VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV INTELIGENTNÍCH SYSTÉMŮ
DEPARTMENT OF INTELLIGENT SYSTEMS

GENEROVÁNÍ RODOKMENŮ Z MATRIČNÍCH ZÁZNAMŮ

FAMILY TREES MAKING FROM PARISH RECORDS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE **ADAM MARHEFKA**
AUTHOR

VEDOUCÍ PRÁCE **Ing. JAROSLAV ROZMAN, Ph.D.**
SUPERVISOR

BRNO 2022

Ústav inteligentních systémů (UITS) Akademický rok 2021/2022

Zadání bakalářské práce

Student: **Marhefka Adam**
Program: Informační technologie
Název: **Generování rodokmenů z matričních záznamů**
Family Trees Making from Parish Records
Kategorie: Umělá inteligence
Zadání:

1. Nastudujte obor genealogie a materiály v ní používané, především matriční knihy. Nastudujte práce, zabývající se automatickým propojováním matričních záznamů (křty, svatby, úmrtí) do větších rodokmenů.
2. Na základě nastudované literatury navrhněte způsob, jak záznamy propojovat do větších celků. Počítejte s tím, že pro danou oblast nebudou v danou chvíli k dispozici všechny záznamy, ty budou teprve průběžně přibývat. Dále systém navrhněte jako pravděpodobnostní, tzn. dítě může mít více rodičů, pro které se budou počítat pravděpodobnosti, které se budou s postupně přidávanými záznamy měnit. Data načítejte z databáze a opět je do databáze ukládejte.
3. Navržený systém implementujte.
4. Testování proveďte na dodané testovací sadě.

Literatura:

- Dintelman, S., Maness, T.: Reconstituting the Population of a Small European Town Using Probabilistic Record Linking: A Case Study, Family History Technology Workshop, BYU 2009
- Malmi, E., Rasa, M., Gionis, A.: AncestryAI: A Tool for Exploring Computationally Inferred Family Trees, Proceedings of International World Wide Web Conference Comittee, 2017

Pro udělení zápočtu za první semestr je požadováno:

- První dva body zadání.

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

Vedoucí práce: **Rozman Jaroslav, Ing., Ph.D.**
Vedoucí ústavu: Hanáček Petr, doc. Dr. Ing.
Datum zadání: 1. listopadu 2021
Datum odevzdání: 11. května 2022
Datum schválení: 3. listopadu 2021

Abstrakt

Práca sa zaoberá oborom genealógie, spracovávaním, porovnávaním a klasifikovaním genealogických dát a následným prepájaním týchto dát do väčších celkov v grafovej databáze. Táto práca priamo nadväzuje na prácu Ing. Tušimovej a ďalej ju rozširuje. Rozšírenia popísané v práci sa budú týkať pridania podpory pre ďalšie formy vstupných genealogických dát, optimalizácií a vylepšení hlavného algoritmu realizujúceho spracovávanie genealogických záznamov. Cieľom práce bude teda zefektívniť algoritmus, zvýšiť jeho presnosť a pridať podporu pre matriky sobášov a úmrtí.

Abstract

This work will discuss the field of genealogy, processing, comparing, and classifying genealogical data and subsequent connection of this processed data into bigger structures in graph database. This work directly continues the work of Ing Tušimová and further expands it. The expansions described in the work will be dedicated to addition of support for more forms of input genealogical data, optimization and expansions of the core algorithm realizing the processing of genealogical records. The goal of this work is therefore to make the core algorithm more efficient and precise as well as the addition of support for parish records of marriages and deaths.

Kľúčové slová

genealógia, grafová databáza, matričné záznamy, spracovávanie záznamov

Keywords

genealogy, graph database, parish records, record processing

Citácia

MARHEFKA, Adam. *Generování rodokmenů z matričních záznamů*. Brno, 2022. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Jaroslav Rozman, Ph.D.

Generování rodokmenů z matričních záznamů

Prehlásenie

Prehlasujem, že som túto bakalársku prácu vypracoval samostatne pod vedením pána Ing. Jaroslava Rozmana Ph.D.. Uviedol som všetky literárne pramene, publikácie a ďalšie zdroje, z ktorých som čerpal.

....
Adam Marhefka
11.5.2022

Poďakovanie

Ďakujem vedúcemu mojej práce, Ing. Jaroslavovi Rozmanovi Ph.D. za jeho ochotu a odbornú pomoc pri vypracovávaní tejto bakalárskej práce.

Obsah

1 Úvod **3**

2 Naštudované materiály a teória **4**
2.1 Kľúčové pojmy a ich vysvetlenie . . . 4
2.2 Podkladová práca . . . 5
2.2.1 Návrh . . . 5
2.2.2 Použité technológie . . . 9
2.2.3 Implementácia . . . 10
2.2.4 Dodatočné skripty . . . 14

3 Návrh **16**
3.1 Rozšírenia . . . 16
3.1.1 Porovnávanie slov . . . 16
3.1.2 Ďalšie vstupné dáta . . . 19
3.2 Analýza efektivity a problémov algoritmu . . . 22
3.2.1 Zistenie doby trvania spracovávania . . . 23
3.2.2 Identifikovanie časovo náročných miest v programe . . . 23
3.2.3 Nový návrh výsledného systému . . . 24

4 Implementácia **27**
4.1 Identifikovanie nedostatkov . . . 27
4.1.1 Počiatočné úpravy . . . 27
4.1.2 Pohlavia osôb . . . 28
4.1.3 Porovnávanie osôb spolu s predkami . . . 28
4.1.4 Práca s dátumami . . . 29
4.1.5 Mazanie databázy . . . 30
4.1.6 Ďalšie optimalizácie . . . 31
4.1.7 Skript get_all_records.py . . . 32
4.2 Implementácia zmeneného systému . . . 32
4.2.1 Rozbor problematiky . . . 32
4.2.2 Reprezentácia dát . . . 32
4.2.3 Štruktúra pre reprezentovanie vzťahov . . . 35
4.2.4 Hierarchia vzťahov uzlov adries . . . 38
4.2.5 Výsledný algoritmus . . . 39
4.3 Nové formy vstupných dát . . . 39
4.3.1 Úpravy pri zmene relačnej databázy . . . 39
4.3.2 Načítanie dát . . . 40
4.3.3 Odhady dátumov . . . 40

4.3.4 Modifikácia riadiaceho algoritmu 40
4.3.5 Porovnávanie 41
4.4 Implementácia nových metrík 41
4.5 Nová možnosť výstupu vyhľadávacích skriptov 42

5 Testovanie **43**
5.1 Testovanie efektivity 43
5.2 Testovanie nových metrík 43
5.3 Vplyv nových záznamov na presnosť 44

6 Záver **45**

Literatúra **46**

Kapitola 1

Úvod

Pokrok v technológii a informatike nám ako spoločnosti umožňuje inováciu vo všetkých oblastiach. Aj pomocné vedy historické ako genealógia sa konečne v dnešnej dobe dočkávajú posunu vpred umožnenému aj vďaka informačným technológiám. Automatizácia v oblasti spracovávania a porovnávania genealogických dát môže ušetriť veľké množstvo potrebnej ľudskej práce. Problematikou automatizácie a inteligentného prepájania záznamov do rodokmeňov sa zaoberá aj práca Ing. Tušimovej [12]. V tejto práci sa detailne venuje spracovávaniu genealogických dát z matrík narodených/pokrstených a následnému generovaniu grafovej databázy obsahujúcej všetky spracované osoby, mestá, záznamy o krste a jednotlivé vzťahy medzi všetkými spracovanými uzlami. Práca Ing. Tušimovej má veľký prínos pre spomínanú oblasť, pretože ju je možné použiť pre zrýchlenie a zefektívnenie práce pri spracovávaní genealogických dát. Avšak vždy je priestor pre zlepšenie. Práve tomuto sa bude venovať moja práca. V prvej časti bakalárskej práce sa budeme zaoberať naštudovanými materiálmi a teoretickými znalosťami, predovšetkým prácou Ing. Tušimovej, na ktorú táto práca nadväzuje. Detailnejšie si rozoberieme časť návrhu a implementácie jej práce. V druhej časti práce sa pozrieme na možné vylepšenia algoritmu, ktoré by viedli ku presnejším výsledkom porovnávaní a klasifikovaní záznamov. Povieme si, akým spôsobom bude realizované pridanie podpory pre ďalšie možnosti vstupných dát, konkrétne matriky sobášov a matriky úmrtí a taktiež si priblížime spôsob zefektívnenia celého algoritmu.

Kapitola 2

Naštudované materiály a teória

V tejto kapitole si vysvetlíme základné pojmy používané v tejto práci, formy genealogických záznamov, detailnejšie si priblížime prácu Ing. Tušimovej a mechanizmus spracovávania genealogických údajov v nej použitý. Všetky informácie použité v tejto kapitole sú prevzaté zo zdrojov [12, 11, 1, 10].

2.1 Kľúčové pojmy a ich vysvetlenie

V tejto podkapitole prejdeme niekoľko základných pojmov týkajúcich sa tejto práce a genealógie všeobecne.

Genealógia

Genealógia je pomocná veda historická zaoberajúca sa štúdiom rodín, rodinnej histórie a tvorbe rodokmeňov. Genealógiu v nejakej forme môžeme nájsť v každom historickom období, nezávisle od národa. Pred zavedením písomných záznamov išlo o orálnu tradíciu predávania informácií, pričom sa spoliehalo iba na pamäť ľudí a prípadné primitívne formy mnemotechnických pomôcok. Počiatky písomných genealogických záznamov siahajú do starovekého Grécka a Ríma, kde sa však ešte nedá hovoriť o genealógii ako o vednej disciplíne, keďže išlo skôr o vedľajšie informácie pri písaných textoch. V období stredoveku sa genealógovia zaoberali najmä rodinnou históriou šľachtických a kráľovských rodov a zostavovaním rodokmeňov týchto rodov. V modernej dobe je genealógia dostupná pre všetkých, vďaka jednoducho dohľadateľným záznamom, ktoré sú dostupné aj širokej verejnosť online alebo v archívoch. Každý sa teda môže stať genealógom, či už skúma iba svoju vlastnú rodinnú históriu alebo sa genealógii venuje profesionálne.

Matrika

Matriky sú formou verejných záznamov, do ktorých sa zapisujú informácie slúžiace ku evidencii obyvateľstva. Keďže cirkev v minulosti zohrávala v spoločnosti významnú úlohu a podieľala sa aj na výkone práva a verejnej správy, pochádzajú prvé záznamy práve od cirkevných inštitúcií. Časom sa však postupne presunuli pod štátnu správu. V matrikách nájdeme všetky skutočnosti dôležité z hľadiska osobného stavu občana. Z tohto dôvodu slúžia ako hlavný zdroj genealogických informácií. Najčastejšou formou vedených matrík sú matriky narodených (alebo krstených), matriky o uzavretých manželstvách a matriky úmrtí. Bližšie podrobnosti o týchto matrikách si povieme v ďalších častiach tejto kapitoly. Až do

roku 1784 boli matriky súkromnou záležitosťou jednotlivých farností a teda nemali jednotný formát a často boli všetky formy matrík vedené v jednej knihe. V tomto roku však za vlády cisára Josefa II. vyšla séria patentov venujúca sa záležitostiam štátnej správy. Jedným z týchto patentov bol aj patent z 1.5.1784, ktorý zaviedol jednotný formát matrík a taktiež rozdelil matriky podľa formy záznamov ktoré v nej boli vedené na matriky narodených, matriky sobášov a matriky úmrtí.

Matrika narodených/krstov

Tieto matriky obsahovali záznamy o pôrodoch a krstoch. V lepšom prípade obsahovali matriky obidva údaje. Všetky údaje vedené v týchto matrikách boli dátum narodenia (alebo krstu), číslo domu, meno narodenej osoby, pohlavie, stav – manželský alebo nemanželský, náboženstvo – katolícke alebo nekatolícke, meno otca, meno matky, meno kmotra a v neskorších záznamoch bolo bežné taktiež meno krstiaceho duchovného a pôrodnej baby.

Matrika úmrtí

Matriky úmrtí obsahovali dátum pohrebu, meno kňaza zodpovedného za zaopatrenie a pochovanie zosnulého, číslo domu, meno, náboženstvo, pohlavie, vek a príčinu smrti zosnulého.

Matrika sobášov

Údaje bežne vedené v týchto matrikách boli dátum sobáša, meno ženícha, meno nevesty, meno kňaza, ktorý ich sobášil a mená svedkov. V novších záznamoch sú bežné aj ďalšie údaje, ako bydlisko ženícha i nevesty, ich vek, náboženstvo a v prípade, že nevesta nebola plnoletá aj povolenie otca ku sobášu.

2.2 Podkladová práca

Táto podkapitola sa bude detailnejšie venovať diplomovej práci Ing. Tušimovej. Priblížime si návrh, použité technológie, implementačné detaily a výstupy práce. Všetky informácie použité v tejto podkapitole boli naštudované a prebrané zo zdroja [12], ktorý slúžil ako hlavný študijný materiál pre túto prácu.

2.2.1 Návrh

Prvou časťou práce je dátová analýza. Najprv ide teda o získanie všetkých potrebných údajov zo záznamu. Údaje ktoré sú pre prácu relevantné sú konkrétne:

- Meno
- Priezvisko
- Titul
- Pohlavie
- Národnosť
- Dátum narodenia
- Dátum krstu
- Interval dátumu narodenia
- Miesta kde osoba žila
- Vierovyznanie
- Dátum úmrtia
- Interval dátumu úmrtia
- Miesto úmrtia
- Povolanie

- Rodinný príslušníci

V prípade, že v zázname chýba presný dátum, dôjde ku odhadu intervalu na základe dostupných informácií. Ak je potom pri spracovávaní ďalších záznamov získaná presnejšia hodnota dôjde ku aktualizácií intervalu a v prípade získania presnej hodnoty sa interval nahradí týmto presným dátumom. Ďalej sa využíva zhlukovanie dát, keďže významovo rovnaké mená a názvy obsiahnuté v matričných záznamoch sa často v zázname od záznamu líšili. Je preto využitá normalizácia, čo je zavedenie jednotnej formy každého mena či názvu. Použitím normalizovanej formy mien a názvov sa eliminuje množstvo chýb, ktoré by mohli vzniknúť používaním nenormalizovaných hodnôt. Ďalej dochádza ku porovnávaniu jednotlivých údajov. Porovnávanie sa líši na základe typu údaju. Všeobecne sa porovnávanie vykonávané v práci dá rozdeliť na porovnávanie slov, porovnávanie dátumov, porovnávanie geografických dát, porovnávanie čísel a finálne porovnanie celého záznamu zostavením vektoru hodnôt z jednotlivých porovnaní.

Porovnávanie slov

Zisťovanie podobnosti slov, ktoré nie sú normalizované je v práci realizované pomocou Levenshteinovej vzdialenosti. Levenshteinova vzdialenosť nám udáva editačnú vzdialenosť dvoch reťazcov. Ide o algoritmus pre zistenie minimálneho počtu jednoznakových operácií potrebných pre transformáciu jedného reťazca na druhý. Týmito operáciami sú vloženie znaku, mazanie znaku a substitúcia znaku.

		0	1	2	3	4	5
			P	**E**	**T**	**E**	**R**
0		**0**	1	2	3	4	5
1	**P**	1	**0**	1	2	3	4
2	**E**	2	1	**0**	1	2	3
3	**T**	3	2	1	**0**	1	2
4	**R**	4	3	2	1	**1**	2
5	**A**	5	4	3	2	2	**2**

Tabuľka 2.1: Príklad určenia Levenshteinovej vzdialenosti dvoch reťazcov

V tabuľke 2.1 prevzatej zo zdroja [12] vidíme príklad určenia Levenshteinovej vzdialenosti reťazca *Peter* a reťazca *Petra*. Vzdialenosť týchto dvoch reťazcov je v tomto prípade 2, keďže pre transformáciu prvého reťazca na druhý je nutné použiť 2 jednoznakové operácie substitúcie. Podobnosť dvoch reťazcov vypočítaná pomocou Levenshteinovej vzdialenosti vyzerá následovne:

$$sim_{levenshtein}(s_1, s_2) = 1.0 - \frac{dist_{levenshtein}(s_1, s_2)}{max(|s_1|, |s_2|)}$$

$sim_{levenshtein}$ udáva výslednú podobnosť reťazcov s_1 a s_2. Tú získame pomocou podielu Levenshteinovej vzdialenosti reťazcov $dist_{levenshtein}$ a maxima max z dĺžok reťazcov a následným odčítaním hodnoty od 1. Výsledné hodnoty sa teda budú blížiť 1 pri vysokej podobnosti reťazcov a 0 pri nízkej podobnosti.

$$sim_{levenshtein}(s_1, s_2) = \begin{cases} 1.0 & ak\ s_1 = s_2 \\ 0.0 & ak\ s_1 \neq s_2. \end{cases}$$

Porovnanie dátumov

Pre porovnávanie dátumov sa okrem Levenshteinovej vzdialenosti prístup kombinuje s porovnávaním pomocou porovnávania veku, pričom sa nakoniec použije hodnota s vyššou podobnosťou dvoch údajov. Pre porovnávanie pomocou veku je nutné určiť hodnotu tolerancie v percentách apc_{max}, kvôli častým nepresným údajom z minulosti. Následne sa určí vekový rozdiel v percentách apc (age percentage difference).

$$apc = \frac{|d_1 - d_2|}{max(|d_1|, |d_2|)} \cdot 100$$

d_1 a d_2 predstavujú hodnoty veku. Na základe rozdielu je určená podobnosť veku.

$$sim_{age} = \begin{cases} 1.0 - \frac{apc}{apc_{max}} & ak \quad apc < apc_{max} \\ 0.0 & inak. \end{cases}$$

Okrem porovnávania veku sa používa kontrola dátumov, keďže častou chybou v zápisoch bolo prehodenie údajov o dni a mesiaci. Preto v prípade, že hodnota o mesiaci presahuje 12 a hodnota údaju o dni nepresahuje túto hodnotu, predpokladá sa zamenenie týchto údajov.

Porovnanie geografických dát

Ďalšou metódou porovnávania použitou v práci je porovnávanie geografických súradníc. Tu sa meria vzdialenosť medzi miestami pozdĺž zemského povrchu v kilometroch. Čím bližšie sú miesta dvoch porovnávaných osôb, tým je vyššia pravdepodobnosť, že sa jedná o tú istú osobu. V prípade, že údaj o presnej adrese nie je dostupný, ale podarí sa vyhodnotiť napríklad iba región, porovnávanie je uskutočnené od stredu regiónu. V tomto prípade ide však iba o približnú hodnotu a údaj a pri porovnávaní je mu priradená nižšia váha.

Porovnanie čísel

Tento spôsob porovnávania je použitý iba pri porovnávaní čísla ulice a je ďalej kombinovaný s Levenshteinovou vzdialenosťou. Podobnosť čísel sa vypočíta nasledujúcim spôsobom:

$$sim_{num} = \begin{cases} 1.0 - \frac{|n_1 - n_2|}{dmax} & ak \quad |n_1 - n_2| < d_{max} \\ 0.0 & inak. \end{cases}$$

Kde n_1 a n_2 predstavujú porovnávané čísla a $dmax$ predstavuje maximálny tolerovaný rozdiel medzi nimi.

Porovnanie záznamov

Záznamy sa skladajú z viacerých atribútov rôznych typov a predstavujú informácie o jednej osobe. Pre porovnanie týchto atribútov sú použité vyššie spomínané metódy. Pre následné porovnanie celých záznamov je zostavený porovnávací vektor z jednotlivých hodnôt porovnávaní atribútov. Jednotlivým atribútom sú potom priradené váhy, ktorými sa prislúchajúca hodnota porovnania pred sčítaním násobí.

ID	Meno	Priezvisko	Pohlavie	Dátum narodenia	Č. domu	Názov ulice	Mesto
a1	alica	mlynárová	žena	18.10.1956	15	božetechova	brno
a2	alica	mikuláková	žena	18.10.1958	68	dožetekova	břeclav
	1.0	0.0	1.0	0.87	0.0	0.73	0.0

Tabuľka 2.2: Príklad vzniknutého porovnávacieho vektoru dvoch osôb

Klasifikácia

Aby sme určili vzťah ktorý prepája dve osoby, je nutné najprv zistiť či nejde o tú istú osobu. To overíme porovnaním pomocou atribútov. Dvojicu porovnávaných osôb si označíme ako r_i a r_j. Po sčítaní všetkých hodnôt výsledného porovnávacieho vektoru týchto osôb dostávame hodnotu celkovej zhody $SimSum[r_i, r_j]$. Na základe tejto hodnoty a nastaviteľných prahov t_u a t_l vieme kategorizovať dvojicu ako zhodu, potencionálnu zhodu a nezhodu.

$$SimSum[r_i, r_j] \geq t_u \implies r \rightarrow Zhoda,$$
$$t_l < SimSum[r_i, r_j] < t_u \implies r \rightarrow Potencionlna \quad zhoda,$$
$$SimSum[r_i, r_j] \leq t_l \implies r \rightarrow Nezhoda$$

Táto klasifikácia, však neberie do úvahy jednotlivé váhy atribútov. Preto je nutné dopočítať tieto váhy.

Pravdepodobnostná klasifikácia

Práca používa pravdepodobnostnú klasifikáciu predstavenú v práci Ivana Fellegiho a Alana Suntera z roku 1969 [8]. Sú zavedené dve populácie A a B ktorým prislúchajú prvky a a b v tomto poradí, pričom sa predpokladá, že existujú prvky, ktoré sú pre populácie A a B spoločné. Následne si zadefinujeme množinu usporiadaných dvojíc definovanú následovne:

$$a \ \times B = \{(a, b) : a \ \in A, b \in B\}$$

Táto množina je výsledkom zjednotenia dvoch disjunktných množín, konkrétne množiny reprezentujúcej zhodné dvojice, teda prvky (v našom prípade osoby) sa zhodujú

$$M = \{(a, b) : a = b, a \ \in A, b \in B\}$$

a množiny reprezentujúcej nezhodné dvojice

$$U = \{(a, b) : a \neq b, a \ \in A, b \in B\}$$

Pri každej dvojici máme už spomínaný porovnávací vektor zložený z hodnôt porovnaní jednotlivých atribútov. Pravdepodobnosť zhody sa potom určí ako podmienená pravdepodobnosť zhodných a nezhodných údajov v zázname.

$$R = \frac{P(\gamma \in \Gamma | r \in M)}{P(\gamma \in \Gamma | r \in U)}$$

Pomocou tejto pravdepodobnosti a prahov určíme kategóriu do ktorej porovnávané osoby patria.

$$R \geq t_u \implies r \rightarrow Zhoda,$$
$$t_l < R < t_u \implies r \rightarrow Potencionlna \quad zhoda,$$
$$R \leq t_u \implies r \rightarrow Nezhoda$$

Táto klasifikácia je základom pravdepodobnostného prepájania záznamov.

Pri predpoklade, že tieto pravdepodobnosti sú pre jednotlivé atribúty záznamov nezávislé, je možné každému atribútu vypočítať váhu. Váha atribútu w_i pre atribút i sa vypočíta na základe pravdepodobnosti m_i, ktorá udáva pravdepodobnosť, že dva záznamy majú rovnakú hodnotu tohto atribútu kvôli tomu že sa jedná o zhodné záznamy

$$m_i = P([a_i = b_i, a \ \in A, b \in B] | r \in M)$$

a pravdepodobnosti u_i, ktorá udáva pravdepodobnosť toho, že dva záznamy majú rovnakú hodnotu atribútu aj napriek tomu, že sa jedná o záznamy, ktoré sa nezhodujú.

$$u_i = P([a_i = b_i, a \ \in A, b \in B] | r \in U),$$

Váha w_i atribútu i je potom určená nasledovne:

$$w_i = \begin{cases} log_2 \frac{m_i}{u_i} & ak \quad a_i = b_i, \\ log_2 \frac{1-m_i}{1-u_i} & a_i \neq b_i \end{cases}$$

2.2.2 Použité technológie

V tejto podkapitole si stručne povieme niečo o technológiách použitých v implementácii.

Python 3

Práca je implementovaná v skriptovacom jazyku python 3, ktorý je efektívny a jednoduchý na použitie. Jedná sa o interpretovaný, dynamicky typovaný jazyk. Obsahuje veľké množstvo knižníc, ktoré implementujú potrebné funkcionality využívané prácou.

Použité knižnice

- Levenshtein - výpočet Levenshteinovej editačnej vzdialenosti slov
- geopy – výpočet vzdialenosti medzi GPS súradnicami
- numpy – matematická knižnica
- datetime – práca s dátumami
- json – spracovanie json súborov
- pandas – spracovanie csv súborov
- time – meranie času výpočtu

Vstupné dáta

Vstupné dáta sa líšia podľa použitia programu. Je možné program spustiť v testovacom móde, kedy je vstup načítaný z csv súborov (z angličtiny „comma separated values“), teda súbory s hodnotami oddelenými čiarkou. V týchto súboroch je množstvo záznamov výrazne menšie ako v použitej MySQL databáze, a preto sú ideálne na testovanie. Ďalšou formou vstupu sú dáta z už spomenutej MySQL databázy. Pre pripojenie do databázy sa využíva ovládač MySQL Connector/Python [4]. Tento ovládač umožňuje jednoducho komunikovať s databázovým serverom typu MySQL. Na dotazovanie nad databázou je využitý dotazovací jazyk SQL. Ďalšou formou vstupu sú dáta so súradnicami GPS. Tie sú obsiahnuté v súboroch typu JSON (JavaScript Object Notation).

Grafová databáza

Pre uloženie spracovaných dát vo forme grafu je využitá grafová databáza Neo4j. Ide o jednu z najviac používaných grafových databáz. Pre pripojenie ku tejto databáze je použitý ovládač Neo4j Python [3]. Na dotazovanie nad grafovou databázou Neo4j sa používa dotazovací jazyk Cypher.

2.2.3 Implementácia

V tejto kapitole si stručne prejdeme implementáciu algoritmu.

Načítanie dát z relačnej databázy

V práci sa pracuje s dátami z relačnej databázy zo školského serveru *fit.vutbr.cz*. Ide o MySQL databázu obsahujúcu tabuľky pre všetky záznamy, ktoré môžu byť obsiahnuté v matrikách narodených ako aj o matrikách samotných. Databáza ďalej obsahuje aj osobitné tabuľky pre normalizované formy záznamov. Na prípravu a normalizáciu je použitá práca Bc. Davida Hříbeka s názvom Poloautomatická normalizace slov z matričních záznamů [9]. Prvým krokom algoritmu je pripojenie do databáz pomocou už spomenutých ovládačov. Následne sa načítajú všetky matričné záznamy, ktoré majú formu tabuľky *birth*.

Táto tabuľka predstavuje matričný záznam z matriky narodených. Vytvára sa nový objekt *record*, do ktorého sa uložia informácie o zázname. Následne sa načítajú údaje o prislúchajúcej matrike a uložia sa do objektu *register*, ktorý reprezentuje matriku a informácie o nej.

birth	
id	int(10) unsigned
register_id	mediumint(8) unsigned
scan	smallint(5) unsigned
pos	tinyint(3) unsigned
lay	char(1)
fin	tinyint(1)
baptism_date	date
sex	enum('m', 'f', 'u')
legitimate	enum('legitimate', 'illegitimate', 'u')
mult	tinyint(3) unsigned
title	varchar(255)
dead_date	date
dead_where	varchar(255)
confirmation_when	date
confirmation_where	varchar(255)
confirmation_sname	mediumint(8) unsigned
church_getoff	date
church_getoff_where	varchar(255)
church_reenter	date
parents_marr_when	date
signs	tinyint(1)
comment	varchar(255)
lang	enum('cz', 'ge', 'lat', 'sc', 'pl', 'sk', 'none')
owner	int(10) unsigned

Obr. 2.1: Tabuľka so záznamom z matriky narodených

Ďalším krokom je získanie všetkých osôb. Tie sú uložené v tabuľke *birthPerson*, ktorú môžete vidieť na obrázku 2.1. Cez jednotlivé získané dáta o osobách sa ďalej prechádza a získajú sa všetky potrebné informácie pre vytvorenie objektov *Person*, ktoré reprezentujú osoby figurujúce v zázname.

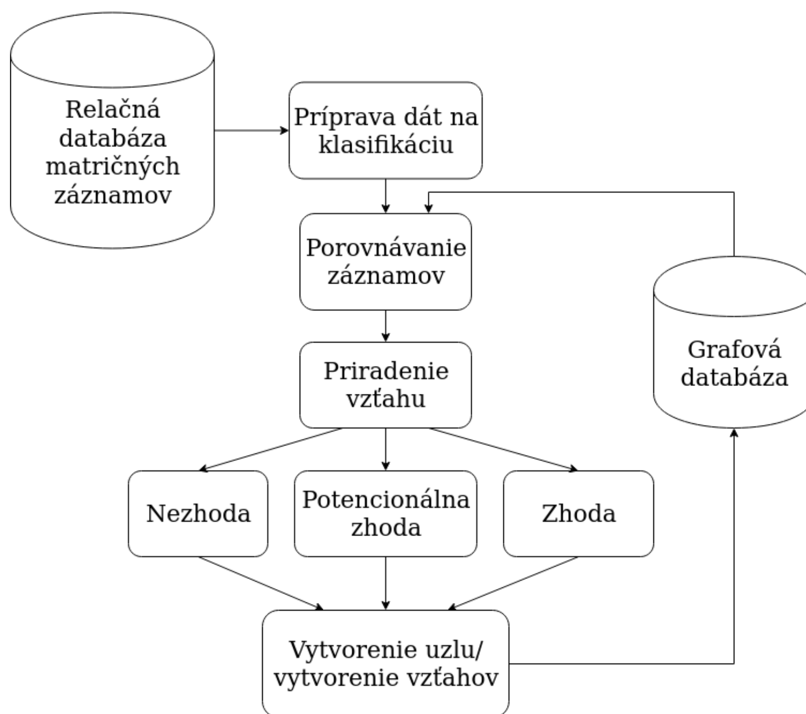
Načítanie dát z grafovej databázy

Z grafovej databázy sa načítavajú dáta pomocou dotazov v jazyku Cypher. Z grafovej databázy potrebujeme získať údaje o osobách kvôli porovnaniu, preto sú dotazy cielené na osoby a nie na záznamy. Dotazy v jazyku Cypher vrátia objekt typu slovník, ktorý má štruktúru uzla z Neo4j. V tomto objekte sú uložené všetky údaje o osobe, s výnimkou adresy, ktorá je daná vzťahom *BÝVA* medzi uzlami osôb a uzlami adries.
GPS súradnice sú načítané z JSON súborov a sú priradené obci ihneď potom čo sa obec priradí osobe.

Porovnávanie a klasifikácia

Porovnanie osôb je rozdelené podľa ich pohlavia, pričom osoby s nedefinovaným pohlavím sú porovnávané aj s mužmi aj so ženami. Prvým krokom porovnania je kontrola či osoba v čase vzniku záznamu mohla žiť. V prípade splnenia tejto kontroly sa prechádza ku časti základného porovnania, pri ktorom sa skontroluje meno, priezvisko, povolanie a mesto v ktorom osoba žila. Ak sa ani jeden z týchto údajov medzi porovnávanými osobami nezhoduje je dvojica označená ako nezhoda. V opačnom prípade sa pokračuje detailným porovnaním ľudí. To je realizované v triede *comparator*. Tu sú údaje porovnávané už spomínanými metódami a z najlepších výsledkov jednotlivých porovnaní sa vytvára slovník *comparison*. Porovnávanie často nekončí iba pri porovnaní osôb pôvodnej dvojice. Ak je to možné porovnáva sa ďalej na základe vzťahov *JE_OTEC* a *JE_MATKA* a porovnanie pokračuje rodičmi osoby z pôvodného porovnania. Rodič zo záznamu je vyhľadaný v grafovej databáze a porovnanie pokračuje. Opäť vzniká objekt *comparison* reprezentujúci porovnávací vektor. Takýmto spôsobom sa pokračuje tak ďaleko ako nám povolí spracovávaný matričný záznam. Všetky výsledky porovnaní sa ukladajú do zoznamu pre výpočet výsledného porovnania.

Po ukončení porovnávania sa spočítajú všetky hodnoty v slovníku *comparison* vynásobené ich príslušnými váhami, pričom hodnoty -1 označujú chýbajúci údaj a sú preto vynechané z výpočtu. Výsledok je potom porovnaný s nastavenými prahmi a podľa toho klasifikovaný. Návrh výsledného systému môžeme vidieť na obrázku 2.2.



Obr. 2.2: Navrh prepojovania záznamov

Uloženie záznamu

Prvým krokom pri ukladaní záznamov je vytvorenie samotného uzlu so záznamom o krste. Následne dôjde ku kontrole či matrika v ktorej bol záznam uložený už existuje ako uzol v grafovej databáze. Ak nie, je vytvorený nový uzol s matrikou a ten je následne prepojený so záznamom o krste použitím vzťahu *JE_V*. V prípade že matrika už existuje, dôjde iba ku vytvoreniu prepojenia. Ďalej sa pokračuje podľa výsledku porovnania dvoch osôb. V prípade že bola dvojica klasifikovaná ako nezhoda dôjde ku vytvoreniu nového uzla s osobou. V prípade, že bola určená potencionálna zhoda dôjde taktiež ku vytvoreniu nového uzla s osobou, ale v tomto prípade je nový uzol prepojený vzťahom *POTENCIONALNI_ZHODA* ku uzlu ku ktorému sa táto zhoda vzťahuje. V oboch prípadoch je nutné vykonať kontrolu, či existuje uzol s adresou na ktorej daná osoba bývala. V prípade, že adresa neexistuje je pridaná nová adresa. Údaj o adrese je zložený z uzlu obce či mesta, ulice a popisného čísla, pričom medzi týmito uzlami funguje hierarchické prepojenie pomocou vzťahu *JE_V*. Osoba je potom prepojená vzťahom *BÝVA* na najdetailnejší údaj. V prípade že adresa už existuje je iba vytvorené prepojenie medzi osobou a najdetailnejším údajom z adresy. Pri vzťahu medzi adresou a osobou sa ukladá aj informácia o tom kedy daná osoba na adrese bývala. V prípade klasifikácie dvojice osôb ako zhody dôjde ku volaniu funkcie *update_node_person* z triedy *person.py*. Tá aktualizuje existujúci uzol o nové hodnoty a dodatočné údaje ako napríklad povolanie, ktorých človek môže mať za život viac uloží do listu. Po aktualizácii osoby v dôsledku zhody dôjde ku prehodnoteniu vzťahov potencionálnych zhôd, ktoré sa môžu po doplnení nových údajov zmeniť. Ak dôjde ku reklasifikovaniu potencionálnych zhôd na zhody proces sa opakuje. Potom čo sú vytvorené všetky uzly začne prepájanie uzlov.

Metódou *create_connection_with_person_in_birth_record* sa iteruje cez osoby a na základe ich roly sú prepojené so záznamom a aj medzi sebou. Po vytvorení vzťahov medzi osobami záznamu sa prechádza na ďalší záznam a proces sa opakuje kým nie je spracovaná celá databáza. V tabuľke 2.3 prevzanej zo zdroja [12] môžeme vidieť prehľad všetkých modulov tvoriacich obsah práce.

Triedy	Význam triedy
main.py	Základný súbor, spracováva pripojenia na databázu a riadi testovanie
create_database.py	Hlavný algoritmus celého programu riadi načítavanie záznamov, vytváranie grafovej databázy a spracováva výsledky porovnávania
relational_database_birth.py	Načítava dáta z relačnej databázy
graph_database.py	Načítava, aktualizuje, vyhľadáva, vytvára uzly a prepojenia v grafovej databáze
csv_source.py	Načítava dáta z CSV súborov
comparator.py	Porovnáva osoby a vyhodnocuje výsledky, obsahuje všetky metódy potrebné k porovnávaniu
record.py	Trieda spravujúca záznam, obsahuje vytváranie uzlu, aktualizovanie uzlu, výpis informácii
person.py	Trieda spravujúca osobu, vytváranie uzlu, aktualizovanie uzlu, výpis informácií
register.py	Trieda spravujúca matriky, vytváranie uzlu, aktualizovanie uzlu, výpis informácií
domicile.py	Trieda spravujúca adresy, prehľadávanie json súborov, výpis informácií
date.py	Trieda spravujúca dátumy
get_persons.py	Skript na základe mena a priezviská nájde všetky zodpovedajúce osoby uložené v grafovej databáze
get_all_records.py	Skript na základe id osoby nájde všetky záznamy v grafovej databáze, v ktorých sa osoba nachádza
get_family_tree.py	Skript na základe id osoby vypíše všetkých predkov a potomkov ktorý sa nachádzajú v grafovej databáze

Tabuľka 2.3: Prehľad tried a ich význam

2.2.4 Dodatočné skripty

Okrem programu realizujúceho hlavný algoritmus načítania, porovnávania, klasifikovania a ukladania záznamov do grafovej databázy obsahuje práca Ing. Tušimovej aj ďalšie skripty realizujúce operácie, ktoré sú z hľadiska genealogického výskumu zaujímavé. Skripty realizujú dotazovanie nad grafovou databázou a vrátia výsledky vyhľadávania podľa zadaných vstupných parametrov.

get_person.py

Skript vráti výsledok vyhľadávania podľa mena a priezviska hľadanej osoby. Zobrazí všetky informácie dostupné o tomto človeku. Prehľadávanie prebieha jednak s pôvodným menom a aj s normalizovanou formou. Výstupom skriptu sú potom všetky informácie dostupné o danom človeku, vrátane jeho ID v grafovej databáze. To môže byť následne použité pri spúšťaní zvyšných skriptov, ktoré ako vstupný parameter vyžadujú ID osoby.

get_all_records.py

Skript realizuje vyhľadávanie na základe ID osoby. Skript získa a vypíše všetky záznamy v ktorých hľadaná osoba figuruje ako aj všetky informácie o danej osobe a jej rolách v konkrétnych záznamoch.

get_family_tree.py

Skript realizuje vyhľadávanie na základe ID osoby. Skript hľadá všetky osoby v prepojení *JE_MATKA, JE_OTEC* oboma smermi. Teda hľadá predkov aj potomkov zadanej osoby. Skript oboma smermi funguje rekurzívne a jeho výstupom je forma rodokmeňu.

Kapitola 3

Návrh

V tejto kapitole si prejdeme návrh rozšírení a modifikácií, ktoré neskôr budú realizované v rámci implementácie. Prejdeme si možnosti vylepšení algoritmu a rozšírení možností vstupných dát. Ďalej si priblížime chyby a nedostatky zdrojovej práce, ktoré bolo nutné vyriešiť. Na záver sa pozrieme aj na možnosti optimalizácie a zvýšenia časovej efektivity skriptu.

3.1 Rozšírenia

V tejto podkapitole si ukážeme akou formou bude pôvodná práca rozšírená. Rozšírenia budú zamerná hlavne na pridanie nových typov záznamov a testovaní nových alternatív ku metrike použitej pre porovnávanie reťazcov v práci.

3.1.1 Porovnávanie slov

V tejto podkapitole sa pozrieme na rôzne alternatívy metrík použitých pri porovnávaní slov, ktoré by mohli viesť ku lepším výsledkom, prípadne aj ku lepšej časovej efektivite programu. Informácie v tejto podkapitole sú prevzaté zo zdroja [7].

Jarova podobnosť

Ak pri porovnávaní narazíme na meno, ktoré v databáze nemá normalizovanú formu, používa sa pre určenie podobnosti mien už spomínaná Levenshteinova editačná vzdialenosť. Existuje však prístup, ktorý by mohol byť pre porovnávanie kratších reťazcov vhodnejší a tým je Jarova podobnosť. Táto metóda je primárne určená pre kratšie reťazce a špecificky napríklad pre mená a priezviská. Jej časová náročnosť by taktiež mala byť lepšia ako použitá Levenshteinova editačná vzdialenosť. Metóda je podobne ako Levenshteinova vzdialenosť založená na editačnej vzdialenosti dvoch reťazcov. Avšak okrem jednoznakových operácií ponúkaných Levenshteinovou vzdialenosťou – vloženie, mazanie a substitúcia, pridáva Jarova metóda ďalšiu operáciu, ktorou je transpozícia susedných znakov. Tento fakt je obzvlášť relevantný, kvôli charakteru dát s ktorými sa táto práca zaoberá. Chyba zamenenia susedných znakov v písaných textoch je jednou z najčastejších chýb, ktorých sa ľudia dopúšťajú. Jarovu podobnosť sim_j dvoch reťazcov s_1 a s_2 vieme určiť týmto spôsobom:

$$sim_{\mathrm{j}} = \begin{cases} 0 & ak \quad m = 0, \\ \frac{1}{3} \cdot \left(\frac{m}{|s_1|} + \frac{m}{s_2} + \frac{m-t}{m}\right) & inak \end{cases}$$

Kde $|s_i|$ udáva dĺžku reťazca s_i, m udáva počet zhodujúcich sa znakov medzi dvoma reťazcami a t udáva počet transpozícií. Dva znaky patriace reťazcom s_1 a s_2 v tomto poradí môžeme prehlásiť za zhodné ak sú znaky rovnaké a zároveň pre ich vzájomnú vzdialenosť d platí nasledujúci vzťah:

$$d \leq \left\lfloor \frac{max(|s_1|, |s_2|)}{2} \right\rfloor - 1$$

Počet transpozícií je určený ako počet zhodujúcich sa znakov, ktorých sekvenčné poradie je v porovnávaných reťazcoch odlišné vydelený dvomi.

Jaro-Winklerova podobnosť

Jaro-Winklerova podobnosť z princípu vychádza z Jarovej podobnosti, ale kladnejšie hodnotí reťazce, ktoré sa zhodujú v prefixe až do dĺžky štyroch znakov. Pre určenie Jaro-Winklerovej podobnosti sim_w dvoch reťazcov použijeme nasledujúci vzťah:

$$sim_w = sim_j + L \cdot p \cdot (1 - sim_j)$$

Kde sim_j je hodnota Jarovej podobnosti dvoch reťazcov, L udáva dĺžku spoločného prefixu reťazcov až do maximálnej dĺžky prefixu 4 a p je konštantná hodnota váhového faktoru so štandardnou hodnotou $p = 0.1$. Výraz (1 - sim_j nám udáva hodnotu Jarovej vzdialenosti (V podstate ide o obrátenú hodnotu Jarovej podobnosti). Táto metóda je obzvlášť relevantná, pretože ponúka výhody Jarovej podobnosti a k tomu pozitívnejšie hodnotí reťazce so zhodným prefixom, čo je práve časté pri variáciách mien, ktoré nemajú uloženú normalizovanú formu.

Návrh implementovania metód

Spomínané metódy je možné do programu implementovať viacerými spôsobmi. Keď zohľadníme fakt, že metódy Jarovej vzdialenosti sú určené primárne pre kratšie reťazce a sú obzvlášť efektívne pre porovnávanie mien, môžeme logicky dôjsť ku záveru, že bude vhodné

testovať dĺžku porovnávaných reťazcov pre určenie metódy, ktorá sa má použiť pre ich porovnanie. Taktiež bude zohľadnený aj fakt, či ide o meno alebo priezvisko, ktoré pri nenormalizovanej forme môžu zdieľať rovnaký prefix. Najlepší prístup bude určený testovaním nad testovacími dátami a sledovaním presnosti klasifikácie záznamov a časovej náročnosti výsledného algoritmu.

Premenná	Typy porovnávania
Meno	Presná zhoda, Levenshteinova vzdialenosť
Priezvisko	Presná zhoda, Levenshteinova vzdialenosť
Povolanie	Presná zhoda, Levenshteinova vzdialenosť
Mesto	Presná zhoda, vzdialenosť miest
Ulica	Levenshteinova vzdialenosť
Popisné číslo	Levenshteinova vzdialenosť, porovnávanie čísel
Dátum narodenia	Levenshteinova vzdialenosť, kontrola dátumov, porovnanie veku
Dátum (zvyšné typy)	Levenshteinova vzdialenosť, kontrola dátumov
Titul	Levenshteinova vzdialenosť
Viera	Levenshteinova vzdialenosť
Značky	Presná zhoda

Tabuľka 3.1: Prehľad atribútov a spôsobov ich porovnávania

V tabuľke 3.1 môžeme vidieť všetky atribúty a spôsob akým sú momentálne porovnávané. Po zmenách v porovnávaní sa tabuľka zmení nasledujúcim spôsobom:

Premenná	Typy porovnávania
Meno	Presná zhoda, Jarova vzdialenosť
Priezvisko	Presná zhoda, Jarova vzdialenosť
Povolanie	Presná zhoda, Jarova vzdialenosť
Mesto	Presná zhoda, vzdialenosť miest
Ulica	Jarova vzdialenosť
Popisné číslo	Jarova vzdialenosť, porovnávanie čísel
Dátum narodenia	Jarova vzdialenosť, kontrola dátumov, porovnanie veku
Dátum (zvyšné typy)	Jarova vzdialenosť, kontrola dátumov
Titul	Jarova vzdialenosť
Viera	Jarova vzdialenosť
Značky	Presná zhoda

Tabuľka 3.2: Prehľad atribútov a nových spôsobov ich porovnávania

Je nutné si však uvedomiť, že pri porovnávaní niektorých atribútov bude stále možno použitá Levenshteinova vzdialenosť, keďže tá môže byť efektívnejšia napríklad pri dlhších reťazcoch. Bude preto nutné určiť istú formu hranice, pri ktorej sú výsledky dosiahnuté použitím Jarovej vzdialenosti efektívnejšie, ako tie dosiahnuté použitím Levenshteinovej vzdialenosti. Taktiež bude otestovaná efektivita Jaro-Winklerovej podobnosti a bude určený najlepší prístup. Táto hranica bude teda určená testovaním. V tabuľke 3.2 môžeme vidieť nové prístupy ku porovnávaniu jednotlivých atribútov.

3.1.2 Ďalšie vstupné dáta

V tejto podkapitole, si priblížime nové formy vstupných dát a urobíme základný návrh pridania podpory.

Matriky úmrtí a sobášov

Práca Ing. Tušimovej momentálne pracuje iba s databázou obsahujúcou záznamy z matrík narodených. Jednou z hlavných úloh mojej práce bude teda pridanie podpory pre záznamy matrík sobášov a úmrtí. Toto bude realizované rovnakým spôsobom ako práca so záznamami z matrík pôrodov. Pri hlavnom dotaze z funkcie v module *create_database.py*, ktorým získavame všetky záznamy z matrík pôrodov si naberieme aj záznamy z matrík úmrtí a sobášov. To znamená, že budeme musieť vytvoriť nové funkcie realizujúce dotazy na SQL databázu v module *relational_database_birth.py* aby sme získali všetky ďalšie záznamy a v nich figurujúce osoby, pričom jednotlivým záznamom priradíme už predpripravený typ, podľa toho o aký záznam pôjde. V práci bude použitá nová databáza typu MySQL s názvom demos. V tejto databáze už budú obsiahnuté aj záznamy o úmrtiach a sobášoch. Dotazy budú teda konkrétne na tabuľku *burial*, v ktorej sú uložené záznamy z matrík úmrtí a tabuľku *marriage*, v ktorej sú zasa záznamy o sobášoch.

Tabuľka 3.3: Štruktúra tabuľky burial

Column	Type	Null
id	int	No
register_id	mediumint	No
scan	smallint	No
pos	tinyint	No
hline lay	char(3)	Yes
fin	tinyint(1)	Yes
check_req	tinyint(1)	Yes
lang	enum('CZ', 'GE', 'LAT', 'SC', 'PL', 'SK', 'NONE')	Yes
sex	enum('M', 'F', 'U', '[M]', '[F]')	Yes
dead_born	tinyint(1)	Yes
viaticum_date	char(10)	Yes
dead_date	char(10)	Yes
burial_date	char(10)	Yes
dead_time	text	Yes
viaticum	tinyint(1)	Yes
burial_place	text	Yes
death_place	text	Yes
years	decimal(5,2)	Yes
months	decimal(5,2)	Yes
weeks	decimal(5,2)	Yes
days	decimal(5,2)	Yes
hours	decimal(5,2)	Yes
minutes	decimal(5,2)	Yes
death_cause	mediumint	Yes
examination	tinyint(1)	Yes

Tabuľka 3.3: Štruktúra tabuľky burial (pokračovanie)

Column	Type	Null
marriage_date	char(10)	Yes
marriage_place	text	Yes
death_address	int	Yes
birth_address	int	Yes
baptised	tinyint(1)	Yes
legitimate	enum('legitimate', 'illegitimate', 'U')	Yes
marriage_years	decimal(5,2)	Yes
comment	text	Yes
owner	int	Yes
score	double	Yes
last_edited_level	int	Yes
last_edited_highest_level	int	Yes

Z tabuľky 3.3 vidíme, že tabuľka záznamu o úmrtí obsahuje všetky potrebné informácie o zázname, ako jazyk v ktorom bol záznam napísaný, číslo skenu a pozícia na skene. Okrem toho záznam obsahuje aj id matriky, ktoré nám určuje matriku, z ktorej záznam pochádza. Ďalej sú tu aj relevantné informácie o zosnulom ako príčina smrti, dátum posledného zaopatrenia, úmrtia a pohrebu, pohlavie zosnulého, a fakt či nešlo o mŕtvorodené dieťa. Ďalej tabuľka samozrejme obsahuje unikátne ID, pomocou ktorého sa identifikujú všetky osoby figurujúce v danom zázname. Počet osôb figurujúcich v zázname o úmrtí je zo všetkých typov záznamov zvyčajne najnižší.

Tabuľka 3.4: Štruktúra tabuľky marriage

Column	Type	Null
id	int	No
register_id	mediumint	No
scan	smallint	No
pos	tinyint	No
lay	char(3)	Yes
fin	tinyint(1)	Yes
check_req	tinyint(1)	Yes
lang	enum('CZ', 'GE', 'LAT', 'SC', 'PL', 'SK', 'NONE')	Yes
banns1	char(10)	Yes
banns2	char(10)	Yes
banns3	char(10)	Yes
marriage_date	char(10)	Yes
kinship_degree	varchar(50)	Yes
groom_age_year	decimal(5,2)	Yes
groom_age_month	decimal(5,2)	Yes
groom_age_day	decimal(5,2)	Yes
bride_age_year	decimal(5,2)	Yes
bride_age_month	decimal(5,2)	Yes
bride_age_day	decimal(5,2)	Yes

Tabuľka 3.4: Štruktúra tabuľky marriage (pokračovanie)

Column	**Type**	**Null**
groom_full_age	char(10)	Yes
bride_full_age	char(10)	Yes
divorce_date	char(10)	Yes
domicile	mediumint	Yes
unres_descr_num_1	char(10)	Yes
unres_descr_num_2	char(10)	Yes
groom_birth_address	int	Yes
groom_dead_address	int	Yes
bride_birth_address	int	Yes
bride_dead_address	int	Yes
signs	tinyint(1)	Yes
comment	text	Yes
owner	int	Yes
score	double	Yes
last_edited_level	int	Yes
last_edited_highest_level	int	Yes

Štruktúru tabuľky reprezentujúcej záznam o sobáši môžeme vidieť v tabuľke 3.4. Okrem rovnakých všeobecných informácií o samotnom zázname a matrike, ako aj v ostatných typoch záznamov, obsahuje informácie ako vek ženícha, vek nevesty, dátum sobášu, dátum rozvodu, a ďalej samozrejme aj unikátne id, pomocou ktorého dokážeme zistiť všetky osoby z tabuľky *person* figurujúce v danom zázname. Naopak oproti záznamom úmrtia, obsahujú záznamy o sobášoch zvyčajne najviac figurujúcich osôb zo všetkých typov záznamov. Štruktúra tabuľky *person* obsahujúcej informácie o jednotlivých osobách zo záznamu môžeme vidieť v tabuľke 3.5.

Tabuľka 3.5: Štruktúra tabuľky person

Column	**Type**	**Null**
id	int	No
birth_id	int	Yes
marriage_id	int	Yes
burial_id	int	Yes
rel	enum	Yes
title	varchar(255)	Yes
sname	mediumint	Yes
domicile	mediumint	Yes
street	varchar(255)	Yes
descr_num	char(10)	Yes
religion	enum('catholic', 'protestant', 'jew', 'none')	Yes
birth_date	char(10)	Yes
dead	tinyint(1)	Yes
waif	tinyint(1)	Yes
category	char(1)	Yes

Tabuľka 3.5: Štruktúra tabuľky person (pokračovanie)

Column	Type	Null
person_relation	mediumint	Yes
widow	tinyint(1)	Yes
legitimate	enum('legitimate', 'illegitimate', 'U')	Yes
dead_date	char(10)	Yes
work_place	text	Yes
age	decimal(3,2)	Yes

Každá osoba uložená v tejto tabuľke predstavuje jednu osobu figurujúcu v práve jednom zázname. Vzťah medzi osobu a záznamom je daný pomocou príslušného ID záznamu. Pôjde teda o jednu z hodnôt *birth_id*, *marriage_id* a *burial_id*. Ako hodnota v týchto stĺpcoch je použité ID záznamu, do ktorého daná osoba prináleží. Je tu preto pridaný cudzí kľúč vo forme daného ID, odkazujúceho na konkrétny záznam. Každá osoba uložená v databáze môže prináležať práve do jedného záznamu. Ďalej tabuľka obsahuje stĺpec *rel*, ktorého hodnotou je jeden zo vzťahov. Tie sú reprezentované typom enum, kde sú všetky jednotlivé vzťahy vymenované. Potom tu máme informácie o dátume narodenia, smrti, veku a ďalších dodatočných informácií. Ďalej uplatníme rovnaký postup porovnávania osôb zo záznamov ako doteraz, ale pri vytvorení nových prepojení medzi dvoma osobami a osobou a záznamom budeme musieť vytvoriť nové funkcie ktoré budú realizovať prepojenie osôb podľa ich úlohy v danom zázname. Pri práci s testovacou sadou pôjde o rovnaký postup.

Rozšírenie funkcií

V module *relational_database_birth.py*, ktorý načítava záznamy z matrík, ale aj v ďalších častiach programu bude potom nutné pridať kontroly zisťujúce o aký typ záznamu sa jedná. Ku každej funkcii určenej pre spracovanie záznamov o krste bude nutné vytvoriť ekvivalentnú funkciu určenú pre iný typ záznamu a prípadne na miestach, kde to bude možné, rozšíriť už existujúce funkcie tak, aby fungovali pre všetky typy záznamov. Vo výstupnej grafovej databáze bude nutné zaviesť niekoľko nových typov uzlov. Pôjde u uzly *Záznam_o_úmrtí* a *Záznam_o_sobáši.* Taktiež dôjde ku pomerne veľkému rozšíreniu typu vzťahov, kde bude nutné pridať vzťahy pre všetky nové spojenia podľa formátu nových dát. Všetky pridané vzťahy a všetky rozšírenia budú popísané ďalej v práci v časti implementácie.

3.2 Analýza efektivity a problémov algoritmu

Hlavným problémom algoritmu v pôvodnej podobe bola jeho vysoká časová náročnosť. Tá je spôsobená neustále sa zvyšujúcim počtom osôb ktoré musíme spracovať. Kvôli charakteru práce je nutné vykonávať stále viac a viac porovnaní, keďže sú osoby porovnávané spôsobom každá s každou. Pri každej porovnávanej osobe nám teda počet porovnaní vzrastie o 1. Preto môžeme hovoriť o polynomiálnej časovej komplexite a vyjadriť ju ako:

$$O(\frac{n^2 - n}{2})$$

kde n je počet osôb, ktoré tvoria vstup algoritmu. Tento predpoklad môžeme učiniť, pretože počet porovnaní priamo úmerne odpovedá dobe trvania a aj keď niektoré porovnania

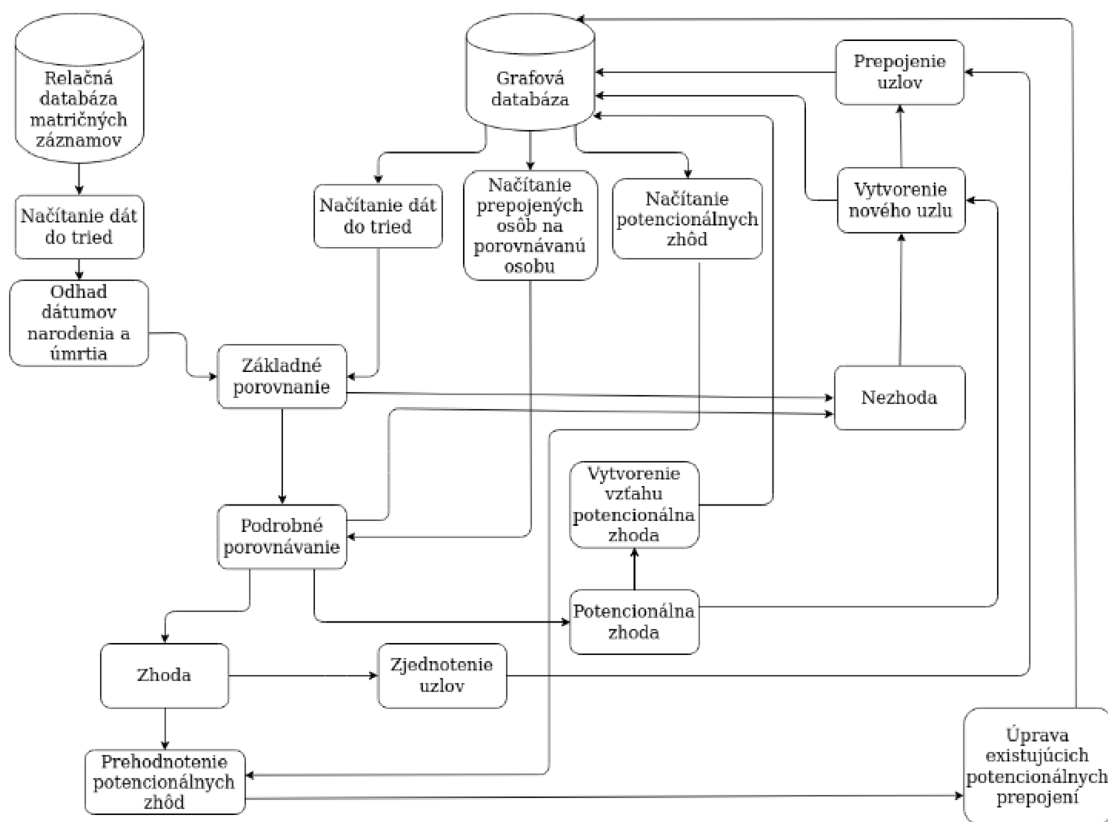
vedú ku identifikácii zhodných osôb a ich následnému zjednoteniu, čo má síce efekt na dobu trvania programu, ide stále o zriedkavú udalosť v porovnaní s bežným scenárom pridania novej osoby, prípadne osoby identifikovanej ako potencionálne zhodnej osoby. Môžeme preto túto skutočnosť zanedbať. Taktiež môžeme pre účely určenia časovej komplexity zanedbať aj fakt, že sú na mieste viaceré optimalizačné metódy, ako napríklad porovnávanie osôb iba s osobami rovnakého pohlavia, alebo neporovnávanie osôb s osobami z rovnakého záznamu. Je nutné sa preto zamerať na zníženie doby trvania spracovávania jednej osoby.

3.2.1 Zistenie doby trvania spracovávania

Pre zistenie priemernej doby spracovania jednej osoby bol použitý nástroj cProfile [6]. Ten je ideálny pre zistenie času behu jednotlivých funkcií a identifikáciu najpomalších úsekov. Keďže potrebujeme zistiť priemernú dobu trvania jedného porovnania a spracovania osoby, môžeme sa pozrieť na celkový čas strávený vo funkcii riadiacej spracovanie jedného záznamu. To síce nie je presná doba spracovania jednej osoby, ale priamo úmerne jej zodpovedá a je to spôsob ako dostaneme najrelevantnejšiu hodnotu ktorá vypovedá o dobe trvania programu bez toho, aby sme zanedbali réžiu nad spracovávaním osôb, ktorá okrem porovnávania zahŕňa vytváranie a prepájanie osôb so záznamami a medzi sebou. Budeme teda skúmať dobu trvania funkcie *compare_record_with_graph_database*. Tá dostane ako parameter záznam a spracuje všetky osoby, ktoré v ňom figurujú. Profiling bol vykonaný nad vzorkou o veľkosti tisíc záznamov krstu z databázy perun. Celkový čas, ktorý bol strávený spracovávaním záznamov, bol 17237 sekúnd, čo sú asi 4 hodiny a 47 minút. Celkovo bolo spracovaných 8271 osôb. Ak teda vydelíme túto dobu trvania počtom spracovaných osôb, dostaneme požadovanú priemernú dobu trvania spracovávania jednej osoby. Po tejto jednoduchej kalkulácii nám vychádza, že táto doba je 2,084 sekundy. Túto hodnotu si neskôr porovnáme s novou hodnotou získanou rovnakým spôsobom po všetkých zmenách vykonaných v tejto práci, aby sme videli o koľko je výsledný algoritmus efektívnejší. Jediné čo sme pri tomto výpočte z celkovej doby trvania programu zanedbali je doba pripojenia ku databázam, mazanie už existujúcich dát z Neo4j databázy a dotazovanie nad MySQL databázou. Doba trvania týchto častí bude rovnaká i po zmenách uskutočnených v tejto práci, a preto ju nebudeme brať do úvahy.

3.2.2 Identifikovanie časovo náročných miest v programe

Proces identifikovanie časovo najpomalších úsekov programu pozostával z vyhľadania funkcií na čo najnižšej úrovni, teda funkcií, ktoré ďalej obsahujú čo najmenší počet, či ideálne žiadne volania ďalších zanorených funkcií. Na to nám opäť slúžil cProfile. Pomocou neho som zistil, že drvivá väčšina času behu programu je strávená v module *graph_database.py*. Konkrétne išlo hlavne o funkciu *run_cql_with_return_person*, ktorá využíva funkcie *run* a *data* z knižnice neo4j na vykonanie dotazov a navrátenia dát získaných v týchto dotazoch. Táto funkcia zodpovedá drvivej väčšine doby behu programu (vyše 16000 sekúnd). Počas celého procesu porovnávania je nutné pracovať s osobami, ktoré sú už uložené v grafovej databáze. Preto dochádza ku neustálemu dotazovaniu nad grafovou databázou na osoby a aj na ich adresy. Keďže pri skončení porovnávania jedného záznamu a teda aj osôb v ňom sa databáza zmenila (boli pridané nové osoby a prepojenia) je nutné toto dotazovanie opakovane vykonávať zakaždým, keď potrebujeme nejaké dáta. Z toho nám vyplýva, že vysoká doba trvania programu je spôsobená najmä spôsobom, akým sa pracuje s grafovou databázou. Problém teda vznikol už v samotnom návrhu pôvodnej práce.



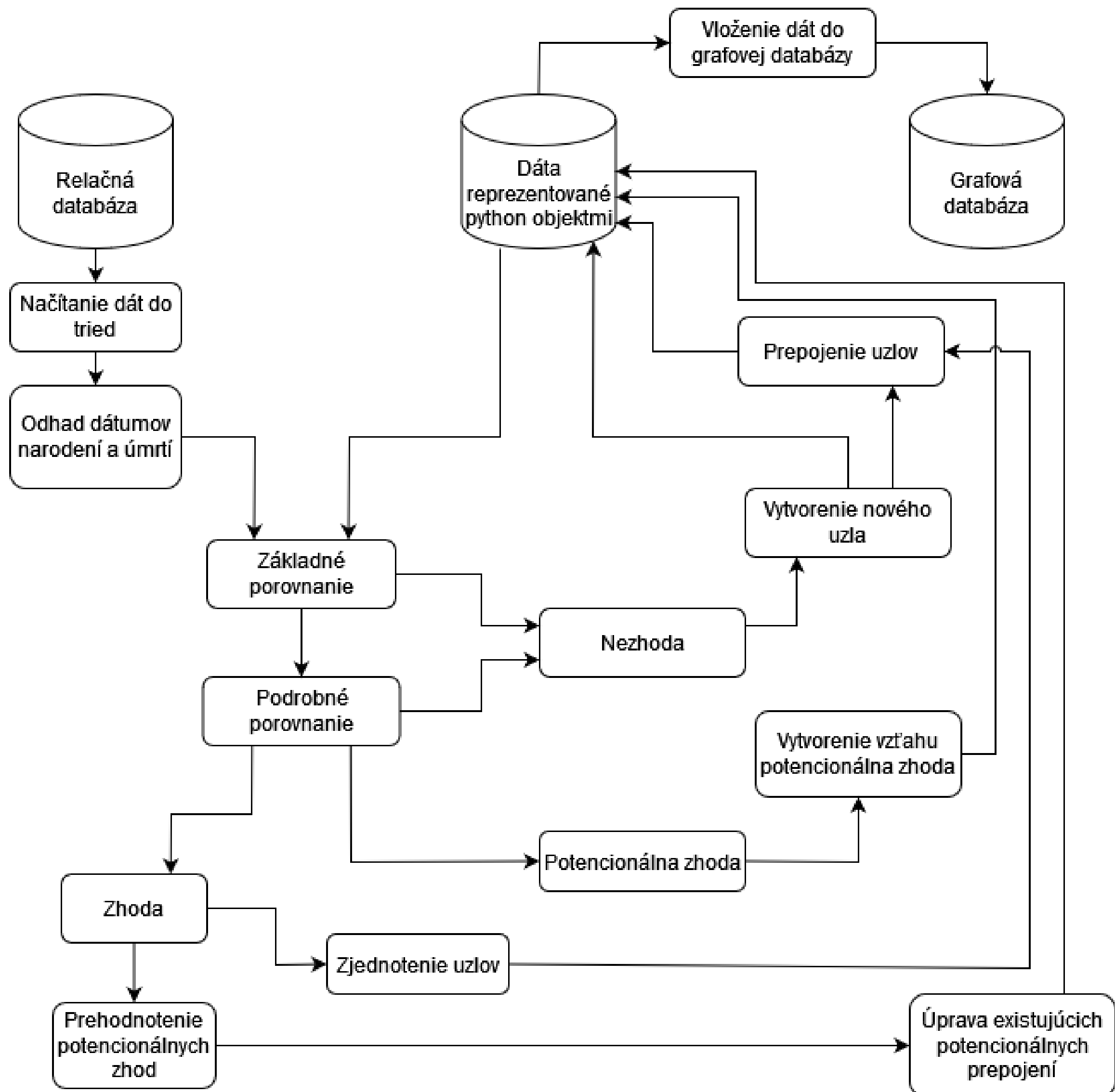
Obr. 3.1: Návrh celého systému v pôvodnej forme

3.2.3 Nový návrh výsledného systému

Keďže už vieme, kde sa skrýva najväčší nedostatok výsledného systému, môžeme sa pozrieť na spôsob, akým by sme ho mohli prerobiť tak, aby bol vo výsledku efektívnejší.

Na obrázku 3.1 prevzatom zo zdroja [12] môžeme vidieť celý návrh pôvodného systému. Keďže vieme, že problém je v práci s grafovou databázou, budú sa zmeny musieť týkať hlavne tejto časti. Pokúsime sa teda eliminovať časť neustáleho načítania dát z grafovej databázy, teda úseky *Načítanie dát do tried*, *Načítanie prepojených osôb na porovnávanú osobu* a *Načítanie potencionálnych zhôd*. Jedným z riešení, ktoré som pri vytváraní výsledného návrhu skúmal bolo udržiavať časť dát z grafovej databázy v objektoch jazyku python a minimalizovať tak počet dotazov na databázu. Tým pádom by sme si dáta natiahli iba raz, a pri ukladaní nových osôb do grafovej databázy by sme si tieto osoby pridali aj do spomínaných objektov. Problémom tohoto spôsobu čiastočnej optimalizácie bol však fakt, že pri ukladaní osôb do databázy často dôjde ku zmene vzťahov a aj informácií o osobách, či prípadnému mazaniu a zjednocovaniu osôb. Okrem toho by sa nejednalo o najlepší spôsob optimalizácie z dôvodu, že stále musíme opakovane pristupovať do grafovej databázy, a to nie len pre ukladanie osôb, ale aj pri zjednocovaní uzlov osôb, prepájaní vzťahov, mazaní uzlov osôb alebo aj aktualizácii pri nájdení zhodnej osoby. Pre dosiahnutie najlepšej optimalizácie výsledného systému by sme teda potrebovali okrem neustáleho načítania dát eliminovať aj časť neustáleho vkladanie dát do grafovej databázy. Tu sa ukázalo ako najefektívnejšie riešenie nepracovať s grafovou databázou počas procesu porovnávania vôbec.

Všetky dáta, ktoré boli pôvodne držané v grafovej databáze by sme teda potrebovali reprezentovať výhradne pomocou objektov jazyku python. Tu si musíme uvedomiť, že aby sme toto mohli vykonať, bude nutné už existujúce objekty reprezentujúce osoby, záznamy ale aj ďalšie uzly z neo4j podstatne rozšíriť takým spôsobom, aby sme mohli reprezentovať všetky spojenia, ktoré budú existovať vo výslednej databáze v reálnom čase. Bude preto nutné vytvoriť novú triedu pre spravovanie všetkých python objektov držaných výhradne v pamäti a pre vykonávanie operácii nad nimi.



Obr. 3.2: Návrh výsledného systému

Z obrázku 3.2 môžeme vidieť, že v takto zmenenom systéme sa budú dáta držať v pamäti na hromade a budú uložené v python objektoch, odpovedajúcich štruktúre a vzťahom z grafovej databázy. Tým by sme eliminovali proces neustáleho vytvárania a vykonávania dotazov v jazyku cypher nad v istom bode už pomerne objemnou databázou. Na konci porovnávania budú dáta do grafovej databázy vložené naraz až vo výslednej podobe. V takomto prístupe však existuje aj nevýhoda oproti pôvodnej implementácii a to je fakt,

že na to, aby sme dostali finálny výstup v podobe grafovej databázy musíme počkať do konca procesu porovnávania a vkladania. Toto však nie je veľmi podstatné, keďže je skript možné jednoducho modifikovať takým spôsobom, aby spracoval iba fixný počet záznamov, napríklad pomocou jednoduchého počítadla v hlavnom cykle (napríklad pri testovacom spúštaní skriptu).

Kapitola 4

Implementácia

V tejto kapitole sa budeme venovať analýze a oprave chýb a prípadných nedostatkov práce, ďalej si povieme niečo ku implementácii nového prerobeného algoritmu, hlavne pri práci s grafovou databázou, kde dochádza ku výraznému spomaleniu algoritmu. Potom sa pozrieme na presný spôsob implementácie rozšírení týkajúcich sa nových matrík (teda nových typov záznamov o sobáši a úmrtí) vrátane zmien potrebných pri prechode na novú relačnú databázu demos, ktorej štruktúra sa mierne zmenila. Na záver si ukážeme novú formu výstupu vo vyhľadávacích skriptoch a ako boli implementované nové metriky použité pri porovnávaní.

4.1 Identifikovanie nedostatkov

Hľadanie nedostatkov a chýb a ich následné riešenie sa ukázalo byť jedným z najkomplikovanejších a časovo najnáročnejších aspektov tejto práce. Bolo tomu tak najmä kvôli veľkému rozsahu zdrojového kódu skriptov, ale aj kvôli vysokému počtu modulov, ktoré sú navzájom všetky prepojené. Ku odhaleniu chýb došlo najmä pri neustálom spúšťaní skriptu nad rozlyčnými dátami, či už z databázy alebo zo súborov csv, a postupným prechádzaním zdrojového kódu. Často sa ani nejednalo vyslovene o chyby, ale skôr nedostatky, ktoré sa ukázali ako zbytočné, či prípadne o funkcionality, ktoré bolo možné implementovať efektívnejším spôsobom. Väčšina chýb sa týkala práce s grafovou databázou Neo4j a dotazov nadňou, ale niekoľko problémov bolo i v reprezentácii dát a samotnom porovnávaní osôb. V tejto podkapitole si priblížime proces identifikovania chýb a konkrétne chyby a nedostatky, na ktoré som počas vypracovávania práce narazil. Pri každom probléme ukážem aj ako som ho vyriešil.

4.1.1 Počiatočné úpravy

V tejto sekcii si ukážeme ako bolo nutné skript upraviť, aby bol opäť funkčný.

Identifikovanie problému

Pred začatím akéhokoľvek implementovania navrhnutých zmien bolo nutné skript najprv upraviť, keďže v stave v akom som ho dostal ho nebolo možné spustiť. Išlo o chybu v module *graph_database.py*, pri ktorej sa vo funkcii, ktorá vytvárala dotaz v jazyku cypher, ktorý mal získať všetky adresy osoby z grafovej databázy pristupovalo ku python štruktúre typu slovník na neexistujúci kľúč. Tento slovník mal reprezentovať osobu z grafovej databázy,

avšak žiaden kľúč pre ID osoby v slovníku nebol. Dôvodom pre toto mohlo byť, že niektoré zo starších verzií grafovej databázy Neo4j alebo ovládača pre túto databázu v štruktúre typu slovník reprezentujúcej uzol získaný z databázy uvádzali aj ID uzla. Rovnaký problém mali aj niektoré ďalšie funkcie v module s grafovou databázou.

Riešenie problému

Keďže ID osoby nebolo v spomínanej funkcii s názvom *get_addresses_of_person_from_graph_db* dostupné, nebolo možné túto querry vytvoriť a spustiť nad databázou. Bolo preto nutné ID osoby nejakým spôsobom získať a propagovať do tejto funkcie. Jednoduchým riešením, ktoré mi napadlo bolo pridať ID samotného uzla ako atribút osoby, tým pádom by sme pri každej osobe získali aj ich ID. Preto som pri vytváraní osôb z grafovej databázy pridal atribút, ktorý sa pri vytvorení uzla nastaví na ID samotného vytváraného uzla.

4.1.2 Pohlavia osôb

Pri určení cieľovej skupiny, s ktorou má daná osoba zo záznamu byť porovnávaná na základe pohlavia dochádzalo pri každej osobe k tomu, že sa porovnávala so všetkými osobami namiesto toho, ako bolo pôvodne zamýšľané (teda iba s osobami rovnakého pohlavia). Zdrojové dáta boli v poriadku a pohlavia osôb boli v databáze i csv súboroch uložené tak, ako mali byť, teda správne označené ako 'M' - muž (male), 'F' alebo 'Ž' - ako žena (alebo female) a pri neidentifikovanom pohlaví ako 'U' - unidentified. Problémom bol spôsob kontroly pohlaví pri spracovávaní osôb. Z dôvodu účelu práce musí každý atribút objektov reprezentujúcich uzly z databázy Neo4j byť uložený ako list. Je tomu tak preto, že niektoré vlastnosti môžu mať v databáze viacero hodnôt a aj kvôli tomu, že pri aktualizácii informácií osoby pri zjednocovaní uzlov osôb si je nutné ponechať všetky informácie, jednak o pôvodnej osobe a aj o osobe, ktorú s ňou zjednocujeme. Pri kontrole pohlavia sa tento fakt však nezohľadňoval a pohlavie sa nesprávne porovnávalo ako objekt typu list, namiesto konkrétnej hodnoty v tomto liste. Tým pádom sa skript vždy vydal do tej istej vetvy a to síce tej, kde pohlavie osoby bolo označené za neidentifikované.

Riešenie tohoto problému bolo triviálne, preto si ho zhrnieme iba veľmi stručne. Na mieste kontroly pohlaví je pridaná jednoduchá kontrola toho, koľko hodnôt má táto osoba v tomto atribúte. V prípade, že je hodnota jedna, tak pohlavie určíme jednoducho pomocou pristúpenia na túto hodnotu. V prípade že sú v liste dve hodnoty, skontrolujeme, či sú obidve hodnoty iba znaky 'Z' alebo 'F', označujúce ženské pohlavie. V opačnom prípade môžeme prehlásiť, že výsledné pohlavie osoby je nedefinované, kvôli rôznym hodnotám tohoto atribútu. To isté môžeme taktiež prehlásiť, ak sú pohlavia v liste viac ako dve.

4.1.3 Porovnávanie osôb spolu s predkami

Pri analýze kódu som narazil na chybu, ktorá spôsobovala, že predkovia osoby zo záznamu boli porovnávaní s nesprávnym predkom osoby z grafovej databázy. Rola osoby je identifikovaná pomocou reťazca, ktorý je rovnaký ako enumeračný typ určený pre roly osôb z databázy. Šlo o prípad, kedy boli porovnávaní rodičia osoby zo záznamu. Tých reprezentujú roly, ktoré majú v programe názov f a m. Chyba nastala ak sme chceli porovnávať rodičov rodičov, teda starých rodičov hlavnej osoby zo záznamu. Pri porovnaní predkov osoby je samozrejme žiadúce porovnávať osoby v rovnakom vzťahu ku pôvodnej osobe. Napríklad pri porovnávaní osoby, ktorá má v zázname rolu otcova matka (teda vzťah reprezentovaný ako

reťazec *f_m*) je žiadúce túto osobu porovnávať s osobou z grafovej databázy, ktorá má ku pôvodnej osobe rovnaký vzťah. Avšak vo funkcii realizujúcej toto porovnanie s názvom *comparison_of_two_person_with_ancestors* tomu tak nebolo. Namiesto toho, aby sme osobu vo vzťahu otcova matka porovnávali aj s otcovou matkou zo záznamu, bola osoba z grafovej databázy porovnávaná s osobou s rolou *m_f*, teda matkin otec. Rovnako tomu bolo aj v opačnom prípade, teda v prípade porovnania matkinho otca, ktorý bol nesprávne porovnávaný s otcovou matkou. Toto mohlo mať mierny negatívny vplyv na výsledky porovnaní. Oprava chyby spočívala iba v zmenení hodnôt reprezentujúcich rolu osoby na správnu hodnotu.

4.1.4 Práca s dátumami

V tejto podsekcii si povieme, ako sa v pôvodnej implementácii pracovalo s dátumami a ukážeme si, ako sa s nimi bude pracovať po vykonaných zmenách.

Modul date.py

Pre prácu s dátumami sa v implementácii používa modul *date.py*. V tomto module sa nachádza trieda *Date*. Jej atribútmi je celý objekt dátumu v podobe objektu typu datetime, rok, mesiac a deň dátumu. Trieda ďalej ponúka viacero funkcií realizujúcich vyžadované operácie nad objektami vytvorenými z tejto triedy, ako napríklad úpravu dátumu pri dni, ktorý je mimo rozsah daného mesiaca, funkciu pre vytvorenie objektu *Date* zo vstupného reťazca či metódu pre výpis dátumu. Pri analýze kódu som došiel ku záveru, že je tento modul zbytočný. Jediná funkcionalita, ktorú ponúka naviac v porovnaní s klasickým python modulom pre reprezentáciu a prácu s dátumami *datetime*, je implementovaná funkcia *change_date_to_nearest_possible*. Tá pri dni v mesiaci, ktorý je mimo rozsah daného mesiaca posunie dátum na najbližší ďalší existujúci deň. Avšak implementáciou podobnej funkcie v module *relational_database_birth.py*, ktorú je nutné aj tak implementovať, kvôli chýbajúcim údajom v dátumoch v novej použitej databáze demos, ktoré bude nutné aj tak nahradiť existujúcou hodnotou. Ďalším dôvodom prečo bolo výhodné tento modul odstrániť bol fakt, že pri prerábaní systému došlo ku viacerým rozsiahlym zmenám a s modulom sa v porovnaní s modulom *datetime* pracuje zbytočne o dosť viac komplikovane, keďže je nutné pre výpis, konverziu či načítanie a uloženie dát volať špeciálne funkcie tohto modulu, nehovoriac o fakte, že modul je v podstate iba komplikovanejší spôsob uchovávania objektu triedy *datetime.datetime* s dodatočnými krokmi.

Odstránenie modulu

Modul som sa rozhodol z implementácie odstrániť úplne a namiesto objektov, ktoré implementuje trieda *Date* som sa rozhodol pre reprezentovanie a prácu s dátumami použiť existujúci modul *datetime*, ktorý ponúka triedu *datetime.datetime*. Ten je pre reprezentáciu dátumov ideálny a ponúka všetky potrebné funkcionality implementované pôvodným modulom, ako funkcie *strptime* a *strftime*, ktoré umožnujú jednoducho konvertovať reťazce na *datetime.datetime* objekty a naopak.

Nutné zmeny po odstránení

Po odstránení modulu bolo nutné vykonať viacero zmien. Išlo o zmeny pri určovaní dátumov podľa rôl osôb, načítaní dát a v podstate na každom mieste v kóde, kde dochádzalo

ku porovnaniam medzi dátumami a prevodom z reťazca na typ reprezentujúci dátum a naopak. V module *relational_database_birth.py* bolo nutné potom implementovať požadovanú funkcionalitu zmenenia neexistujúcich dátumov na existujúce a to pomocou funkcie *replace_double_questionmarks*, ktorá vykonáva doplnenie chýbajúcich dát takým spôsobom, že v prípade chýbajúceho dňa a mesiaca je deň alebo mesiac nastavený na prvý deň v mesiaci alebo teda prvý mesiac v roku. Funkcia zohľadňuje aj počet dní, ak vieme o aký mesiac sa jedná a naopak, ak by bol deň mimo rozsah mesiaca, vyberie iba odpovedajúce mesiace, ktoré môžu daný počet dní mať. Okrem funkcie sú na mieste spracovania dátumov aj ďalšie kontroly, či napríklad nedošlo ku zameneniu údaju o dni s údajom o mesiaci. Toto môžeme prehlásiť, ak je údaj dňa menší ako 12 a údaj o mesiaci väčší ako 12. V takomto prípade sa predpokladá zamenenie hodnôt a hodnoty údajov sú vymenené.

4.1.5 Mazanie databázy

V tejto sekcii si priblížime problém, ktorý vznikol pri mazaní databázy, ktorá bola naplnená vysokým objemom dát.

Identifikovanie problému s mazaním

Počas testovania nad databázou som narazil na problém, ku ktorému dochádzalo pri snahe zmazať celú databázu pri novom spustení skriptu. Keďže Neo4j community edition neponúka možnosť databázu zmazať pomocou drop dotazu (funkcionalita je dostupná iba pre Neo4j enterprise edition), bola databáza mazaná vytvorením dotazu odpovedajúcemu všetkým prvkom databázy a následnému zmazaniu všetkých vzťahov a uzlov.

```
MATCH (n) DETACH DELETE n
```

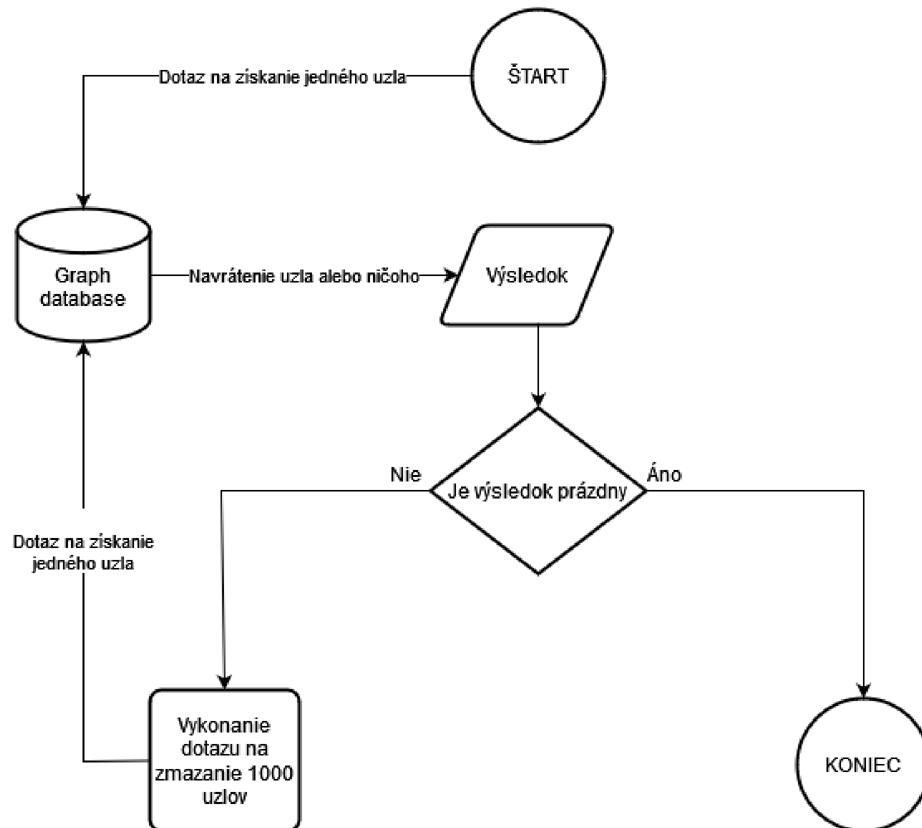
Obr. 4.1: Dotaz používaný na mazanie databázy

Na obrázku 4.1 môžeme vidieť dotaz realizujúci mazanie v pôvodnej implementácii. Problém nastáva vtedy, keď je v databáze veľký objem dát a z tohto dôvodu pri snahe o vytvorenie takéhoto dotazu dôjde ku chybe typu java heap error. To znamená, že pri snahe pomocou dotazu označiť príliš veľké množstvo dát dôjde jazyku java, v ktorom je databáza Neo4j implementovaná pamäť na hromade.

Možné riešenia

Na vyriešenie tohoto problému sa ponúkalo viacero spôsobov. Prvým by bolo zo skriptu za použitia python modulu s názvom *os* manuálne pristúpiť a zmazať zdrojové súbory databázy. Avšak problémom tohto riešenia je, že skript by potom musel byť vždy spúšťaný s administrátorskými právami, čo nie je veľmi praktické. Taktiež by vznikol problém pri zisťovaní cesty k týmto súborom, keďže databáza Neo4j môže byť nainštalovaná na rôznych miestach a súbory s databázou môžu mať podľa platformy, na ktorej je databáza nainštalovaná rôzne umiestnenie. Toto riešenie preto nie je veľmi vhodné. Ďalšou možnosťou by bolo manuálne zvýšenie množstva pamäťe na hromade, ktoré java priraďuje aplikácii Neo4j. Avšak tu sa nejedná o úplné riešenie problému, ale skôr iba o odsunutie. Potom by tu bola možnosť prechodu z Neo4j Community Edition na Neo4j Enterprise Edition. Táto edícia databázy je však platená a kvôli jedinej požadovanej funkcionalite by to bolo zbytočné. Najlepším riešením sa ukázalo byť mazanie databázy v menších dávkach (tzv. batch deletion).

To ale nie je možné vykonať pomocou jediného dotazu. Bolo preto nutné pridať jednoduchý algoritmus realizujúci opakované cypher dotazy dovtedy kým databáza nie je prázdna.



Obr. 4.2: Schéma algoritmu mazania

Na obrázku 4.2 môžeme vidieť algoritmus mazania databázy. Na začiatku sa vykoná jednoduchý dotaz na získanie jediného uzla. Vo výsledku dotazu následne skontrolujem, či nie je prázdny. V prípade, že je výsledok prázdny môžeme prehlásiť databázu za prázdnu a teda pokračovať vo vykonávaní samotného skriptu. V prípade, že výsledok dotazu nie je prázdny, vykonáme dotaz na zmazanie tisícich uzlov z databázy. Ďalej pokračujeme rovnakým dotazom na získanie jediného uzla z databázy a proces sa takto opakuje, až kým databáza nie je prázdna.

4.1.6 Ďalšie optimalizácie

V skripte bolo vykonaných množstvo ďalších menších optimalizácií, ktoré samé o sebe veľmi nestoja za zmienku. Išlo najmä o odstránenie zbytočných funkcií realizujúcich triviálne operácie. Jedná sa o funkcie, ktoré mohli byť napríklad nahradené jednou podmienkou, alebo realizujúcich minimálne množstvo kódu, teda iba pridávali extra réžiu spôsobenú svojím volaním. Ďalšou optimalizáciou, ktorá možno stojí za zmienku je nahradenie neustáleho používania operátora + jazyka python pre konkatenáciu reťazcov. Ten môže byť pri konkatenácii väčšieho množstva reťazcov výrazne menej efektívny ako metóda *join* z modulu *string*. Tá môže byť podľa informácií zo zdroja [2] až 4-krát efektívnejšia pri konkatenácii viacerých reťazcov ako operátor +. Takéto optimalizácie nemusia mať pri spracovávaní menšieho množstva dát veľmi výrazný efekt, ale pri spracovávaní veľkého objemu dát sa už

môže jednať o signifikantné množstvo ušetreného času, hlavne ak si uvedomíme, že počas behu skriptu dochádza ku konkatenácii reťazcov pomerne často.

4.1.7 Skript get_all_records.py

Tento skript bol v pôvodnom stave implementácie nefunkčný. Pri získavaní dát o úlohe osoby v špecifickom zázname a pri získavaní dátumu záznamu sa ku navrátenej štruktúre s týmito údajmi pristupovalo nesprávne. Tento problém bol jednoducho vyriešiteľný úpravou spôsobu, akým sa k týmto dátam pristupuje.

4.2 Implementácia zmeneného systému

V tejto podkapitole si povieme ako prebiehala a čo všetko zahŕňala implementácia nového navrhnutého systému. Postupne si prejdeme jednotlivé zmeny, ktoré táto optimalizačná zmena priniesla.

4.2.1 Rozbor problematiky

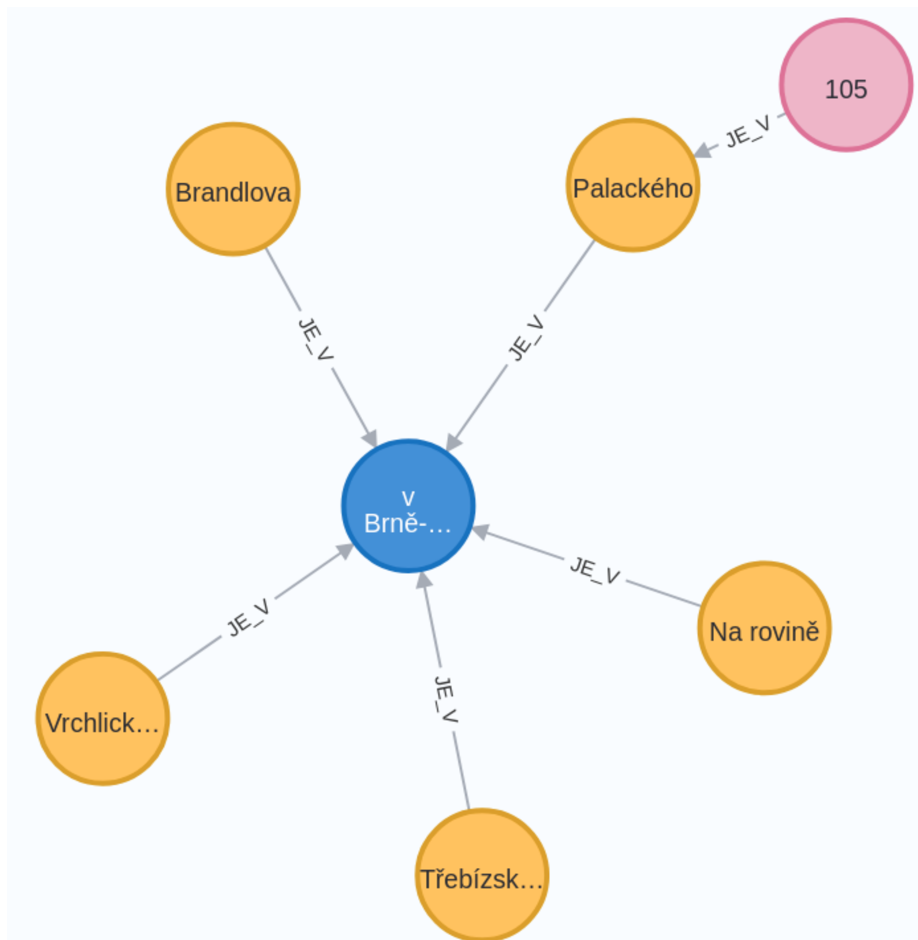
Hlavným riadiacim bodom celého algoritmu je modul *create_database.py*. Ten riadi celý proces porovnávania a spracovávania dát. Celý proces pozostáva z načítania dát z relačnej databázy pomocou metód z triedy *RelationalDatabaseHandle*, ktorá je implementovaná v module *relational_database_birth.py*, následnému načítania všetkých osôb z grafovej databázy pomocou metód z triedy *GraphDatabaseHandle*, ktorá je implementovaná v module *graph_database.py* a ich následnému porovnávaniu pomocou funkcií z modulu *comparator.py*. Počas celého procesu sa celý čas pracuje priamo nad grafovou databázou. To znamená, že ak napríklad klasifikujeme osobu ako nezhodu a chceme ju vložiť do databázy, musíme hneď zavolať metódu triedy *GraphDatabaseHandle*, ktorá vytvorí a vykoná dotaz priamo nad databázou v reálnom čase. Týmto sa pri každej spracovávanej osobe zakaždým mení štruktúra grafovej databázy, preto je nutné ju po každom spracovanom zázname celú znova načítať aj s jej novou štruktúrou pomocou dotazov v jazyku Cypher. Takýto algoritmus je pomerne časovo náročný a neefektívny.

Keďže už vieme, ako by mal výsledný optimalizovaný systém vyzerať, ukážeme si ako sa bude celý algoritmus meniť. Vieme už, že dáta nechceme zakaždým ukladať priamo do databázy a opakovane ju celú za každým záznamom znova načítať. Preto musí existovať nejaká medzivrstva, ktorá dokáže reprezentovať dáta vkladané do grafovej databázy Neo4j, teda spôsob ako reprezentovať štruktúru dát uložených v tejto databáze. V implementácii tejto medzivrstvy bude nutné reprezentovať teda nie len samotné uzly, ale aj všetky vzťahy ktoré sú medzi nimi.

4.2.2 Reprezentácia dát

Keďže potrebujeme reprezentovať veľké množstvo rôznych uzlov, ktoré majú medzi sebou rôzne typy vzťahov, bude nutné vybrať čo najuniverzálnejší spôsob reprezentácie dát, ktorý bude fungovať pre všetky typy uzlov a vzťahov. V implementácii už existujú triedy, ktoré reprezentujú jednotlivé uzly z databázy Neo4j. Avšak zatiaľ medzi nimi nie je žiaden spôsob reprezentácie vzťahov. Jedná sa o triedy *Person*, *Record* a *Register*. Pre reprezentovanie uzlov adries, teda miest, ulíc a popisných čísel je použitá trieda *Domicile*, ktorá slúži skôr na implementáciu ukladania adries osôb, nie samotných miest, ulíc, popisných čísel a vzťahov

medzi nimi. Jedná sa teda skôr o spôsob ako uchovať informáciu o adrese osoby a nie celkovú štruktúru mesta, na ktoré sú prepojené ulice, na ktoré sú následne prepojené všetky popisné čísla v ulici. Toto je v pôvodnej implementácii implementované neustálym kontrolovaním existencie duplicitných uzlov opäť priamo nad databázou.



Obr. 4.3: Ukážka štruktúry uzlov adries

Na obrázku 4.3 môžeme vidieť jednoduchý demonštračný príklad. Modrý uzol reprezentuje mesto alebo dedinu, oranžový uzol ulicu a ružový uzol popisné číslo. Uzly sú následne hierarchicky prepojené podľa príslušnosti vzťahom *JE_V*. Ak by sme v databáze mali uloženú nasledujúcu štruktúru dát mesta a adries v ňom a pri spracovávaní osoby by sme sa dostali ku adrese osoby (objekt typu *Domicile*), musel by byť opäť vykonaný dotaz nad databázou, aby sme zistili, či takáto adresa už v databáze existuje a následne v prípade, že už existuje, osobu prepojiť na túto adresu. V opačnom prípade by musel byť ešte aj vytvorený nový uzol pre novú adresu, opäť v reálnom čase nad databázou.

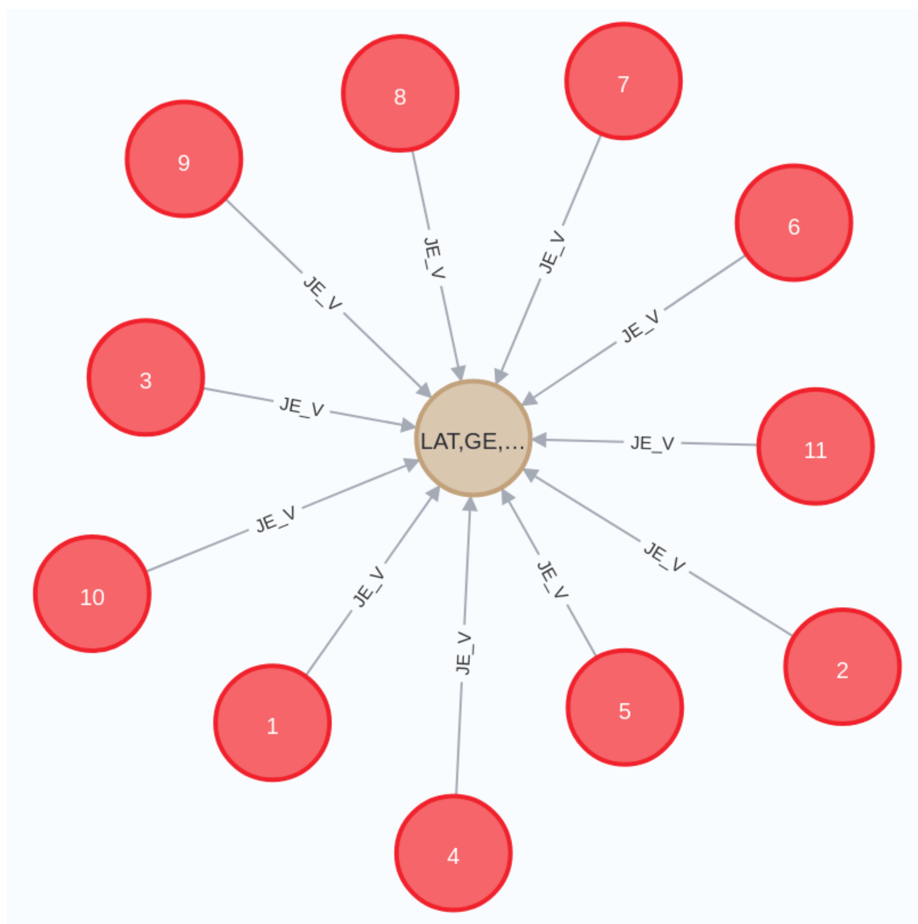
Pridanie modulu data_representation.py

Pridávanie reprezentácie vzťahov a všetkých uzlov vo výslednej databáze si rozoberieme hierarchicky, to znamená, že začneme uzlom záznamu, ktorý nesie aj informáciu o matrike z ktorej pochádza, následne si prejdeme reprezentáciu osôb a na záver si ukážeme ako budú reprezentované uzly adries. Ešte predtým si musíme vytvoriť jednoduchý spôsob prístupu

ku všetkým objektom a ich vzťahom. Preto som vytvoril nový modul implementujúci triedu *DataRepresentation*. Tá bude tvoriť už spomínanú medzivrstvu pred ukladaním dát do grafovej databázy. Bude obsahovať objekty reprezentujúce všetky uzly, ich vzťahy a všetky potrebné metódy pre operácie, ktoré bude nutné vykonávať nad touto vzniknutou štruktúrou. V module *create_database.py* si potom iba vytvoríme novú inštanciu tohoto objektu a tak budeme môcť jednoducho pristupovať ku všetkým dátam a metódam. Dá sa povedať, že tento modul implementuje funkcionality, ktoré boli vykonávané v module *graph_database.py* priamo nad databázou, ale vykonáva ich nad navrhnutou reprezentáciou dát.

Reprezentácia dát pomocou objektov

Keďže potrebujeme reprezentovať veľké množstvo uzlov, bude najvhodnejšie uchovávať ich v objekte typu list z jazyku python. Takto môžeme vyriešiť ukladanie samotných objektov. Spôsob ukladania jednotlivých vzťahov sa bude pri každom type vzťahov trochu líšiť preto si o nich povieme, až keď sa dostaneme ku danému vzťahu. Objekty budú teda uložené ako atribúty v novej vzniknutej triede *DataRepresentation* a budú mať formu listu. Ak teda začneme hierarchycky samotnými záznamami, tie sú reprezentované objektom *Record*. Preto bude vhodné vytvoriť list všetkých objektov záznamov. Pri kažom spracovávanom zázname si teda daný záznam uložíme do tohto listu. Ďalej nasleduje objekt reprezentujúci matriku. Tým je objekt implementovaný triedou *Register*. Tu budeme opakovať rovnaký postup a opäť vytvoríme list všetkých týchto objektov. Tak ako v pôvodnej implementácii bude ale nutné kontrolovať, či sa už matrika záznamu v liste nenachádza. Špeciálnu reprezentáciu vzťahu, ktorý medzi sebou má matrika a záznam matriky teda už nemusíme pridávať, keďže záznam má ako jeden zo svojich atribútov ID matriky, z ktorej pochádza. Preto ho budeme vedieť vo výsledku jednoducho prepojiť na správnu matriku z listu matrík. Príklad štruktúry databázy, ktorú sme takto práve implementovali môžeme vidieť na obrázku 4.4. Na ňom môžeme vidieť hnedý uzol reprezentujúci matriku a červené uzly reprezentujúce záznamy z matriky, ktoré sú na matriku prepojené vzťahom *JE_V*.



Obr. 4.4: Výsledná štruktúra záznamov prepojených na matriku

Najnáročnejšou časťou bolo implementovať reprezentáciu prepojení osôb s inými osobami a osôb so záznamami. Pre reprezentovanie vzťahov osôb som sa rozhodol vytvoriť nové atribúty objektu *Person*. Tento objekt po novom obsahuje atribút vo forme listu pre reprezentovanie všetkých vzťahov vedúcich z osoby (či už na iné osoby alebo na záznamy) okrem vzťahov potencionálnych zhôd, ktoré sú uložené v osobitnom liste. Okrem toho bolo nutné vedieť aj všetky vzťahy vedúce do uzla osoby, preto som vytvoril ďalší atribút typu list, ktorý uchováva aj tieto vzťahy.

Ďalšími uzlami a vzťahmi, ktoré bolo nutné uchovať sú mestá, ulice a popisné čísla. Pri spracovávaní osoby máme v už spomínanom objekte *Domicile* informácie o adrese osoby. Bude preto nutné na základe informácií obsiahnutých v tomto objekte vytvárať novú štruktúru. Opäť bude vhodné pre uchovanie všetkých objektov miest vytvoriť v triede *DataRepresentation* atribút typu list, do ktorého budú všetky objekty miest ukladané.

4.2.3 Štruktúra pre reprezentovanie vzťahov

Teraz keď už vieme, kde si budeme vzťahy uchovávať, je nutné navrhnúť štruktúru pre reprezentovanie každého typu vzťahu. Toto budú štruktúry uložené v spomínaných listoch. Ako najlepší spôsob pre reprezentáciu vzťahu sa ukázal byť python objekt typu slovník. Ten uchováva dáta v pároch formou kľúčov, ku ktorým prislúchajú hodnoty. Takým spôsobom si môžeme uchovať všetky atribúty vzťahov a zároveň reprezentovať smer vzťahov a aj sa

odkazovať na uzol na ktorý vzťah smeruje. Smer vzťahov je implicitne vždy smerujúci z uzla okrem prípadu už spomínaného listu s všetkými vzťahmi smerujúcimi do uzla. Pre určenie uzla, s ktorým je pôvodný uzol (reprezentovaný objektom) vo vzťahu, používame referenciu na objekt reprezentujúci cieľový uzol. To v praxi v podstate znamená jednoduché priradenie objektu, keďže v jazyku python je priraďovanie každej hodnoty iba vytvorenie premennej odkazujúcej na rovnaký objekt.

Vzťahy medzi osobami

Na obrázku 4.5 môžeme vidieť príklad vzniknutého vzťahu osoby. V tomto prípade sa jedná o vzťah medzi dvoma osobami. Slovník tvoriaci takýto vzťah sa skladá z dvoch kľúčov. Tými sú *type*, ktorého hodnotou je názov vzťahu, ktorý budeme vkladať do databázy, v tomto konkrétnom príklade je to vzťah *JE_OTEC*. Ďalším kľúčom je *person*, ktorého hodnotou je už samotný odkaz na objekt reprezentujúci cieľový uzol, v tomto prípade by išlo o dieťa osoby. Zakaždým, keď vytvárame takýto vzťah, je vytvorený aj vzťah v liste vzťahov smerujúcich do uzla osoby v atribúte objektu reprezentujúceho cieľový uzol. Takýto vzťah bude mať rovnakú hodnotu kľúča *type* a hodnotou kľúča *person* bude osoba, z ktorej vzťah smeruje. Pre uloženie vzťahov potencionálnych zhôd je potom použitý osobitný atribút. Pri reprezentácii vzťahov je v prípade potencionálnej zhody prítomný aj kľúč s porovnávacím skóre osôb.

```
{
  "type": "JE_OTEC",
  "person": person_variable
}
```

Obr. 4.5: Slovník reprezentujúci vzťah osoby s inou osobou

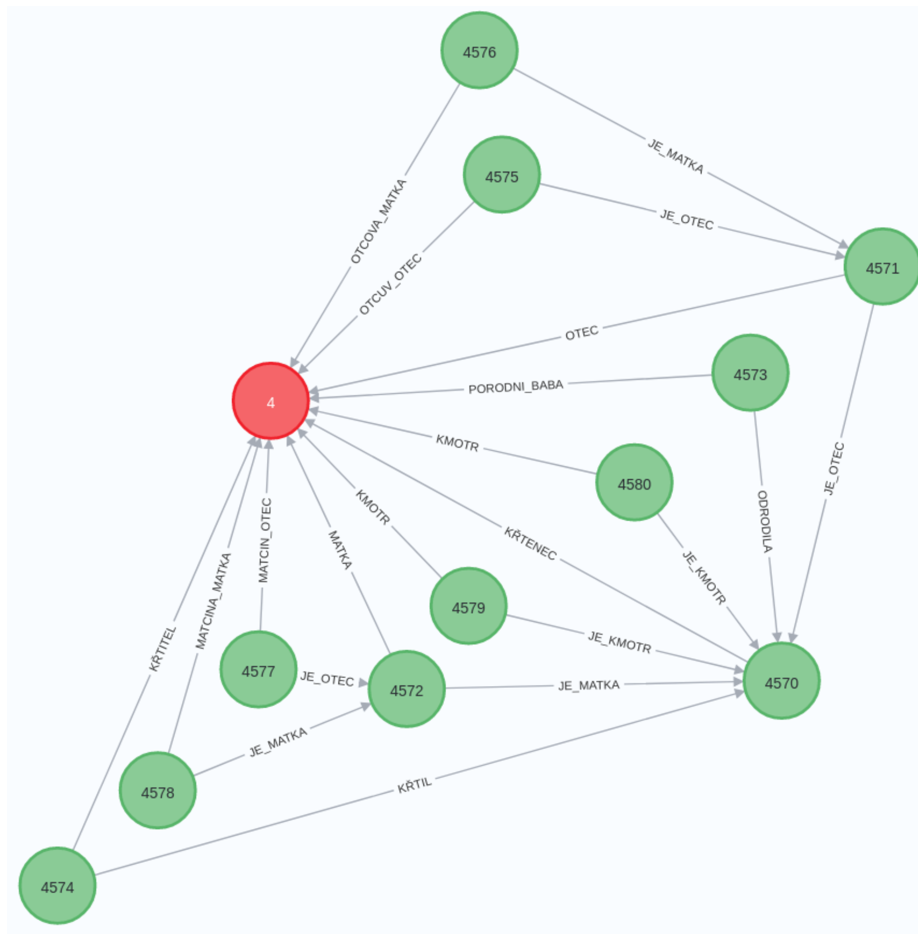
Vzťah medzi osobou a záznamom

Ďalším typom vzťahov, ktoré môže osoba mať sú vzťahy ku záznamom. V týchto vzťahoch potrebujeme ukladať opäť typ vzťahu, objekt na ktorý sa vzťahom odkazujeme a k tomu ešte aj dátum vytvorenia záznamu. Na obrázku 4.6 môžeme vidieť štruktúru slovníka reprezentujúceho takýto vzťah. Opäť je tu kľúč *type*, ktorého hodnotou je názov vzťahu. Ďalej tu máme kľúč *record*, ktorého hodnota je odkaz na samotný objekt reprezentujúci záznam. Potom je tu ešte kľúč *date*, ktorého hodnota je objekt triedy *datetime.datetime* reprezentujúci dátum záznamu.

```
{
  "type": "OTEC",
  "record": record_variable,
  "date": date
}
```

Obr. 4.6: Slovník reprezentujúci vzťah osoby so záznamom

Na obrázku 4.7 môžeme vidieť, akú štruktúru dát sme vytvorenými vzťahmi dokázali reprezentovať. Ide o príklad záznamu o krste a všetkých osôb, ktoré v ňom figurujú vrátane všetkých vzťahov medzi osobami a osobami a záznamom. Červený uzol tu reprezentuje záznam o krste a zelené uzly reprezentujú osoby figurujúce v zázname.



Obr. 4.7: Príklad osôb a ich vzťahov v databáze

Vzťah medzi osobou a uzlom adresy

Tento typ vzťahu nebolo nutné reprezentovať novou štruktúrou, keďže každý objekt *Person* už má atribút typu list objektov typu *Domicile*, ktorý nám hovorí na akých adresách osoba počas života bývala. Jediné čo bolo nutné v tomto atribúte modifikovať bolo pridanie špecifického ID k atribútu ulice a popisného čísla. Toto bolo vykonané z implementačných dôvodov pri vytváraní štruktúr reprezentujúcich hierarchiu vzťahov medzi uzlami adries (teda mestami, ulicami a popisnými číslami). Keďže sme prestali robiť neustále dotazy na adresy v databáze, stratili sme informácie o existujúcich mestách a uliciach a popisných číslach v nich. To ako sme si tieto vzťahy reprezentovali si popíšeme v ďalšej sekcii. Vzťahy osôb k najdetailnejšiemu uzlu adresy (teda vzťahy *BÝVA*) si vieme prepojiť jednoducho podľa atribútu s adresami osoby.

4.2.4 Hierarchia vzťahov uzlov adries

Ako už vieme, adresy uložené v databáze majú vlastnú hierarchiu vytvorenú na základe príslušnosti ulíc mestám a popisných čísel uliciam, či prípadne dedinám. Keďže sú všetky mestá uložené v liste v atribúte triedy *DataRepresentation* bolo ideálne ukladať celú hierarchiu týchto vzťahov do štruktúry reprezentujúcej mesto. Najlepšou štruktúrou pre uloženie mesta a jeho vzťahov sa opäť ukázal byť slovník, keďže potrebujeme ukladať viacero pomenovaných údajov s ich hodnotami a zároveň aj vzťahy mesta (teda ulice a popisné čísla prepojené na mesto).

```
{
  "id": uuid.uuid4(),
  "name": "V Brně-Husovicích",
  "normalized_name": None,
  "gps": [0.0,0.0]
  "streets": [
               {
                 "street_name": ("Palackého", uuid.uuid4()),
                 "numbers": [
                              ("105", uuid.uuid4())
                            ]
               },
               {
                 "street_name": ("Na rovině", uuid.uuid4()),
                 "numbers": []
               },
               {
                 "street_name": ("Brandlova", uuid.uuid4()),
                 "numbers": []
               },
               {
                 "street_name": ("Třebízského", uuid.uuid4()),
                 "numbers": []
               },
               {
                 "street_name": ("Vrchlického", uuid.uuid4()),
                 "numbers": []
               }
            ],
  "numbers": []
}
```

Obr. 4.8: Príklad slovníka reprezentujúceho mesto a jeho ulice a popisné čísla

Na obrázku 4.8 môžeme vidieť vzniknutú štruktúru mesta, ktorá reprezentuje všetky uzly a vzťahy, ktoré môžeme vidieť na obrázku 4.3. Mesto je teda reprezentované slovníkom, ktorého kľúče *id*, *name*, *normalized_name* reprezentujú ID mesta, jeho názov a aj jeho normalizovaný názov v tomto poradí, následne je tu kľúč *gps*, ktorý reprezentuje gps

súradnice mesta (v prípade, že sme nenašli hodnotu súradníc získavanú zo súborov JSON, je predvolená hodnota [0.0,0.0]), kľúč *streets*, ktorého hodnotou je list slovníkov reprezentujúcich ulice mesta a posledným kľúčom slovníka mesta je kľúč *numbers*, ktorého hodnotou je zasa list všetkých objektov typu tuple, reprezentujúcich jednotlivé popisné čísla v meste (v prípade, že by sa jednalo o dedinu, alebo v prípade, že by chýbal údaj o ulici, do ktorej popisné číslo patrí).

Pri štruktúre reprezentujúcej ulicu sa jedná opäť o slovník, ktorého kľúče sú *street_name* a *numbers*. Ich hodnotou je objekt typu tuple jednoznačne identifikujúci ulicu a list objektov typu tuple reprezentujúcich popisné čísla v tomto poradí. Objekt typu tuple reprezentujúci ulicu sa skladá z názvu ulice vo forme reťazca a z ID ulice, ktoré jednoznačne ulicu identifikuje. Pri objekte typu tuple reprezentujúcom popisné čísla ide o presne rovnaký formát dát, čiže popisné číslo vo forme reťazca a id. Pre vytvorenie hodnôt ID bola použitá funkcia modulu *uuid*, *uuid4()*, ktorá vygeneruje náhodné ID.

Vytváranie štruktúry

Štruktúra miest v takejto podobe vzniká pri spracovaní adries spracovávanej osoby. Pre každú adresu z listu adries osoby je zavolaná funkcia *create_domicile* modulu *DataRepresentation*. Tá prejde list miest a skontroluje, či sa v ňom nová adresa už nenachádza, pričom kontrola začína zhora nadol, najprv sa kontroluje, existencia mesta, potom ulíc v ňom a nakoniec aj popisných čísel. Vždy sa pridá iba potrebný najdetailnejší údaj, takže nevytvárame žiadne duplicitné mestá ani ulice.

4.2.5 Výsledný algoritmus

Vytvorenie novej podoby algoritmu ktorú som navrhol teda pozostávalo z implementácie modulu *data_representation.py* vrátane všetkých jeho metód a atribútov v podobe už spomínaných štruktúr, nahradenia všetkých volaní funkcií modulu *graph_database.py* volaniami funkcií modulu *data_representation.py*, vykonávajúcich ekvivalentné operácia nad mnou navrhnutými datovými štruktúrami a následným vložením celej výslednej štruktúry uloženej v objektoch do grafovej databázy Neo4j. Týmto spôsobom sme eliminovali a nahradili všetky časti pôvodného algoritmu, ktoré sme si vytýčili v jeho návrhu.

4.3 Nové formy vstupných dát

V tejto podkapitole si rozoberieme proces pridávania nových vstupných dát v podobe matrík úmrtí a matrík sobášov. Pozrieme sa na jednotlivé časti algoritmu, ktoré museli byť modifikované a na nové funkcionality, ktoré museli byť implementované.

4.3.1 Úpravy pri zmene relačnej databázy

Ešte pred začiatkom pridávania podpory pre nové vstupné dáta bolo nutné zmeniť spôsob, akým sa z databázy načítavajú dátumy. V pôvodnej databáze perun nebolo nutné prevádzať dátumy z reťazcov na *datetime.datetime* objekty. Túto konverziu bolo nutné pridať pri každom načítanom dátume do modulu zodpovedného za načítanie dát, teda *relational_database_birth.py*. Taktiež formát dátumov v databáze bol často nekonzistentný, s chýbajúcimi údajmi o mesiacoch či dňoch a niekedy aj rokoch. Často sa v hodnotách vyskytovali neexistujúce dni v mesiaci ako napríklad 31.9 alebo 30.2. To vždy spôsobovalo spadnutie programu pri konverzii takéhoto reťazca na objekt typu *datetime.datetime*, ktorý

takéto chybné údaje nepodporuje. Na miesto som preto implementoval kontroly, ktoré prípadné chýbajúce údaje doplnia reálnymi datami. Chýbajúci deň v mesiaci je preto nastavený na prvý deň v mesiaci a chýbajúci mesiac zasa na prvý mesiac v roku. Taktiež je na mieste kontrola, či údaje o mesiaci a dni nie sú zamenené. V takom prípade sú vymenené. V budúcnosti by som odporučil vykonať opravu dát v databáze, prmárne dátumov, tak aby tam boli iba reálne a úplné hodnoty, keďže nesprávne hodnoty môžu viesť ku nekonzistenciám.

4.3.2 Načítanie dát

Do modulu načítavajúceho dáta z relačnej databázy *relational_database_birth.py* bolo nutné pridať funkcie vykonávajúce načítanie dát z tabuliek záznamov o úmrtí a sobášov, načítanie všetkých osôb figurujúcich v danom zázname a vyplnením novovzniknutých objektov typu *record* a *Person* všetkými údjami získanými z databázy. Taktiež bolo nutné pridať nové funkcie pre získanie dátumu záznamu, keďže dátumy záznamov rozličného typu mali rozličné názvy stĺpcov v ktorých bol dátum uložený a v prípade chýbajúceho záznamu bolo nutné dátum získať implicitne z príslušného odpovedajúceho dátumu z tabuľky osoby, ktorá zastupovala v zázname hlavnú rolu. Objekty *Person* museli byť taktiež rozšírené o nové atribúty, ktorými boli údaje v nových záznamoch. Boli nimi napríklad dátum viatika, pohrebu, sobášu, rozvodu, fakt či dieťa nebolo mrtvorodené a príčina smrti osoby.

4.3.3 Odhady dátumov

Ďalšie zmeny bolo nutné implementovať vo funkcii vykonávajúcej odhad intervalu dátumu narodenia a úmrtia osoby. Tam boli pridané nové odhady vďaka novým typom dátumov a novým rolám osôb a taktiež kontextu udaného typom záznamu. Pri určovaní odhadov bolo nutné ku existujúcim pravidlám pridať nasledujúce pravidlá.

- Dátum viatika musel byť najskôr mesiac pred úmrtím osoby
- Dátum pohrebu musel byť najneskôr mesiac po úmrtí osoby
- Vek matky ženícha alebo nevesty musel byť minimálne 30 rokov a maximálne 60 rokov
- Vek otca ženícha alebo nevesty musel byť minimálne 30 rokov a maximálne 70 rokov
- Vek starých rodičov ženícha alebo nevesty musel byť minimálne 45 rokov a maximálne 85 rokov

4.3.4 Modifikácia riadiaceho algoritmu

Hlavný algoritmus riadiaci porovnávanie a načítanie záznamov bol modifiovaný tak, aby po skončení spracovávania záznamov jedného typu pokračoval záznamami ďalšieho typu. Zároveň po skončení spracovávania jedného záznamu bola pridaná kontrola na typ záznamu, podľa ktorej sa určí aká funkcia na prepojenie osôb zo záznamu bude zavolaná. Bolo teda nutné implementovať aj tieto nové funkcie, ktoré prejdú všetkými osobami záznamu a vytvoria všetky spojenia medzi osobami a záznamami, pomocou metód implementovaných v module *DataRepresentation*. Nové typy záznamov pridali pomerne veľké množstvo nových rôl osôb. Nové roly pridané záznamami úmrtí sú nasledovné.

- ZESNULÝ
- SYN_ZESNULÉHO
- DCERA_ZESNULÉHO
- MANŽEL_ZESNULÉHO
- MANŽELKA_ZESNULÉHO
- POCHOVAL
- HROBNÍK
- ZAOPATŘIL
- ZAOPATŘOVATEL

A nakoniec roly pridané záznamami o sobášoch sú nasledovné:

- ŽENÍCH
- JE_MANŽEL
- NEVĚSTA
- JE_MANŽELKA
- ŽENICHUV_OTEC
- ŽENICHOVA_MATKA
- NEVĚSTIN_OTEC
- NEVĚSTINA_MATKA
- MATKA_ŽENICHOVA_OTCE
- OTEC_ŽENICHOVA_OTCE
- MATKA_ŽENICHOVY_MATKY
- OTEC_ŽENICHOVY_MATKY
- MATKA_NEVĚSTINA_OTCE
- OTEC_NEVĚSTINA_OTCE
- MATKA_NEVĚSTINY_MATKY
- OTEC_NEVĚSTINY_MATKY
- BYL_SVĚDEK
- ŽENICHUV_SVĚDEK
- BYLA_DRUŽIČKOU
- DRUŽIČKA
- SVĚDEK
- NEVĚSTIN_ZESNULÝ_MANŽEL
- ŽENICHOVA_ZESNULÁ_ŽENA
- VDOVA_PO
- VDOVEC_PO
- ODDÁVAJÍCÍ

Medzi týmito rolami sú všetky nové formy spojení osôb medzi sebou a aj osôb so záznamom. Je nutné podotknúť, že nie všetky osoby figurujúce v zázname o sobáši musia byť nutne prepojené na hlavnú osobu figurujúcu v zázname, teda na nevestu alebo ženícha. Je tomu tak preto, aby sme sa vyhli príliš vysokému množstvu vzťahov v grafovej databáze a aby sme nestratili prehľadnosť. Preto roly svedkov a oddávajúceho kňaza nie sú prepojené na hlavné osoby. Ich príslušnosť k nim však vyplýva implicitne zo záznamu.

4.3.5 Porovnávanie

Zmeny v module realizujúcom porovnanie boli iba minimálne, išlo iba o pridanie nových atribútov osoby do zostavovaného porovnávacieho vektoru. Spôsob, akým sa tieto atribúty porovnávajú je rovnaký ako iné atribúty rovnakého typu.

4.4 Implementácia nových metrík

Toto bude iba veľmi stručná podkapitola, v ktorej si povieme akým spôsobom boli pridané nové metriky pre účel otestovania ich dopadu na efektivitu výsledného algoritmu. Funkcia

implementujúca Jarovu i Jaro-Winklerovu vzdialenosť je dostupná z modulu Levenshtein, ktorý práca už používa. Otestované budú ale aj iné implementácie, a to implementácia týchto porovnávacích metrík z modulu pyjarowinkler a jellyfish.

Zmeny v implementácii sa budú týkať modulu *comparator.py*, kde bude stačiť jednoducho nahradiť použitie funkcie implementujúcej Levenshteinovu vzdialenosť inou funkciou implementujúcou alternatívnu metriku. Následne sa metrika otestuje nad testovacou sadou.

4.5 Nová možnosť výstupu vyhľadávacích skriptov

V čase riešenia práce vznikla požiadavka aj na rozšírenie možnosti výstupu vyhľadávacích skriptov *get_all_records.py*, *get_person.py* a *get_family_tree.py*. Tie doposiaľ produkovali výstup iba v podobe textu na štandardnom výstupe. Pre zvýšenie použiteľnosti dát získaných z týchto skriptov bolo žiadúce produkovať univerzálnejší výstup. Tým bol výstup vo formáe JSON. Pre vytvorenie štruktúr bolo nutné mierne upraviť základný algoritmus týchto skriptov. Bol pridaný prepínač *-j*, ktorý namiesto pôvodného výstupu v podobe textu na štandardnom výstupe produkuje text vo formáte JSON na štandardný výstup. Pri skripte *get_all_records.py* a *get_person.py* produkujú jednoduchý JSON, kde sú informácie o osobe či záznamoch zobrazené v jednotlivých políčkach za sebou. Pri skripte *get_family_tree.py* možno stojí za zmienku fakt, že výstup bol upravený takým spôsobom, že ku položkám v štruktúre osoby je pridaná položka s názvom *matka* a *otec*, ktorej hodnota je celá štruktúra osôb, ktoré majú s touto osobou rodičovský vzťah. Týmto vzniká JSON forma vývodu pre danú osobu. Pre zobrazenie potomkov platí to isté s tým, že deti osoby sú zobrazené ako hodnota položky *deti*. V tej sú zaradom vypísané štruktúry všetkých osôb, ktoré sú deťmi tejto osoby.

Kapitola 5

Testovanie

V tejto kapitole si ukážeme jednotlivé testy, ktoré výsledný skript podstúpil a ich výsledky. V testoch zameraných na výslednú efektivitu budeme porovnávať skript vo výslednej podobe so skriptom používajúcim pôvodný algoritmus. Pri testoch metrík budeme porovnávať výsledné štatistické hodnoty jednotlivých metrík s pôvodnou Levenshteinovou vzdialenosťou.

5.1 Testovanie efektivity

V tejto podkapitole si ukážeme rozdiel v efektivite, ktorý priniesli všetky zmeny implementované touto prácou. Keďže už vieme, aká bola priemerná doba porovnávania jednej osoby v pôvodnom skripte (2,084 sekúnd), môžeme túto hodnotu porovnať s výslednou hodnotou nového skriptu, získanou rovnakým spôsobom. Celková doba trvania nového skriptu pri spracovávaní tisícich záznamov krstu z databázy demos bola 1366 sekúnd a spracovaných bolo 7933 osôb. Síce sa nejedná o úplne rovnakú testovaciu vzorku, ale ide o rovnaký typ záznamov, v ktorých je takmer rovnaký počet osôb, pričom je výsledná doba trvania aj tak vydelená počtom osôb, aby sme získali priemernú hodnotu. Táto nová hodnota je 0,172 sekúnd na osobu. Aby sme teda určili výsledné zrýchlenie, môžeme vydeliť pôvodnú priemernú hodnotu porovnávania jednej osoby s novou priemernou hodnotou porovnávania. Takto nám vychádza, že je výsledný algoritmus vyše 12-krát rýchlejší.

5.2 Testovanie nových metrík

Ako prvé sa pozrieme na výsledky skriptu nad dodanou testovacou sadou, v podobe csv súborov. Pre účely porovnania metrík použijeme záznamy o krste, konkrétne pôjde o testovaciu sadu s 482 záznamami, vktorých figurujú osoby otca, matky a dieťaťa, pričom bol použitý prah dosahujúci najlepšie výsledky v základnej implementácii používajúcej Levenshteinovu vzdialenosť. Horná hodnota prahu bola 0,95 a spodná hodnota prahu bola 0,8. V nasledujúcej tabuľke môžeme vidieť prehľad jednotlivých metrík a výsledky štatistických hodnôt, ktoré dosahujú.

Tabuľka 5.1: Prehľad štatistických výsledkov metrík

Použitá metrika	Precision	Recall	F-measure
Levenshteinova vzdialenosť	0,775	0,964	0,859
Jarova vzdialenosť (Levenshtein)	0,543	0,986	0,700
Jarova vzdialenosť (pyjarowinkler)	0.542	0.987	0.700
Jarova vzdialenosť (jellyfish)	0,543	0,987	0,701
Jaro-Winklerova vzdialenosť (Levenshtein)	0,543	0,987	0,701
Jaro-Winklerova vzdialenosť (pyjarowinkler)	0.542	0.987	0.700
Jaro-Winklerova vzdialenosť (jellyfish)	0.543	0.987	0.701

Na tabuľke 5.1 môžeme vidieť, že jednotlivé implementácie metrík sa skoro vôbec nelíšia. Zároveň môžeme pozorovať, že nové alternatívne metriky produkujú horšie štatistické výsledky pri testovacom porovnaní, rozdiel je hlavne viditelný pri presnosti, teda pomere správne identifikovaných zhôd ku všetkým identifikovaným zhodám.

5.3 Vplyv nových záznamov na presnosť

Kvôli veľkému rozdielu v počtoch pozitívnych porovnaní a negatívnych porovnaní, ktoré skript učiní, je vhodné použiť štatistickú metriku, ktorá tento fakt zohľadnuje. Tou je balancovaná presnosť (anglicky balanced accuracy) [5]. Tá sa počíta ako priemer pomeru správne identifikovaných zhôd ku všetkým zhodám a pomeru správne identifikovaných nezhôd ku všetkým nezhodám. Takýto výsledok pre nás má výpovednú hodnotu, pretože zohľadnuje nepomer v počte týchto dvoch tried.

Tabuľka 5.2: Prehľad štatistických výsledkov metrík

Typ záznamov	Precision	Recall	F-measure	Balanced accuracy
Iba záznamy krstov	0.855	0.537	0.660	0.768
Iba záznamy úmrtí	0.684	0.786	0.732	0.890
Iba záznamy sobášov	0.634	0.675	0.655	0.836
Všetky typy záznamov	0.814	0.475	0.600	0.737

V tabuľke 5.2 môžeme vidieť výsledné hodnoty štatistických údajov podľa typu vstupných záznamov. Hodnoty prahov použitých pre tento test boli 0,90 pre horný a 0,7 pre spodný. Pri použití všetkých typov záznamov pozorujeme mierny pokles v štatistických hodnotách presnosti algoritmu, okrem hodnoty *precision*. Horšie výsledky algoritmu môžu byť spôsobené i faktom, že vstupné testovacie dáta nie sú v normalizovanej podobe. Množstvo údajov je napísaných v rozličných variantách toho istého slova. Časté sú aj skrátené formy mien, kde je v stĺpci mena alebo priezviska uvedená skratka alebo skrátená forma mena a zároveň aj celé meno osoby. To môže negatívne ovplyvniť porovnávanie, ale poskytuje nám to pohľad ako úspešný by bol algoritmus pri spracovávaní dát v takej forme, v akej ich nájdeme priamo v matrikách (nenormalizovanej). Ďalším faktom, ktorý je nutné zohľadniť je, že množstvo osôb v porovnávacej sade nemalo priradené ID (bolo neznáme), a teda porovnania s osobami, ktoré ID nemali, nemali na štatistické výsledky skriptu žiaden vplyv, okrem faktu, že pri zjednotení s osobami, ktoré ID priradené mali, došlo ku aktualizácii hodnôt a prípadnému prehodnoteniu vzťahov danej osoby.

Kapitola 6

Záver

Cieľom práce bolo rozšíriť prácu Ing. Tušimovej o nové typy genealogických záznamov, zvýšiť časovú efektivitu skriptu pri použití nad databázou a vyskúšať nové alternatívy ku pôvodnej metrike, ktorú skript používal. Začiatku vypracovávania práce predchádzalo naštudovanie oboru genealógie a samotnej implementácie od Ing. Tušimovej. Nové typy záznamov boli úspešne pridané a skript vo výsledku zvládne spracovať všetky typy matričných záznamov, oproti pôvodnej verzii, ktorá fungovala iba so záznamami o krste. Efektivita skriptu pri použití nad databázou bola vo výsledku zvýšená viac ako dvanásťnásobne, ako je možné vidieť v sekcii s testovaním. Podarilo sa nám zistiť, že použitie nových metrík, konkrétne Jarovej a Jaro-Winklerovej vzdialenosti vedie ku zhoršeným výsledkom porovnávania skriptu. Ďalej sme si ukázali nedostatky práce a rôzne implementačné problémy a ich riešenia.

Vývoj práce by mohol pokračovať ďalej, bolo by možné implementovať grafické rozhranie pre vyhľadávacie skripty, pridať podporu pre ďalšie formy historických záznamov ako napríklad sčítania ľudí, či vyskúšať alternatívy ku metódam použitým v procese porovnávania.

Literatúra

[1] História, vznik a vývoj matrík. [http://www.kotesova.info/files/1510-historia-vznik-a-vyvoj-matrik-na-slovensku-1297777123.pdf]. MATRIČNÝ ÚRAD KOTEŠOVÁ. [Online; accessed 12.1.2022].

[2] *Do Not Use "+" to Join Strings in Python* [https://towardsdatascience.com/do-not-use-to-join-strings-in-python-f89908307273]. 2020.

[3] *Using Neo4j from Python* [https://neo4j.com/developer/python/]. 2020.

[4] *MySQL Connector/Python Developer Guide* [https://dev.mysql.com/doc/connector-python/en/]. MySQL, 22. dec 2021.

[5] *What is Balanced Accuracy? (Definition & Example)* [https://www.statology.org/balanced-accuracy/], 6. októbra 2021.

[6] *The Python Profilers* [https://docs.python.org/3/library/profile.html]. 2022.

[7] Cohen, W. W., Ravikumar, P. a Fienberg, S. E. A comparison of String Distance Metrics for Name-Matching Tasks. [https://www.cs.cmu.edu/~wcohen/postscript/ijcai-ws-2003.pdf]. 2003. [Online; accessed 14.1.2022].

[8] Fellegi, I. P. a Sunter, A. B. A theory for record linkage. *Journal of the American Statistical Association.* Taylor & Francis. 1969, zv. 64, č. 328, s. 1183–1210.

[9] Hříbek, D. a Rozman, J. Poloautomatická normalizace slov z matričních záznamů. 2019.

[10] Lednická, B. *Rodopisné stránky* [http://rodokmen.nase-koreny.cz/matriky/obsah_matrik.htm]. [Online; accessed 13.1.2022].

[11] Pine, L. G. *Genealogy* [https://www.britannica.com/topic/genealogy]. Encyclopedia Britannica, 5. sep 2021. [Online; accessed 12.1.2022].

[12] Tušimová, L. a Rozman, J. Generování rodokmenů z matričních záznamů. Vysoké učení technické v Brně, Fakulta informačních technologií. 2020.