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 ARCHIVNÍCH ZÁZNAMŮ

FAMILY TREES MAKING FROM ARCHIVE RECORDS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

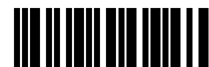
AUTOR PRÁCE **ADAM HALTMAR**
AUTHOR

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

BRNO 2024

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

Zadání bakalářské práce



154297

Ústav: Ústav inteligentních systémů (UITS)
Student: **Haltmar Adam**
Program: Informační technologie
Název: **Generování rodokmenů z archivních záznamů**
Kategorie: Umělá inteligence
Akademický rok: 2023/24

Zadání:

1. Nastudujte obor genealogie a materiály s nimiž genealogie pracuje (matriky, lánové rejstříky, urbáře, atd.). Nastudujte bakalářské práce uvedené v seznamu literatury, které se zabývají automatickým propojováním osob v matričních záznamech (křty, svatby, úmrtí) do větších rodokmenů. Nastudujte projekt DEMoS vyvíjený na naší fakultě a způsob propojování osob v něm.
2. Navrhněte rozšíření a vylepšení systému DEMoS tak, aby byla zvýšena úspěšnost a rychlost propojování. Data načítejte z MySQL databáze a ukládejte do Neo4j databáze.
3. Navržené rozšíření implementujte.
4. Vámi rozšířený systém důkladně otestujte.

Literatura:

- POJEZDÁL, Denis. Generování rodokmenů z matričních záznamů. Brno, 2023. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií.
- MARHEFKA, Adam. Generování rodokmenů z matričních záznamů. Brno, 2022. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií.
- TUŠIMOVÁ, Lucia. Generování rodokmenů z matričních záznamů. Brno, 2020. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií.

Při obhajobě semestrální části projektu 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: 1.11.2023
Termín pro odevzdání: 9.5.2024
Datum schválení: 6.11.2023

Abstrakt

Tato práce popisuje funkci již existujícího programu pro zpracování, porovnání a propojení archivních matričních záznamů o osobách do struktury vhodné pro uložení do grafové databáze s vedlejším účinkem tvorby rodokmenů. Dále se věnuje jeho optimalizaci a zlepšení přesnosti výsledků propojování. Ukazuje nevhodné postupy a jejich alternativní řešení pro zrychlení výpočetní doby běhu programu a rozšiřuje program o další funkce jako například standardizaci jmen pro navýšení úspěšnosti propojování. Výsledkem těchto postupů je dosažení zrychlení přes 5000 %.

Abstract

This thesis is about optimalization and increasing precision of already existing program for processing, comparing and linking archive records about people in a way to store them in a graph database while constructing family pedigrees as a side effect. It shows where the algorithm was approached badly and makes better alternatives for lowering the computation time. It also presents new methods for increasing the precision of record linking like name standardization and compares tests with the original program. Result of these changes is accelerating the program by more than 5000 %.

Klíčová slova

genealogie, tvorba rodokmenů, propojování záznamů, grafová databáze, optimalizace

Keywords

genealogy, pedigree creation, record linking, graph database, optimalization

Citace

HALTMAR, Adam. *Generování rodokmenů z archivních záznamů.* Brno, 2024. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Jaroslav Rozman, Ph.D.

Generování rodokmenů z archivních záznamů

Prohlášení

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

....
Adam Haltmar
8. května 2024

Obsah

1 Úvod **4**

2 Teorie **5**
- 2.1 Genealogie 5
- 2.2 Matriky a další prameny 5
  - 2.2.1 Kniha narození, rodný a křestní list 5
  - 2.2.2 Kniha manželství a oddací list 6
  - 2.2.3 Kniha úmrtí, úmrtní list 6
  - 2.2.4 Sčítací operáty 6
  - 2.2.5 Lánový rejstřík a Poddanská přiznávací fase 6
  - 2.2.6 Urbář a Pozemková kniha 6
- 2.3 Dosavadní implementace 6
  - 2.3.1 Vstupní data 7
  - 2.3.2 Načtení dat 9
  - 2.3.3 Porovnání 10
  - 2.3.4 Propojení 14
  - 2.3.5 Sjednocení 14
  - 2.3.6 Ukládání 15
  - 2.3.7 Výstupní data 15
  - 2.3.8 Statistické metriky 16
  - 2.3.9 Statistiky stavu grafové databáze 17

3 Implementace **19**
- 3.1 Profilování 20
- 3.2 Výpočet vzdálenosti měst 20
- 3.3 Hašování a porovnávání objektů `Person` 21
- 3.4 Kopírování seznamů 23
- 3.5 Garbage Collector 23
- 3.6 Preventivní odstraňování při sjednocování 24
- 3.7 Výpočet překryvu dob života 24
- 3.8 Zavedení proměnné ___slots___ 26
- 3.9 Standardizace jmen 27
- 3.10 Druhá fáze profilování 28
- 3.11 Redukce výpočtů Levenshteinovy vzdálenosti 31
- 3.12 Další drobné změny 31
  - 3.12.1 Selektivní import funkcí 32
  - 3.12.2 Zkratové vyhodnocování 32
  - 3.12.3 Sekvenční procházení hašovací tabulky 33

3.12.4 Vektorizace aplikace vah 33

4 Testování **34**
4.1 Počáteční výsledky 34
4.2 Průběžné výsledky 35
4.2.1 Vliv standardizace 36
4.3 Konečné výsledky 37
4.4 Další vývoj 39
4.4.1 Skóre podobnosti mezi městy 39
4.4.2 Minimální údaje pro sloučení osob 41
4.4.3 Porovnání dat vzdáleností ke dnešnímu dni 41
4.4.4 Operace „<“mezi osobami 42
4.4.5 Ručně zvolené hodnoty 42
4.4.6 Další standardizace 42

5 Závěr **43**

Literatura **44**

Seznam obrázků

2.1 Schéma systému. Převzato z [5], upraveno. 7
2.2 Příklad výpočtu společné doby života dvou osob. Osoba A se sice mohla narodit ve stejnou dobu jako osoba B, určitě však ve stejnou dobu neumřely. 14
2.3 Příklad grafové databáze. Převzato z [8]. 16

3.1 Doba běhu programu v závislosti na počtu zpracovaných osob v porovnání s kvadratickou složitostí 19
3.2 Porovnání rychlosti původní a nové implementace výpočtu vzdálenosti . . . 21
3.3 Porovnání rychlosti porovnávání objektů `UUID` a `int` 22
3.4 Porovnání hašování identifikátoru a načtení předem uložené hodnoty 22
3.5 Srovnání délky výpočtu překryvu života dvou osob po přidání jednotlivých optimalizačních prvků 26
3.6 Čas funkce `get_family_relatives` v závislosti na počtu osob 29
3.7 Čas staré a nové implementace funkce `get_family_relatives` v závislosti na počtu osob 30

4.1 Doba běhu programu v závislosti na počtu zpracovaných osob v porovnání s kvadratickou složitostí 35
4.2 Srovnání délky běhu programu před a po všech optimalizacích 38
4.3 Délka běhu optimalizovaného programu 39
4.4 Skóre podobnosti měst vzhledem ke vzdálenosti těchto měst od sebe 40
4.5 Navrhované funkce F_1 a F_2 pro výpočet skóre podobnosti přes vzdálenost mezi městy 41

Kapitola 1

Úvod

V posledních letech se genealogie stává mnohem častějším koníčkem lidí po celém světě a s rostoucím množstvím digitalizovaných matričních záznamů se otevírá nové pole možností pro vývoj automatizovaných nástrojů, které umožní rychlejší a efektivnější vyhledávání a propojování těchto záznamů, než jakého jsou schopni řadoví odborníci.

Jeden takový program je na FIT VUT sériově vyvíjen studenty jako téma jejich závěrečných prací už několik let a tématem mé práce je ve vývoji tohoto programu pokračovat. V této práci analyzujeme použité metody propojovacího programu, identifikujeme jejich slabiny a zkusíme navrhnout a implementovat alternativní řešení, která povedou k výraznému zkrácení výpočetní doby běhu programu, případně i zlepšení úspěšnosti v porovnání záznamů a propojování osob. V první části se seznámíme s teorií genealogie a rekapitulujeme řešení předešlých prací. Následně si stručně popíšeme předešlou implementaci a základní funkčnost programu, vysvětlíme si, jaké postupy program používá na řešení problémů se spojováním, a ukážeme výsledky časové analýzy výkonnosti programu.

V druhé části pak postupně projdeme nejvíce kritická místa programu, nalezneme problémy v jejich aktuální implementaci a navrhneme alternativní postupy k docílení téhož výsledku za minimalizace výpočetní doby. Navržené postupy implementujeme, otestujeme a porovnáme s původní verzí, abychom si ukázali výsledky našeho snažení.

Dále se budeme věnovat rozšíření programu o standardizaci jmen, a to jak z hlediska zvýšení efektivity, tak co se týče zrychlení v ohledu porovnávání záznamů, poukážeme na další chyby a nedostatky v programu, které v rámci práce nebyly vyřešeny a navrhneme další postup, kterým by se vývoj mohl ubírat.

Kapitola 2

Teorie

V této kapitole se seznámíme se základními pojmy, které se týkají genealogie, podíváme se na různé typy matričních záznamů a dalších historických pramenů, které program umí zpracovat, co vše z nich můžeme vyčíst a následně shrneme obsahy předešlých prací, na které tato navazuje [4, 5, 8].

2.1 Genealogie

Genealogie je historická věda, která se zabývá mezilidskými vztahy rodového původu. Tedy genetickou návazností lidí na sebe. Základním pramenem genealogie jsou matriky. Ty se však povinně zavedly až v 18. století. Do té doby se musíme spoléhat na úřední listiny a knihy, což znamená, že nejvíce informací je o majetných a významných lidech, nikoliv poddaných, kterých bylo mnohem více. V moderní době se jako podpůrný nástroj k tradiční genealogii používají i testy DNA. Genealogický výzkum začíná jedincem, od kterého pak můžeme zkoumat jeho předky nebo jeho potomky.

2.2 Matriky a další prameny

Matriky jsou úřední seznamy jmen a případně dalších údajů, které se váží k nějaké události. Dělí se na živé a mrtvé podle uplynulé doby od posledního zápisu. Zatím co k živým matrikám se kvůli ochraně osobních údajů běžný člověk nedostane, ty mrtvé jsou veřejnosti běžně dostupné. Kromě matrik k nalezení více informací o konkrétních osobách můžeme použít i jiné archivní záznamy jako například soupisy majetku, nebo záznamy o sčítání obyvatel.

2.2.1 Kniha narození, rodný a křestní list

Kniha narození, rodné nebo případně křestní listy jsou záznamy s detaily o narození dítěte případně jeho křtu. Kromě jména a příjmení dítěte se zde ze zákona musí nacházet i datum a místo narození, pohlaví a údaje o rodičích, což zahrnuje jejich jména, datum a místo narození, státní příslušnost a místo trvalého pobytu. Kromě toho zde můžeme nalézt i údaje o porodní bábě, křtícím knězi, prarodičích, praprarodičích a kmotrech dítěte.

2.2.2 Kniha manželství a oddací list

Knihy manželství a oddací listy jsou matriční záznamy o uzavřených sňatcích. Mezi povinné údaje v těchto záznamech patří jména a příjmení manželů, jejich datum a místo narození, datum a místo narození jejich rodičů, datum uzavření manželského sňatku a jména svědků. Někdy zde lze najít i místo bydliště a datum narození svědků, údaje o oddávajícím knězi a prarodičích novomanželů nebo informaci o tom, jestli některý ze zúčastnělých není vdovec/vdova.

2.2.3 Kniha úmrtí, úmrtní list

Knihy úmrtí a úmrtní listy jsou dokumenty s údaji o zemřelých osobách. Nalezneme v nich datum a místo úmrtí, jméno, příjmení, případně rodné příjmení, datum a místo narození, manželský stav, pohlaví, státní občanství, místo trvalého bydliště a jméno žijícího manžela nebo manželky. Také zde ale mohou být informace o rodičích a prarodičích zemřelého, jeho potomcích, pokud nějaké měl, třetích osobách, které měly s úmrtím něco společného, například byly svědky smrtelné nehody, pečovateli, nebo hrobníkovi, který mrtvého pohřbil.

2.2.4 Sčítací operáty

Sčítací operáty, neboli populační censy, soupisy obyvatelstva, byly obecné záznamy o lidech sloužící hlavně k vojenským a daňovým účelům. Typicky se zde nachází údaje o obyvatelích nějakého statku, takže většinou otce rodiny, který statek vlastní a jeho rodičích, manželce a dětech, kteří tam s ním žijí.

2.2.5 Lánový rejstřík a Poddanská přiznávací fase

Lánový rejstřík neboli lánová vizitace je seznam poddanských usedlostí a pozemků ze 17. století. Přestože lánové rejstříky obsahují enormní množství informací co se týče majetku na různých statcích, pro naše účely jsou pouze minimálně relevantní. Pro nás užitečnými informacemi zde jsou údaje o majiteli pozemku, jeho ženě, pokud na ni jeho pozemek připadl po jeho smrti, případně jeho potomkovi, který pozemek zdědil. Poddanská přiznávací fase je pak obdobný dokument z 18. století, kde je sice osob uvedeno více, ale nejsou mezi nimi nutně stanoveny jejich vzájemné vztahy.

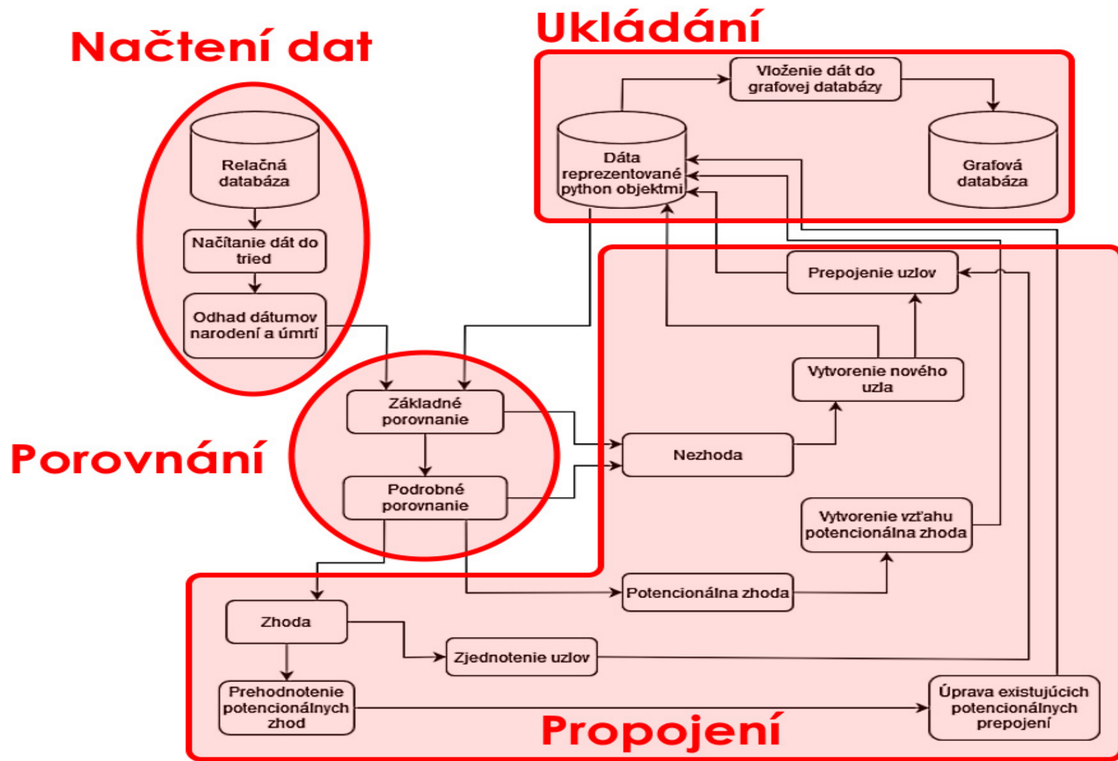
2.2.6 Urbář a Pozemková kniha

Urbáře a z nich následně vzniklé pozemkové knihy jsou soupisy majitelů panství pro evidenci platů a dávek jejich poddaných. Kromě dat a majitele, případně majitelů, pokud jich bylo více, se žádné důležité informace z urbářů nedozvíme. I tak jsou ale hodnotným zdrojem, protože nám mohou potvrdit, kde se daná osoba v nějakém období nacházela a také můžeme lépe zaměřit dobu narození nebo úmrtí, pokud nám tyto informace o dané osobě chybí.

2.3 Dosavadní implementace

V této sekci se seznámíme s fungováním programu pro porovnávání archivních záznamů a propojování osob vyvíjeným po řadě studenty L. Tušimovou, A. Marhefkem a D. Pojezdálem. Ukážeme si strukturu vstupních dat, hlavní tok programu, využívané algoritmy důležité pro vyhodnocování shod a výstupní data.

Jak lze vidět na následujícím diagramu (obr. 2.1), program lze rozdělit na 4 logické celky – načtení dat, porovnávání osob, vytváření vztahů mezi osobami a uchování této reprezentace v paměti plus její perzistentní uložení do grafové databáze.



Obrázek 2.1: Schéma systému. Převzato z [5], upraveno.

2.3.1 Vstupní data

Většina vstupních dat je uložena v relační MySQL databázi na fakultním serveru do struktury tabulek s na sebe navzájem se odkazujícími prvky. Ať už se jedná o obecnou tabulku zastřešující všechny vlastnosti pro každou archiválii, tabulku dat pro konkrétní typ matriky, tabulku se jmény osob, názvy států, nebo seznam vinic, aktuálně se v databázi nachází přes 80 různých tabulek, takže jen demonstračně popíšu strukturu tří z nich.

Název sloupce	Datový typ
id	int
birth_id	int
marriage_id	int
burial_id	int
rel	enum
title	string
sname	int
domicile	int
street	string
descr_num	string
religion	enum
birth_date	string
dead_date	string
work_place	string
age	float

Tabulka 2.1: Struktura tabulky `person` pro ukládání údajů o konkrétních osobách

Kromě automaticky generovaného primárního klíče se v tabulce s daty osob nachází cizí klíče na tabulku se záznamem, kde je tato osoba vedena. Přestože je struktura zdánlivě navržená tak, že by jedna osoba mohla odkazovat jak na záznam o narození, sňatku i úmrtí, reálně je vyplněn vždy pouze jeden z těchto sloupců, protože osoby v tabulce `Person` jsou generovány z údajů z oněch konkrétních archiválií a propojení stejných osob se záznamy různých typů je až jedním z výsledků našeho programu.

Ve sloupci `rel` (relation) je pak typ vztahu této osoby **k záznamu**. Například vztah `father` k záznamu o narození dítěte znamená, že dotyčná osoba je otcem narozeného dítěte. Výčtový typ pro sloupec `rel` pak obsahuje ještě speciální hodnotu `main`, která značí hlavní osobu neboli „vlastníka" záznamu. Pro onen záznam o narození by to tedy byl sám novorozenec.

Dále se zde nachází sloupce pro název ulice a číslo orientační, náboženství, datum narození a úmrtí, kde zase typicky známe pouze jedno z nich, podle toho, z jakého záznamu osoba pochází, věk (zemřelého) a další.

Název sloupce	Datový typ
id	int
archive_id	int
fond_id	int
signature	string
inv	int
type	enum
lang1	enum
lang2	enum
lang3	enum
scan	int

Tabulka 2.2: Část tabulky `register` pro obecné údaje o archiválii

V tabulce `register` nalezneme data, která jednak zastřešují všechny typy záznamů, ale taktéž spojují konkrétní reálnou písemnost s její digitalizovanou verzí v databázi. Najdeme zde odkaz na archiv, ze kterého záznam pochází, signaturu, konkrétní sken, jazyky zápisu nebo jeho typ (viz 2.2)

Název sloupce	Datový typ
id	int
register_id	int
sex	enum
dead	int
birth_date	string
baptism_date	string
owner	int
parents_marr_when	string

Tabulka 2.3: Část tabulky `birth` pro data ze zánamů o narození dítěte

Konkrétní údaje z matriky pak jsou třeba v tabulce `birth` pro záznamy o narození, kde najdeme respektive můžeme najít pohlaví novorozence, jestli se narodil mrtvý, datum jeho křtu, odkaz na dítě jakožto osobu v tabulce `person` (položka `owner`), ale někdy například i datum svatby jeho rodičů, což může být velice užitečné.

2.3.2 Načtení dat

Jak už bylo řečeno, vstupní data pro živou verzi programu jsou uložena v MySQL databázi (viz 2.3.1). K programu ovšem existuje i testovací sada dat, která má podobnou strukturu jako data z databáze, akorát že data v této sadě jsou uložena v CSV souborech lokálně. O načítání dat se stará modul `relational_database.py` respektive `csv_source.py` v případě práce s CSV soubory a načítání probíhá po jednotlivých záznamech. Nejdříve se zjistí, zdali mateřská archiválie záznamu již v databázi existuje a potenciálně se tam přidá, pak se pro každou osobu vytvoří instance objektu `Person`, která reprezentuje konkrétní osobu a kromě konkrétních údajů dostupných v záznamu probíhá pro každou osobu ještě odhad doby jejího narození a úmrtí podle následujících předpokladů:

- Datum narození zemřelého je rok úmrtí − jeho věk ± 1 rok
- Ženich, nevěsta a svědci na svatbě mají alespoň 15 let
- Matce narozeného dítěte je minimálně 15 a maximálně 45 let
- Otci narozeného dítěte je minimálně 15 a maximálně 60 let
- Datum narození / křtu a úmrtí jsou od sebe maximálně 100 let
- Hrobník, porodní bába, kněz, pečovatel a všechny další osoby vykonávající nějaké povolání mají minimálně 18 let
- Datum pohřbu je maximálně 1 měsíc po datu úmrtí
- Kmotrům je minimálně 18 let

- Osobám uvedeným v lánovém rejstříku bylo 18 k datu 1. ledna 1668
- Osobám uvedeným v poddanské přiznávací fasi bylo 18 k datu 1. ledna 1747
- Osobám uvedeným v urbáři a pozemkové knize je minimálně 16 let
- Pokud jsou v záznamu další osoby, například rodiče, jejichž věk lze z těchto pravidel také omezit, učiní se tak

Dále je potřeba data zpracovat do formátu, v jakém s nimi program může efektivně pracovat, což zahrnuje například převod dat do formátu `date` z knihovny `datetime`, protože, jak můžeme vidět z tabulek 2.1 a 2.3, v databázi jsou data uložena jako textový řetězec. Nakonec jsou uloženy informace o aktuálně zpracovávaném záznamu a přechází se do fáze porovnání.

2.3.3 Porovnání

V této fázi dostaneme seznam osob z nějakého archivního záznamu a jednu po druhé ji budeme porovnávat se všemi lidmi, kteří již v databázi uložení jsou. V rámci snížení průchodů tímto cyklem jsou lidé v databázi rozdělení do seznamů podle jejich pohlaví na muže, ženy a neznámé, čímž se docílí, že se spolu nebudou porovnávat lidé rozdílného pohlaví. Dále se z tohoto seznamu vyčlení ještě všichni příbuzní osoby, která se chystá být porovnávána, a zbylí lidé z aktuálního záznamu. Před samotným postupem při porovnávání vlastností si ještě vysvětlíme, jak se porovnávají různé datové typy.

Slova

Ať už jde o jména, příjmení, názvy měst, ulic, povolání, nebo titul, vše jsou to textové řetězce, ve kterých se mohou vyskytovat a velice často opravdu vyskytují chyby nebo jsou zapsány v různých jazycích, zdrobněle, zkratkovitě nebo neformálně. V databázi proto existuje tabulka normalizovaných jmen, kde najdeme pro různé varianty téhož jména jeho unifikovanou podobu. Tato tabulka ale ani zdaleka neobsahuje všechna jména, se kterými se při zpracovávání záznamů setkáme a z toho důvodu je potřeba nějakým způsobem vyjádřit, jak moc jsou si 2 slova podobná s větší přesností než pouze na SHODNÁ, JINÁ. V programu se k tomuto osvědčilo používat Levenshteinovu vzdálenost. [10]

Levenshteinova vzdálenost mezi 2 slovy je definována jako minimální počet znaků, které je potřeba v jednom slově přidat, změnit a odstranit, aby z něj vzniklo slovo druhé a výsledná vzdálenost L pak je

$$L(w_1, w_2) = i \cdot w_i + s \cdot w_s + d \cdot w_d$$

kde w_1 a w_2 jsou slova, mezi kterými chceme vypočítat Levenshteinovu vzdálenost, i, s a d je počet operací přidání znaku (insertion), změny znaku (substitution) a smazání znaku (deletion), které je potřeba provést a w_i, w_s a w_d jsou váhy těchto operací.

Sama vzdálenost pro porovnání ale nestačí, předpokládejme, že všechny váhy jsou 1, kdybychom pak měli 2 zjevně různé osoby se jmény „Max“ a „Bob“, jejich vzdálenost by byla 3, protože nemají nic společného a musí se provést 3 substituce. Ovšem pro jména „Johan“ a „Johannes“, která obě značí stejné jméno a jeden člověk je v průběhu svého života mohl v různých dokumentech užívat obě, je vzdálenost také 3, protože se musí provést 3

vložení znaku. Proto se v programu zavedlo používání tzv. normalizované Levenshteinovy vzdálenosti L_N, která se vypočítává jako

$$L_N(w_1, w_2) = 1 - \frac{L(w_1, w_2)}{max(|w_1|, |w_2|)}$$

Tím v našem případě dostaneme

$$L_N(Bob, Max) = 1 - \frac{L(Bob, Max)}{max(|Bob|, |Max|)} = 1 - \frac{3}{max(3, 3)} = 1 - \frac{3}{3} = 1 - 1 = 0$$

$$L_N(Johan, Johannes) = 1 - \frac{L(Johan, Johannes)}{max(|Johan|, |Johannes|)} = 1 - \frac{3}{8} = 1 - 0,375 = 0,625$$

Tedy zatím co jména „Bob“ a „Max“ mají skóre 0, neboli jsou zcela odlišná, mezi jmény „Johan“ a „Johannes“ máme 62, 5% shodu.

Data (datum)

Všechna data se porovnávají 3 různými způsoby a za konečný výsledek se následně považuje nejvyšší skóre z nich. Prvním způsobem je převedení data na textovou podobu a jeho porovnání pomocí Levenshteinovy vzdálenosti (viz. 2.3.3 sekce Slova). Tím je možné obejít chyby v zápisu, opisu typu „napsal 8, ale vypadá to jako 6“, případně chyb „o jedničku“, když si někdo nepamatuje přesné datum a splete se o 1 den, měsíc nebo rok.

Druhým způsobem je eliminování různých typů zápisu přehozením pozice dne a měsíce. Máme-li třeba 3. květen, někde toto datum v číselné formě může být zapsáno jako „3/5“, ale jinde zase jako „5/3“

Třetím způsobem je výpočet skóre pomocí poměru věků osob ke dnešnímu dni, který se počítá prakticky jako

$$S = F\left(\frac{A}{B}\right)$$

kde S je skóre podobnosti, A je věk mladšího, B je věk staršího a F je aktivační funkce pro stanovení maximální tolerance, definovaná jako

$$F(x) = \begin{cases} x & \text{pro } x > 0,9 \\ 0 & \text{jinak} \end{cases}$$

To by podle [5] mělo řešit nepřesnosti vzniklé tím, že si lidé v minulosti přesně nepamatovali svůj věk.

Čísla

Čísla se porovnávají prvním a třetím způsobem pro datum. U třetího způsobu se akorát nevypočítává věk, ale pracuje se rovnou s oním číslem. Viz 2.3.3 sekce Data (datum).

Souřadnice

V programu je uložen seznam měst s jejich zeměpisnými souřadnicemi, abychom mohli vypočítat vzdálenost mezi městy. Vzhledem k velice omezeným způsobům přepravy se v minulosti lidé moc často nestěhovali a když už, tak rozhodně ne moc daleko. Proto, můžeme

použít vzdálenost mezi městy jako jeden z faktorů, který bude snižovat pravděpodobnost shody osob. K tomu je využíván tento vzorec

$$S = \begin{cases} 1 - \frac{1}{e^{100-d}} & \text{pro } d < 100 \\ 0 & \text{jinak} \end{cases}$$

kde S je výsledné skóre podobnosti a d je vzdálenost mezi městy v kilometrech.

Osoby

Co se týče porovnávání celých osob, program využívá pravděpodobnostní klasifikaci popsanou Ivanem Fellegi a Alanem Sunterem v jejich práci *A Theory for Record Linkage* z roku 1969 [2]. Skóre podobnosti dvou osob je vypočten součtem skóre porovnání všech dílčích údajů o osobách s aplikováním předem definovaných vah na tyto údaje, které vypovídají o tom, jak moc by tyto údaje měly o podobnosti osob rozhodovat. Každá osoba má až 25 údajů, které se mezi sebou mohou porovnat, ale vždy se porovnávají pouze ty, které jsou u obou osob vyplněné. Konečný součet je pak normalizován do intervalu $< 0, 1 >$ vydělením maximálním možným skóre všech porovnávaných údajů. Mějme například tyto osoby

#	Jméno	Příjmení	Datum narození	Město	Titul
1	Petr	Němec	3.7.1836	Ostrava	Ing
2	Pepa	Krupička	7.3.1836	Praha	–
3	Josef	Krupička	3.7.1852	Praha	–

Tabulka 2.4: Příklady údajů osob

s těmito váhami

Jméno	Příjmení	Datum narození	Město	Titul
1,0	2,0	0,8	0,7	0,5

Tabulka 2.5: Příklady vah

Budeme porovnávat osobu 1 s osobou 2 a osobu 2 s osobou 3. Výsledky porovnání dílčích údajů vypadají takto:

Údaj 1	Údaj 2	Skóre
Petr	Pepa	0,500
Němec	Krupička	0,125
3.7.1836	7.3.1836	0,995
Ostrava	Praha	0,429
Ing	–	–

Tabulka 2.6: Porovnání osob 1 a 2

Údaj 1	Údaj 2	Skóre
Pepa	Josef	0
Krupička	Krupička	1
7.3.1836	3.7.1852	0,910
Praha	Praha	1
–	–	–

Tabulka 2.7: Porovnání osob 2 a 3

Jméno a příjmení je normalizovaná Levenshteinova vzdálenost, jak bylo vysvětleno v 2.3.3, pro datum narození u osob 1 a 2 nám prvním způsobem porovnání dat Levenshteinovou vzdáleností vyjde skóre 0,75, druhým způsobem, přehozením dne a měsíce sice dostaneme totožné datum, ale skóre po procesu přehození je omezeno na 0,5 a třetím způsobem dostaneme hodnotu 0,995, která se zvolí, protože je nejvyšší. U měst se taktéž volí

nejvyšší výsledek, takže i přesto, že Praha je od Ostravy 280 kilometrů vzduchem a z porovnání souřadnic vyjde skóre 0, Levenshteinovou vzdáleností je podobnost 0,43. Titul u druhé osoby chybí, takže se žádné porovnání neprovádí. Obdobně dojde k porovnání u osob 2 a 3, kde za zmínku zase stojí datum, kterému skóre Levenshteinovou vzdáleností vyjde 0,5, druhý způsob se neaplikuje, protože se nejedná o stejný rok a třetím způsobem dostaneme 0,910.

Na příslušné skóre je ještě potřeba aplikovat příslušnou váhu, což si můžeme představit jako skalární součin vektoru se všemi skóre a vektoru všech vah. Jelikož se ne všechny údaje vždy porovnají, doplní se vektor skóre nulami. Celkové skóre S je pak součet dílčích skóre s aplikovanými váhami

$$S_1 = (0.5, 0.125, 0.995, 0.429, 0) \cdot (1, 2, 0.8, 0.7, 0.5) = 0.5 + 0.25 + 0.796 + 0.3 + 0 = 1.846$$

$$S_2 = (0, 1, 0.91, 1, 0) \cdot (1, 2, 0.8, 0.7, 0.5) = 0 + 2 + 0.728 + 0.7 + 0 = 3.428$$

Toto skóre se ale ještě musí normalizovat vydělením počtem porovnaných atributů, což jsou v tomto případě 4. Pravděpodobnosti shody P_1 pro shodu mezi první a druhou osobou a P_2 pro shodu mezi druhou a třetí osobou tedy jsou

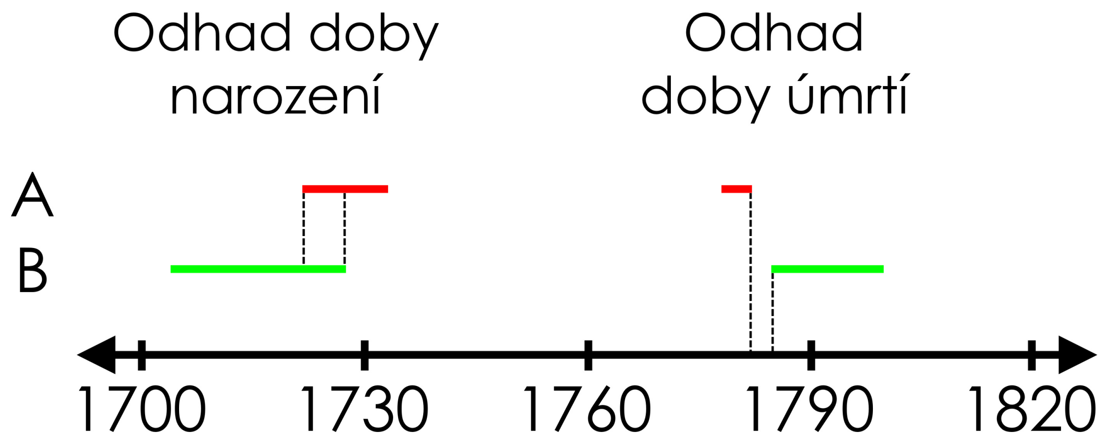
$$P_1 = \frac{S_1}{4} = \frac{1.846}{4} = 46\,\%$$

$$P_2 = \frac{S_2}{4} = \frac{3.428}{4} = 86\,\%$$

Toto číslo ale stále nemusí být konečné. Abychom se opravdu ujistili, že se jedná o jednu osobu, pokud o nich máme údaje, porovnají se tímto způsobem i shodní předci obou osob a jako finální skóre podobnosti dvou osob se pak považuje vážený průměr z výsledku jejich porovnání a porovnání jejich předků.

Před porovnáním dvou osob se ale provádí ještě rychlé předporovnání, kde se podle těch nejvíce základních údajů zjistí, zda-li osoby splňují nutné prerekvizity a k detailnímu porovnání všech jejich údajů má vůbec dojít. Toto předporovnání se skládá ze dvou částí – potvrzení společné doby života a nalezení společného údaje.

Jelikož se předpokládá, že ne všechna data jsou vždy úplně přesná a zavedená pravidla pro odhady dob narození a úmrtí (viz 2.3.2) nejsou většinou schopná tyto události konkrétně zaměřit, jak s datem narození, tak s datem úmrtí, se pracuje jako s intervaly. Pokud osoby nemají průnik v intervalu odhadu doby narození i v intervalu odhadu doby úmrtí, jedná se o různé osoby a žádné další porovnání už se neprovádí.



Obrázek 2.2: Příklad výpočtu společné doby života dvou osob. Osoba A se sice mohla narodit ve stejnou dobu jako osoba B, určitě však ve stejnou dobu neumřely.

Když mají osoby společnou dobu života, ještě se porovná jejich jméno, příjmení, zaměstnání a město, ve kterém žijí. Pokud nic z toho nepřesáhne podobnost 90%, k dalšímu porovnání také nedojde.

2.3.4 Propojení

Poté, co je osoba z nového záznamu porovnána se *všemi*[1] lidmi z databáze, přechází se do fáze propojování. Program využívá dva prahy pro rozhodování o dalších akcích – práh shody a práh potenciální shody s hodnotami 0.9 respektive 0.7. Pokud při porovnání skóre podobnosti žádné osoby nepřesáhlo práh shody, vytvoří se pouze vazba potenciální shody se všemi osobami, které přesáhly práh potenciální shody. Pokud se ale našla alespoň jedna osoba, jejíž skóre podobnosti přesáhlo práh shody, ze všech takových se vybere jedna z nejvyšším skóre podobnosti a dojde ke sjednocení údajů nové osoby s touto osobou. Jelikož sjednocením údajů dochází vlastně k aktualizaci údajů o osobě, která už v databázi je, znovu se přepočítá skóre podobnosti se všemi osobami, které jsou s touto vedené jako potenciální shoda, aby se zjistilo, jestli se jejich podobnost nenavýšila, a může tak dojít ke sloučení i zde, nebo nesnížila, aby se vztah potenciální shody mohl zrušit.

2.3.5 Sjednocení

Při sjednocení dvou osob se provedou prakticky jen 2 operace – kontrola potenciálních shod a sjednocení údajů. Při porovnávání mohla nastat situace, že se osoba vyhodnotila jako shoda s nějakou jinou osobou, ale zároveň jako potenciální shoda například s otcem této osoby. Při sjednocení tedy dochází k odstranění vztahů potenciálních shod s příbuznými. Samotné sjednocení údajů je pak přímočará záležitost. Jelikož nechceme žádné údaje ztratit, ukládají se všechny. Když tedy dojde ke sloučení osoby se jménem „Pepa“ s osobou se jménem „Petr“, je výsledkem osoba s množinou jmen Pepa, Petr. Jediná významná věc při slučování je změna intervalů pro odhady dob narození a úmrtí, které se omezí na průniky těchto intervalů a jsou tak díky tomu přesnější.

[1]Nejedná se o úplně všechny, viz sekce Porovnání

2.3.6 Ukládání

Aby se při porovnání zamezilo vytváření extrémního množství dotazů na grafovou databázi, byl zvolen velice agresivní přístup nahrání a po dobu běhu programu udržování celé databáze v paměti v tzv. datové reprezentaci. Ta funguje jako abstrakce grafové databáze, různé typy objektů v ní jsou uloženy v seznamech a pro vytváření vztahů mezi nimi se používá třída `Relation`, kterou lze reprezentovat orientovaný vztah mezi dvěma objekty, tak jak tomu je v grafové databázi. Jakmile se všechna vstupní data zpracují, objekty a vztahy z datové reprezentace se 1 ku 1 uloží do Neo4j grafové databáze.

Grafová databáze je typ NoSQL databáze, kde se uložené údaje ukládají grafickou formou – pomocí vrcholů a hran. Kronkrétně v tomto projektu se využívá grafová databáze Neo4j, která pro dotazy nad svými daty využívá jazyk Cypher.

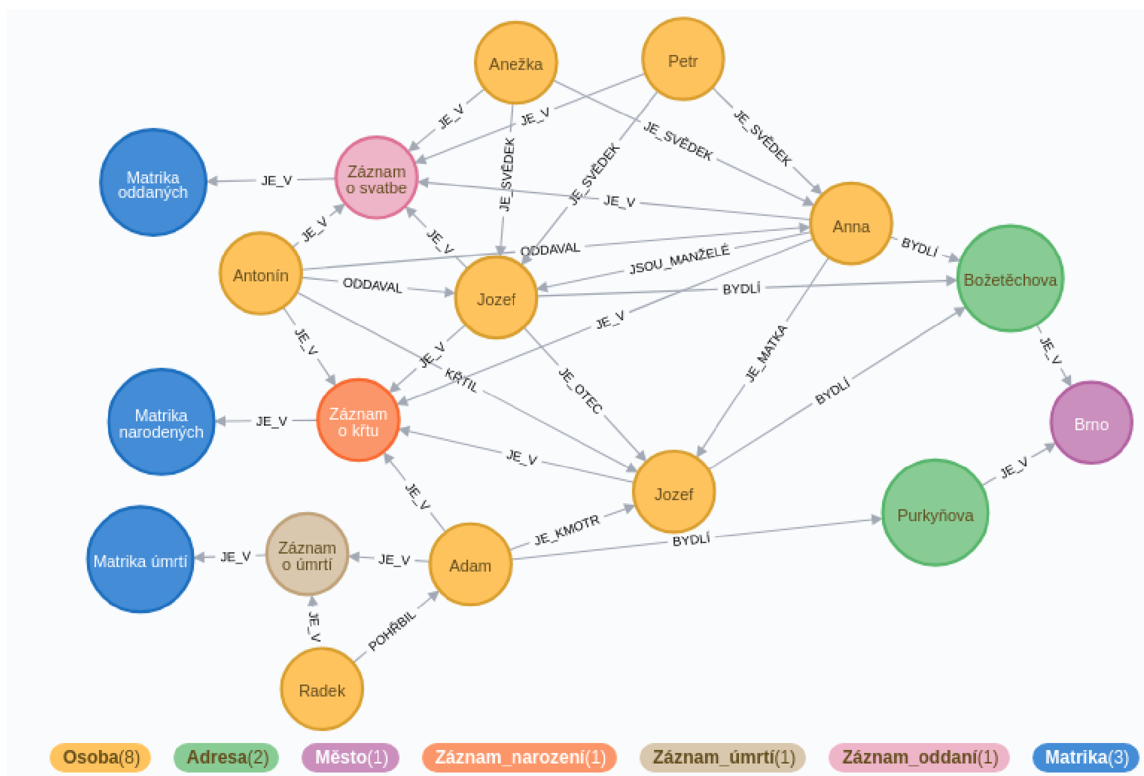
2.3.7 Výstupní data

Výstupem programu je nahrání datové reprezentace na server s grafovou databází. Tento druh databáze ukládá data do grafové struktury jako vrcholy a následně mezi nimi definuje vztahy do hran spojujících tyto vrcholy. Tento způsob ukládání dat je pro ukládání vztahů mezi osobami velice výhodný, protože na rozdíl od relační databáze nemáme potřebu pro každý typ vztahu definovat nový sloupec, který třeba u většiny záznamů ani nebude vyplněn, nebo rovnou celou spojovací tabulku.

Aktuálně se do grafové databáze ukládá 13 druhů vrcholů:

- Lánový rejstřík
- Matrika
- Město
- Osoba
- Poddanská přiznávací fase
- Číslo popisné
- Pozemková kniha
- Sčítací operát
- Ulice
- Urbář
- Záznam o křtě
- Záznam o svatbě
- Záznam o úmrtí

Mezi vrcholy pak existuje 40 různých druhů hran, které je mohou propojovat. Na obrázku 2.3 můžeme vidět výřez z vizuální reprezentace grafové databáze. Když se zaměříme na vrchol „Anna“, můžeme zjistit, že se jedná o osobu, která bydlí na ulici Božetěchova, která se nachází někde ve městě Brně, její manžel se jmenuje Jozef, mají spolu syna Jozefa, oddával je Antonín a za svědky jim byli Petr a Anežka.



Obrázek 2.3: Příklad grafové databáze. Převzato z [8].

2.3.8 Statistické metriky

Abychom mohli ověřit úspěšnost propojování, disponuje program při práci s daty v přiložených CSV testovacích souborech, u kterých správné výsledky propojování známe, mechanismem pro zařazení výsledků do klasifikačních tříd a výpočet statistickým metrik [7]. Po provedení každého porovnání se tedy zkontroluje správný výsledek a podle kombinace programem určeného a správného výsledku se toto porovnání zařadí do jedné z kategorií

- Správně pozitivní (True positive) – TP – o dvou záznamech se vyhodnotilo, že patří téže osobě, a byla to pravda
- Falešně pozitivní (False positive) – FP – o dvou záznamech se vyhodnotilo, že patří téže osobě, ale nebyla to pravda
- Správně negativní (True negative) – TN – o dvou záznamech se vyhodnotilo, že patří různým osobám, a byla to pravda
- Falešně negativní (False negative) – FN – o dvou záznamech se vyhodnotilo, že patří různým osobám, ale nebyla to pravda

Přičemž naším cílem je, aby se co nejvíce výsledků vyhodnotilo správně, tedy aby co nejvíce porovnání bylo v kategori TP, nebo TN a naopak co nejméně v kategoriích FP a FN.

Z kombinací těchto kategorií pak lze vypočítat další výkonnostní metriky pro lepší představu o kvalitě výsledků vyhodnocování. V programu jsou využívány následující:

Přesnost

Přesnost, neboli precision, je míra spolehlivosti udávající, jak moc se můžeme spoléhat na to, že pozitivní výsledek byl vyhodnocen správně. Jde tedy o poměr správně pozitivních porovnání ze všech, která vyšla jako pozitivní. Vyjádřeno vzorcem:

$$precision = \frac{TP}{TP + FP}$$

Citlivost

Citlivost, neboli recall, nám udává, jak přesně program dokáže pozitivní výsledek vyhodnotit opravdu jako pozitivní. Jde tedy o poměr správně pozitivních porovnání vůči všem, která měla vyjít jako pozitivní.

$$recall = \frac{TP}{TP + FN}$$

F-skóre

F-skóre, F-míra, F_1 skóre, nebo dále pouze jen F_1, je metrika pro vyhodnocení úspěšnosti testu kombinací přesnosti a citlivosti vypočítaná jako jejich harmonický průměr.

$$F_1 = 2 \cdot \frac{precision \times recall}{precision + recall}$$

Specificita

Specificita (specificity), je procento negativních případů, které byli správně vyhodnoceny. Tedy:

$$specificity = \frac{TN}{TN + FP}$$

Vyvážená přesnost

Vyvážená přesnost, neboli Balanced accuracy, je pak metrika spolehlivosti, která, na rozdíl od normální přesnosti, bere ohled na silně nevyvážené kategorie. To je v našem případě velice podstatné, protože jen velice málo porovnání je ve skutečnosti shoda, zatím co skoro všechna jsou ve skutečnosti neshoda. Vyvážená přesnost BA je dána jako průměr citlivosti a specificity.

$$BA = \frac{recall + specificity}{2}$$

2.3.9 Statistiky stavu grafové databáze

Statistické metriky z předchozí sekce jsou užitečné pro získávání informací o průběhu porovnávání, ale nedozvíme se z nich, v jakém stavu je celá databáze po skončení programu. Abychom zjistili, kolik uzlů se v databázi nachází a jaký je jejich stav, program vypočítává i další sadu statistik, vytvořených přesně pro tento účel. V testovací sadě má každá osoba předdefinovaný identifikátor, abychom mohli určit, jestli je výsledek sloučení dvou osob správný nebo špatný. Pokud se sloučí dvě osoby s rozdílným identifikátorem, výsledná osoba si uchová všechny jejich identifikátory. To umožňuje zavést čtyři kategorie, do kterých osoby ve výsledné databázi můžeme klasifikovat.

- Perfect (**P**) – konkrétní identifikátor osoby je v celé databázi pouze jednou a osoba s tímto identifikátorem žádný jiný nemá
- Mixed (**M**) – konkrétní identifikátor osoby je v celé databázi pouze jednou, ale osoba s tímto identifikátorem má i nějaký další
- Split (**S**) – konkrétní identifikátor je v databázi vícekrát, ale všechny osoby s tímto identifikátorem žádný jiný nemají
- Split_mixed (**SM**) – konkrétní identifikátor má více osob a některé z těchto osob mají i další identifikátory

Kategorie **P** znamená 100% úspěšnost programu při vyhodnocování těchto osob.

Kategorie **S** jsou případy, kdy mezi sebou osoby mohou mít vztah potenciální shody, ale program o nich ještě neměl dost informací, aby došlo k jejich sloučení. Osoby v této kategorii jsou statisticky neutrální – sice nedošlo k chybě, ale ani k úspěšnému propojení.

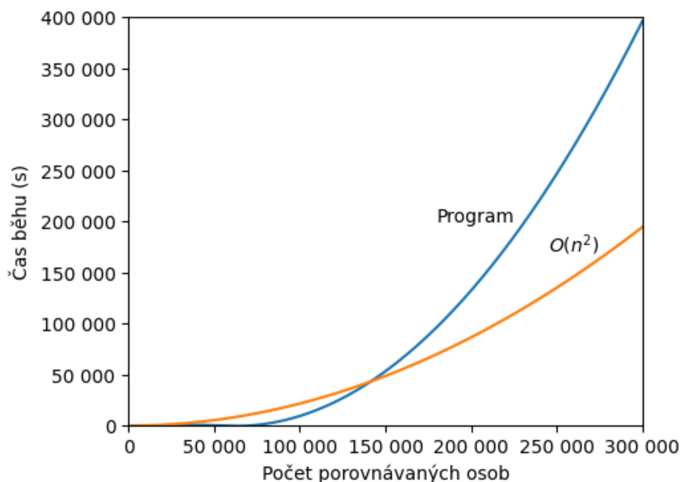
Kategorie **M** a **SM** jsou pak chybové. V případě **SM** se lidé slučují, jak se jim za chce, a tato sloučení tak mají nulovou hodnotu a v pro **M** buď došlo ke správnému sloučení všech záznamů a stejné osobě, ale připletla se tam i nějaká cizí, nebo se jsou to zdegenerované záznamy z kategorie **SM** o osobách, o kterých byl v databázi pouze jeden záznam (a tedy jejich identifikátor byl v databázi pouze jednou už od samého začátku).

Kapitola 3

Implementace

Vzhledem k tomu, že na programu už pracovalo několik lidí a typicky jejich jediný způsob „komunikace" byly informace z programové dokumentace od jejich předchůdců, nacházel se program ve velice špatném stavu. Některé funkce se stejnou funkcionalitou jsou definované vícekrát, mnohdy v modulech/třídách, kam vůbec nepatří. Dále program obsahoval spoustu mrtvého kódu a spoustu zřejmě netestovaného kódu, který možná někdy v minulosti fungoval, ale nyní s novými typy archivních záznamů v relační databázi generoval neošetřené výjimky. Přestože jsem vytvářením všemožných záplat na tento děravý kód strávil významnou dobu, nebudu se jim věnovat, protože nejsou věcně důležité.

Hlavním problémem programu je, že je velice pomalý. Testovací sadu, ve které se nachází pouze 30 000 osob, je schopný zpracovat za 15 minut. Když ho ale spustíme nad celou databází, ve které se aktuálně nachází kolem 300 000 osob, rázem doba běhu vystřelí na zhruba **110 hodin** neboli 6600 minut. To je pouze 10x nárůst ve velikosti vstupních dat, ale 440x nárůst doby vyhodnocování. Pokud bychom se bavili o porovnávání každý s každým, měla by složitost programu být $O(n^2)$. V našem případě ale nové osoby porovnáváme pouze s lidmi, kteří už v databázi jsou a prakticky polovinu z nich jsme schopni vyřadit rovnou na základě jejich pohlaví. Měli bychom se tak zhruba pohybovat někde v rozmezí $\frac{n^2}{8}$ až $\frac{n^2}{2}$. Reálný časový průběh však vypadá takto:



Obrázek 3.1: Doba běhu programu v závislosti na počtu zpracovaných osob v porovnání s kvadratickou složitostí

3.1 Profilování

Aby se dalo zjistit, kde přesně se nachází problémy, které program takto zpomalují, musíme změřit čas, který byl v průběhu programu stráven v jednotlivých funkcích. K tomu slouží Profilování. K profilování byl použit nástroj `lsprof` přes rozhraní `cProfile`. Profilování ovšem přidává programu obří režii, zvlášť velkou když je v programu velké množství volání jednoduchých funkcí. Počáteční profilování bylo tedy provedeno na několika menších částech databáze, kdy největší byla zpracování 15 000 záznamů z různých kategorií (cca 83 000 osob). Z výsledků úvodního profilování vyplynulo, že většina výpočetní doby programu se tráví pouze v těchto několika funkcích[1]:

Funkce	Čas (s)	Čas (%)
main	6230	100
compare_record_.../<listcomp>	1650	26,5
levenshtein_distance	1550	24,9
check_lifetime_overlap	848	13,6
person/__eq__	775	12,4
unite_nodes	593	9,5
town_distance	492	7,9
person/__hash__	331	5,3
gc.collect	234	3,8

Tabulka 3.1: Časový podíl funkcí na běhu programu při spuštění nad 15 000 záznamy

Funkce `main` reprezentuje celý program a je zde uvedena pouze pro přehlednost. List comprehension ve funkci `compare_record_with_graph_database` se stará o vybírání osob z databáze, se kterými se nová osoba bude nebo nebude porovnávat. Funkce `levenshtein_distance` vypočítává Levenshteinovu vzdálenost mezi slovy, `person/__eq__` a `__hash__` jsou funkce pro porovnání dvou osob na shodu a vytvoření hashe z objektu osoby pro práci s hašovací tabulkou, do které se osoby ukládají pro jejich rychlé vyhledávání. `Unite_nodes` slouží ke spojení údajů dvou osob do jedné při vyhodnocení shody těchto osob. Funkce `town_distance` slouží k výpočtu vzdálenosti mezi dvěma městy z jejich zeměpisných souřadnic a `gc.collect` je volání „Garbage Collectoru“ pro uvolnění nevyužívané paměti.

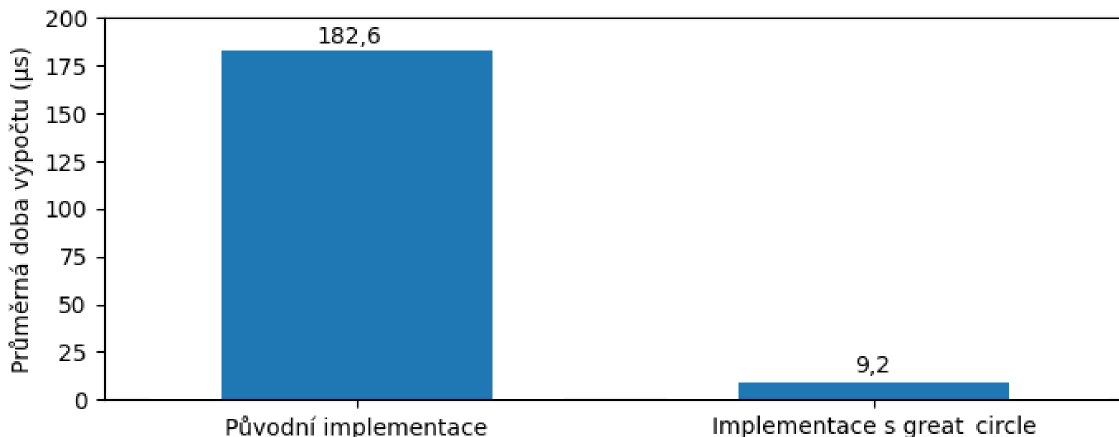
3.2 Výpočet vzdálenosti měst

V programu se na výpočet vzdálenosti mezi dvěma městy využíval komplexní algoritmus na výpočet přesné vzdálenosti mezi dvěma body na geoidu, která dává sice velice přesné výsledky, ale o to náročnější je její výpočet. Navíc ona extra přesnost, kterou tento algoritmus poskytuje, se nejvíce projevuje až na velkých vzdálenostech. Vzhledem k tomu, že my počítáme s relativně malými vzdálenostmi (cokoliv nad 100km už nás nezajímá) a na extrémní přesnosti by nám také nemuselo záležet (viz dále), je zcela zbytečné se tohoto způsobu výpočtu vzdálenosti držet.

Použitím funkce `great_circle` (ciz [9]) z knihovny `geopy`, která počítá vzdálenost mezi dvěma body jako vzdálenost bodů na povrchu koule s poloměrem průměrného poloměru

[1] Jelikož uvedený list comprehension pracuje s objekty typu `Person`, je pravděpodobné, že část jeho doby má průnik s funkcemi __hash__ a __eq__ třídy `Person`. Totéž platí i o funkci `unite_nodes`. Ostatní funkce však na sobě nezávislé jsou.

Země, jsme schopni pouze s minimální ztrátou přesnosti tento výpočet významně urychlit. Na vzdálenostech do 100km je maximální chyba/odchylka vůči původnímu výpočtu pouze 0.32%, což i na maximální vzdálenosti dělá jen 320 metrů. Mnohem zajímavější je ale rychlost výpočtu, která je vůči původní implementaci nižší skoro o 95 %. Z původních 492 sekund při spuštění nad 15 000 záznamy dostáváme něco kolem 25 sekund.



Obrázek 3.2: Porovnání rychlosti původní a nové implementace výpočtu vzdálenosti

3.3 Hašování a porovnávání objektů `Person`

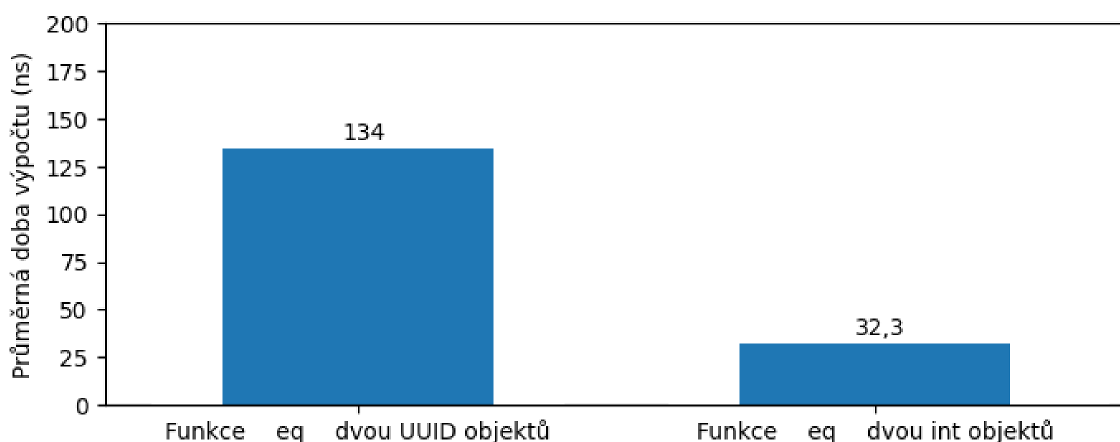
Stejně tak jak tomu je v relačních databázích, i grafové databáze potřebují při ukládání dat mít u každého záznamu primární klíč, který jej bude jednoznačně identifikovat. Za primární klíč můžeme v relačních databázích označit nějakou kombinaci sloupců, pro kterou víme, že se na více řádcích určitě nebude opakovat. Běžnější řešení ale je mít primární klíč pouze jako 1 samostatný sloupec. Kdybychom však v našem případě u osob chtěli použít nějaký údaj, který o nich již máme uložený, jako primární klíč, daleko bychom nedošli, protože jedna z nutných vlastností primárního klíče je, že vždy musí nabývat nějaké hodnoty a my u žádného údaje nikdy nemáme jistotu, že určitě byl v záznamu vyplněn. Druhým faktorem je pak i to, že jednoznačný identifikátor občanů, jako je například rodné číslo, se na našem území zavedl až v roce 1946 a my pracujeme typicky s mnohem staršími záznamy.

Z tohoto důvodu se pro každou nově příchozí osobu do systému vygeneruje tzv. univerzálně (nebo globálně) unikátní identifikátor, neboli UUID respektive GUID (viz [3]). Což by samo o sobě bylo naprosto v pořádku a nemělo by skoro ani smysl se o tom zmiňovat. Problém ovšem nastává v situaci, kdy začneme pracovat s hašovacími tabulkami nebo sekvenčně prohledávat seznamy, kvůli čemuž se nad UUID objekty (podle kterých se osoby identifikují) musí neustále volat funkce pro porovnání a vytvoření hashe. A i když tyto operace nepatří mezi nejdražší, při miliardách volání, kterých vzhledem k N^2 složitosti dosáhneme už při relativně málo osobách, se každá nanosekunda projeví.

Řešení? Přetypování a cachování

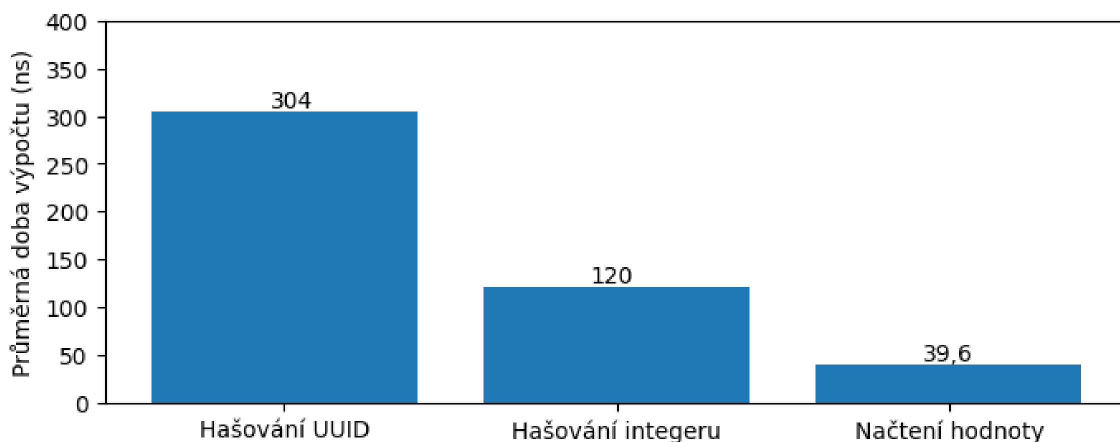
UUID je ve své podstatě pouze 128bitová hodnota, neboli číslo. Pokud pracujeme pouze s tímto číslem a píšeme kód „bezpečně“ – tedy nepotřebujeme extra režii, například typovou kontrolu při každém porovnání, kterou třída $UUID$ provádí, můžeme namísto $UUID$

objektu pracovat pouze s objekty typu *int*. Z testů pak vychází, že porovnání objektů *int* je až čtyřikrát rychlejší než porovnání objektů *UUID*.



Obrázek 3.3: Porovnání rychlosti porovnávání objektů `UUID` a `int`

Totéž v menší míře platí i pro hašování těchto datových typů. Hašování je samozřejmě náročnější proces než pouhé zjištění rovnosti, ale i tak nám změna typu z `UUID` na `int` poskytne 153% zrychlení tohoto procesu. To ale není vše. Protože v hašovací tabulce jsou na většinu osob po čas průběhu porovnávání „kladeny dotazy“ vícekrát, identifikační číslo konkrétní osoby se od jejího vytvoření už nemění a hašovací funkce pro stejný vstup dává vždy tentýž výstup, není potřeba ten stejný hash vypočítávat opakovaně. Každý objekt `Person` si nyní při vytvoření společně s číselnou hodnotou svého UUID vytvoří a uloží i jeho hash, aby v budoucnu nemusela hashovací funkce být volána opakovaně. Tímto zcela zanedbatelným zvednutím nároku na paměť jsem tak zrychlil práci s hashováním osob o dalších 203 % vůči ukládání `int` místo `UUID` objektu a celkově o 668 % oproti původnímu řešení.



Obrázek 3.4: Porovnání hašování identifikátoru a načtení předem uložené hodnoty

3.4 Kopírování seznamů

Nejvíce vytěžující funkcí v úvodním profilování byl list comprehension ve funkci `compare_record_with_graph_database`. Podstatou této funkce je zpracovat všechny osoby uvedené v jednom konkrétním archivním záznamu. Kromě vytvoření vztahů mezi osobami podle jejich role v záznamu, tato funkce ještě vymezuje, s jakými osobami z databáze se nově příchozí bude porovnávat. To zahrnuje vytvoření seznamu příbuzných dotyčného, vytvoření seznamu všech osob z aktuálního archivního záznamu a nakonec vytvoření seznamu všech osob z databáze, které nejsou v ani jednom z těchto dvou seznamů. K tomu se využívá tzv. list comprehension, neboli způsob zápisu vytvoření nového seznamu na základě podmínek a jiných iterovatelných objektů, v tomto případě dvou seznamů.

Každý již zpracovaný člověk v databázi se tak musí otestovat na přítomnost v seznamu příbuzných `relatives` a v seznamu osob z aktuálního záznamu `current` a protože to jsou seznamy, tak to v praxi znamená sekvenční průchod obou z nich a porovnávání každý s každým, navíc s absolutně minimální šancí na úspěch, protože v jednotlivých záznamech nebývá více než 20 lidí, rodiny také nebývají nikterak rozsáhlé a v celé databázi mohou být desítky až stovky tisíc osob. To vše ještě korunuje fakt, že se tím až na pár výjimek vyrábí kopie celého seznamu s osobami v databázi. Tento seznam sice obsahuje jen reference na objekty, ne objekty samotné, takže v řádech desítek až stovek tisíc osob se bavíme o kopírování pouze několika jednotek MB. I to se ale nasčítá a při zpracování celé databáze už jsou to desítky GB. V případě, že bychom databázi dále chtěli rozšiřovat, mohl by se z tohoto stát závažný problém.

Elegantním a jednoduchým řešením je nejdříve převést seznamy `relatives` a `current` na množiny. Množiny jsou datové struktury na bázi hašovací tabulky. To znamená, že se ze sekvenčního procházení seznamu, kde vyhledání konkrétní osoby má lineární složitost, $O(n)$, dostaneme na přímý přístup k hodnotě, neboli konstantní složitosti, $O(1)$.

Dále jsem z těchto dvou množin udělal jednu množinu `exclude`, kde jsou všichni příbuzní i lidé z aktuálního záznamu, kteří mají být vyčlenění z porovnání a redukoval tím tak počet nutných testů na přítomnost v množinách ze dvou na jeden.

Třetí krok pak byl delegovat rozhodování o tom, s kým se osoba bude a nebude porovnávat, na funkci řídící porovnávání. Tedy místo kopírování celého seznamu s odkazy na všechny osoby a vyčlenění množiny `exclude` pro vytvoření vyfiltrovaného seznamu osob, se kterými se nová bude porovnávat, posílám porovnávací funkci celý seznam všech osob a množinu `exclude`, ať si za běhu provádí filtraci sama bez kopírování.

Delegováním filtrace do stádia porovnávání se na jednu stranu zvedla časová náročnost porovnávání na vzorku 15 000 záznamů o zhruba 300 sekund, ale na stranu druhou tím zároveň zcela zaniklo oněch 1650 sekund ve funkci `compare_record_with_graph_database`, takže celý program byl oproti původní verzi při úvodním profilování zrychlen o skoro 28 %.

3.5 Garbage Collector

Python je interpretovaný jazyk, což znamená, že ke běhu Python programu je potřeba externí program, který kód našeho programu bude provádět, tzv. interpret. Jednou z výhod interpretu je, že za nás spravuje veškerou paměť, kterou náš program chce alokovat a v momentu, kdy si náš program řekne o další, ale on ji od operačního systému není schopný dostat, ať už z důvodu celkového nedostatku paměti, nebo kvůli manuálně nastavenému omezení, spustí svůj podprogram zvaný Garbage Collector, který pomocí sledovacího algo-

ritmu zjistí, které části alokované paměti může uvolnit. K tomu ale potřebuje, aby po čas tohoto hledání program s pamětí nepracoval, a proto ho zastaví.

Kromě toho, že když je program zastaven, logicky nemůže pracovat, je samotné prohledávání paměti celkem náročná operace. Proto je zcela nevhodné volat tento proces manuálně vždy po provedení sloučení 100 osob. Pokud by to mělo být potřeba, mohl by se tento limit zvednout alespoň na 1000, nebo jinou hodnotu, při které dopad na celkový výkon bude zanedbatelný. Každopádně já jsem po zvážení manuální volání Garbage Collectoru odstranil úplně, protože, jak už bylo řečeno, v případě potřeby se spustí automaticky.

Přesný výsledek této změny nelze jednoznačně určit, protože je závislý na hardwarových podmínkách stroje, na kterém je program spuštěn, ale maximální a v mém osobním případě i reálné zrychlení je 4 %, neboť disponuji dostatečným množstvím paměti, aby za celý průběh programu nemusela být uvolňována.

3.6 Preventivní odstraňování při sjednocování

Program si pro urychlení práce vede seznamy osob konkrétního pohlaví (muže, ženy a neznámé), aby nově příchozí osoby nemusel porovnávat se všemi lidmi opačného pohlaví, kde už předem víme, že bude výsledek negativní. To může dělat, protože lidé v minulosti pohlaví během života neměnili. Bohužel uvnitř databáze se nám zavedením neznámé hodnoty pohlaví časem pohlaví osob měnit může – když jsme ji našli v záznamu, ze kterého její pohlaví zřejmé nebylo, ale pak se k nám dostane další záznam o této osobě, kde její pohlaví bude specifikováno.

Funkce `unite_nodes` fungovala tak, že na vstup dostala osobu z databáze, a osobu z nově příchozího záznamu, která se s ní měla sloučit. Osoba z databáze byla odstraněna ze seznamu pro osoby stejného pohlaví, kterého aktuálně nabývala, pak proběhlo sjednocení údajů těchto osob a finální osoba s aktualizovanými údaji byla zase přidána do příslušnoho seznamu osob se stejným pohlavím.

Vyhledat ale osobu v seznamu pro její odstranění znamená sekvenčně seznam procházet a každou jeho položku testovat, dokud se tato konkrétní osoba nenajde. To znamená celkem náročná operace a vzhledem k tomu, že ve většině případů osoby své pohlaví při slučování údajů nemění a byly tedy odstraněny ze seznamu, kam jsou hned zase vráceny, i celkem zbytečná. Výrazně lze zrychlit průběh této funkce jen tím, že se počáteční pohlaví osoby uloží, údaje osob se spolu sloučí, porovná se počáteční a konečné pohlaví a až v případě, že se pohlaví osoby sloučením změnilo, je tato osoba z jednoho seznamu vyňata a do jiného seznamu vložena.

Už po aplikování předešlých změn se čas strávený v této funkci snížil z původních 593 sekund na 357 sekund a přidáním této kontroly změny pohlaví jakožto podmínku pro přeřazení do jiného seznamu, se z těchto 357 sekund na vzorku 15 000 záznamů stalo pouhých 137 sekund, přičemž konkrétně odstraňování ze seznamů, jehož podíl v této funkci činil 270 sekund z 357, se snížilo na 52 sekund. To znamená 160% zrychlení oproti verzi po předešlých změnách nebo 333% zrychlení proti času z úvodního pofilování.

3.7 Výpočet překryvu dob života

Jedna z nejjednodušších funkcí celého programu je funkce `check_lifetime_overlap` pro výpočet překryvu dob narození a úmrtí dvou osob. Přestože se jedná pouze o 1 logický

výraz, je tato funkce jedna z nejnáročnějších, alespoň co se celkové doby strávené v ní týče, protože se jedná o takovou vstupní bránu k porovnání dvou osob.

Kompletní porovnání dvou osob je velice náročná operace, v programu je proto využíván jednoduchý test na překryv odhadované doby narození a úmrtí těchto dvou lidí a až v případě, že se potvrdí, že tyto osoby žily ve stejném období, porovnávají se jejich zbylé osobní údaje. Dále, pokaždé, co se najde shoda s jinou osobou a dojde ke sloučení údajů, odhad dob narození a úmrtí se může změnit, takže je nutné překontrolovat všechny potenciální shody těchto osob. To, po předešlých změnách redukujících počet volání funkcí __hash__ a __eq__, dělá z této funkce nejvolanější funkci v celém programu a druhou nejnáročnější funkci na výpočetní čas.

Načítání vstupních dat v programu řeší pro každý druh archiválie jiná funkce a různé funkce vytvářeli různí lidé až v moment, kdy ve školní databázi pro tyto archiválie vznikly tabulky. To mělo za důsledek, že každá z těchto funkcí si data načítá víceméně jak se autorovi zamanulo, a ta jsou tak nekonzistentní. U většiny údajů toto nehraje žádnou roli, typicky řetězce se ukládají 1 ku 1 a žádný prostor pro rozmanitost tam není. Opravdu velký problém je ale v ukládání datumů. Ty se někde ukládají do typu `datetime` ze stejnojmenné knihovny `datetime`, ale jinde jen jako typ `date` z této knihovny.

Tento problém už sice byl předešlým autorům známý, ale z nějakého absurdního důvodu se jej někdo rozhodl řešit ne při načítání dat, ale právě ve funkci na výpočet překryvu dob života. Konkrétně první věc, co tato funkce před samotným výpočtem učiní, je kontrola všech čtyř hodnot pro počátek a konec intervalů pro narození a úmrtí na typ `datetime`, a když se o `datetime` jedná, transformuje ho na typ `date`.

Jak už jsme si řekli dříve, toto je nejvolanější funkce celého programu, protože se jedná o vstupní bránu k porovnávání, ke které se minimálně dostanou každé dvě osoby, co se mají porovnat. Pro 83 000 osob v 15 000 záznamech to znamená stovky milionů průchodů touto funkcí. Jako `datetime` se ukládá jen minimum ze všech načtených dat a převod na `date` stačí provést pouze jednou. To v praxi znamenalo, že ve více než 99,9999 % případech se v této funkci provedla úplně zbytečná kontrola datového typu čtyř hodnot.

Předělal jsem tedy všechny funkce pro načítání dat tak, aby datum ukládaly jako typ `date` a tuto kontrolu z výpočtu překryvu doby života oddělal. Jen zavedením této datové konzistence se čas strávený v této funkci snížil na 22,7 % původní hodnoty. To znamená 348% zrychlení, což není špatný začátek.

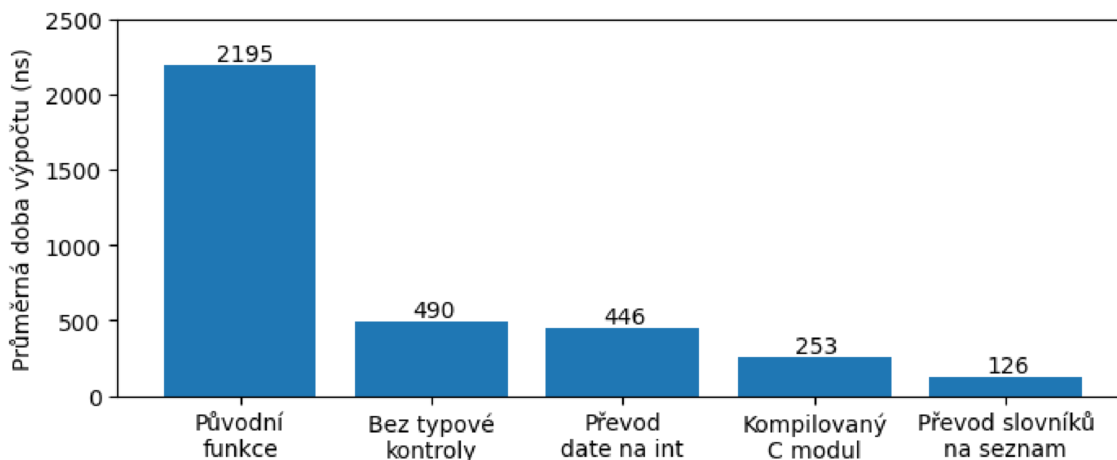
Údaje o těchto datech se u každé osoby ukládají do dvou různých slovníků, jeden pro narození, druhý pro úmrtí, které obsahují položky „od“ a „do“ s typem `date`. Slovník je sice velice efektivní struktura pro vyhledávání dat, ale v tomto případě ukládáme vždy pouze 2 hodnoty, mezi kterými nic hledat nepotřebujeme, prostě chceme buď tu první, nebo druhou, ale většinou obě, a neustálé hašování klíčů pro přístup k těmto hodnotám zbytečně zpomaluje celou funkci. Také počítání s `date` není zcela efektivní, kromě extra režie s kontrolami datových typů a přepočtem časových zón se datum porovnává po složkách. Jen uložením dat jako jednoho čísla, která se celá porovnají jednou operací, docílíme zhruba 10% zrychlení funkce pro výpočet překryvu dob života.

Toho můžeme docílit tak, že původní datum z typu `date` převedeme na počet dní od nějakého pevného bodu T_0. Za T_0 jsem zvolil datum 1. ledna roku 1500, aby počet dní k tomuto bodu vycházel pro všechna data v databázi jako pozitivní číslo, ale zároveň tato čísla nebyla zbytečně velká. To znamená, že pokud bychom měli osobu, jejíž datum narození máme vytyčené někdy do období mezi 5. lednem 1500 a 10. únorem 1500, tento odhad by se převedl na interval „od“ 4 „do“ 40.

Navíc teď v tomto stavu používáme jednoduchý datový typ, konkrétně integer, a to nám vzhledem k tomu, že celá funkce na porovnávání dob narození a úmrtí používá pouze základní aritmeticko-logické operace, umožňuje tuto funkci přepsat do kompilovaného C modulu za pomoci jazyka Cython. Tím jsem se zbavil režie, kterou s sebou nesou Python objekty, čímž jsem dosáhl dalšího 76% zrychlení.

Pořád se do této funkce ale předávají argumenty ze slovníků, které po všech těchto úpravách mají zásadní podíl na celkovém času. To jsem vyřešil převodem těchto slovníků na seznamy, čili prostá pole, u kterých se nemusí nic hašovat a C modul přímo přistupuje na potřebné indexy. Což vyústilo v další 100% zrychlení.

Celkově tedy bylo dosaženo více než 1600% zrychlení v rámci funkce na výpočet překryvu dob života, což činí 94 % úspory času a celý program byl tímto zrychlen o 15 %.



Obrázek 3.5: Srovnání délky výpočtu překryvu života dvou osob po přidání jednotlivých optimalizačních prvků

3.8 Zavedení proměnné __slots__

V Pythonu každá instance třídy obsahuje slovník, který uchovává její stav prostřednictvím atributů. Tento přístup poskytuje vysokou flexibilitu, protože můžeme přidávat nové atributy instancím již definované třídy, ale také může vést k nadměrné spotřebě paměti, protože každá instance třídy si musí vést vlastní slovník, a snížení výkonu, kvůli následnému vyhledávání v těchto slovnících. V této sekci se zaměřím na využití proměnné __slots__, která může významně přispět k efektivnějšímu využití paměti a zrychlení programu, zvlášť pokud vytváří velké množství instancí různých tříd.

Proměnná __slots__ je speciální třídní proměnná, kterou lze ve třídě definovat pro omezení dynamického vytváření atributů. Deklarací __slots__ určujeme pevnou strukturu atributů, které mohou být instancemi třídy přiřazeny a eliminuje se tak vytváření slovníku __dict__ pro každou instanci, což zredukuje paměťovou náročnost a zrychlí přístup k těmto atributům, protože nebude potřeba vyhledávat klíče ve slovníku.

Nevýhodou zavedení __slots__ je nemožnost dynamicky přidávat instancím nové proměnné, což ovšem v programu nikde není potřeba, a možná nekompatibilita s některými knihovními funkcemi, která se ale při aplikaci v našem programu nikde neprojevila.

Deklarací __slots__ jsem prakticky zadarmo dosáhl 6% zrychlení celého programu a pravděpodobně byla i snížena paměťová náročnost, což však měřeno nebylo.

3.9 Standardizace jmen

Protože spousta jmen v databázi obsahuje různé překlepy, nebo i když opis z archiválie je přesný, samotný zápis se může lišit mezi matrikami, například jedna osoba je v různých záznamech vedená pod jmény „Joan“, „Joannes“, „Johannes“, „Johanes“, „Johan.“, „Joh.“, nebo „Joan.“, je pro správnou funkci programu potřeba mezi těmito slovy najít nějakou podobnost. K tomu slouží funkce `levenshtein_distance` pro výpočet normalizované Levenshteinovy vzdálenosti mezi dvěma slovy. Jedna nevýhoda tohoto přístupu je, že výpočet funkce Lehvenstainovy vzdálenosti mezi skoro každými dvěma osobami je velice zdlouhavý. Pro případ s 15 000 záznamy je to po předchozích optimalizačních změnách 48 % celkové doby běhu programu. Jelikož se k tomuto účelu využívá knihovní funkce, není možné ji jednoduše upravit a navíc lze předpokládat, že z hlediska výkonu bude navržena spíše dobře. Ještě další nevýhodou je, že ani výsledky tohoto výpočtu nejsou moc přesné. Například pro již zmíněná slova „Joh.“ a „Joannes“ je skóre podobnosti 0,29, což nestačí ani na práh potenciální shody, natož na sloučení těchto osob. V databázi sice existuje tabulka normalizovaných jmen, která by všechny varianty téhož jména měla unifikovat na jedno slovo, ale v současném stavu se v ní nachází pouze 47 jmen a tím pádem se normalizace využívá v porovnání pouze u 1,46 % osob. Ve zbylých případech, kdy standardizované jméno dostupné není, vypočítával program podobnost jmen normalizovanou Lehvenstainovou vzdáleností.

Já jsem standardizaci jmen rozšířil o seznam standardizovaných jmen dostupných na webu projektu DEMoS[2], díky čemuž se počet porovnání s normalizovanými jmény zvedl z 1,46 % na 93 %, a čas ve funkci `levenshtein_distance` se snížil o 88 %, čímž se celý program zrychlil o 59 %. Aby se při každém spuštění programu nemusela jména z webu stahovat, program nebyl na funkčnosti webu závislý a neměnil výstupy zejména při testování, v případech, kdy na webu dojde k aktualizaci nebo opravě jmen, byla aktuální verze seznamu jmen z dubna 2024 uložena ve formátu vhodném pro načtení do slovníku do JSON souboru.

Bohužel využití normalizace nepřineslo pouze pozitivní výsledky, ale i nějaké negativní. False Positive případy u porovnávání narostly o 7,7 % a False Negative až o 29 %.

Nárůst False Positive případů se dal očekávat, protože normalizací jmen se zvedla podobnost osob, které mají stejné jméno, přičemž skóre podobnosti ostatních vlastností zůstalo stejné, takže typicky osoby stejného jména, o kterých víme jen velice málo a jejich nenormalizovaná verze jmen prahu shody nedosáhla, nyní nějakého z prahů podobnosti dosáhnout mohou. To by se do nějaké míry dalo řešit snížením váhy pro jméno nebo využitím spolehlivosti skóre podobnosti, které udává jak moc se můžeme spolehnout na skóre podobnosti podle toho, kolik údajů se mezi osobami porovnávalo, jejíž výpočet v programu už sice je, ale jeho výsledky se nikde nevyužívají, aby jméno samotné nestačilo pro vyvolání sloučení osob.

Nárůst False Negative případů už mi pak tak jasný není. Jak přesně ke špatnému vyhodnocení pozitivních případů dochází, se mi zjistit nepodařilo, ale je možné, že kvůli tomu, že standardizovaná forma některých položek ze seznamu na webu byla vytvořena počítačem, se různé formy stejného jména klasifikovaly jako různá jména. Každopádně z analýzy False Negative případů jsem zjistil, že mnoho z nich jen těsně nedosahuje hranice pro potenciální

[2] `http://www.genealogickadatabaze.cz/`

shodu. Snížením této hranice z původních 0,7 na 0,6 se mi povedlo docílit ještě o 30 % nižších případů False Negative než před normalizací.

3.10 Druhá fáze profilování

Po všech výše uvedených změnách se podíváme na jejich souhrnný vliv provedením dalšího profilování programu.

Funkce	Čas (s)	Čas (%)
main	2 203	100
unite_nodes	274	14,4
get_family_relatives	240	10,9
komunikace s databází	241	10,9
levenshtein_distance	188	8,5
check_set_valid	117	5,3
check_lifetime_overlap	49	2,2
person/__hash__	40	1,8
person/__eq__	23	1
town_distance	2	0,1

Tabulka 3.2: Časový podíl funkcí na běhu programu při spuštění nad 15 000 záznamy po provedení některých optimalizací

Jak můžeme vidět, nejvíce problémové funkce detekované při úvodním profilování se mi úspěšně povedlo odstranit a na tomto konkrétním vzorku byl celý program zrychlen o 185 %. Samozřejmě tento seznam obsahuje jen nepatrnou část všech funkcí, které se v programu volají. Účelem této tabulky je identifikovat nové náročné funkce (s procentuální zátěží nad 3 %) a demonstrovat rozdíl času u funkcí, které byly optimalizovány. Zároveň zde nemá smysl uvádět kořenové funkce v programové hierarchii, které samy nic nepočítají, ale provolávají všechny ostatní funkce, čímž v nich program stráví třeba 40 % svého času.

Nutno také podotknout, že ve výše uvedených funkcích je velký časový překryv. Zejména u funkce `get_family_relatives`, která z 98 % přispívá svým časem k času funkce `unite_nodes`.

Nyní, když je program dostatečně optimalizován, mohu provést profilovaní i většího vzorku dat. To je potřeba udělat zejména kvůli tomu, aby se odhalily například exponenciálně náročné funkce, které se na menších vzorcích ještě nestihly dostatečně projevit. Takto vypadají výsledky profilování na vzorku 30 000 záznamů, což odpovídá 153 000 lidem.

Funkce	Čas (s)	Čas (%)
main	8 358	100
get_family_relatives	2 288	27,4
levenshtein_distance	580	6,9
komunikace s databází	452	5,4
check_set_valid	412	4,9

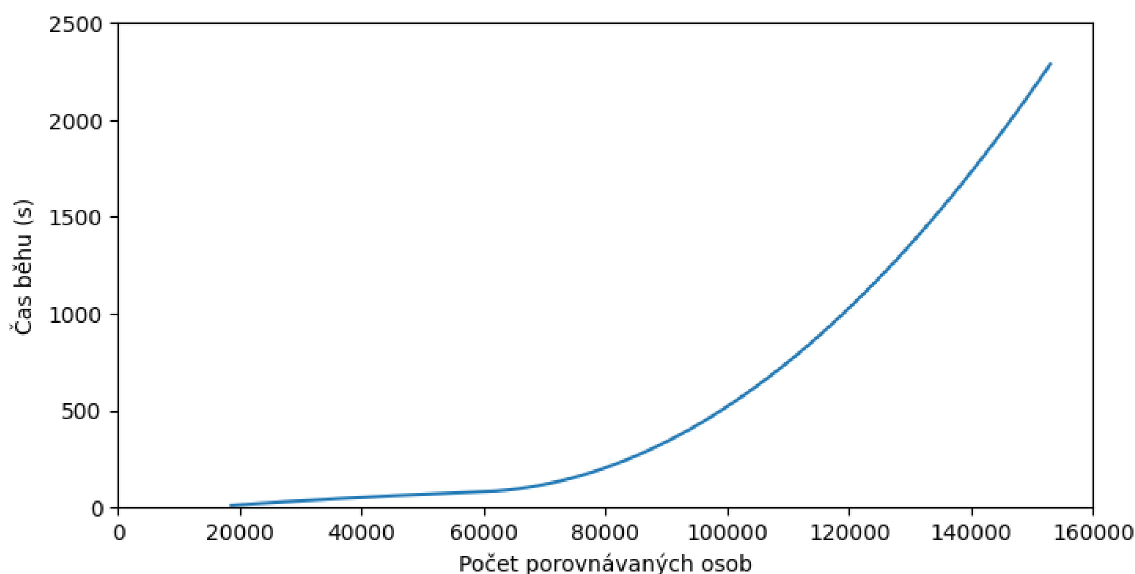
Tabulka 3.3: Výsledky profilování pro N_{30000}

Jak lze vidět, tři z těchto funkcí rostou zcela podle předpokladu, a sice komunikace s databází, ta má prakticky lineární průběh a nic s ní dělat nemůžeme, funkce `check_set_valid`

pro kontrolu jestli množina neobsahuje nulové a prázdné hodnoty (dvě osoby, jejichž příjmení neznáme a jsou tak obě reprezentována prázdným řetězcem, neznamená, že mají příjmení stejné) roste zhruba kvadraticky, což také bylo očekáváno, totéž platí pro `levenshtein_distance`. Překvapením je funkce `get_family_relatives`, která se na menších vzorcích tvářila neškodně, ale nyní při nárůstu osob o pouhých 83 % navýšila svůj potřebný čas o 853 %.

Smyslem funkce `get_family_relatives` je při nalezení shody, před sloučením osob, vyhledat celý rodokmen druhé osoby, aby se odstranily všechny potenciální shody s rodinnými příslušníky této osoby. Ačkoli se náročnost tohoto procesu na malých vzorcích zdála zanedbatelná, s čím více osobami a rodinami pracujeme, tím jsou rodiny větší a širší a jejich prohledávání je tak zdlouhavější. Každý člověk má 2 rodiče a ti mají také 2 rodiče atd. Pro N prohledávaných generací, pak bude mít každá osoba minimálně 2^N příbuzných.

Tento fakt můžeme pozorovat i na datech:



Obrázek 3.6: Čas funkce `get_family_relatives` v závislosti na počtu osob

Počet záznamů	Počet osob	Čas (s)	Čas (%)
3 000	19 000	11	4,8
6 000	36 000	45	8
15 000	83 000	240	10,5
30 000	153 000	2 288	27,4

Tabulka 3.4: Čas funkce `get_family_relatives` v závislosti na počtu osob

Mezi prvními dvěma testy je nárůst osob pouze 1,9krát, ale čas strávený hledáním příbuzných je více než čtyrnásobný. To by ještě bylo v pořádku, ale čím více osob program zpracovává, tím horší tento poměr je. Mezi prvním a posledním testem je sotva 8krát více osob, ale 208krát více času hledáním příbuzných.

Tento proces je sice důležitý, aby náhodou nedošlo ke sloučení s předkem a nevznikaly tak cykly v rodokmenu, ale je zcela zbytečné vyhledávat v databázi celý rodokmen. Každá

nová osoba už má nějaký odhad doby života podle toho, z jakého typu archivního záznamu byla vzata. Dále program počítá se spoustou limitací pro různé události. Například že matka mohla porodit nejdříve v 15 letech nebo že maximální doba života člověka je 100 let. Díky těmto limitacím můžeme výrazně omezit hloubku prohledávání rodokmenu.

Originálnímu řešení možná také moc nepomohl způsob implementace přes iterativní průchod oboustrannou frontou s nějakou abstrakcí směru, který určuje, jestli se zpracuje prvek zepředu, zezadu, nebo z obou stran. Upřímně jsem s absencí jediného komentáře vůbec nepochopil, o co se autor snažil, a při hlubší analýze jsem v kódu našel i několik mrtvých větví.

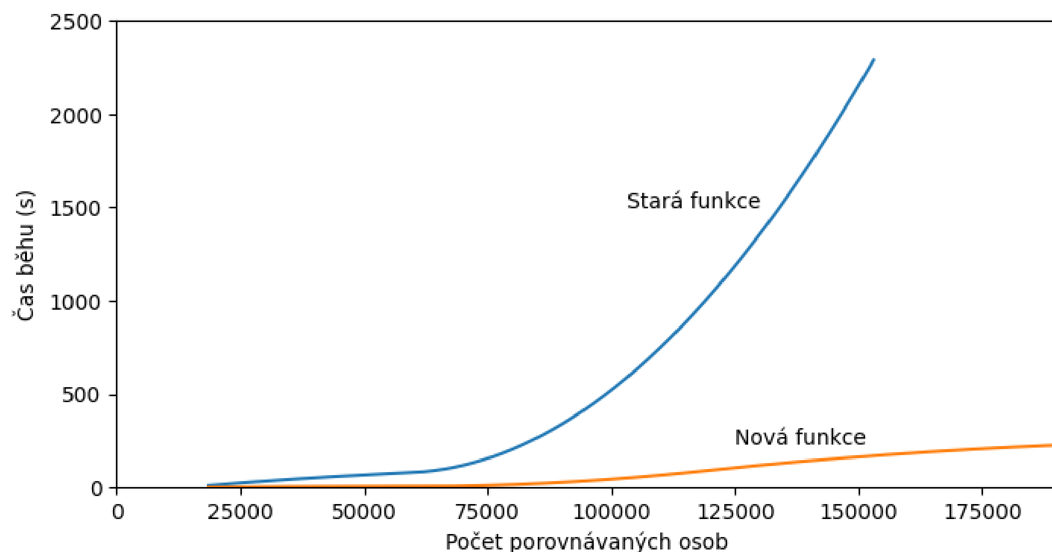
Rozdělil jsem tedy tuto funkci na dvě nové. Jedna pouze prochází vztahy konkrétní osoby a vrací seznam všech dalších osob, se kterými zpracovávaná osoba má jeden ze vztahů uvedených v seznamu vztahů, který funkce bere jako vstupní argument. Druhá funkce pak jen zavolá první funkci nejdříve pro osobu, jejíž příbuzné chceme nalézt, čímž zjistíme příbuzné vzdálené o 1 generaci. Následně ji volá pro tyto příbuzné k nalezení příbuzných vzdálených 2 generace atd.

Výsledky s využitím mojí implementace dopadly následovně:

Počet záznamů	Počet osob	Čas (s)	Čas (%)
3 000	19 000	1,8	0,8
6 000	36 000	5,1	0,84
15 000	83 000	18,5	0,9
30 000	153 000	170	2,7
45 000	189 000	225	2,5

Tabulka 3.5: Růst času `get_family_relatives` v závislosti na počtu osob po optimalizaci funkce

Rychlost růstu byla výrazně omezena a pro N_{30000} se mi povedlo dosáhnout zrychlení 1245 %, což je velice dobrý výsledek.



Obrázek 3.7: Čas staré a nové implementace funkce `get_family_relatives` v závislosti na počtu osob

Počet záznamů	Počet osob	Čas (s)	Čas (%)
3 000	19 000	11	4,8
6 000	36 000	45	8
15 000	83 000	240	10,5
30 000	153 000	2 288	27,4

Tabulka 3.6: Růst času ve funkci `get_family_relatives` v závislosti na počtu osob

Z grafu na obrázku 3.7 můžeme vidět, nejen že nová funkce je o poznání rychlejší, ale zároveň i to, že rychlost růstu času nové funkce se pro vyšší N snižuje. Díky omezení hloubky prohledávání rodokmenu totiž existuje teoretické maximum pro počet vyhledaných osob a bez ohledu na počet generací rodiny, které jsou v databázi uloženy, funkce toto maximum nemůže překročit.

3.11 Redukce výpočtů Levenshteinovy vzdálenosti

Program pracuje s různými prahy pro shodu a neshodu dvou údajů. Například ve funkci `basic_check_of_two_person` se za práh shody považuje přesáhnutí hodnoty 0,9. V případě porovnávání slov je skóre podobnosti rovno normalizované Levenshteinově vzdálenosti (viz 2.3.3 sekce Slova). Problém tohoto přístupu je, že už jen vypočítat Levenshteinovu vzdálenost je náročné a následně se musí ještě normalizovat a porovnat s nějakou prahovou hodnotou, typicky ve formátu čísla s plovoucí řádovou čárkou. O Levenshteinově vzdálenosti víme, že jde o počet operací přidání, odebrání nebo změny znaku, což nám umožňuje stanovit odhad minimální vzdálenosti dvou slov na rozdíl jejich délek – tento rozdíl udává počet znaků, které se musí odebrat/přidat, i kdyby zbytek slov byl stejný. Jestliže pak nejdelší jméno v databázi má například 20 znaků, můžeme pro práh 0,9 s jistotou říci, že pokud rozdíl délek dvou slov je alespoň 2, Levenshteinova vzdálenost mezi těmito slovy je také minimálně 2 a normalizovaná Levenshteinova vzdálenost je pak maximálně 0,9, což není více než 0,9 a výsledkem je tedy neshoda. Výhodou této metody je, že ve spoustě případů vůbec Levenshteinovu vzdálenost nemusíme počítat. Navíc zjištění délky řetězce v Pythonu má konstantní složitost (a pro normalizaci vzdálenosti se musí zjišťovat tak jak tak) a porovnání celých čísel je zhruba o 10 % rychlejší než porovnaní čísel desetinných. Vyhodnocení takového výrazu je pak 7-10krát rychlejší než výpočet normalizované vzdálenosti v závislosti na délce porovnávaných slov. Samozřejmě ne vždy výpočet na této podmínce skončí a v takových případech vzdálenost konvenčním způsobem spočítat potřeba je. Průměrné zrychlení výpočtu normalizované Levenshteinovy vzdálenosti tedy z testů nad aktuální databází činí pouze 290 %. Celá aplikace se pak zrychlila o 7,5 %.

3.12 Další drobné změny

V této sekci si ukážeme další drobné optimalizační změny, které na programu byly provedeny pro zlepšení jeho výkonu. Na rozdíl od ostatních změn, kterým byla věnována celá sekce, tyto změny měly za následek jen minimální zlepšení výkonu, nebo sice urychlily nějakou funkci výrazně, ale program této funkci nevěnoval dost času, aby se tato změna výrazně projevila i na době běhu celého programu.

3.12.1 Selektivní import funkcí

V programu byl typicky využíván přístup importování celého modulu a následné přistupování k potřebným funkcím z tohoto modulu přes referenci tečkovou notací. To ovšem přidává extra krok, který interpret musí provést při vyhledávání této funkce ve jmenném prostoru, tedy extra potřebný čas. Selektivním importem konkrétních funkcí do modulu, ze kterého je chceme volat, se tomuto extra kroku, který by interpret při každém volání musel provést, můžeme vyhnout. Za normálních okolností je takováto změna prakticky nepozorovatelná, ale v případě funkce `control_if_compare`, která je společně s `check_lifetime_overlap` (sekce 3.7) nejvíce volanou funkcí celého programu, už rozdíl viditelný je.

Smyslem funkce `control_if_compare` je rozhodnout, zda má dojít k detailnímu porovnání dvou osob. To provede nejdříve voláním funkce `check_lifetime_overlap` pro tyto dvě osoby pro kontrolu překryvu jejich dob života a pokud tento test vyjde, zavolá ještě funkci `basic_check_of_two_person`, která provede porovnání jmen, příjmení, bydlišť a názvu povolání. Následně vrací hodnotu `True`, pokud obě volané funkce vrátily `True`, jinak vrací `False`. I přesto, že funkce samotná tedy vlastně skoro nic nedělá, a přesto v ní program při testu s 15 000 záznamy trávil 355 sekund. Obě volané funkce byly odkazovány přes názvy modulů, ve kterých se nacházely. Po přidání importů pro tyto funkce, změnou na jejich přímé volání a provedení dalšího testu, byl čas strávený ve funkci `control_if_compare` pouze 320 sekund. Zavedením selektivního importu funkcí tak bylo docíleno zhruba 11% zrychlení u funkce `control_if_compare`.

3.12.2 Zkratové vyhodnocování

Python, jakožto i spousta dalších jazyků, vyhodnocuje logické výrazy zkratově. To znamená, že pokud bychom měli například tyto dva logické výrazy, které jsou významově ekvivalentní

$$\texttt{t}_1 \ \texttt{AND} \ \texttt{t}_2 \ \texttt{OR} \ \texttt{t}_3 \tag{3.1}$$

$$\texttt{t}_3 \ \texttt{OR} \ \texttt{t}_2 \ \texttt{AND} \ \texttt{t}_1 \tag{3.2}$$

a porovnávali bychom postup jejich vyhodnocení pro následující hodnoty

Proměnná	Hodnota
t_1	`False`
t_2	`True`
t_3	`False`

pro výraz 3.1 by se vyhodnotila pouze proměnná t_1, protože její hodnota je `False` a tím i výsledek celého výrazu, nehledě na to, jakých hodnot nabývají proměnné t_2 a t_3.

Pro výraz 3.2 by se ale musela nejdříve vyhodnotit proměnná t_3, která má hodnotu `False` a v operaci `OR` nám sama nic neřekne, dále proměnná t_2 s hodnotou `True`, díky které je celá levá strana výrazu pravdivá a následně i proměnná t_1, aby celý výraz mohl být vyhodnocen.

To znamená, že pouze nevhodným pořadím provádění operací se může doba vyhodnocení tohoto výrazu až ztrojnásobit.

Výraz shodný s výrazem 3.2 používá funkce `check_date_cross`, která porovnává podobnost dvou dat na základě prohození pořadí dne a měsíce v zápisu jednoho z nich. Hodnota t_3 je rovna shodě dnů, hodnota t_2 je rovna shodě měsíců a hodnota t_1 je pak shoda roku.

Teoreticky, pokud jsou data v průběhu roku zastoupena rovnoměrně, je pravděpodobnost pravdivosti výrazů t_2 a t_3 zhruba $\frac{12}{365}$. Kvůli pořadí operací `OR` a `AND` se vždy budou muset provést alespoň 2 vyhodnocení a to konkrétně v

$$\frac{12}{365} + \left(\frac{353}{365}\right)^2 = 96.8\,\%$$

případů. Ve zbylých 3,2 % se budou provádět všechna tři vyhodnocení. To znamená, že se průměrně provede 2,03 vyhodnocení. Pokud bychom tento výpočet přepsali do tvaru výrazu 3.1, nastaly by i situace, kdy se provede pouze jedno vyhodnocení – když hodnota t_1 bude `False`, což z testů na aktuálních datech z databáze nastává v 97 % případů. Pravděpodobnost provedení dvou vyhodnocení by pak byla

$$0.03 \cdot \frac{12}{365} = 0.1\,\%$$

a ve zbylých 2,9 % by se provedla vyhodnocení tři, což by znamenalo průměrně provedených 1,06 vyhodnocení, neboli zrychlení této funkce o 92 %.

3.12.3 Sekvenční procházení hašovací tabulky

V programu se hojně místo obyčejných seznamů používají množiny pro zabránění vkládání duplicitních záznamů a zvýšení rychlosti vyhledávání. Množiny fungují na bázi hašovací tabulky, takže vyhledávání v nich má konstantní časovou složitost. Přestože třída `set` disponuje funkcí `intersection`, jejíž průměrná složitost je $O(\min(len(s), len(t)))$, kde s a t jsou množiny [6], na několika místech v programu se používá klasický sekvenční průchod se dvěma vnořenými cykly, kdy se každý prvek z první množiny porovnává s každým prvkem z druhé množiny. Podle počtu prvků v těchto množinách využití funkce `intersection` třídy `set` urychlí nalezení společných prvků dvou množin až o 1250 %.

3.12.4 Vektorizace aplikace vah

Při porovnávání dvou osob se nejdříve vypočítají dílčí skóre podobností pro všechny údaje o těchto osobách a následně se provede skalární součin s předem definovaným vektorem vah. Pro tento výpočet se využívaly slovníky a každá položka slovníku s dalšími skóre podobnosti se vynásobila s korespondující vahou ve slovníku s váhami. K oběma těmto slovníkům jsem vytvořil alternativu v podobě seznamu respektive pole z knihovny `numpy` a aplikaci vah na toto pole jsem vektorizoval pomocí funkcí z knihovny `numba`.

Kapitola 4

Testování

V této kapitole si ukážeme jaký vliv měly provedené změny na rychlost výpočtu programu a zároveň jak tyto změny ovlivnily úspěšnost propojovaní na testovací sadě. Dále v několika bodech uvedu, čeho všeho jsem si při práci na tomto projektu všiml a šlo by toho využít při dalším vývoji.

Všechny výsledky a udané hodnoty v této práci se vážou k tomuto referenčnímu stroji:

Komponenta	**Specifikace**
Procesor	Intel Core i7-7700 @ 3.60 GHz
RAM	32GB DDR4
Operační systém	Windows 10 verze 22H2
Python verze	3.11.9

Tabulka 4.1: Specifikace referenčního stroje pro testování optimalizace programu

4.1 Počáteční výsledky

Abychom mohli výsledky změn porovnat, ukážeme si nejdříve výsledky všech pozorovaných statistik, jak vypadli na začátku práce před provedením jakýchkoliv úprav. Význam a výpočet těchto statistik je uveden v sekcích 2.3.8 a 2.3.9.

Kategorie	Počet porovnání
TP	15 672
FP	2 080
TN	1 171 568
FN	5 904
Celkem	1 195 224

Tabulka 4.2: Úvodní rozložení kategorií výsledků porovnání

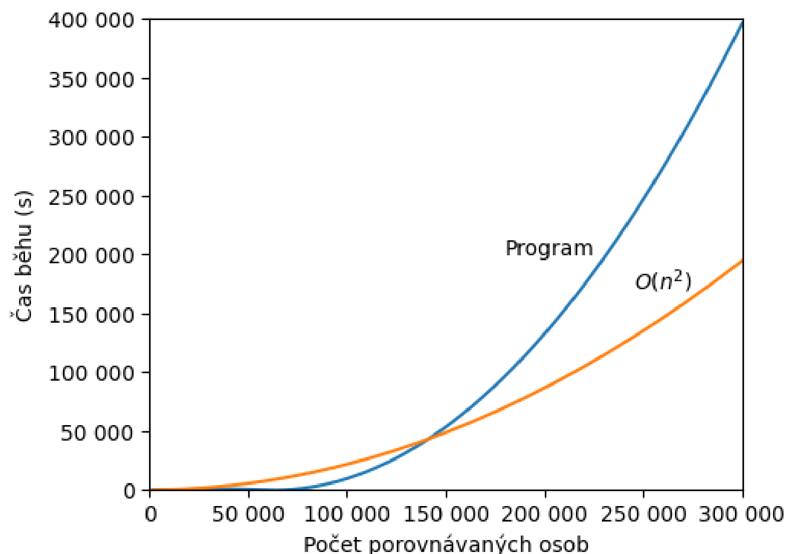
Metrika	Hodnota metriky (%)
Precision	88,28
Recall	72,63
F_1	79,69
BA	86,23

Tabulka 4.3: Úvodní hodnoty statistických metrik

	Celkem	P	M	S	SM
Různá ID	3 058	943	348	1 132	635
Počet uzlů	5 742	943	136	3 474	1 189

Tabulka 4.4: Úvodní hodnoty statistik grafové databáze

Kromě těchto statistik je pro testování podstatný ještě graf s časovým průběhem programu v závislosti na počtu porovnávaných osob. Ten na začátku mé práce vypadal následovně:



Obrázek 4.1: Doba běhu programu v závislosti na počtu zpracovaných osob v porovnání s kvadratickou složitostí

4.2 Průběžné výsledky

Jestli se v průběhu přepracovávání programu někde něco nepokazilo, bylo průběžně testováno po každé změně uvedené v kapitole 3. Samozřejmě, nějaká chyba se někde vždy vyskytne, ale vzhledem k povaze prováděných změn až na standardizaci jmen byla po opravě všech chyb úspěšnost propojování stejná jak před implementací této změny. Co se týče dílčích výsledků jednotlivých změn z hlediska snížení časové náročnosti, ty jsou uvedeny v sekcích těchto změn v kapitole 3.

4.2.1 Vliv standardizace

Jak už bylo řečeno, jediná provedená změna, která se dotkla úspěšnosti propojování, byla zavedení standardizace jmen osob (3.9). Porovnání statistik před a po standardizaci jsou uvedena v následujících tabulkách.

Kategorie	Před	Po
TP	15 672	15 553
FP	2 080	2 240
TN	1 171 568	611 022
FN	5 904	7 624
Celkem	1 195 224	636 439

Tabulka 4.5: Hodnoty kategorií binární klasifikace před a po zavedení standardizace jmen

Metrika	Před (%)	Po (%)
Precision	88,28	87,41
Recall	72,63	67,11
F_1	79,69	75,92
BA	86,23	83,37

Tabulka 4.6: Hodnoty statistických metrik před a po zavedení standardizace jmen

Jak z těchto tabulek můžeme vidět, zavedení standardizace jmen drasticky snížilo počet TN případů, protože velké množství porovnání se díky standardizaci vůbec neprovádí. Zároveň se zvedly FP případy, což je zapříčiněno tím, že se do porovnání dostaly osoby s jiným uvedeným jménem, ale stejným normalizovaným jménem, které jsou ovšem různé. Program nemá žádná omezení na minimální počet údajů, které pro sloučení musí o osobách znát. Pokud tedy známe dvě osoby pouze jménem, mají spolu 100% shodu a sloučí se. Zvýšení případů FN je pravděpodobně zaviněno tím, že seznam standardizovaných jmen není dokonalý a dvě osoby stejného jména po standardizaci mohou mít jména různá. Propad TP je pak zapříčiněn kombinací předešlých důvodů.

Při analýze FN porovnání jsem zjistil, že skóre velkého množství těchto porovnání velmi těsně nedosahovalo prahu potenciální shody, tehdy stanoveného na 0,7. Snížil jsem tedy tento práh na 0,6, což přineslo následující výsledky:

Kategorie	Před	Po, práh 0,7	Po, práh 0,6
TP	15 672	15 553	15 661
FP	2 080	2 240	2 229
TN	1 171 568	611 022	459 513
FN	5 904	7 624	4 168
Celkem	1 195 224	632 983	481 571

Tabulka 4.7: Hodnoty kategorií binární klasifikace po snížení prahu potenciální shody

Metrika	Před (%)	Po, práh 0,7 (%)	Po, práh 0,6 (%)
Precision	88,28	87,41	87,54
Recall	72,63	67,11	78,98
F_1	79,69	75,92	83,04
BA	86,23	83,37	89,25

Tabulka 4.8: Hodnoty statistických metrik po snížení prahu potenciální shody

Snížením prahu potenciální shody na 0,6 znovu ubylo TN porovnání, protože místo zamítnutí shody ve spoustě případech vznikl vztah potenciální shody. Totéž platí i pro FN případy, které jsem touto změnou chtěl potlačit, což se tedy povedlo až nad očekávání dobře. Dále můžeme vidět nárůst TP díky potenciálním shodám, které se při dalších zpracovaných osobách přehodnotí na shodu úplnou.

Co se týče statistik, až na přesnost, která se snížila pouze nepatrně, jsou všechny výrazně lepší a to nejen proti původním po standardizaci, ale dokonce i proti těm před zavedením standardizace.

4.3 Konečné výsledky

Většina provedených úprav v programu mířila na snížení jeho časové náročnosti, i proto statistiky z výsledků porovnávání nedosahují velkých změn. Přesnost porovnávání se nepatrně zhoršila, ale na druhou stranu můžeme vidět mírné zlepšení co se týče citlivosti, F-míry a vyvážené přesnosti.

Kategorie	Počet porovnání
TP	15 672
FP	2 225
TN	458 222
FN	4 180
Celkem	480 299

Tabulka 4.9: Konečné hodnoty kategorií binární klasifikace

Metrika	Hodnota (%)
Precision	87,56
Recall	78,94
F_1	83,03
BA	89,23

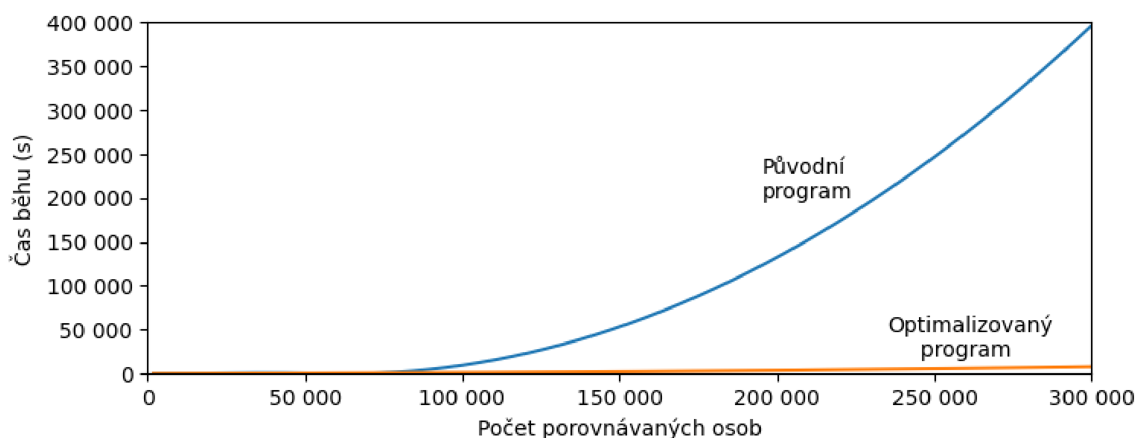
Tabulka 4.10: Konečné hodnoty statistických metrik

Dalšími měřenými hodnotami jsou kategorie definované v práci D. Pojezdála [5]. Ani tady jsme se žádných drastických změn nedočkali. Ve výsledné databázi je o 156 méně uzlů a méně osob se nám povedlo klasifikovat dokonale – tedy buď se nějaký z jejich záznamů sloučil s cizí osobou, nebo se nějaké záznamy této osoby nesloučily spolu, což ale neznamená, že mezi nimi nemůže existovat vztah potenciální shody a nesloučí se někdy v budoucnu.

	Celkem	P	M	S	SM
Různá ID	3058	903	399	1074	682
Počet uzlů	5608	903	170	3323	1212

Tabulka 4.11: Úvodní hodnoty statistik grafové databáze

Co je ovšem hlavním výsledkem této práce, jsou výsledky v rámci úspory času při porovnávání. Časový průběh funkce na začátku práce je uveden v sekci Profilování na grafu 3.1. Po všech úpravách z kapitoly 3 vypadá srovnání původního a aktuálního programu takto:



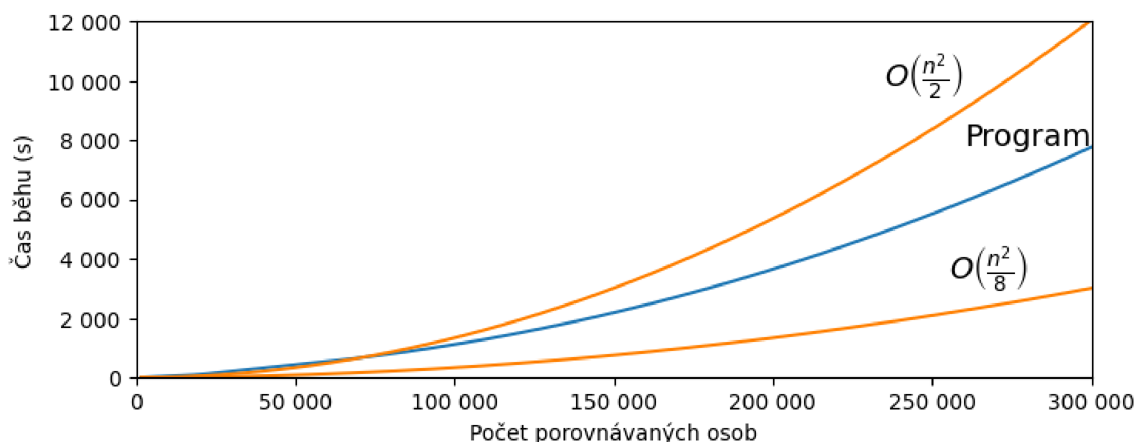
Obrázek 4.2: Srovnání délky běhu programu před a po všech optimalizacích

Program při zpracování celé databáze dosáhl zrychlení 5000 % a rychlost růstu nutného času se zásadně zpomalila. Porovnání dob běhu pro různé počty zpracovávaných osob jsou vidět v následující tabulce:

Počet osob	Původní čas (s)	Nový čas (s)
1 800	7	6
9 000	N/A	40
18 600	200	92
30 000	700	218
83 000	2 570	840
150 000	53 300	2 184
300 000	396 000	7 762

Tabulka 4.12: Časový podíl funkcí na běhu programu při spuštění nad 15 000 záznamy

Samotný průběh konečné verze programu:



Obrázek 4.3: Délka běhu optimalizovaného programu

Nejzávažnější chyby byly odstraněny a jak bylo předpovězeno v kapitole 3, program se pohybuje v teoretických hranicích mezi složitostmi $\frac{n^2}{8}$ a $\frac{n^2}{2}$.

4.4 Další vývoj

Před samotným závěrem této práce bych chtěl ještě vypíchnout nějaké části programu, u kterých jsem si všiml, že by je bylo možné vylepšit, nebo alespoň by stálo za to je v budoucnu otestovat, protože v současném stavu to nebylo možné.

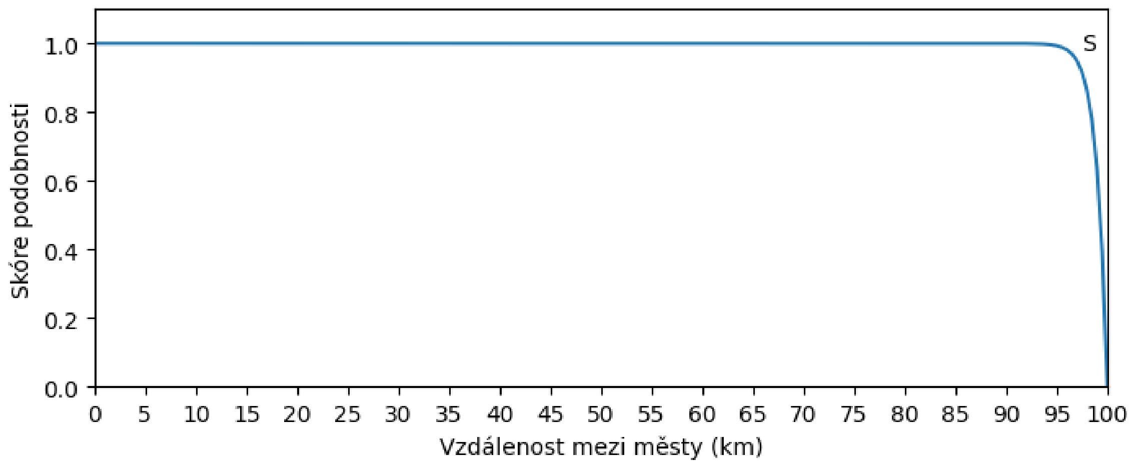
4.4.1 Skóre podobnosti mezi městy

Jak je detailněji popsáno v sekci 2.3.3, skóre podobnosti dvou měst se vypočítává přes vzdálenost těchto měst pomocí vzorce

$$S = \begin{cases} 1 - \frac{1}{e^{100-d}} & \text{pro } d < 100 \\ 0 & \text{jinak} \end{cases}$$

Tento vzorec má definovat funkci pro generování skóre podobnosti takovým způsobem, že města, která jsou blízko u sebe, budou mít skóre vysoké a toto skóre pak s narůstající vzdáleností bude klesat k nule, až pro všechny města nad 100 km od sebe bude skóre vždy rovno 0.

Problém této funkce ale je, že její graf vypadá takto:



Obrázek 4.4: Skóre podobnosti měst vzhledem ke vzdálenosti těchto měst od sebe

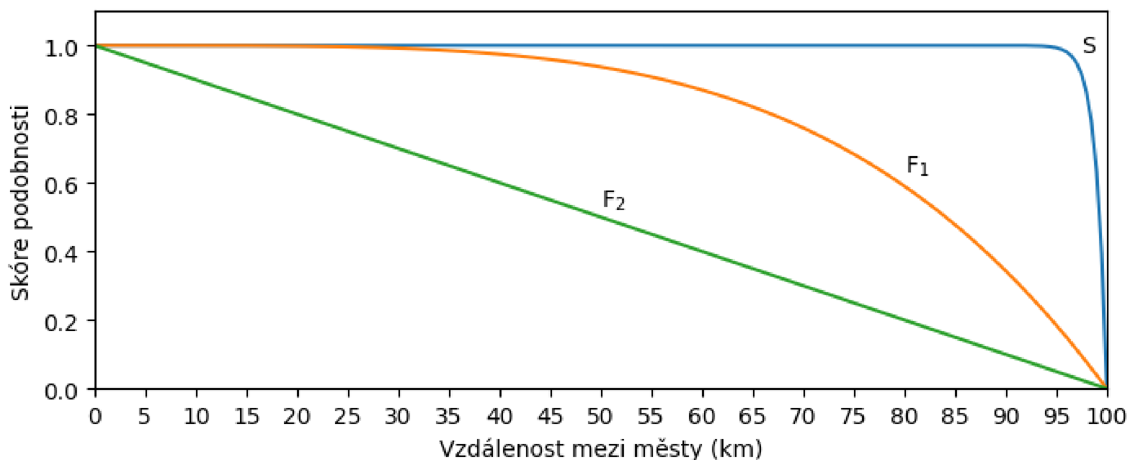
Neboli všechna města do vzdálenosti 95 kilometrů od sebe mají skóre podobnosti 95 % nebo vyšší, přičemž pro města pod 60 km jsou vždy vyhodnocena jako zcela shodná, protože jejich skóre je tak blízké hodnotě 1, že takto malý rozdíl už 64bitová hodnota s plovoucí řádovou čárkou neumí rozlišit a skóre zaokrouhlí na 1.

Kromě toho se skóre podobnosti měst vypočítává jako největší z normalizované Levenshteinovy vzdálenosti a právě skóre vzdálenosti, takže prakticky pro všechna města do 98 kilometrů od sebe se normalizovaná Levenshteinova vzdálenost vypočítává zcela zbytečně. Dalším faktem je, že právě kvůli Levenshteinově vzdálenosti a výběrem nejvyššího skóre je toto omezení, že města mohou být maximálně 100 kilometrů od sebe zcela irelevantní, protože Nová ves někde na Moravě, sice bude mít vzdáleností skóre podobnosti rovno nule s Novou vsí na Ústecku, ale kvůli přístupu „největší z“, dostane 100% shodu díky skóre z normalizované Levenshteinovy vzdálenosti.

Navrhoval bych tedy vzorec změnit na nepřímou úměru, nebo rovnou lineární průběh, aby výpočet této hodnoty dával nějaký smysl a případně zavést i opatření, které by vyřešilo problém se stanovením výsledného skóre jakožto „největší z“. Tyto návrhy s aktuálním stavem testovací databáze nelze otestovat, protože se v ní nenachází dostatek osob bydlících v dostatečném počtu různých městech v různých vzdálenostech od sebe, aby jakákoliv změna na této funkci měla měřitelný dopad na výsledky propojování.

$$F_1 = \begin{cases} 1 - \left(\frac{d}{100}\right)^4 & \text{pro } d < 100 \\ 0 & \text{jinak} \end{cases}$$

$$F_2 = \begin{cases} 1 - \frac{d}{100} & \text{pro } d < 100 \\ 0 & \text{jinak} \end{cases}$$



Obrázek 4.5: Navrhované funkce F_1 a F_2 pro výpočet skóre podobnosti přes vzdálenost mezi městy

4.4.2 Minimální údaje pro sloučení osob

Program stále vyhodnocuje dost osob za False Positive, tedy provede jejich sloučení, přestože jsou to rozdílné osoby. Z analýzy těchto FP případů vyplynulo, že velmi mnoho z nich je sloučeno právě z důvodu nedostatku informací. Tedy v situaci, kdy jediný, nebo jeden z mála, údajů, který se mezi osobami porovnává, je jméno. Po standardizaci se tento problém ještě zhoršil, protože najednou existuje více osob se stejným jménem. To by šlo omezit zavedením extra váhy pro údaje podle jejich četnosti. Tedy pokud například 5 % osob má jméno „Jiří“, skóre podobnosti pro jméno „Jiří“ bude mít menší váhu, než kdyby se tyto osoby jmenovaly „Ahmed“, což je jméno, na které v Evropě narazíme opravdu jen zřídka. Druhá možnost, která by tento problém prakticky úplně eliminovala, by bylo zavedení limitu na počet údajů, které osoby musí mít, aby došlo k jejich sloučení nebo alespoň úpravou váhy jména a způsobu výpočtu finálního skóre, aby jméno samotné nestačilo pro překročení prahu shody.

4.4.3 Porovnání dat vzdáleností ke dnešnímu dni

V sekci 2.3.3 o porovnávání dat jsme si představili tzv. „třetí způsob“, kdy se spočítá kolik by osobám bylo právě dnes a pokud podíl těchto hodnot spadá do intervalu $(0.9, 1.\overline{1})$, určí se tato hodnota jako skóre podobnosti mezi daty jejich narození. Tento způsob výpočtu skóre mi přijde zcela absurdní, už jen z důvodu, že se výpočet vztahuje ke dnešnímu dni, takže program zítra bude dávat jiné výsledky než jaké dává dnes. Další zvláštní vlastností tohoto postupu je, že čím jsou osoby dále v minulosti, tím větší volnost v rozdílu jejich dat narození jim dáváme. Kdybychom například porovnávali osobu narozenou v roce 1945 s osobou z roku 1955, jejich skóre podobnosti by tímto výpočtem vycházelo

$$\frac{2024 - 1955}{2024 - 1945} = \frac{69}{79} = 0.87$$

Tedy méně než 0,9 a shoda by to nebyla. Pokud by se tyto osoby narodily o 200 let dříve, už by rozdíl 10 let mezi jejich dobami narození problém nedělal.

$$\frac{2024 - 1755}{2024 - 1745} = \frac{269}{279} = 0.96$$

Naopak, osoby z 18. století by od sebe mohly být narozeny dokonce i 30 let a stále by takto vypočítané skóre podobnosti přesahovalo hodnotu 0,9. Lidé v minulosti si možná úplně přesně nepamatovali svůj věk, ale jak bylo zjištěno při propojování záznamů ze sčítání lidu z 19. století v Americe, 99,9 % osob si svůj věk pamatuje s odchylkou maximálně jednoho roku a 100 % osob svůj věk uvedlo s odchylkou do dvou let [1]. Předpokládat, že se někdo ve svém věku splete o několik desítek let akorát snižuje úspěšnost propojování.

4.4.4 Operace „<“ mezi osobami

Výsledkem porovnávání konkrétní osoby je seznam uspořádaných dvojic, kde první prvek v těchto dvojicích je skóre podobnosti mezi osobami a na druhém místě je osoba, se kterou porovnání tohoto skóre nabývá. Z tohoto seznamu se následně vybere jedna z dvojic, jejíž skóre má nejvyšší hodnotu (předpokládejme, že více než je práh shody). To ale neznamená, že se stejným skóre podobnosti v tomto seznamu neexistuje více dvojic a tedy více osob. I přesto se ale vybere pouze jedna, se kterou se porovnávaná osoba sloučí a zbylé se zahodí, ačkoli mají úplně stejné právo na to být sloučeny. O tom, která osoba z těch s nejvyšším skóre se sloučí, rozhoduje vestavěná funkce `max`, která pro uspořádané n-tice porovná hodnotu prvního prvku a pokud se rovnají, porovná hodnotu dalšího, tak dlouho, dokud jsou shodné. Protože na druhém místě v těchto dvojicích je objekt typu `Person`, musel pro něj být zaveden zcela nesmyslný operátor $<$ definovaný jako $\forall p_1, p_2 : p_1 < p_2$, kde p_1 a p_2 jsou osoby.

To má za následek, že výsledky porovnávání jsou závislé na pořadí porovnávání a přestože průměrně by se to na statistikách nemělo projevovat, aktuální testovací sada je dostatečně omezená, aby kvůli tomuto pouhá změna pořadí výběru osoby z osob s nejvyšším skóre podobnosti posunula hodnoty statistických metrik až o 3 procentní body.

4.4.5 Ručně zvolené hodnoty

Ať už jde o práh shody a práh potenciální shody, které jdou měnit přímo z konfiguračního souboru, porovnávací váhy pro různé atributy osob na výpočet celkového skóre podobnosti, nebo spoustu magických konstant na různých místech v kódu, všechny tyto hodnoty byly voleny ručně a přestože i ručně kdysi mohly být zvoleny blízko ideální hodnotě, jejich význam se v průběhu práce na tomto projektu časem mění a co byla ideální hodnota někdy v minulosti, nemusí být ideální hodnota i v současnosti. Hodilo by se tedy vyvinout systém, který bude minimálně měřit vliv těchto hodnot, případně je i automaticky upravovat pro zlepšení výsledků porovnávání.

4.4.6 Další standardizace

Standardizací jmen jsme dosáhli celkem úspěchu jak z hlediska úspěšnosti propojování, tak co se týče zrychlení doby běhu programu. Standardizovaná jsou ale pouze jména osob, dalším krokem by mohlo být zavedení standardních forem i pro příjmení, případně města, názvy ulic, nebo jiných textových řetězců, jako jsou tituly a názvy povolání.

Kapitola 5

Závěr

Hlavním cílem této práce byla optimalizace časové náročnosti programu pro propojování osob z archivních genealogických záznamů pocházejících například z matrik, urbářů nebo lánových rejstříků. V první části byl popsán vědní obor genealogie a dále byly vysvětleny základní genealogické pojmy, se kterými se v rámci práce na tomto programu můžeme setkat. Bylo vysvětleno, jak se mezi sebou záznamy porovnávají, podle jakých kritérií je program spojuje a jaké metriky používá k hodnocení svých výsledků. Dále byly představeny různé způsoby optimalizace programového kódu, od pouhého využití lepších knihovních funkcí až po celkové přepsání větších celků programu výhodnějším způsobem nebo využitím kompilovaného modulu v rámci interpretovaného jazyka, jakým je právě Python, ve kterém je program napsán. Pro všechny tyto změny byly provedeny řádné testy a po srovnání testů se závěry předešlých prací vyplynulo, že následkem těchto provedených změn bylo urychlení programu o více než 5 000 %. Testována byla i úspěšnost propojování pro zjištění vlivu optimalizačních změn na funkcionalitu programu, a přestože hlavním cílem této práce nebylo zvýšit úspěšnost propojování, díky zavedení standardizace jmen se balancovaná přesnost navýšila z 86,23 % na 89,25 %, F-míra ze 79,69 % na 83,04 %, citlivost ze 72,63 % na 78,98 % a to vše jen za cenu 0,74 procentního bodu u přesnosti, která z 88,28 % spadla na 87,54 %.

Během práce se tedy prokázalo, že efektivní optimalizace může vést k významnému zlepšení výkonu programu, což je zcela zásadní pro výpočetní úlohy vyžadující velké množství zdrojů, jako je například právě porovnávání osob.

Literatura

[1] FEIGENBAUM, J. J. *Automated Census Record Linking: A Machine Learning Approach.* 2016.

[2] FELLEGI, I. P. a SUNTER, A. B. A theory for record linkage. *Journal of the American Statistical Association*, 1969, sv. 64, č. 328, s. 1183–1210. ISSN 0162-1459.

[3] LEACH, P. J.; SALZ, R. a MEALLING, M. H. *A Universally Unique IDentifier (UUID) URN Namespace* RFC 4122. RFC Editor, červenec 2005. Dostupné z: https://doi.org/10.17487/RFC4122.

[4] MARHEFKA, A. *Generování rodokmenů z matričních záznamů.* Brno, CZ, 2022. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Dostupné z: https://www.fit.vut.cz/study/thesis/24660/.

[5] POJEZDÁL, D. *Generování rodokmenů z archičních záznamů.* Brno, CZ, 2023. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Dostupné z: https://www.fit.vut.cz/study/thesis/144926/.

[6] PYTHON WIKI. *TimeComplexity — Python Wiki, The Free Encyclopedia.* 2024. Dostupné z: https://wiki.python.org/moin/TimeComplexity. [Online; vid. 2024-5-4].

[7] SHESKIN, D. J. *Handbook of parametric and nonparametric statistical procedures.* 5. vyd. Chapman & Hall/CRC, 2020. ISBN 9781439858011.

[8] TUŠIMOVÁ, L. *Generování rodokmenů z matričních záznamů.* Brno, CZ, 2020. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Dostupné z: https://www.fit.vut.cz/study/thesis/22818/.

[9] WEISSTEIN, ERIC W. . *Great Circle. From MathWorld–A Wolfram Web Resource.* 2023. Dostupné z: https://mathworld.wolfram.com/GreatCircle.html. [Online; vid. 2024-5-3].

[10] WIKIPEDIA CONTRIBUTORS. *Levenshtein distance — Wikipedia, The Free Encyclopedia.* 2024. Dostupné z: https://en.wikipedia.org/wiki/Levenshtein_distance. [Online; vid. 2024-5-3].