Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky

Bakalářská práce

Automatická těžba dat z veřejných zdrojů pro vizualizaci

Plzeň 2023 Jakub Šmrha

ZÁPADOČESKÁ UNIVERZITA V PLZNI
Fakulta aplikovaných věd
Akademický rok: 2022/2023

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení:	**Jakub ŠMRHA**
Osobní číslo:	**A19B0206P**
Studijní program:	**B0613A140015 Informatika a výpočetní technika**
Specializace:	**Informatika**
Téma práce:	**Automatická těžba dat z veřejných zdrojů pro vizualizaci**
Zadávající katedra:	**Katedra informatiky a výpočetní techniky**

Zásady pro vypracování

1. Seznamte se s nástroji a postupy pro zpracování částečně strukturovaných dat popisujících historické události.
2. Seznamte se s dostupnými datovými zdroji a jejich strukturou.
3. Seznamte se s možnostmi vizualizace dat a nástroji, které pro ni lze využít.
4. Navrhněte konfigurovatelný nástroj pro získávání dat z veřejných zdrojů jako Wikipedia nebo DBpedia.
5. Navržený nástroj implementujte.
6. Implementaci otestujte zejména s důrazem na přehlednost a použitelnost.

Rozsah bakalářské práce: **doporuč. 30 s. původního textu**
Rozsah grafických prací: **dle potřeby**
Forma zpracování bakalářské práce: **tištěná/elektronická**

Seznam doporučené literatury:

Dodá vedoucí bakalářské práce

Vedoucí bakalářské práce: **Ing. Richard Lipka, Ph.D.**
Katedra informatiky a výpočetní techniky

Datum zadání bakalářské práce: **3. října 2022**
Termín odevzdání bakalářské práce: **4. května 2023**

L.S.

Doc. Ing. Miloš Železný, Ph.D.
děkan

Doc. Ing. Přemysl Brada, MSc., Ph.D.
vedoucí katedry

V Plzni dne 25. října 2022

Prohlášení

Prohlašuji, že jsem bakalářskou práci vypracoval samostatně a výhradně s použitím citovaných pramenů.

V Plzni dne 1. května 2023

Jakub Šmrha

Abstract

In this work, a tool for creating timelines is implemented. The tool simplifies data mining and can semi-automatically create a timeline from the extracted data. At the same time, it can export the extracted data to the format required by the timeline visualization tool. This thesis describes the processing procedures of semi-structured data, available data sources, their structure and visualization possibilities.

Abstrakt

V této práci je implementován nástroj pro tvorbu časových linií. Nástroj zjednodušuje těžbu dat a dokáže z nich poloautomaticky vytvořit časovou linii. Zároveň dokáže exportovat vytěžená data do formátu, který požaduje nástroj pro vizualizaci časové osy. Tato práce líčí postupy zpracování částečně strukturovaných dat, dostupné datové zdroje, jejich strukturu a možnosti vizualizace.

Poděkování

Rád bych poděkoval vedoucímu bakalářské práce Ing. Richardu Lipkovi Ph.D. za trpělivost a cenné rady. Dále bych chtěl poděkovat své přítelkyni Veronice Fedasové za grafiku a otestování nástroje a také všem ostatním testerům Bc. Martinu Jakubaškovi a Ing. Janu Šmrhovi.

Obsah

1 Úvod

Zajímá-li nás datum vzniku a zániku Svaté říše římské včetně dat vlády jejich císařů a chtěli bychom je graficky zobrazit na časové ose, museli bychom procházet různé zdroje, mezi které patří např. infoboxy Wikipedie, u každého zdroje zjišťovat, zdali subjekt odpovídá našim parametrům, a nakonec najít správný vizualizační nástroj, který nám všechna tato data zobrazí na časové linii. Rozhodně se jedná o časově náročnou aktivitu a lze ji alespoň částečně automatizovat. Z tohoto důvodu vznikla tato bakalářská práce a společně s ní nástroj pro automatickou těžbu dat z veřejných zdrojů pro vizualizaci.

Nástroj má za úkol vyhledat a zpracovat data z veřejných zdrojů na základě různých parametrů. Tato data jsou uložena v tzv. Resource Description Framework (RDF) formátu, o kterém se v této práci dozvíme více v sekci 2.3.2. Nástroj je schopen tato data převést do časové linie za pomocí uživatelského vstupu. Data jsou uživateli prezentována tak, aby byla co nejvíce přehledná a čitelná.

Mezi hlavní účely tohoto nástroje patří urychlení procesu těžby dat, umožnění volné konfigurace a možnost vizualizace vztahů mezi souvisejícími prvky na časové ose. Zaměřujeme se především na přehlednost a použitelnost nástroje.

2 Typy datových zdrojů

Máme na výběr ze 3 typů zdrojů, z jakých můžeme „těžit“ (zpracovávat) data:

- Strukturovaná data,
- nestrukturovaná data,
- částečně strukturovaná (polostrukturovaná) data.

Strukturovaná a nestrukturovaná data si v následujících sekcích krátce představíme včetně jejich výhod a nevýhod. Částečně strukturovaná data budou popsána detailněji – těmito daty se budeme podrobněji zabývat, jelikož s nimi nástroj úzce pracuje, a budou opět představeny výhody, nevýhody a důvody, proč byl tento zdroj vybrán. Seznámíme se také s definicemi různých pojmů, které se hojně v této práci používají.

2.1 Strukturovaná data

Jedná se o data, která mají předem definovanou strukturu – dodržují tabulkovou strukturu datových modelů. Příkladem strukturovaných dat je databáze.

2.1.1 Databáze

Databáze je strukturovaná a trvalá sbírka informací o některých aspektech světa organizovaná a uložená způsobem, který usnadňuje efektivní vyhledávání a úpravy. Struktura databáze je určena abstraktním datovým modelem[12].

Datový model je abstraktní model, který organizuje prvky dat, vysvětluje relace mezi nimi a jejich vlastnostmi entit z reálného života. Datový model může např. specifikovat, že datový prvek reprezentující člověka je složen z jiných prvků, které zase definují křestní jméno, příjmení, výšku a rodné číslo[19]. Primárně je to tato struktura, která odlišuje databázi od datového souboru. Nejčastěji používaný datový model je relační, který reprezentuje data jako sadu tabulek. Databáze obsahuje instance entit a vztahy mezi nimi[12].

2.1.2 Výhody a nevýhody

Výhodou těchto dat je, že předem známe strukturu a víme, co můžeme od těchto dat očekávat. Podle různých pravidel také víme, že se tam daná informace buď musí, nebo nemusí vyskytovat. Příkladem je primární klíč v relačních databázích – obvykle se jedná o unikátní číslo, které identifikuje danou položku (řádek) v databázi. Další výhodou je jednoduché pochopení struktury – tj. není obtížné poznat, s čím pracujeme. Tvorba linie by byla také triviální, nicméně z nevýhod vyplyne, proč nelze tohoto tak jednoduše dosáhnout.

Hlavní nevýhodou je nedostatek veřejně dostupných dat. Mějme jako příklad panovníka. Jelikož se jedná o člověka, očekáváme v datovém modelu položky typu jméno, příjmení, apod., a jedná se také o panovníka, proto mimo jiné očekáváme, v jaké dynastii či jakých dynastiích panoval. Takovéto informace je obtížné reprezentovat jednotnou tabulkovou strukturou. Je zcela možné, že na webu nalezneme strukturovaná data o panovnících, nicméně se nebude jednat o obecný model, který by vyhověl zpracování dat, protože účelem nástroje není těžit pouze v okruhu panovníků.

2.2 Nestrukturovaná data

Nebo-li nestrukturované informace, jsou informace, které nemají předdefinovanou strukturu, anebo nejsou organizované předem definovaným způsobem[20].

Příkladem nestrukturovaných dat je text nějakého článku či text této bakalářské práce. Zpracování takovýchto dat je velice náročné a vyžaduje znalosti a zkušenosti v oboru zpracování přirozeného jazyka, proto si následující pojmy popíšeme pouze v krátkosti.

2.2.1 Text mining

Text mining je objevování a získávání zajímavých, netriviálních znalostí a smysluplných vzorů[6] z volného nebo nestrukturovaného textu. To zahrnuje vše od vyhledávání informací (tj. vyhledávání dokumentů nebo webových stránek) do klasifikace textu a shlukování k extrakci entity a vztahu[9]. Text mining je označován jako nejvýznamnější metoda analýzy a zpracování nestrukturovaných dat. Dolování textu je spojení nástroje těžby dat, vyhledávání informací, statistiky, strojového učení apod., a lze jej tedy považovat za multidisciplinární obor. Zvládne také Zpracování přirozeného jazyka (NLP). K tomu patří několik základních kroků metody:

1. Nestrukturovaná data jsou sesbírána z různých zdrojů, jako jsou webové stránky a soubory v různých formátech.

2. Poté jsou detekovány anomálie a odstraněny z vytěžených dat pomocí různých nástrojů pro těžbu dat.

3. Následně je extrahovaná informace konvertována do strukturovaného formátu.

4. Vzory ve vytěžených datech jsou analyzovány pomocí tzv. manažerského informačního systému (MIS).

5. Všechna hodnotná a relevantní informace je uložena v databázi[6].

2.2.2 NLP

Zpracování přirozeného jazyka (NLP) je pokus extrahovat plnější význam reprezentace z volného textu. Dá se to vyjádřit jako vyřešení toho, kdo co komu udělal, kdy, kde, jak a proč. NLP obvykle používá lingvistických pojmů, jako jsou slovní druhy (podstatné jméno, sloveso, přídavné jméno atd.) a gramatickou strukturu (buď reprezentovanou jako fráze jako podstatná jmenná fráze nebo předložkové spojení, anebo závislostní vztahy jako podmět nebo předmět). Musí se vypořádat s tím, čemu předchozí podstatné jméno, zájmeno nebo jiné zpětně odkazující fráze odpovídají, a také s nejednoznačnostmi (jednak slov, jednak gramatickou strukturou, např. co se daným slovem popř. předložkovým spojení upravuje)[9].

V nedávné době se rozvinul trend okolo umělé inteligence, především tzv. „chatbot“ ChatGPT od společnosti OpenAI, který byl trénován na LLM GPT[11]. Tento chatbot je schopný rychle zodpovědět prakticky jakýkoliv dotaz. Dokáže tedy odpovědět na dotaz ohledně tvorby linie včetně požadovaného formátu, nicméně níže v sekci 2.2.3 se dozvíme jeho nedostatky.

2.2.3 Výhody a nevýhody

Výhodou těchto dat je ten, že v nich dokážeme nalézt prakticky všechny informace. Jelikož už existuje poměrně přesný model, který dokáže zodpovědět až překvapivě komplexní dotazy, mohl by v tom uživatel vyhledávat a vytvořit prakticky jakoukoliv linii.

Nevýhodou je poměrně vysoká pravděpodobnost, že umělá inteligence vrátí nesprávné výsledky a záleží na uživateli, aby tato data validoval a případně opravil. Lze také tato data přetvořit do určitého formátu, který specifikujeme,

nicméně v tomto ohledu očekáváme jisté technické znalosti od uživatele. Z výsledků testů v příloze A uvidíme nedostatky tohoto chatbota.

2.3 Částečně strukturovaná data

Takováto data leží na pomezí strukturovaných a nestrukturovaných dat. Nemají pevně danou strukturu, nicméně se řídí určitými pravidly, která by se měla dodržovat.

Jedná se o formu strukturovaných dat, která nedodržuje tabulkovou strukturu datových modelů (strukturovaných dat) asociovaných s relačními databázemi či jinými formami tabulek. Nicméně obsahuje značky nebo jiné popisovače pro oddělení sémantických prvků a vynucení hierarchii záznamů a polí v datech[18].

2.3.1 Báze znalostí

Předpokládejme množinu entit I (např. *Praha*), množinu binárních relací P (např. *nacházející se v*) a množinu literálů L (řetězce nebo čísla). Bázi znalostí K modelujeme jako *množinu tvrzení* $r(s, o)$, také nazývány jako *fakta*, s podmětem $s \in I$, relací (přísudkem) $r \in P$ a předmětem $o \in I \cup L$[10].

2.3.2 RDF

Resource Description Framework (RDF) je datový model pro reprezentaci informací ve World Wide Webu. RDF specifikace obsahuje množinu rezervovaných IRI a RDF Schema (RDFS) slovník, který má předdefinovanou sémantiku. Tento slovník je designován k popisu speciálních relací mezi zdroji, jako jsou např. typování a dědičnost tříd a vlastností[1]. V tomto slovníku se nachází speciální IRI, mezi které např. patří:

rdfs:subClassOf používá se k vyjádření, že všechny instance jedné třídy jsou instancemi jiné

rdfs:subPropertyOf používá se k vyjádření, že všechny RDF zdroje (*resource*) související s jednou vlastností (*property*) jsou také spojeny s jinou

rdfs:range používá se k vyjádření, že hodnoty vlastnosti jsou instancemi jedné nebo více tříd

rdfs:domain používá se k vyjádření, že jakýkoli RDF zdroj, který má danou vlastnost, je instancí jedné nebo více tříd

rdfs:type používá se k vyjádření, že RDF zdroj je instancí nějaké třídy[4]

RDF je standardní model pro výměnu dat na webu. RDF má funkce, které usnadňují slučování dat, i když se základní schémata liší, a konkrétně podporuje vývoj schémat v čase, aniž by bylo nutné měnit všechny spotřebitele dat[16]. RDF byl původně navržený jako datový model pro metadata. Používá se jako obecná metoda pro popis a výměnu grafových dat.

Hlavní myšlenkou RDF je ke zdroji přiřadit výraz ve tvaru *„podmět – přísudek (vlastnost) – předmět“* (*„subjekt – predikát – objekt“*). RDF poskytuje různé syntaktické notace a formáty pro serializaci dat, přičemž Terse RDF Triple Language (TTL) je v současnosti nejpoužívanější notací[17]. Nejčastěji používané souborové formáty jsou později popsány a doprovozeny krátkými ukázkami v sekci 2.3.4 na straně 11.

RDF trojice

Předpokládejme nekonečnou množinu URI referencí U, nekonečnou množinu prázdných uzlů B (anonymní zdroje) a nekonečnou množinu literálů L. Pak trojice $<s, p, o> \in (U \cup B) \times U \times (U \cup L \cup B)$ je *RDF trojice*, kde s je subjekt (podmět), p je predikát (přísudek, vlastnost) a o je objekt (předmět)[8].

RDF graf (dataset)

Jedná se o konečnou množinu RDF trojic[8]. Graf se dá představit jako sada objektů, které jsou propojeny. RDF graf je orientovaný graf, tzn. spojení mezi objekty mají směr.

SPARQL

SPARQL Protocol and RDF Query Language (SPARQL) je v podstatě dotazovací jazyk pro porovnávání grafů. Dotaz SPARQL je ve tvaru $H \leftarrow B$. Tělo dotazu B je komplexní vzorový výraz nad RDF grafem, který může zahrnovat trojice RDF s proměnnými, konjunkcemi, disjunkcemi, volitelnými částmi a omezeními nad hodnotami proměnných. Hlava H dotazu je výraz, který naznačuje, jak sestavit odpověď na dotaz[1].

Vyhodnocení dotazu Q oproti RDF grafu G se provádí ve dvou krocích:

1. Tělo Q je porovnáno s G, aby se získala sada vazeb pro proměnné,
2. pomocí informací v hlavě Q jsou tyto vazby zpracovávány pomocí klasických relačních operátorů (`projection`, `distinct` atd.) k vytvoření odpovědi na dotaz. Tato odpověď může mít různé formy, např. odpověď ano/ne, tabulku hodnot nebo nový graf RDF.[1].

Konkrétní definice SPARQL dotazu je následující: Předpokládejme existenci nekonečné množiny proměnných V disjunktní s U. Graf SPARQL je definován rekurzivně následovně:

1. N-tice $(U \cup V) \times (U \cup V) \times (U \cup V)$ je vzor grafu (*vzor trojice*).
2. Pokud P_1 a P_2 jsou vzory grafu, pak výrazy (P_1 AND P_2), (P_1 OPT P_2) a (P_1 UNION P_2) jsou vzory grafu.
3. Pokud P je vzor grafu a R je vestavěná podmínka SPARQL, pak výraz (P FILTER R) je vzor grafu.

Vestavěná podmínka SPARQL je Booleovská kombinace termínů konstruovaná pomocí ekvivalence ($=$) mezi prvky $(U \cup V)$ a konstantou a unárním predikátem `bound`$(\cdot)$ u deklarovaných proměnných[1].

SPARQL dotaz a výrazy použité v definici lze dále přesněji specifikovat, nicméně pro účely této bakalářské práce to je již nadbytečné. Nejdůležitějším faktem zůstává, že se jedná o dotazovací jazyk nad grafem RDF pomocí trojic, ve kterých lze použít proměnné. Jazyk syntaxí silně připomíná jazyk SQL, používají se také stejné výrazy, jako jsou např. `DISTINCT`, `UNION` či `ORDER BY`. V následujících odstavcích si představíme jednoduché příklady dotazů SPARQL. Později si představíme komplexnější příklady, konkrétně v implementační sekci, tj. sekci 6.2 na straně 29.

Dotaz SPARQL obsahuje v tomto pořadí:

1. Deklarace prefixů pro zkracování IRI
2. Definice datové sady, která uvádí, jaké RDF grafy jsou dotazovány
3. Klauzule výsledku určující, jaké informace mají být z dotazu vráceny
4. Vzor dotazu určující, na co se má dotazovat v základní datové sadě
5. Modifikátory dotazů, dělení, řazení a jiné přeskupování výsledků dotazů

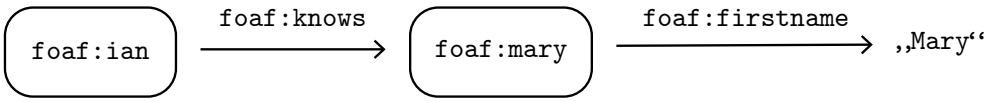
Příklad dotazu: *„Najděte všechny lidi v souboru Friend of a Friend (FOAF), kteří mají jména a e-mailové adresy. Vraťte všechna data o těchto lidech.“*

```
1 PREFIX foaf: <http://xmlns.com/foaf/0.1/>
2 SELECT *
3 WHERE {
4    ?person foaf:name ?name .
5    ?person foaf:mbox ?email .
6 }
```

Zdrojový kód 2.1: Příklad SPARQL dotazu

V této ukázce můžeme vidět jeden deklarovaný prefix FOAF, který je zkratkou IRI `http://xmlns.com/foaf/0.1/`. Datová sada není zde uvedena, jelikož se jedná o vymyšlený příklad. Později bude představena jednoduchá sada včetně grafického zobrazení pro jednodušší čtení. Dále můžeme vidět klauzuli `SELECT *`, která říká, že chceme vrátit všechny informace z dotazu. Konkrétně nám dotaz vrátí proměnné `?person`, `?name` a `?email` – tj. URI člověka se jménem a e-mailem. Modifikátor dotazu zde žádný není.

Nyní si představíme jednoduchou bázi znalostí zapsanou ve formátu Terse RDF Triple Language (TTL).



Obrázek 2.1: Příklad dvou trojic

```
foaf:ian foaf:knows foaf:mary.
foaf:mary foaf:firstName "Mary".
```

Zdrojový kód 2.2: Příklad dvou trojic

Na obrázku 2.1 lze vidět příklad jednoduchého grafu znalostí složeného ze dvou trojic. Tento obrázek je doprovázen zdrojovým kódem 2.2, který byl zapsán v TTL formátu. Tento graf lze přečíst následovně: „Podmět `foaf:ian` má vlastnost `foaf:knows` s hodnotou `foaf:mary`. Podmět `foaf:mary` má vlastnost `foaf:firstName` s hodnotou „`Mary`“. Jinak řečeno, `ian` zná `mary`, která má vlastnost – její křestní jméno je „`Mary`“.

RDF rozšiřuje spojovací strukturu webu o použití IRI k pojmenování vztahu mezi věcmi. Jak bylo řečeno, zkratky, jako je např. FOAF, se obvykle definují na začátku SPARQL dotazu v části *prefix*. Lze si představit místo FOAF samotné IRI:

```
<http://xmlns.com/foaf/0.1/ian>
  <http://xmlns.com/foaf/0.1/knows>
  <http://xmlns.com/foaf/0.1/mary>.
<http://xmlns.com/foaf/0.1/mary>
  <http://xmlns.com/foaf/0.1/firstName>
  "Mary".
```

Zdrojový kód 2.3: Rozšířený SPARQL dotaz k obrázku 2.1

Nicméně ze zdrojového kódu 2.3 můžeme vidět, že data nejsou již tak čitelná a zároveň se IRI opakují na různých místech.

2.3.3 Ontologie

Ontologie jsou konceptuální modely, které zachycují a explicitně uvádějí slovní zásobu používanou v sémantických aplikacích, čímž zaručují komunikaci bez nejednoznačností[3]. Ontologie zahrnuje reprezentaci, formální pojmenování a definici kategorií, vlastností a vztahů mezi pojmy, daty a entitami.

Slovo ontologie pochází z řeckého *ontos* (bytost) + *logos* (slovo). Slovo bylo zavedeno do filozofie v devatenáctém století německými filozofy, aby odlišili studium bytí jako takového od studia různých druhů bytostí v přírodních vědách. V počítačové vědě byly ontologie přijaty v umělé inteligenci, aby se usnadnilo sdílení a opětovné použití znalostí[3].

2.3.4 Souborové formáty

Mezi hojně užívané souborové formáty patří:

- `JSON`
- `XML`
- `CSV`
- `TTL`

JavaScript Object Notation (JSON) je minimalistický čitelný formát pro strukturování dat. Jedná se o nejrozšířenější a nejvíce používaný formát k přenosu dat mezi serverem a webovou aplikací. Objekt JSON obsahuje data ve formě klíč/hodnota. Klíče jsou řetězce a hodnoty jsou typy JSON (např. řetězec, číslo, pole, ...). Klíče a hodnoty jsou odděleny dvojtečkou. Každá položka je oddělena čárkou. Složené závorky „{...}“ představují objekt JSON. Každý JSON soubor začíná a končí složenými závorkami.

Extensible Markup Language (XML) je značkovací jazyk podobný jazyku HTML, ale bez předdefinovaných značek k použití. Místo toho se definují vlastní značky navržené pro speciální potřeby. Jedná se o účinný způsob ukládání dat ve formátu, který lze ukládat, vyhledávat a sdílet. Každý XML soubor musí mít kořenový prvek.

Comma-separated Values (CSV) je formát souboru s oddělovačem, který k oddělení hodnot používá čárku. Každý řádek souboru je datovým záznamem a každý záznam se skládá z polí oddělených čárkami. Obvykle se používá pro tabulková data.

Terse RDF Triple Language (TTL) je rozšířením tzv. N–Triples syntaxe, které přebírá nejužitečnější a nejvhodnější věci přidané z tzv. Notace 3, přičemž je ponechává v modelu RDF. Soubor v TTL formátu umožňuje zapsat RDF graf v kompaktní textové formě. Skládá se ze sekvencí direktiv, tvrzení generujících RDF trojice, anebo prázdných řádek. Umožňuje přidat komentáře pomocí znaku „`#`“ před daným textem. Jednoduché RDF trojice jsou sekvence termínů (podmět, přísudek, předmět) oddělené mezerou a zakončené tečkou „.“ po každé trojici. Toto koresponduje k N–Triples syntaxi[2].

2.3.5 Výhody a nevýhody

Mezi hlavní výhody částečně strukturovaných dat rozhodně patří flexibilní strukturování dat. Přidání datové položky je triviální – do báze znalostí stačí pouze přidat RDF trojici např. ve formátu uvedeném ve zdrojovém kódu 2.2 na straně 10. Tato data lze také snadno dotazovat pomocí vlastního dotazovacího jazyka SPARQL, který funguje na stejné bázi jako SQL. Pomocí RDFS lze také *částečně* zajistit validitu dat. Umožňuje např. možnost omezit data na určitý datový typ, nicméně záleží na aplikaci, jak s tímto faktem naloží, jelikož se pouze jedná o metadata. Správnost dat uvnitř báze znalostí musí tedy zajistit daná aplikace.

Nevýhodou jsou potenciálně chybějící data. U člověka např. očekáváme datum narození či jeho pohlaví, nicméně tyto položky mohou chybět, anebo namísto datového typu datum obdržíme řetězec. Důvod byl objasněn v minulém odstavci, kdy za správnost dat zodpovídá daná aplikace.

3 Datové zdroje

V této sekci si představíme volně dostupné báze znalostí, s kterými by mohl nástroj pracovat. Seznámíme se pouze s takovými bázemi znalostí, které považujeme za potenciálně užitečné pro náš nástroj. Popíšeme rozsáhlost jednotlivých datových zdrojů a jejich užitečnost vzhledem k této bakalářské práci. Rozsáhlost dat byla spočtena pomocí SPARQL dotazu přibližně k datu odevzdání této bakalářské práce. Jelikož některé báze znalostí čerpají z Wikipedie, zmíníme také zdroj Wikipedia, ze kterého rovněž můžeme čerpat, ač se jedná o nestrukturovaná data. Popíšeme také tzv. infoboxy, které se ve Wikipedii nachází. V sekci 5.3 na straně 25 bude odůvodněn výběr zvolené báze znalostí.

3.1 YAGO

Yet Another Great Ontology (YAGO) je rozsáhlá znalostní báze s obecnými znalostmi o lidech, městech, zemích, filmech a organizacích[21]. YAGO je jedním z prvních akademických projektů, které automaticky budují bázi znalostí. Hlavní myšlenkou YAGO bylo sklízet informace o entitách z infoboxů a kategorií Wikipedie a kombinovat tato data s ontologickou páteří odvozenou z tříd ve WordNetu. Znalostní báze byla tvořena tak, že každý fakt prošel následujícím procesem:

1. Filtrování
2. Kontrola omezení
3. Deduplikace

Tento proces zvýšil kvalitu konečné báze znalostí na 95%. Tato přesnost byla ručně ověřena[13]. YAGO obsahuje více než *477 miliónů trojic*.

3.2 DBpedia

DBpedia je projekt, jehož cílem je extrahovat strukturovaný obsah z informací vytvořených v projektu Wikipedie. Umožňuje dotazovat vztahy a vlastnosti zdrojů, včetně odkazů na další související datové sady, k čímž DBpedie používá RDF záznamy[14].

DBpedia obsahuje více než *1 miliardu trojic*, díky čemuž se stala velice atraktivní a populární bází znalostí. DBpedia začala jako malý projekt, který se později vyvinul ve velký projekt řízený komunitou. DBpedia obsahuje přibližně 20 jazykových kapitol, které se zabývají vylepšením a extrakcí dat z Wikipedie v různých jazycích. Kapitoly jsou součástí vedení DBpedie, kteří převzali odpovědnost za přispění k její infrastruktuře[14].

Počet trojic je spočten napříč všemi jazyky – jedná se o globální počet trojic, které DBpedia obsahuje. DBpedii lze dotazovat přes SPARQL endpoint `https://dbpedia.org/sparql`. Konkrétně lze dotazovat URL

`https://dbpedia.org/sparql?default-graph-uri=http://dbpedia.org&query=`

do které můžeme vložit vlastní SPARQL dotaz. Do této URL lze vložit další různá metadata, mezi které patří např. formát a limit výsledků. Tato URL také poskytuje webové rozhraní, ve kterém lze zkoušet vlastní SPARQL dotazy.

RDF formát dat je intuitivní a lidsky čitelný. Uvedeme zde krátký příklad, který následně ve zkratce popíšeme.

```
<rdf:RDF>
  <rdf:Description rdf:about="DBR/Charles_IV,_Holy_Roman_Emperor">
    <dbp:predecessor rdf:resource="DBR/John_of_Bohemia"/>
    <dbp:successor rdf:resource="DBE/Sigismund,_Holy_Roman_Emperor"/>
    <dbp:successor rdf:resource="DBR/Wenceslaus_IV_of_Bohemia"/>
    <dbp:title rdf:resource="DBR/King_of_Bohemia"/>
    <dbp:title rdf:resource="DBR/King_of_the_Romans"/>
    <rdfs:label xml:lang="cs">Karel IV.</rdfs:label>
  </rdf:Description>
</rdf:RDF>
```

Zdrojový kód 3.1: Příklad RDF DBpedia

Zdrojový kód 3.1 je v XML formátu. Kořenovým prvkem tohoto souboru je `<rdf:RDF>`. Uvnitř tohoto prvku můžeme vidět jeden element `<rdf:Description ...>`, který slouží pro popis jedné entity. Konkrétně se jedná o entitu uvnitř atributu `rdf:about="..."`, tedy hodnotu

`"DBR/Charles_IV,_Holy_Roman_Emperor"`

Hodnotou tohoto atributu je URI ukazující na subjekt trojic uvnitř elementu `<rdf:Description ...>`. V tomto zdrojovém kódu jsme použili vlastní zkratku „DBR“, jinak by přetékal text. Tato zkratka ukazuje na IRI `dbpedia.org/resource`.

Uvnitř elementu `<rdf:Description ...>` můžeme vidět 6 elementů – `dbp:predecessor`, 2× `dbp:successor`, 2× `dbp:title` a `rdfs:label`. Všechny tyto elementy jsou RDF trojice, kdy název těchto elementů reprezentuje predikát (`dbp:predecessor`, ...). Hodnotou atributu `rdf:resource="..."` je RDF zdroj a hodnotou elementu je literál, přičemž obě hodnoty reprezentují subjekt. Subjekt může být pouze jedna z těchto hodnot, kombinace hodnoty atributu a elementu není validní. V tomto příkladu je tedy 6 RDF trojic, přičemž u 5 prvních trojic, tj. na řádcích 3–7, je subjektem RDF zdroj (`rdf:resource="..."`) a u poslední trojice literál `Karel IV.` na řádku 8.

3.3 Wikidata

Wikidata je bezplatná, kolaborativní, vícejazyčná sekundární databáze, shromažďující strukturovaná data za účelem poskytování podpory pro Wikipedii, Wikimedia Commons, dalším wiki hnutím Wikimedia a všem na světě[5].

Wikidata obsahuje více než *14 miliard trojic*. Stejně jako u DBpedie, jedná se o globální počet trojic. Wikidata je centrální úložiště, ke kterému mají přístup ostatní wiki projekty, mezi které třeba patří wiki spravované nadací Wikimedia Foundation. Obsah načítaný dynamicky z Wikidat nemusí být udržován v každém jednotlivém wiki projektu. Ve Wikidata lze například centralizovat statistiky, data, místa a další běžná data[5].

Úložiště Wikidata se skládá hlavně z položek, z nichž každá má štítek, popis a libovolný počet aliasů. Položky jsou jednoznačně identifikovány písmenem „Q“ následované číslem. Tvrzení popisují podrobné charakteristiky položky a sestávají z vlastnosti a hodnoty. Vlastnosti ve Wikidatech jsou jednoznačně identifikovány písmenem „P“ následované číslem[5].

3.4 Wikipedia

Wikipedie je jednou z největších encyklopedií na světě. Extrahování a integrace strukturovaných dat z Wikipedie do znalostních grafů může přinést velké výhody mnoha aplikacím. DBpedia ukázala úspěch a dopad takové strategie, ale používá pouze infoboxy. Kromě těchto infoboxů má Wikipedie také miliony vysoce kvalitních tabulek pokrývajících širokou škálu domén. Využití těchto tabulek může potenciálně pomoci přidat nebo udržet znalosti v bázi znalostí aktuální[15].

Jako druhotný zdroj by nástroj mohl využívat dat přímo z Wikipedie. Jak bylo dříve zmíněno, v částečně strukturovaných datech by mohl nastat problém, kdy některá data u entit chybí, anebo nejsou zpracována podle

našich představ. Tato data bychom mohli právě doplnit z Wikipedie.

3.4.1 Infobox

Infobox je tabulka s pevným formátem, která se obvykle přidává do pravého horního rohu článků. Představuje shrnutí nějakého sjednocujícího aspektu, který články sdílejí. Také někdy slouží pro zlepšení navigace k dalším souvisejícím článkům.

```
{{Infobox royalty
| jmeno = Karel IV.
| rok narozeni = 1316
| rok umrti = 1378
}}
```

Zdrojový kód 3.2: Příklad infoboxu

Ze zdrojového kódu 3.2 lze vidět, že strukturou připomíná formát JSON – formát klíč–hodnota. Znak „|“ si můžeme představit jako „,“ a „=“ jako „:“. JSON formát byl popsán v sekci 2.3.4 na straně 11.

Jedná se také o částečně strukturovaná data, tj. řídí se podle určitých pravidel, nicméně se může opět stát, že některá data budou chybět. Tato pravidla nejsou striktně dodržována – spíše fungují jako šablona, podle které se infobox řídí. Některé báze znalostí čerpají právě z dat v infoboxech, nicméně jsou tato data tvořena komunitou a mohou obsahovat chyby.

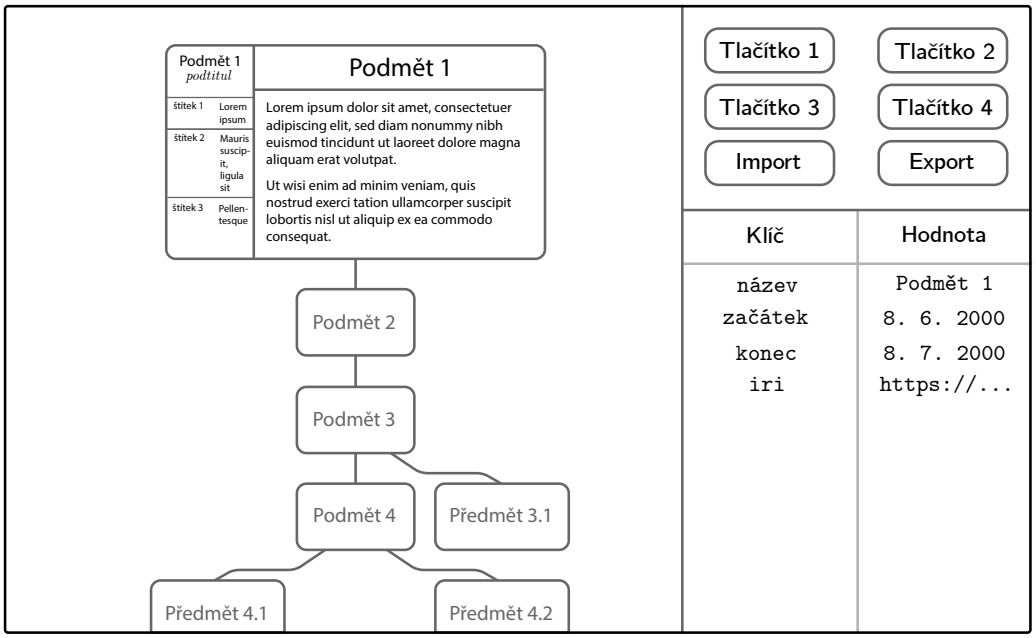
Každý infobox může také obsahovat tzv. podinfoboxy (vnořené infoboxy), např. Karel IV. býval vládcem několika království a měl tedy několik titulů, je tedy třeba doplnit o každém tomto faktu podinfobox.

4 Vizualizace dat

V této kapitole si představíme dva návrhy vizualizace nástroje, nad kterými jsme uvažovali. První návrh bude detailně popsán – odůvodníme umístění a použití jednotlivých grafických prvků. Stejně jako první návrh bude i druhý návrh detailně popsán, nicméně zde také popíšeme vizualizaci na základě textu bakalářské práce Bc. Michala Fialy, bývalého studenta Fakulty aplikovaných věd, který modifikoval verzi nástroje pro zobrazování dat na časové ose diplomové práce Ing. Michala Kacerovského, také bývalého studenta Fakulty aplikovaných věd.

Z hlediska implementace je jeden návrh komplexnější a jeden jednodušší. U obou návrhů bude graficky zobrazena vizualizace doprovozena o popisek jednotlivých částí návrhu. V obou návrzích zmiňujeme tzv. hlavní a vedlejší část vizualizace, kdy v hlavní části se vyskytuje grafické rozhraní, s kterým bude uživatel převážně pracovat, a ve vedlejší části se vyskytují dodatečné informace či nástroje pro manipulaci s linií.

4.1 První návrh



Obrázek 4.1: První návrh

Na obrázku 4.1 můžeme vidět první návrh vizualizace. Ve zkratce nyní popíšeme jednotlivé části, a poté detailněji rozložení jednotlivých částí.

- Počáteční podmět – „bublina“ s názvem „Podmět 1“
  - Na levé straně infobox ze zdroje `wikipedia.org`
  - Obsahem titulku je název podmětu – štítek nebo transformované URI
  - Krátký popis podmětu pod titulkem – jeho abstrakt
- Přísudek – čára mezi „Podmět 1“ — má vlastnost — „Podmět 2“
  - „Podmět 3“ — má vlastnost — „Předmět 3.1“
  - „Podmět 3“ — má vlastnost — „Podmět 4“
- Grafické rozhraní pro modifikaci linie – v pravé horní části okna
  - Zde bude možnost přidávat či odebírat relace (tj. přísudky) s jinými předměty, vytvoření nové linie z daného podmětu, přejmenování podmětu, úprava popisku apod.

- Bližší, zajímavé informace o označeném podmětu – v pravé dolní části okna
    - Mezi tyto informace může patřit např. věk, datum narození či úmrtí (pokud se jedná o osobu), začátek a konec vládnutí (pokud se jedná o panovníka) apod.

Účelem tohoto rozložení je především přehlednost informací a jednoduchá konfigurovatelnost. Jednotlivé části lze graficky zobrazit podle našich představ, hranicí je pouze naše představivost. V tomto rozložení bychom také umožnili posouvat, přibližovat či oddalovat kameru v ortografickém způsobu promítání[1]. Tento způsob zobrazení je uživatelsky přívětivý, jednoduchý a přehledný.

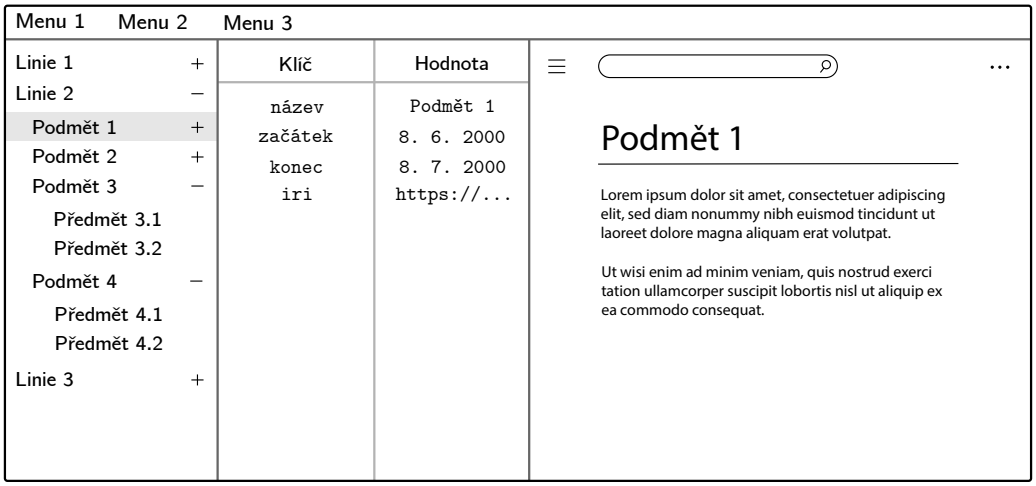
Hlavní částí tohoto zobrazení by bylo samozřejmě zobrazení samotné linie. Jak bylo výše zmíněno, bylo by možné posouvat, přibližovat či oddalovat toto zobrazení. Každý podmět by byl zobrazen jako nějaká „bublina informací“ (na obrázku 4.1 zobrazen jako obdélník).

Aby byl následující popis přehlednější, budeme používat označení *entita* pro jednotlivé části (tj. podměty, předměty) linie. Každou entitu by bylo možné posunout dle libosti. Mohl by nastat případ, kdy se uživateli nelíbí rozmístění entit, proto mu umožníme jejich přemístění. Každou entitu lze označit – jak jednotlivě, tak hromadně, a provést s nimi nějakou akci, jako je např. již zmíněné přemístění, smazání, anebo přidání relace mezi nimi. Pokud bychom označily entity jednotlivě, nebude možné zobrazit upřesňující informace o konkrétní entitě ve vedlejší části zobrazení, proto by se tato část nijak nemodifikovala. Označené entity by byly tučně vyznačeny jako indikátor označení.

Vedlejší částí jsou dodatečné informace či konfigurační možnosti označené entity či označených entit. Tato část je statická a nacházela by se na pravé straně okna. Tyto informace by mohly být také zobrazeny na druhé straně okna, jedná se pouze o preferenci. Přepnutí na toto alternativní zobrazení by bylo možné pomocí nastavení nástroje, které by bylo uloženo v konfiguračním souboru. Zobrazení nahoře nebo dole by z hlediska toku informací o linii, který je shora dolů, nedává smysl. Informace by pouze bránily ve výhledu – museli bychom kameru posunout více dolů anebo nahoru pro zobrazení větší části linie, anebo ji v horším případě oddálit. Nicméně, ani jedno řešení není uživatelsky přívětivé.

[1] Ortografická kamera je taková kamera, ve které se všechny objekty zobrazují ve stejném měřítku.

4.2 Druhý návrh



Obrázek 4.2: Druhý návrh

Na obrázku 4.2 můžeme vidět druhý návrh vizualizace. Opět, stejně jako v předešlém návrhu, prvně popíšeme jednotlivé části, a poté popíšeme jejich rozložení.

- Menu pro manipulaci s linií – v horní liště okna
- Stromová reprezentace linie – v levé části okna
  - Jelikož v této části budeme také zobrazovat předměty, mezi kterými uživatel vybral právě jeden při řešení nejasnosti, kdy si nástroj nevěděl rady, kudy pokračovat, bude mít linie stromovou reprezentaci než-li lineární.
  - V této části bude možné také pracovat s jednotlivými entitami, jako je např. mazání.
- Bližší informace o vybraném podmětu – ve středu okna
  - Tento grafický element bude mít tabulkovou a stromovou reprezentaci. Aby byl nástroj co nejvíce obecný, všechna data budou reprezentována pomocí klíče a hodnoty. Hodnota může být typu pole, anebo také klidně tabulky vnořené uvnitř pole pro uložení relací mezi ostatními entitami.
- Přísudek – vyjádřený v bližších informacích a v stromové struktuře sémanticky relace „Podmět 1“ — má vlastnost — „Podmět 2“

  - „Podmět 3“ — má vlastnost — „Předmět 3.1“
  - „Podmět 3“ — má vlastnost — „Předmět 3.2“
  - „Podmět 3“ — má vlastnost — „Podmět 4“
- Vestavěný prohlížeč – v pravé části okna
  - V této části se budou zobrazovat informace z webové stránky `wikipedia.org`, aby si uživatel mohl ověřit, že má v linii správnou entitu.
  - Uživatel bude moci pomocí menu přepnout na vizualizační část namísto informace z `wikipedia.org`.

Tento návrh se skládá ze dvou částí, přičemž jsme si nyní jednu představili:

- Konfigurační část (tuto jsme právě představili)
- Vizualizační část

Tyto dvě části jsou od sebe odděleny. Vizualizační část tohoto návrhu úzce souvisí s grafickou knihovnou Ing. Michala Kacerovského, kterou později modifikoval v bakalářské práci Bc. Michal Fiala, jehož nástroj budeme v tomto návrhu využívat. Tento nástroj bude popsán v sekci 5.2.1, není tedy nutné zacházet do detailů. O interních záležitostech této knihovny, jako je např. datový formát, se dozvíme v sekci 6.6.4 na straně 23.

Hlavní částí je levá polovina okna, ve které se nachází grafické rozhraní pro tvorbu linií. V této části se bude linie modifikovat. Uživatel také může vidět celou strukturu linie.

Vizualizační nástroj byl napsán v jazycích `Javascript`, `HTML` a `CSS`, vizualizační část se bude tedy zobrazovat v prohlížeči. Nástroj bude toto potenciálně moci zobrazit uvnitř aplikace pomocí vestavěného prohlížeče.

5 Návrh nástroje

V této kapitole představíme obecný návrh nástroje a vybraný návrh vizualizace, který jsme si představili v minulé kapitole, a odůvodníme jeho výběr.

5.1 Obecný návrh

Než-li se budeme věnovat vizualizaci nástroje, potřebujeme zmínit představu o nástroji ze stránky funkčnosti. Představa o tomto nástroji je následující:

Aplikační rozhraní (API) Součástí nástroje bude API, díky kterému bude možnost rozšiřovat nástroj podle našich představ, především ze stránky vizualizační. API bude vysvětleno v samostatné sekci 5.4 na straně 27.

Nezávislá vizualizace Cílem nástroje je *těžit* data a *vizualizovat* je na časové lince. Tyto dvě oblasti budou *odděleny* – tj. jedna část nástroje bude zodpovědná za těžbu dat a druhá část za zpracování těchto dat a jejich následnou vizualizaci.

Uživatelsky přívětivé GUI Tento nástroj cílí na neodborné uživatele. Grafické rozhraní by mělo tedy být uživatelsky přívětivé a, pokud možno, co nejjednodušší na pochopení. Neměli bychom tedy zmiňovat žádné termíny „RDF trojice, RDF graf, . . . “, nicméně se slovům „podmět, přísudek, předmět“ nejspíše nevyhneme. Součástí by tedy měla být nějaká uživatelská příručka.

Poloautomatizovaná těžba Data by se měla těžit poloautomatizovaně. Tím myslíme, že očekáváme, že uživatel bude mít nad těžbou dat určitou kontrolu a, pokud lze proces automatizovat, pak nástroj bude pracovat samostatně. Jako modelovou entitu zvolíme Karla IV., protože se jedná o lidský subjekt (proč jsou preferované bude odůvodněno na začátku následující kapitoly), kdy uživatel bude moci vybrat:

- **časové údaje** – počátek a konec jednotlivých entit (např. datum narození a úmrtí),
- **predikát** – podle tohoto predikátu se bude tvořit linie (např. jeho předchůdce),
- **konkrétní entitu** – při nejednoznačnosti, tj. v situaci, kdy SPARQL dotaz vrátí více než jeden předmět, se nástroj dotáže uživatele, jak se má dále pokračovat.

5.2 Vizualizace

V minulé kapitole v sekci 4 jsme se seznámili s dvěma návrhy vizualizace – jedním komplexnějším na implementaci, nicméně kvalitnějším a diskutabilně přehlednějším návrhem, a jedním jednodušším, nicméně minimalističtějším a pravděpodobně méně složitým na pochopení. V této sekci odůvodníme výběr návrhu.

Vybrali jsme *druhý návrh*, tj. návrh popsaný v sekci 4.2. Máme k tomu hned několik odůvodnění:

Nástroj disponuje API Nástroj je především implementován jako aplikační rozhraní. K dispozici máme rozhraní pro reprezentaci dat, rozhraní pro vizualizaci a rozhraní pro kontrolér vizualizace. Nezáleží tedy na konkrétní implementaci vizualizace, nástroj pracuje se zpětnými voláními do kontroléru. Kvůli tomuto faktu lze napojit na nástroj prakticky jakoukoliv implementaci vizualizace, byť pravděpodobně s mírnými modifikacemi. API je detailněji popsáno v sekci 5.4.

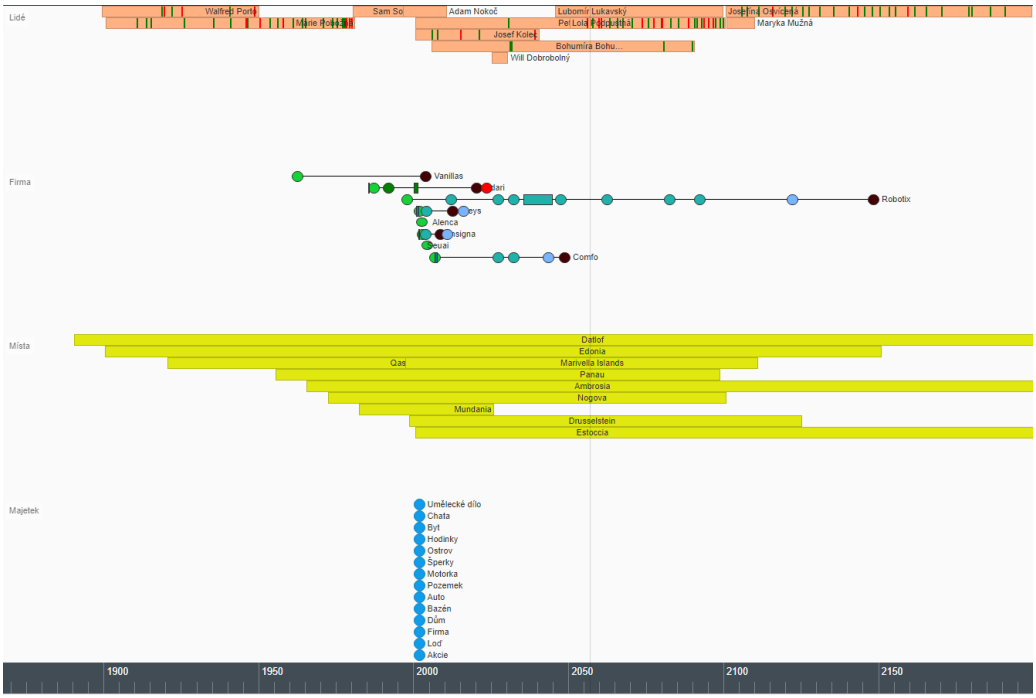
Nenáročnost implementace Jak bylo v druhém návrhu zmíněno, budeme používat již existující aplikaci pro zobrazení dat na časové linii. Integrovat tuto aplikaci do nástroje by mělo být triviální, jelikož se jedná o aplikaci napsanou v jazyce JavaScript a můžeme tedy využít prohlížeče pomocí Webview. V případě prvního návrhu bychom nejspíš museli implementovat vlastní vizualizační komponentu kvůli nedostatku grafických knihoven zaměřených na zobrazení RDF grafu.

Rozdělení vizualizace/konfigurace Tento návrh nám umožňuje rozdělit vizualizační a konfigurační část nástroje. Prvním krokem je sestrojit linii, kterou uživatel uvidí ve stromové struktuře, a druhým krokem je její zobrazení na časové linii.

5.2.1 Vizualizace na časové linii

Jak jsme v sekci 4.2 na straně 21 zmínili, druhý návrh bude využívat vizualizačního nástroje, který nám poskytnul Ing. Michal Kacerovský a modifikoval Bc. Michal Fiala. V této sekci popíšeme, jak tento nástroj funguje. Tento nástroj poskytuje vizualizaci dat na časové linii. Byl implementován v jazykách JavaScript, CSS a HTML, zobrazení bude tedy v prohlížeči.

Na obrázku 5.1 můžeme vidět ukázkovou vizualizaci smyšlených dat. Tato data se týkala lidí, firem a majetku, kdy lidem a firmám se přihodily různé události[7]. Vizualizace je rozdělena na pásy, přičemž na obrázku 5.1 jsou viditelné celkem 4 pásy



Obrázek 5.1: Vizualizace na časové linii *(obrázek převzat z BP)*[7]

- Lidé
- Firma
- Místa
- Majetek

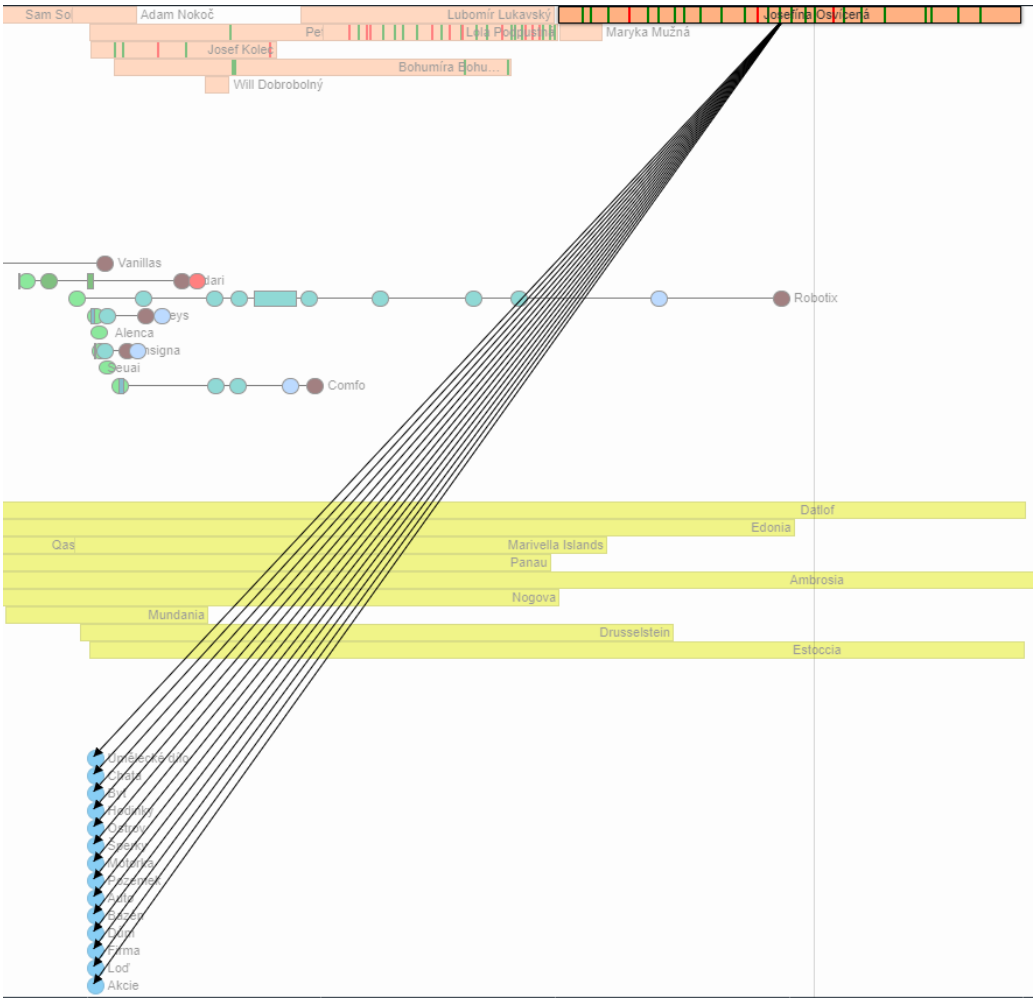
V pásu „Lidé“ můžeme vidět zelené a červené čáry. Tyto čáry reprezentují kladné (zelená barva) a záporné (červená barva) události, které můžeme zobrazit po kliknutí na daného člověka v linii. Tento pás je ideální pro zobrazení obecných dat na časové linii.

Co vyjadřují pásy „Firma“ a „Místa“ je poměrně zřejmé, zaměříme se tedy ještě na poslední pás – „Majetek“. V tomto pásu můžeme vidět body naskládané pod sebou. Tyto body nazýváme tzv. momenty.

Prvek se zobrazí jako moment právě tehdy, když neuvedeme konec trvání – tj. existuje pouze počátek a konec není definován[7].

Na obrázku 5.2 můžeme vidět detailnější zobrazení vztahů mezi prvky. Této vlastnosti můžeme využít pro vyjádření návaznosti mezi dvěma prvky na časové linii.

Součástí tohoto nástroje je také tzv. testovací nástroj, který provede uživatele sekvencí otázek, během kterých bude sledovat uživatelovu aktivitu[7].



Obrázek 5.2: Detailnější zobrazení vztahů *(obrázek převzat z BP)*[7]

Tento nástroj nebude v našem případě použit, nicméně ve vizualizaci se bude ukazovat na pravé straně okna (na obrázcích je to oříznuto, na pravé straně je ve skutečnosti lišta s otázkami a dalšími nástroji).

5.3 Vybraná báze znalostí

Vybrali jsme bázi znalostí *DBpedia*. Důvodem je převážně lepší čitelnost dat a také fakt, že byly zpracovány infoboxy, ve kterých se obvykle vyskytují odkazy na předchůdce či následovníky entity – a to nás v této práci zajímá nejvíce.

Přemýšleli jsme také nad bázemi znalostí YAGO, která je strukturou velice podobná DBpedii, ale není tak obsáhlá, a Wikidata kvůli její úzké spojitosti

s Wikipedií a také její udržovatelnosti vzhledem k datům z Wikipedie. Nicméně, data poskytnutá od DBpedie jsou značně čitelnější – a tedy také se s nimi lépe pracuje, proto jsme raději zvolili DBpedii.

Nyní si ukážeme rozdíly ve zpracování RDF poskytnutých bázemi znalostí DBpedia a Wikidata v XML formátu. Použijeme opět příklad RDF generovaný DBpedií ze sekce 3.2.

```
1 <rdf:RDF>
2   <rdf:Description rdf:about="DBR/Charles_IV,_Holy_Roman_Emperor">
3     <dbp:predecessor rdf:resource="DBR/John_of_Bohemia"/>
4     <dbp:successor rdf:resource="DBE/Sigismund,_Holy_Roman_Emperor" />
5     <dbp:successor rdf:resource="DBR/Wenceslaus_IV_of_Bohemia" />
6     <dbp:title rdf:resource="DBR/King_of_Bohemia" />
7     <dbp:title rdf:resource="DBR/King_of_the_Romans" />
8   </rdf:Description>
9 </rdf:RDF>
```

Zdrojový kód 5.1: RDF DBpedia

```
1 <rdf:RDF>
2   <rdf:Description rdf:about="WD/entity/statement/Q155669-F5...">
3     <ps:P39 rdf:resource="WD/entity/Q95975116"/>
4     <pq:P580 rdf:datatype="W3#dateTime">1346-01-01</pq:P580>
5     <pqv:P580 rdf:resource="WD/value/a7850f8127..."/>
6     <pq:P1365 rdf:resource="WD/entity/Q155167"/>
7     <rdf:type rdf:resource="WD/prop/novalue/P1366"/>
8   </rdf:Description>
9  </rdf:RDF>
```

Zdrojový kód 5.2: RDF Wikidata

Na první pohled můžeme vidět, že oba zdrojové kódy (5.1 a 5.2) začínají kořenovým elementem `<rdf:RDF>`. Následuje element `<rdf:Description rdf:about="...">`, kdy v hodnotě atributu `rdf:about` se vyskytuje u DBpedie:

`https://dbpedia.org/resource/Charles_IV,_Holy_Roman_Emperor`

(ve zdrojovém kódu 5.1 na řádce 2 `"DBR/Charles_IV,_Holy_Roman_Emperor"`) a u Wikidata:

`https://wikidata.org/entity/statement/Q155669-F5...`

(ve zdrojovém kódu 5.2 na řádce 2 `"WD/entity/statement/Q155669-F5..."`). Hodnota tohoto atributu (`rdf:about`) *je subjektem všech RDF trojic* v tomto příkladu.

Dále můžeme vidět jednotlivé predikáty s korespondujícími objekty. Jeden z nich je např. ve zdrojovém kódu 5.1 na řádce 3, který říká, že subjekt má predikát `dbp:predecessor` s objektem `DBR/John_of_Bohemia`. Jinak řečeno, Karel IV. má předchůdce Jana Lucemburského. Tato trojice by mohla v TTL formátu vypadat následovně:

```
1 dbr:Charles_IV dbp:predecessor dbr:John_of_Bohemia .
```

Zdrojový kód 5.3: Trojice zapsaná v TTL formátu

Příklad zdrojového kódu RDF z báze znalostí YAGO zde neuvádíme z důvodu, že je strukturou velice podobný DBpedii.

5.4 Aplikační rozhraní

Jak již bylo zmíněno, aplikace disponuje aplikačním rozhraním. Účelem aplikačního rozhraní je umožnit implementovat nástroj tak, aby dokázal komunikovat s různými SPARQL koncovými body a navázat data na jakoukoliv vizualizaci. Přesněji – aby umožnil parsovat jakýkoliv zdroj dat v RDF formátu.

V aplikačním rozhraní se musí tedy vyskytovat rozhraní, které umožňuje parsovat data v RDF formátu. Na začátku vyhledávání je nutné získat počáteční podmět, kterým bude linie začínat, a přísudek, kterým se bude linie řídit. Musí také tedy disponovat rozhraním anebo implementací, které tato data získá – buď tedy např. od uživatele nebo souborového zdroje. Data musí být také nějak reprezentována, musí tedy také poskytovat univerzální přístup k RDF datům.

Dalším problémem je nejednoznačnost směru linie. Tento problém je detailněji popsán v sekci 6.3, nicméně ve zkratce nástroj musí rozpoznat za určitých podmínek, jak má dále pokračovat. Konkrétním příkladem je Karel IV. a tvorba linie na základě jeho předchůdců. Jelikož vládl několika územím, měl několik předchůdců. Není tedy jednoznačné, jakým směrem má nástroj pokračovat, proto musí uvnitř API existovat mechanismus, který tento problém vyřeší.

Součástí API bude také serializace a deserializace linie, jinak řečeno persistence. Linie bude serializována v textovém formátu pro možnost jakékoliv modifikace, kterou nástroj neposkytuje.

6 Implementace nástroje

V této kapitole se dozvíme o interních stránkách nástroje, mezi které patří API, důležité části kódu a jeho rozdělení do jednotlivých balíků. Nástroj je důkladně zdokumentován pomocí Javadoc, převážně tedy API část nástroje. V složitějších částech zdrojového kódu jsou doplňující komentáře.

6.1 Aplikační rozhraní

Nástroj disponuje aplikačním rozhraním a základní implementací, tj. grafickým rozhraním a napojením na koncový bod `https://dbpedia.org/sparql`. Aplikace je rozdělená na aplikační rozhraní a samotnou implementaci. Aplikační rozhraní se nahází v balíku `cz.zcu.jsmahy.datamining.api` a implementace v balíku `cz.zcu.jsmahy.datamining` (samozřejmě vyjma balíku `api`). Aplikační rozhraní existuje proto, že jsme chtěli umožnit aplikaci volně rozšiřovat. Samotné chování aplikace *není* vázáno na GUI – tyto části jsou odděleny. Celé API bylo napsáno a zdokumentováno v anglickém jazyce.

Frontend aplikačního rozhraní je navržen tak, že se může použít jakákoliv vizualizace dat. Nástroj především data sbírá, vizualizace těchto dat je tedy druhotná, nicméně nástroj stále disponuje vizualizací dat na časové linii.

Backend aplikačního rozhraní jsme se pokusili navrhnout tak, aby příliš nezávisel na knihovnách třetí strany, avšak se tomu nevyhneme. Nicméně, mělo by být možné použít jakoukoliv doplňující knihovnu pro sběr dat. Stěžejními třídami jsou `DataNode`, `DataNodeFactory` (implementace těchto tříd slouží k reprezentaci dat v paměti), `ResponseResolver` (implementace této třídy slouží k vyřešení nejasností při tvorbě linie, o řešení nejednoznačnosti linie se dozvíme více v sekci 6.3), `SparqlEndpointTask`, `RequestProgressListener` (implementace těchto tříd slouží k samotné tvorbě linie, přičemž `RequestProgressListener` slouží jako posluchač událostí uvnitř `SparqlEndpointTask`).

Aplikace používá framework Guice pro injekci závislostí. Tento framework požaduje implementaci tzv. `Module`, ve které registrujeme implementace tříd a vazby mezi třídami, které poté Guice použije pro poloautomatickou injekci závislostí. Nejdůležitější třídou je `AbstractModule`, kterou rozšiřuje třída `DataMiningModule`, která je základní implementací `Module`, v níž se např. nachází výchozí implementace `DataNodeFactory`. Modelová implementace bude představena později v programátorské příručce v příloze F.

6.1.1 Struktura dat

Jak bylo výše zmíněno, pro reprezentaci dat používáme třídy `DataNode`, `DataNodeFactory`.

`DataNode` Jedná se o rozhraní, ve kterém se ukládají metadata o RDF trojici. Rozhraní také disponuje odkazem na rodiče (nicméně se rodič při serializaci vynechává) a potomky, lze tedy takto vytvořit graf. Protože tvoříme linii, která má spíše stromovou strukturu, součástí rozhraní je také metoda, která říká, zda se jedná o kořen. Kořen nelze přidat jako potomka jiné `DataNode`. Rozhraní také obsahuje jednoznačný identifikátor a další pomocné funkcionality, mezi které patří např. traverzování grafem.

`DataNodeFactory` Jedná se o továrnu `DataNode`. Obsahuje pomocné metody pro tvoření jedné instance `DataNode` během tvorby linie.

6.1.2 Řešení nejasností

Jak bylo zmíněno, pro řešení nejasností používáme třídu `ResponseResolver`, která uvnitř nástroje obsahuje výchozí implementace pro různé případy. Uvnitř nástroje používáme termín „řešení nejednoznačností“ pro určení počátečního podmětu, data počátku a konce entity a nejednoznačnosti, kdy nástroj neví, kudy v linii pokračovat. Dá se tedy říci, že lze použít toto rozhraní tehdy, když požadujeme od uživatele vstup.

`ResponseResolver` Jedná se o rozhraní, které má za úkol vyřešit nejednoznačnosti. V balíku `resolvers` již existují implementace tohoto rozhraní pro všechny zmíněné nejednoznačnosti.

6.2 Implementace dotazu na linii

Existuje několik způsobů na dotazování linie. My si ukážeme dva způsoby a představíme jejich výhody a nevýhody.

6.2.1 Rekurzivní SPARQL dotaz

Prvním, nicméně náročnějším způsobem, je tvorba tzv. rekurzivního dotazu, nebo-li tzv. „property path“.

Úryvek 6.1 je ukázkou rekurzivního dotazu. V tomto dotazu hledáme všechny lidi, ke kterým se dostaneme pomocí predikátu „`foaf:knows`“ a dotazujeme se na jejich jméno pomocí „`foaf:name`“. Tento dotaz se dá přečíst

```
PREFIX : <http://example/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>

SELECT ?name
WHERE
  { ?x foaf:mbox <mailto:alice@example> .
    ?x foaf:knows+/foaf:name ?name .
  }
```

Zdrojový kód 6.1: Dotaz na všechny lidi dosažitelné od Alice

následovně: „Vyber mi výsledky, kdy: hledám podmět *x (člověk)* pomocí linky `foaf:knows`“. Nebo-li řečeno „`?x foaf:mbox <mailto:alice@example> .`“ se dá představit jako „`x` je někdo/něco, co má e-mail adresu `alice@example`“ a „`?x foaf:knows+/foaf:name ?name .`“ se dá představit jako „od `x` (tj. Alice) začni hledat pomocí `foaf:knows` a vyber mi jejich jméno“.

Lépe se dá tento úryvek představit s nějakými daty. Mějme například data (pro ukázku nebudou zapsaná ve formátu, jaký formát RDF požaduje):

- Alice zná Boba,
- Bob zná Karla,
- Karel zná Vaška,
- Jan zná Jakuba.

Výsledkem dotazu budou jména „Bob, Karel, Vašek“. Jan ani Jakub se ve výsledcích neobjeví, protože nelze vytvořit cestu od Alice tak, aby se k nim dostala.

Výhodou tohoto dotazu je rychlost – dotážeme se serveru a ten nám během chvíle vrátí všechny požadované výsledky. Nevýhodou je ztráta možnosti výběru, po jaké lince máme pokračovat, pokud narazíme na nejasný výsledek. Do tohoto dotazu lze přidat jistá omezení, která by zúžila, resp. vyjasnila výběr, nicméně se stále může stát, že nám dotaz vrátí více než dva výsledky. Kvůli tomuto nedostatku jsme nuceni použít druhý způsob tvorby linie, tj řetězení SPARQL dotazů.

6.2.2 Řetězení SPARQL dotazů

Druhým způsobem je vytvořit SPARQL dotaz, přečíst data a použít tato data v následujícím SPARQL dotazu. Rozdíl oproti předešlému způsobu je

ten, že vytváříme dotazy na každý výsledek oproti jednomu komplexnímu dotazu.

Mějme příklad Karla IV., kdy potřebujeme najít všechny jeho předky. Ihned narazíme na problém, jelikož nevíme, jakého předka vybrat. Byl českým králem, kdy jeho předchůdcem je Jan Lucemburský, a zároveň císařem Svaté říše římské, kdy jeho předchůdcem je Ludvík IV. Bavor. Rekurzivní dotaz tento problém ignoruje – vybere oba předchůdce, a to je nežádoucí. Linie by poté nebyla lineární, protože by v ní byli namíchaní předci z různých dynastií či říší. Lze tedy v každém kroku kontrolovat, zda daný dotaz vrátil více než dva výsledky, a poté vyřešit tuto nejednoznačnost, o které se dozvíme více v následující sekci.

Výhodou tohoto způsobu je jednoduchost a hlavně možnost výběru při každém kroku tvorby cesty, pokud je pokračování cesty nejasné. Nevýhodou je nutnost tvorby mnoha dotazů, čehož výsledkem je zpomalení aplikace.

6.3 Řešení nejednoznačnosti linie

K predikátu pro daný subjekt může korespondovat více než jeden objekt. Příkladem může být rodič, který má více než jednoho potomka. Tento problém musí umět nástroj vyřešit buď sám, anebo za pomocí lidské interakce.

Nástroj by tento problém mohl částečně řešit sám na základě restrikcí, které by byly nastaveny před začátkem tvorby linie. Nicméně, nastavení restrikcí by mohlo být poměrně komplikované z uživatelského hlediska, proto se v nástroji tento systém nevyskytuje.

Namísto toho se nástroj pokaždé dotazuje uživatele, jakým objektem chce dále pokračovat. Nástroj také zobrazí v stromové struktuře všechny možnosti, které byly uživateli nabídnuty (viz. návrh 2 – obr. 4.2 na straně 20). Tento způsob umožní přesnější tvorbu linie.

6.4 Tvorba linie

Prvním krokem je vytvořit název linie, který bude sloužit jako počáteční bod (kořen) pro celou linii. V aplikaci je možné mít několik paralelních linií uvnitř stromové struktury naráz – díky tomu lze zobrazit data paralelně.

Druhým krokem je stanovit počáteční data. Těmi jsou:

kým má linie začít Zobrazí se dialog, ve kterém uživatel napíše subjekt, který hledá. Jsou podporovány všechny jazyky, pro které existuje stránka Wikipedie. Dotaz musí korespondovat s URL Wikipedie (vyjma

speciálních znaků, jako jsou escape znaky, znak _ apod.). *(„Albert Einstein“)*

datum počátku a konce Zobrazí se tabulka se všemi trojicemi, které odpovídají datumu či číslicovému formátu. Validní formáty se dají nastavit v konfiguračním souboru. Uživatel prvně vybere predikát, který odpovídá počátku *(„byl narozen“)* intervalu. Poté se zobrazí stejná tabulka, ve které vybere predikát, který odpovídá konci *(„zemřel“)*.

jak se má linie vytvořit Zobrazí se tabulka s trojicemi, které jsou filtrovány na kandidáty pro tvorbu linie *(„doktorandský poradce“)*. Mezi vyfiltrované trojice patří RDF zdroje bez zbytečných zdrojů, mezi které patří např. odkaz na Wikipedii.

Poté program začne automaticky dohledávat dané trojice. Pokud nastane situace, že dotaz vrátí více než jeden subjekt, pak se program obrací na uživatele. K tomu slouží již zmíněné rozhraní `ResponseResolver`. Uživateli jsou představeni všichni kandidáti, po kterých lze dále v linii pokračovat. Jsou také představeni ti, kteří tu linii dokončí (literály, tj. už na ně nic v datech nenavazuje).

Všechny tyto objekty je dále možné zobrazit uvnitř linie pod daným subjektem, u kterého tato nejednoznačnost nastala. Například jeden z doktorandských poradců v linii počínající Albertem Einsteinem je Christian Heinrich Bünger, který měl poradce dva. Pod tímto subjektem budou možné zobrazit všechny doktorské poradce, nicméně bude moci být vybrán pouze jeden, kterým bude program pokračovat.

6.5 Knihovny

Jak bylo v předešlé kapitole zmíněno, budeme používat JavaScript knihovnu pro vizualizační část nástroje. Nicméně, zmíníme další možnosti, nad kterými jsme při tvorbě návrhů přemýšleli. Pro tvorbu frontendu jsme měli na výběr několik knihoven, které si nyní v krátkosti představíme.

JavaFXSmartGraph Jelikož jsme se rozhodli pro druhý návrh (viz 4.2 na straně 20), nebudeme tvořit grafy. Nicméně, jednalo se o kandidáta u prvního návrhu.

JGraphT Ze stejného důvodu jako JavaFXSmartGraph jsme se rozhodli tuto vizualizaci nepoužít.

AWT Bohužel již poměrně zastaralé API, které je součástí jazyka Java. Není ideální pro tvorbu uživatelského rozhraní. Nicméně, pokud bychom chtěli tvořit vlastní vizualizaci grafů, použili bychom právě toto API, anebo Swing.

Swing Podobně jako AWT se jedná o starší API, které se ujalo o trochu více.

JavaFX Jedná se pravděpodobně o nejznámější externí knihovnou pro tvorbu uživatelského rozhraní. Tato knihovna disponuje kvalitním grafickým zpracováním a je zároveň dlouhodobě udržovaná a vyvíjená. Pro druhý návrh je ideálním kandidátem.

Přemýšleli jsme také o knihovnách pro práci s grafovými daty pro backend. Jelikož ve skutečnosti tvoříme orientovaný graf, tyto knihovny by nám měly zjednodušit práci s implementací grafových algoritmů – nicméně opak je pravdou, protože jsme v nástroji nemuseli použít prakticky žádné grafové algoritmy. Původně nástroj používal pro reprezentaci grafu knihovnu Apache Jena, nicméně tato knihovna byla nahrazena vlastní implementací a aplikace se také několikanásobně zrychlila. Nakonec jsme se tedy rozhodli nepoužít *žádnou* knihovnu pro práci s grafy. Vše bylo nahrazeno datovou strukturou popsanou v sekci 6.1.1.

Dále potřebujeme přečíst data v RDF formátu a tvořit SPARQL dotazy, které se pošlou na různé koncové body. Existují různé knihovny, které tyto služby poskytují, proto si zde vypíšeme pár z nich, nad kterými jsme přemýšleli.

DKPro JWPL Tato knihovna pracuje pouze nad statickými daty – tzv. dumpy Wikipedie. Nicméně nám znemožňuje pracovat nad reálnými daty, a proto nebyla vybrána.

Apache Jena Tato knihovna poskytuje vše, co potřebujeme – komunikaci se SPARQL endpointem, tvorbu SPARQL dotazu (lze ho vytvořit programově či napsat ho úplně sám), reprezentaci a manipulaci s daty v RDF formátu. Jedná se o knihovnu velice jednoduchou na použití. Základní datovou strukturou je tzv. `Model`. Tato knihovna je pro naší práci ideální, a proto jsme se ji rozhodli používat.

Aplikace také používá pomocné knihovny, mezi které patří např. Lombok pro zlepšení čitelnosti kódu, anebo Jackson pro serializaci dat do formátu JSON (anebo její zpětné deserializaci).

6.6 Samotná implementace nástroje

Nástroj byl implementován v jazyce Java verze 17 implementace OpenJDK. Na začátku projektu jsme používali sestavovací systém Maven, nicméně jsme od tohoto systému později přešli na Gradle. Projekt je také verzován pomocí Git. Aplikace je logována pomocí knihovny Log4J.

6.6.1 Grafické rozhraní

Implementace GUI se nachází uvnitř balíku `app.controller`. Stěžejní třídou je `MainController`, která je napojená na FXML soubor `main.fxml`. Veškerá inicializace se nachází uvnitř metody

```
public void initialize(URL location, ResourceBundle resources) {
    ...
}
```

Zde se nastavují vazby mezi prvky, např. tedy zobrazení dat v tabulce po kliknutí na entitu uvnitř stromové struktury či modifikace názvu entity v tabulce a následné projevení změny uvnitř stromové struktury.

6.6.2 Implementace tvorby linie

Veškerá logika tvorby linie se nachází ve třídě `DBPediaEndpointTask`, která původně byla jen implementací napojení na endpoint DBpedia. Nicméně, v posledních fázích programu se tato implementace stala poměrně obecnou implementací na jakýkoliv endpoint, avšak již z časových důvodů nebylo toto abstrahováno do vlastní třídy nacházející se v API. Jelikož se jedná o jednu ze stěžejních tříd programu, popíšeme si detailněji její implementaci.

Hlavní logika se nachází v metodě `call`

```
public synchronized R call() {
  // ...
}
```

uvnitř které vytvoříme `Model`, do kterého na začátku načteme data pomocí metody `model.read(URL)`. Tato metoda načte z URL RDF data, které můžeme pomocí metod uvnitř `Model` přečíst. Doménu URL lze měnit v konfiguračním souboru `config.yml`. Tato doména je přidána před počáteční podmět.

Třída `DBPediaEndpointTask` úzce pracuje s dvěma důležitými rozhraními – `ResponseResolver` a `RequestProgressListener`. Během tvorby linie se tato třída obrací právě na tyto dvě rozhraní, přičemž implementace `ResponseResolver` má za úkol vyjasnit nejednoznačnosti, když k nim dojde. Již na začátku tvorby linie narazíme na nejednoznačnost, kdy z pohledu programátora po výběru počátečního podmětu nevíme, po jaké lince máme pokračovat a jaké časové rozmezí máme vybrat – na to se musíme zeptat uživatele. Toto se odehrává v metodě

```
1  InitialSearchResult initialSearch(QueryData inputMetadata) {
2      InitialSearchResult pathPredicateResult =
           requestOntologyPathPredicate(inputMetadata);
3      if (pathPredicateResult != InitialSearchResult.OK) {
4          return pathPredicateResult;
5      }
6
7      InitialSearchResult startAndEndDateResult =
           requestStartAndEndDatePredicate(inputMetadata);
8      if (startAndEndDateResult != InitialSearchResult.OK) {
9          return startAndEndDateResult;
10     }
11
12     return InitialSearchResult.OK;
13 }
```

Uvnitř metod `requestXXX` používáme několik různých implementací rozhraní `ResponseResolver`, které se nachází uvnitř balíku `resolvers`. Na následující straně ve zdrojovém kódu 6.2 ukážeme jednu konkrétní implementaci tohoto rozhraní a popíšeme jednotlivé sekce kódu.

```
public class OntologyPathPredicateResolver extends
    DefaultResponseResolver<Collection<Statement>> {

    @Override
    protected void resolveInternal(Collection<Statement> candidates,
        SparqlEndpointTask<?> requestHandler) {
        Platform.runLater(() -> {
            RDFNodeChooserDialog dialog = new RDFNodeChooserDialog(
                candidates, ...);
            dialog.showDialogueAndWait(stmt -> result.addMetadata("
                ontologyPathPredicate", stmt.getPredicate()));

            markResponseReady();
            requestHandler.unlockDialogPane();
        });
    }
}
```

Zdrojový kód 6.2: Vybírání cesty

Třída rozšiřuje výchozí implementaci `DefaultResponseResolver`, která v sobě obsahuje pomocné metody pro řešení nejednoznačností a implementuje společné metody. Především implementuje:

```
ArbitraryDataHolder result = ...

abstract void resolveInternal(D inputMetadata, SparqlEndpointTask<?>
    requestHandler);

public void resolve(D inputMetadata, SparqlEndpointTask<?>
    requestHandler) {
    result.clearMetadata();
    resolveInternal(inputMetadata, requestHandler);
}

public ArbitraryDataHolder getResponse() {
    ...
    return result;
}
```

Dále tato třída vyvolává dialog `RDFNodeChooserDialog`. Můžeme si povšimnout ve zdrojovém kódu 6.2, že se vše odehrává uvnitř callbacku pro `Platform.runLater(...)`. Důvod této konstrukce je ten, že tuto metodu

voláme z jiného vlákna – pro `SparqlEndpointTask` se před tvorbou linie vytvoří vlastní vlákno. Poté se volají dvě metody na řádcích 9 a 10 – `markResponseReady()` a `requestHandler.unlockDialogPane()`. Po zavolání metody `resolve(...)` se uzavře vlákno a čeká se na odpověď. První metoda označí, že daný resolver má připravenou odpověď, a druhá uvolní vlákno, aby nemuselo čekat ve smyčce.

Během tvorby linie třída `DBPediaEndpointTask` volá v různých situacích callbacky rozhraní `RequestProgressListener`. Tyto situace jsou detailně popsány uvnitř samotného rozhraní.

6.6.3 Automatický redirect

Jelikož RDF zdroj může ukazovat na stránku, která je pouze odkazem na další stránku, musí být nástroj schopný se s tímto nějak vypořádat. Při dotazu na danou stránku na endpoint DBpedia je uvnitř RDF dat predikát „wikiPageRedirects“ s příslušným zdrojem, na který stránka odkazuje.

```
1  public Resource redirectIfPossible(Resource currentPage, Model model) {
2      return redirectIfPossible(currentPage, model, MAX_REDIRECTS /* 30
           */);
3  }
4  private Resource redirectIfPossible(Resource currentPage, Model model,
       int redirectsLeft) {
5      if (redirectsLeft <= 0) {
6          return currentPage;
7      }
8
9      final StmtIterator stmts = model.listStatements(currentPage,
           wikiPageRedirects, ...);
10     if (!stmts.hasNext()) { // no redirects found
11         return currentPage;
12     }
13
14     final Resource redirect = stmts.next();
15     model.read(redirect.getURI());
16     return redirectIfPossible(redirect, model, redirectsLeft - 1);
17 }
```

Zdrojový kód 6.3: Automatický redirect

Implementace je rozdělena do dvou metod, jelikož ji voláme rekurzivně (viz zdrojový kód 6.3). Tento vzor je poměrně běžný – zavoláme první metodu (označenou `public`), která slouží jako první volání rekurzivní metody. Tato metoda obvykle předá argument, který vyjadřuje maximální hloubku, ve

zdrojovém kódu 6.3 je to `MAX_REDIRECTS`, která později uvnitř druhé metody (označenou `private`) zajišťuje, že se program za žádných okolností nezastaví na nedostatek zásobníku – ovšem tedy za předpokladu, že podmínka je správně implementována. Implementace není úplná – vynechali jsme méně důležité kroky, aby byl zdrojový kód co nejkratší (např. `stmts.next()` nevrací `Resource`, ale `Statement`, ze kterého dokážeme požadovaný `Resource` získat přes určité transformace).

V druhé metodě vytvoříme SPARQL dotaz (řádek 9), přičemž subjektem je nynější stránka a predikátem je „wikiPageRedirects“. Každá stránka, která slouží k přesměrování, má tento predikát. Tento dotaz nám vrátí objekt, kterým je odkaz na další stránku, anebo nám nevrátí nic. Prvně se tedy koukneme, zda nám něco vrátil, a pokud nic nevrátil, nepřesměrováváme nikam (řádek 10–12). Pokud nám dotaz něco vrátil (řádek 14), tak přesměrujeme stránku (řádek 15).

Protože se teoreticky může stát, že přesměrovaná stránka `redirect` bude opět přesměrovávat na další stránku, voláme metodu rekurzivně (řádek 16) s touto stránkou jako argument do dalšího volání namísto jejího vracení. Rekurzivní volání nám buď vrátí stránku, která dále přesměrovává, jelikož jsme udělali příliš mnoho přesměrování (řádek 6), anebo poslední přesměrovanou stránku (řádek 11). Stav, kdy bylo příliš mnoho přesměrování, by teoreticky neměl nastat, chceme si být ale jisti, že se nezacyklíme, anebo nenalezneme stránku s mnoha přesměrováními.

6.6.4 Datový model vizualizačního nástroje

V minulé kapitole jsme se seznámili s vizualizačním nástrojem, který budeme používat. Nicméně, nepopsali jsme formát dat, které zpracovává, a integraci s tímto nástrojem.

Datový formát

```
1  {
2      "id":<Number>,
3      "stereotype":<String>,
4      "name":<String>,
5      "begin":<String ISO8601>,
6      "end":<String ISO8601>,
7      "description":<String>,
8      "properties": { ... },
9      "subItems": { ... }
10 }
```

Zdrojový kód 6.4: Datový formát časové linie *(převzato z BP)*[7]

id Vyjadřuje identifikaci prvku v grafu, díky tomuto můžeme prvek v grafu najít, anebo ho spojit s jiným prvkem pomocí hrany.

stereotype Vyjadřuje typ prvku, podle toho typu dokážeme poznat jaký použít *renderer* pro jeho vykreslení.

name Obsahuje hlavní název prvku, který se zobrazí uživateli.

begin Vyjadřuje datum začátku prvku v grafu.

end Vyjadřuje datum konce prvku v grafu. Pokud tento údaj neuvedeme, považujeme prvek za moment.

description Nepovinný údaj, který obsahuje popis prvku.

properties Nepovinný údaj, který obsahuje doplňující informace o prvku.

subItems Pole všech pod-prvků, aby prvek mohl obsahovat podmnožinu dalších prvků.[7]

Dále popíšeme formát pod-prvků `subItems`.

```
{
    "id":<Number>,
    "name":<String>,
    "begin":<String ISO8601>,
    "end":<String ISO8601>,
    "css":<String>
}
```

Zdrojový kód 6.5: Datový formát pod-prvku *(převzato z BP)*[7]

id Vyjadřuje identifikaci pod-prvku v grafu, díky tomuto můžeme pod-prvek v grafu najít, anebo ho spojit s jiným prvkem pomocí hrany. Toto identifikační číslo musí být unikátní v celé množině dat (nemůže ho obsahovat ani žádný nad-prvek).

name Obsahuje hlavní název pod-prvku, který se zobrazí uživateli.

begin Vyjadřuje datum začátku pod-prvku v grafu.

end Vyjadřuje datum konce pod-prvku v grafu. Pokud tento údaj neuvedeme, považujeme pod-prvek za moment.

css Nepovinný údaj, který obsahuje doplňující kaskádové styly, které se přiřadí pod-prvku při vykreslování. Pro definování tříd existuje soubor (*../src/css/customStyles.css*), ve kterém by se měly styly objevovat, je to doporučeno pro lepší přehlednost a údržbu stylů aplikace.[7]

Integrace nástroje

Nyní známe formát dat, můžeme tento nástroj integrovat do našeho nástroje. V uživatelské příručce je řečeno, že pro úpravu dat musíme upravit soubor `src/data/data.js`. V tomto souboru se na začátku nachází definice funkce, která vrací JSON.

```
1 define([],function(){
2   return {
3   ...
4   }
5 }
```

Zdrojový kód 6.6: Soubor data.js

Ve zdrojovém kódu 6.6 můžeme vidět soubor `src/data/data.js`. Můžeme rozdělit tento soubor na 3 části:

prefix část od začátku souboru po `return {`

tělo část uvnitř `return { ... }`

sufix zbytek souboru

Nástroj musí tedy:

1. generovat z vytěžených dat data ve formátu JSON,
2. vytvořit soubor `src/data/data.js` pokud neexistuje a smazat obsah,
3. zapsat prefix, data ve formátu JSON a sufix.

Poté musí nástroj tato data zobrazit. Opět je v uživatelské příručce řečeno, že máme otevřít soubor `src/index-sub-items-source.html` pro zobrazení časové linie. Vzhledem k našemu návrhu vizualizace pravá část, ve které se nachází Webview a zobrazuje se v ní stránka `wikipedia.org` pro příslušný podmět, zní jako ideální kandidát pro integraci vizualizace časové linie.

Nicméně, kvůli interním problémům v knihovně JavaFX nelze tohoto využít. Konkrétně se data po přepsání souboru `src/data/data.js` neobnovují. Zkoušeli jsme různé možnosti, mezi které patří obnovování stránky pomocí API JavaFX, spouštění vlastního JavaScript skriptu, který obnoví stránku, nicméně ani jeden způsob nefunguje. Z tohoto důvodu jsme nuceni spouštět výchozí prohlížeč operačního systému a vložit do URL cestu k souboru `src/index-sub-items-source.html`. Tento způsob nám umožní data obnovovat, obvykle tedy pomocí klávesy F5, anebo příslušného tlačítka v prohlížeči.

Implementace exportování se nachází uvnitř třídy `FialaBPSerializer`, která rozšiřuje rozhraní `DataNodeSerializer`. Datový model reprezentují třídy `FialaBPExportFormatRoot`, která obsahuje seznam uzlů a hran, poté `FialaBPExportNodeFormat`, která reprezentuje uzel, a nakonec datový model pro hranu ve třídě `FialaBPExportEdgeFormat`.

```
1   public void serialize(OutputStream out, DataNode root) {
2       List<? extends DataNode> dataNodes = root.getChildren();
3       List<FialaBPExportNodeFormat> nodes = getNodes(dataNodes);
4       List<FialaBPExportEdgeFormat> edges = getEdges(dataNodes);
5       FialaBPExportFormatRoot exportRoot = new
            FialaBPExportFormatRoot(nodes, edges);
6       serialize(out, exportRoot);
7   }
8
9   synchronized void serialize(OutputStream out,
        FialaBPExportFormatRoot root) {
10      try (PrintWriter pw = new PrintWriter(out, true,
            StandardCharsets.UTF_8)) {
11          pw.print(PREFIX);
12          objectMapper.writeValue(pw, root);
13          pw.print(SUFFIX);
14      }
15  }
```

Při serializaci musíme implementovat metodu `serialize(OutputStream, DataNode)`, která nám předá referenci pro výstup a kořen datové struktury `DataNode`. Implementace `DataNodeSerializer` se využívá uvnitř třídy `MainController` během požadavku zobrazení časové linie. Po exportování se zavolá callback `onDisplayRequest`, který má za úkol časovou linii zobrazit. Serializace se také využívá při exportování dat pro uložení postupu[1].

[1]Uložení postupu bohužel není zcela funkční. Při implementaci importování dat jsme narazili na slepou ulici, jelikož knihovna Jackson nedokázala rozumně importovat data. Důvod známe – v souboru chybí metadata o třídě, kterou jsme exportovali, nicméně, jak se ukázalo, tuto informaci není tak jednoduché přidat.

Obrázek 6.1: UML diagram důležitých rozhraní a implementací

7 Testování nástroje

Nástroj byl testován jednak pomocí unit testů, jednak pomocí uživatelské interakce. Namísto typické knihovny JUnit pro unit testy jsme použili v tomto nástroji Spock. Toto rozhodnutí bylo především experimentální. V uživatelských testech jsme se zaměřovali na úplnost a srozumitelnost programu.

Testovali jsme také přesnost a úplnost programu. Pokusili jsme se vytvořit různé linie a porovnávali jsme výstup našeho programu a ChatGPT s očekávaným výstupem. Výsledky testů jsou v příloze A.

7.1 Unit testování

Otestováno je především API, u kterého jsme dosáhli pokrytí kódu 60%. Pokrytí je nízké především z toho důvodu, že uvnitř některých metod používáme kód pro tvorbu GUI (např. upozornění), které je testováno uživatelsky. Netestovali jsme také třídy typu Plain Old Java Object (POJO). Vyjma těchto metod či tříd bychom dosáhli přibližně 85–90% pokrytí kódu.

Aplikace je testována za použití testovací knihovny Spock psaná v programovacím jazyce Groovy. Zvolili jsme tento jazyk a tuto knihovnu z osobního důvodu – jednotkové testy připadají výrazně přehlednější oproti knihovně JUnit a lze je také deskriptivněji popsat v jednotlivých částech testu.

Spock se nechal inspirovat knihovnou JUnit společně s dalšími testovacími knihovnami, mezi které patří například jMock, anebo samotným jazykem Groovy, který je navržený pro psaní testů. V následující ukázce porovnáme jednotkový test, který je psaný za použití knihoven JUnit a Spock. Modelový případ by měl ukázat, že „nelze přidat jakémukoliv potomkovi kořen datové struktury `DataNode`“.

```
void shouldThrowIAEWhenAddingRootToTheChildren() {
    DataNode root = nodeFactory.newRoot("Root");

    // Test case 1: Root node
    DataNode node = nodeFactory.newRoot();
    // Root is added to another root
    // Throw an IAE because that's not allowed
    assertThrows(IllegalArgumentException.class, () -> node.addChild(
        root));

    // Test case 2: Regular node
    node = nodeFactory.newNode("");
    // Root is added to a data node
    // Throw an IAE because that's not allowed
    assertThrows(IllegalArgumentException.class, () -> node.addChild(
        root));
}
```

Zdrojový kód 7.1: Jednotkový test JUnit

```
def "Should throw IAE when adding root to the children of any node type
    "() {
    given: "A root node"
    DataNode root = nodeFactory.newRoot("Root")

    when: "Root is added to a data node"
    node.addChild(root)

    then: "Throw an IAE because that's not allowed"
    thrown(IllegalArgumentException)

    where: "Node can be any node type -- that is a root or a regular
        node"
    node << [nodeFactory.newRoot(), nodeFactory.newNode(_)]
}
```

Zdrojový kód 7.2: Jednotkový test Spock

Komentáře jsou pravděpodobně příliš dlouhé, nicméně jsou zde přidány pro ukázku, jak by mohla vypadat dokumentace testu. Na první pohled můžeme vidět, že Spock spojil oba modelové případy `Test case 1` a `2` do jednoho pomocí následující konstrukce:

```
node << [nodeFactory.newRoot(), nodeFactory.newNode(_)]
```

Vidíme tedy pouze jedno tvrzení `thrown(...)` v ukázce 7.2 namísto dvou tvrzení `assertThrows(...)` v ukázce 7.1. Dále můžeme vidět, že kód je rozdělen do několika částí:

1. `given`
2. `then`
3. `when`
4. `where`

`given` definuje „máme dáno“, `when` definuje „za jakých podmínek se má něco stát“, `then` definuje „co se má stát“ a `where` definuje „s jakými daty“. Můžeme si také povšimnout rozdílu dokumentace – namísto typických komentářů začínajícími dvojitým lomítkem „//“ jsou komentáře přidány hned za uvedené části.

Nakonec si můžeme povšimnout signatury metody. V typických JUnit testech je názvem metody to, co testujeme v jejím těle. Ve Spocku se definuje metoda následovně:

```
def "Co testujeme uvnitř těla"() { (...) }
```

7.2 Uživatelské testování

Uživatelské rozhraní bylo testováno pomocí lidské interakce. Vytvořit uživatelské testy v testovací knihovně TestFX za použití jazyka Groovy společně s knihovnou Spock se ukázalo být poměrně náročné na nastavení – dokumentace je zastaralá a aplikace vyhazovala výjimky při spuštění.

Nástroj nejlépe funguje při hledání lidských subjektů, kdy dokážeme zadat data jejich narození a úmrtí jako počátek a konec intervalu. Tyto dva prvky bývají u lidí konzistentní a obvykle mívají dva predikáty pro narození a úmrtí (tj. celkem 4 predikáty – 2 predikáty pro narození a 2 predikáty pro úmrtí), přičemž jeden z nich se vyskytuje skoro vždy. Těmito predikáty jsou:

- `dbp:birthDate`
- `dbp:deathDate`
- `dbo:birthDate`
- `dbo:deathDate`

Predikát `dbp` značí kořenové IRI `https://dbpedia.org/property/` a `dbo` značí `https://dbpedia.org/ontology/`. Predikáty produkující konzistentnější výsledky jsou ty, které začínají prefixem `dbp`.

V následujících odstavcích představíme zpětnou vazbu od některých testerů, mírně modifikovanou pro lepší čitelnost, avšak s nezměněným významem, a odpověď či řešení jejich problémů. Testeři neprošli školením ohledně programu, pouze jim byla předána distribuovatelná verze programu a informace o návodu uvnitř aplikace, kterou jsme jim doporučili si přečíst před použitím programu. Testerům bylo pouze řečeno, že mají vyzkoušet vytvořit nějakou linii – nebyla jim předem určena jaká linie. U některých případů jsme poskytli mírnou asistenci při tvorbě linie – tester se prvně neorientoval, nicméně s lehkou pomocí bylo vše v pořádku – či jsme navrhli nějaké konkrétní linie, u kterých bylo předem známo, že se dobře tvoří.

„Chvilku mi trvalo, než jsem přišel na to, že parametr „od–do“ nemá vliv na tvorbu linie.“ V textu nápovědy nástroje se nachází popis tohoto parametru, ačkoliv nebylo explicitně řečeno, že tento parametr nemá na tvorbu linie vliv – že pouze sbírá z daného podmětu časový údaje, které se zobrazí na časové ose. Uživatel si myslel, že se jedná o rozsah, odkud nástroj sbírá výsledky, např. u prvního podmětu „Karel IV.“ by vybral časové údaje „narodil se“ a „zemřel“ a to ho navedlo k myšlence, že se linie bude pouze tvořit z tohoto časového rozsahu.

Řešení: Úprava nápovědy tak, aby bylo jasné, že tento parametr nehraje roli v tvorbě linie, nýbrž pouze sbírá časové údaje.

„Tvorba linie byla poměrně náročná, jelikož jsem musela linii několikrát předělávat, protože se zasekla.“

Řešení: Bohužel, řešení tohoto problému není tak jednoduché, jelikož data v bázi znalostí nejsou konzistentní. Potenciálním řešením tohoto problému by bylo napojení na jinou bázi znalostí, nicméně v daném časovém rozmezí nelze napojení na jiný koncový bod SPARQL implementovat.

8 Závěr

Tato práce nastínila problematiku zpracování polostrukturovaných dat a společně s ní byl implementován nástroj, který s těmito daty pracuje. Cílem nástroje bylo umožnit běžnému uživateli jednoduše sestavit linii s daty, které lze jednoduše modifikovat a konfigurovat. Součástí programu je také vizualizace těchto dat na časové ose pomocí front-end knihovny poskytnuté od Bc. Michala Fialy, kterou jsme do programu integrovali tak, že jsme exportovali data do požadovaného formátu popsaného v jeho bakalářské práci.

Nástroj byl dostatečně otestován jednak po funkční stránce pomocí jednotkových testů, jednak po stránce uživatelské. Bylo vytvořeno API, pomocí kterého lze nástroj rozšířit, a proto lze nástroj využít v jiné bakalářské práci pro doplnění funkcionalit. Nicméně, implementace API byla druhořadá při tvorbě programu, proto je možné, že nebude tak intuitivní s tím pracovat.

Během implementace nástroje jsme narazili na mnoho problémů ohledně inkonzistence DBpedie, přesto si nástroj s těmito daty dokáže poměrně dobře poradit. Báze znalostí Wikidata je z hlediska dat konzistentnější a zároveň data lépe kategorizuje, jedná se tedy o kvalitnější bázi dat, avšak je hůře čitelná, proto jsme raději pracovali s DBpedií.

Seznam zkratek

API Aplikační rozhraní. 22, 23, 27, 28, 33, 34, 40, 43, 47, 79

BP Bakalářská práce. 24, 38, 39, 50, 51

CSS Cascading Style Sheets. 23

CSV Comma-separated Values. 11, 51, 59

FOAF Friend of a Friend. 9, 10

GPT Generative Pre-trained Transformer. 6

GUI Grafické rozhraní. 22, 34, 43, 80

HTML Hypertext Markup Language. 11, 23

IRI Internationalized Resource Identifier. 7, 9, 10, 14, 46

JSON JavaScript Object Notation. 11, 16, 33, 40, 51, 58

LLM Large Language Model. 6

NLP Zpracování přirozeného jazyka. 5, 6

OS Operační systém. 55, 67–69, 75

POJO Plain Old Java Object. 43

RDF Resource Description Framework. 3, 7–10, 12–15, 22, 23, 26, 27, 29, 30, 32–34, 37, 48, 51, 55, 62, 63

RDFS RDF Schema. 7, 12

SPARQL SPARQL Protocol and RDF Query Language. 8–10, 12–14, 22, 27, 30, 33, 38, 46, 48, 51, 60

SQL Structured Query Language. 9, 12

TTL Terse RDF Triple Language. 8, 10–12, 27, 51, 59

URI Uniform Resource Identifier. 8, 10, 14, 18

URL Uniform Resource Locator. 14, 34, 40

WWW World Wide Web. 7

XML Extensible Markup Language. 11, 14, 26, 51, 58

YAGO Yet Another Great Ontology. 13, 25, 27

Seznam obrázků

Seznam zdrojových kódů

Literatura

[1] Arenas, M. – Gutierrez, C. – Pérez, J. An Extension of SPARQL for RDFS. In Christophides, V. – Collard, M. – Gutierrez, C. (Ed.) *Semantic Web, Ontologies and Databases*, s. 1–20, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. ISBN 978-3-540-70960-2.

[2] Beckett, D. – Berners-Lee, T. *TTL (Terse RDF Triple Language)* [online]. [cit. 2023-03-29]. Dostupné z: `https://www.w3.org/TeamSubmission/turtle/`.

[3] Breitman, K. K. – Casanova, M. A. – Truszkowski, W. *Ontology in Computer Science*, s. 17–34. Springer London, London, 2007. doi: 10.1007/978-1-84628-710-7_2. Dostupné z: `https://doi.org/10.1007/978-1-84628-710-7_2`. ISBN 978-1-84628-710-7.

[4] Brickley, D. – Guha, R. – McBride, B. *RDFS* [online]. [cit. 2023-03-29]. Dostupné z: `https://www.w3.org/TR/rdf-schema/`.

[5] Contributors, W. *Wikidata* [online]. [cit. 2023-04-15]. Dostupné z: `https://www.wikidata.org/wiki/Wikidata:Main_Page`.

[6] De, S. et al. Chapter 1 - An introduction to data mining in social networks. In De, S. et al. (Ed.) *Advanced Data Mining Tools and Methods for Social Computing*, Hybrid Computational Intelligence for Pattern Analysis. https://www.sciencedirect.com/science/article/pii/B9780323857086000084: Academic Press, 2022. s. 1–25. doi: https://doi.org/10.1016/B978-0-32-385708-6.00008-4. ISBN 978-0-323-85708-6.

[7] Fiala, M. Implementace testovacího nástroje pro časovou osu. Master's thesis, Západočeská univerzita v Plzni, Univerzitní 2732/8, 301 00 Plzeň, 2019.

[8] Heist, N. – Paulheim, H. Entity Extraction from Wikipedia List Pages. In Harth, A. et al. (Ed.) *The Semantic Web*, s. 327–342, Cham, 2020. Springer International Publishing. ISBN 978-3-030-49461-2.

[9] Kao, A. – Poteet, S. R. *Overview*, s. 1–7. Springer London, London, 2007. doi: 10.1007/978-1-84628-754-1_1. Dostupné z: `https://doi.org/10.1007/978-1-84628-754-1_1`. ISBN 978-1-84628-754-1.

[10] Lajus, J. – Galárraga, L. – Suchanek, F. Fast and Exact Rule Mining with AMIE 3. In Harth, A. et al. (Ed.) *The Semantic Web*, s. 36–52, Cham, 2020. Springer International Publishing. ISBN 978-3-030-49461-2.

[11] OpenAI. [online]. [cit. 2023-04-27]. Dostupné z: https://openai.com/blog/chatgpt.

[12] Özsu, M. T. *Database*, s. 931–932. Springer New York, New York, NY, 2018. doi: 10.1007/978-1-4614-8265-9_80734. Dostupné z: https://doi.org/10.1007/978-1-4614-8265-9_80734. ISBN 978-1-4614-8265-9.

[13] Pellissier-Tanon, T. – Weikum, G. – Suchanek, F. *YAGO 4: A Reason-able Knowledge Base* [online]. [cit. 2023-04-15]. Dostupné z: https://suchanek.name/work/publications/eswc-2020-yago.pdf.

[14] University, L. – Mannheim, U. *DBPedia* [online]. [cit. 2023-04-06]. Dostupné z: https://www.dbpedia.org/about/.

[15] Vu, B. et al. A Graph-Based Approach for Inferring Semantic Descriptions of Wikipedia Tables. In Hotho, A. et al. (Ed.) *The Semantic Web – ISWC 2021*, s. 304–320, Cham, 2021. Springer International Publishing. ISBN 978-3-030-88361-4.

[16] W3 Contributors. *RDF* [online]. [cit. 2023-03-10]. Dostupné z: https://www.w3.org/egov/wiki/Resource_Description_Format.

[17] Wikipedia contributors. Resource Description Framework — Wikipedia, The Free Encyclopedia, 2023. Dostupné z: https://en.wikipedia.org/w/index.php?title=Resource_Description_Framework&oldid=1139034032. [Online; accessed 10-March-2023].

[18] Wikipedia contributors. Semi-structured data — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Semi-structured_data&oldid=1076318900, 2022. [Online; accessed 13-March-2023].

[19] Wikipedia contributors. Data model — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Data_model&oldid=1109677104, 2022. [Online; accessed 13-March-2023].

[20] Wikipedia contributors. Unstructured data — Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Unstructured_data&oldid=1133968256, 2023. [Online; accessed 13-March-2023].

[21] YAGO. *YAGO* [online]. [cit. 2023-04-15]. Dostupné z: https://yago-knowledge.org/.

A Výsledky testů

Náš nástroj	ChatGPT–3.5	Očekávaný výstup
Windows 11	Windows 11	Windows 11
Windows 10	Windows 10	Windows 10
Windows 8.1, Windows 8*	Windows 8	Windows 8.1
Windows 8	—	Windows 8
Windows 7	Windows 7	Windows 7
Windows Vista	Windows Vista	Windows Vista
Windows XP	Windows XP	Windows XP
Windows 2000, Windows Me**	Windows 95	Windows 2000, Windows Me**
Windows 98	—	Windows 98
Windows 95	—	Windows 95
Windows 3.1x	—	Windows 3.1x
Windows 3.0	—	Windows 3.0
Windows 2.1x	—	Windows 2.1x
Windows 2.0x	—	Windows 2.0x
Windows 1.1x	—	Windows 1.1x
Windows 1.0x	Windows 1.0	Windows 1.0x
MS-DOS	—	MS-DOS

Tabulka A.1: Linie OS Windows

**Nástroj nám nabídl tyto dvě verze zároveň, nicméně „Windows 8.1“ není RDF zdroj, musí se tedy opět tvořit linie z „Windows 8“. Pokud bychom však zvolili Windows 8, vše by bylo v pořádku.*

***Zde jsme vybrali „Windows Me“, abychom vytvořili podobnou linii vygenerovanou pomocí ChatGPT.*

V tabulce A.1 můžeme vidět linii operačního systému Windows. Našemu nástroji jsme zadali prvotní podmět „Windows 11“ a přísudek „dbp:precededBy“, podle kterého se tvořila daná linie.

Umělé inteligenci ChatGPT jsme předali dotaz „Would you please make a timeline, starting from Windows 11 and ending at the earliest version of Windows?“, přeloženo „Mohl bys mi prosím vytvořit časovou linii začínající

Windows 11 a končící nejstarší verzí Windows?“. ChatGPT odpověděl „Here's a timeline starting from Windows 11 and going back in time to the earliest version of Windows“, přeloženo „Zde je časová linie začínající Windows 11 a, v čase do minulosti, končící nejdřívější verzí Windows“. Za touto odpovědí následovala linie operačního systému Windows. Můžeme vidět, že náš nástroj si s tímto obecným dotazem poradil lépe než ChatGPT.

Náš nástroj	ChatGPT–3.5	Očekávaný výstup
Albert Einstein	Albert Einstein	Albert Einstein
Alfred Kleiner	Alfred Kleiner	Alfred Kleiner
Johann Jakob Müller	August Kundt (, Adolf Eugen Fick)	Johann Jakob Müller
Adolf Eugen Fick	Hermann von Helmholtz	Adolf Eugen Fick
Franz Ludwig Fick	Johannes Peter Müller	Franz Ludwig Fick
Christian Heinrich Bünger	Christian Gottfried Ehrenberg	Christian Heinrich Bünger
Justus Christian Loder, Gottfried Christoph Beireis *	Carl Friedrich Gauss	Justus Christian Loder, Gottfried Christoph Beireis *
Heinrich August Wrisberg, August Gottlieb Richter **	Johann Pfaff	Heinrich August Wrisberg, August Gottlieb Richter **
Georg Gottlob Richter	Abraham Gotthelf Kaestner	Georg Gottlob Richter
Johann Ludwig Hannemann	Johann Christoph Wichmannshausen	Johann Ludwig Hannemann

Tabulka A.2: Doktorandští poradci Alberta Einsteina

**Zde jsme vybrali „Justus Christian Loder“*
***Zde jsme vybrali „Gottlieb Richter“*

V tabulce A.2 můžeme vidět linii doktorandských poradců Alberta Einsteina. Našemu nástroji jsme zadali prvotní podmět „Albert Einstein“ a přísudek „dbp:doctoralAdvisor“, podle kterého se tvořila daná linie.

Umělé inteligenci ChatGPT jsme předali dotaz „Would you please generate a timeline of doctoral advisors, starting from Albert Einstein. In essence, I would like to know Albert Einstein's doctoral advisor, and the doctoral advisor's doctoral advisor.“, přeloženo „Mohl bys prosím vytvořit časovou

osu doktorandů, počínaje Albertem Einsteinem. V podstatě bych rád znal doktorského poradce Alberta Einsteina a doktorandského poradce jeho doktorského poradce.“. Odpověděl „Sure, here is a timeline of doctoral advisors starting from Albert Einstein“, přeloženo „Jistě, zde je časová linie doktorandských poradců počínaje Albertem Einsteinem“.

Nicméně, již třetí výsledek byl špatně – po „Alferd Kleiner“ nabídl „August Kund“. Pokusili jsme se ho opravit tak, že jsme mu napsali, že to není správně – „po Alfred Kleiner má následovat Adolf Eugen Fick“ a zda by to neopravil. ChatGPT se omluvil, opravil tuto chybu, nicméně zbytek linie byl špatně – viz tabulka A.2.

B Datové formáty

```
{
  "http://dbpedia.org/resource/Albert_Einstein":
  {
    "http://dbpedia.org/property/birthDate":
    [
      {
        "type": "literal",
        "value": "1879-03-14",
        "datatype": "http://www.w3.org/2001/XMLSchema#date"
        }
    ],
    "http://dbpedia.org/property/deathDate":
    [
      {
        "type": "literal",
        "value": "1955-04-18",
        "datatype": "http://www.w3.org/2001/XMLSchema#date"
        }
    ],
    "http://dbpedia.org/property/doctoralAdvisor":
    [
      {
      "type": "uri",
      "value": "http://dbpedia.org/resource/Alfred_Kleiner"
      }
    ]
  }
}
```

Zdrojový kód B.1: Příklad JSON formátu

```
<!-- http://dbpedia.org/resource nahrazeno zkratkou "DBR" -->

<rdf:RDF>
  <dbp:birthDate rdf:datatype="http://www.w3.org/2001/XMLSchema#date">
    1879-03-14
  </dbp:birthDate>
  <dbp:deathDate rdf:datatype="http://www.w3.org/2001/XMLSchema#date">
    1955-04-18
  </dbp:deathDate>
  <dbp:doctoralAdvisor rdf:resource="DBR/Alfred_Kleiner"/>
</rdf:RDF>
```

Zdrojový kód B.2: Příklad XML formátu

```
jmeno, rok narozeni, rok umrti
Karel IV., 1316, 1378
```

Zdrojový kód B.3: Příklad CSV formátu

```
dbr:Albert_Einstein dbp:birthDate "1879-03-14"^^xsd:date ;
dbr:Albert_Einstein dbp:deathDate "1955-04-18"^^xsd:date ;
dbr:Albert_Einstein dbp:doctoralAdvisor dbr:Alfred_Kleiner .
```

Zdrojový kód B.4: Příklad TTL formátu

C SPARQL

```
SELECT (COUNT(*) as ?count)
WHERE {
  ?s ?p ?o.
}
```

Zdrojový kód C.1: SPARQL příkaz pro spočtení trojic

```
# deklarace prefixu
PREFIX foo: <http://example.com/resources/>
...
# definice datove sady
FROM ...
# select
SELECT ...
# vzor dotazu
WHERE {
    ...
}
# modifikatoru dotazu
ORDER BY ...
```

Zdrojový kód C.2: Formát SPARQL příkazu

D Datové zdroje

```
{{Infobox - osoba
 | jmeno =
 | obrazek =
 | velikost obrazku =
 | popisek =
 | rodne jmeno =
 | datum narozeni =
 | misto narozeni =
 | datum umrti =
 | misto umrti =
 | pricina umrti =
 ...
}}
```

Zdrojový kód D.1: Příklad šablony infoboxu

```
{{Infobox - osoba 2
 | obrazek = Charles IV-John Ocko votive picture-fragment.jpg
 | popisek = Podobizna Karla IV. na [[Votivni obraz Jana Ocka z Vlasimi
     |Votivnim obrazu<br />Jana Ocka z Vlasimi]]
 | misto narozeni = [[Praha]]<br>{{Vlajka a nazev|ceske kralovstvi}}
 | misto umrti = [[Praha]]<br>{{Vlajka a nazev|ceske kralovstvi}}
 {{Infobox - osoba 2/titul
 | titul = [[Seznam panovniku Svate rise rimske|Cisar Svate rise rimske
     ]]
 | obdobi = [[1355]]--[[1378]]
 | korunovace = [[5. duben|5. dubna]] [[1355]]
 | predchudce = [[Ludvik IV. Bavor]]
 | nastupce = [[Zikmund Lucembursky]]
}}
{{Infobox - osoba 2/titul
 | titul = [[Seznam panovniku Svate rise rimske|rimsko-nemecky kral]]
 | obdobi = [[1346]]/[[1347|7]]--[[1378]]
 | korunovace = [[26. listopad]]u [[1346]]
 | predchudce = [[Ludvik IV. Bavor]]
 | nastupce = [[Vaclav IV.]]
}}
{{Infobox - osoba 2/titul
 | titul = [[Seznam predstavitelu ceskeho statu|cesky kral]]
 | obdobi = [[1346]]--[[1378]]
 | korunovace = [[2. zari]] [[1347]]
 | predchudce = [[Jan Lucembursky]]
 | nastupce = [[Vaclav IV.]]
}}
```

```
...
}}
```

Zdrojový kód D.2: Infobox Karla IV. (zkrácený)

```
<!-- http://dbpedia.org/resource nahrazeno zkratkou "DBR" -->
<!-- http://www.w3.org/2001/XMLSchema nahrazeno zkratkou "W3" -->

<rdf:RDF>
<!-- (...) -->
  <rdf:Description rdf:about="DBR/Charles_IV,_Holy_Roman_Emperor">
    <dbp:predecessor rdf:resource="DBR/John_of_Bohemia"/>
    <dbp:successor rdf:resource="DBE/Sigismund,_Holy_Roman_Emperor" />
    <dbp:successor rdf:resource="DBR/Wenceslaus_IV_of_Bohemia" />
    <dbp:years rdf:datatype="W3#integer">1346</dbp:years>
    <dbp:years rdf:datatype="W3#integer">1355</dbp:years>
    <dbp:years xml:lang="en"></dbp:years>
    <dbp:coronation rdf:datatype="W3#gMonthDay">--01-06</dbp:coronation
        >
    <dbp:coronation rdf:datatype="W3#gMonthDay">--04-05</dbp:coronation
        >
    <dbp:coronation rdf:datatype="W3#gMonthDay">--09-02</dbp:coronation
        >
    <dbp:coronation rdf:datatype="W3#gMonthDay">--11-26</dbp:coronation
        >
    <dbp:title rdf:resource="DBR/Holy_Roman_Emperor" />
    <dbp:title rdf:resource="DBR/Count_of_Luxembourg" />
    <dbp:title rdf:resource="DBR/King_of_Bohemia" />
    <dbp:title rdf:resource="DBR/King_of_the_Romans" />
  </rdf:Description>
<!-- (...) -->
</rdf:RDF>
```

Zdrojový kód D.3: Příklad RDF generovaným endpointem DBpedia

```
<!-- http://www.wikidata.org nahrazeno zkratkou "WD" -->
<!-- http://wikiba.se nahrazeno zkratkou "WBA" -->
<!-- http://www.w3.org/2001/XMLSchema nahrazeno zkratkou "W3" -->
<!-- Unikatni ID "WD/value/<ID>" byly zkraceny -->
<!-- Dotaz "WD/entity/statement/<statement>" byl zkracen -->
<!-- V casove znacce byla odebrana casova zona -->

<rdf:RDF>
  <rdf:Description rdf:about="WD/entity/statement/Q155669-F5...">
    <rdf:type rdf:resource="WBA/ontology#Statement"/>
    <rdf:type rdf:resource="WBA/ontology#BestRank"/>
    <wikibase:rank rdf:resource="WBA/ontology#NormalRank"/>
    <ps:P39 rdf:resource="WD/entity/Q95975116"/>
    <pq:P580 rdf:datatype="W3#dateTime">1346-01-01</pq:P580>
```

```
15      <pqv:P580 rdf:resource="WD/value/a7850f8127..."/>
16      <pq:P582 rdf:datatype="W3#dateTime">1353-01-01</pq:P582>
17      <pqv:P582 rdf:resource="WD/value/e6f2b82284..."/>
18      <pq:P1365 rdf:resource="WD/entity/Q155167"/>
19      <pq:P1534 rdf:resource="WD/entity/Q2719360"/>
20      <rdf:type rdf:resource="WD/prop/novalue/P1366"/>
21   </rdf:Description>
22  </rdf:RDF>
```

Zdrojový kód D.4: Příklad RDF generovaným endpointem Wikidata

E Uživatelská příručka

E.1 Spuštění programu

Aplikace je distribuována v ZIP souboru. Tento soubor rozbalte a otevřete vytvořený adresář (měl by se nazývat DataMining). Aplikace byla spouštěna a testována na platformě Windows ve verzích Windows 11 a Windows 10.

Adresář má následující strukturu:

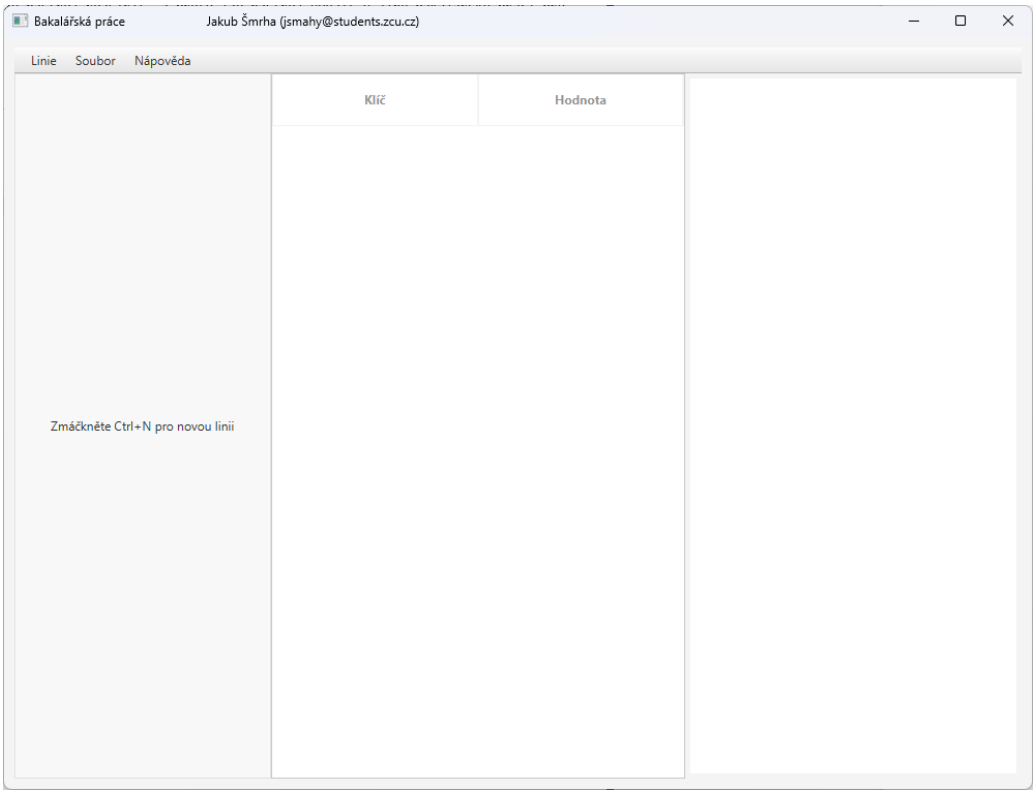
- `bin/`
- `frontend/`
- `start.bat`

V adresáři `bin/` se nachází aplikace. Knihovny, na kterých aplikace závisí, se nachází v adresáři `bin/libs/`. V adresáři `frontend/` se nachází front-endové implementace, přičemž se tam momentálně nachází implementace testovacího nástroje pro časovou osu od Bc. Michala Fialy. Pro spuštění aplikace stačí spustit soubor `start.bat`.

```
1 @echo off
2 start javaw -cp "bin/DataMining-1.0.jar;bin/libs/*" cz.zcu.jsmahy.
    datamining.Main2
```

Zdrojový kód E.1: Soubor `start.bat`

Skript E.1 pouze spouští aplikaci. Příkaz `start` spustí samostatné okno příkazového řádku pro spuštění aplikace. Příkaz `javaw` spustí samotnou aplikaci bez konzole. Parametr `-cp` tohoto příkazu pouze udává cestu k aplikaci a knihovnám. Parametr `cz.zcu.jsmahy.datamining.Main2` pouze udává třídu obsahující `main` metodu.



Obrázek E.1: Okno ve výchozím stavu

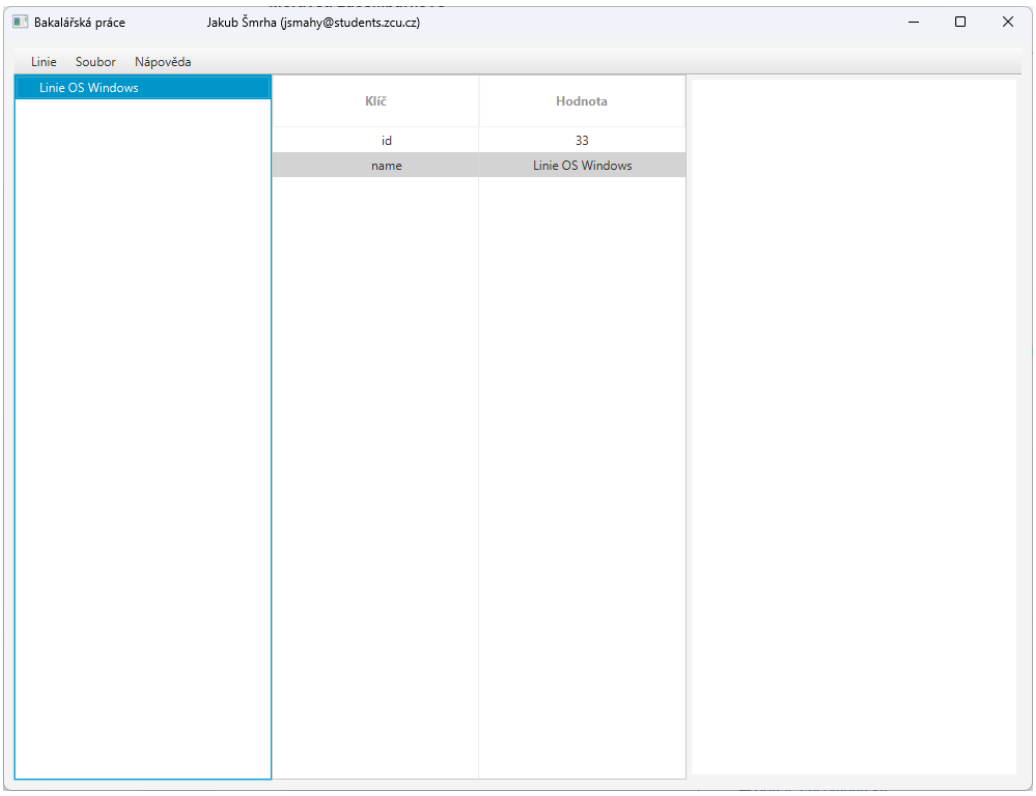
Po spuštění skriptu se zobrazí okno na obrázku E.1. Na levé straně, kde je zobrazen text „Zmáčkněte CTRL + N pro novou linii“, bude zobrazena linie ve stromové struktuře. Napravo od ní je tabulka s metadaty o konkrétním podmětu, která je nyní prázdná, jelikož nemáme zvolený žádný podmět. Na pravé straně je Webview, který je také prázdný ze stejného důvodu. Na horní liště můžeme vidět menu s třemi položkami

- Linie
- Soubor
- Nápověda

V položce „Linie“ se nachází práce s linií, jako je např. tlačítko pro vytvoření linie či tlačítko pro zobrazení linie. V položce „Soubor“ se nachází importování a exportování linie. V položce „Nápověda“ se nachází návod, jak pracovat s tímto nástrojem.

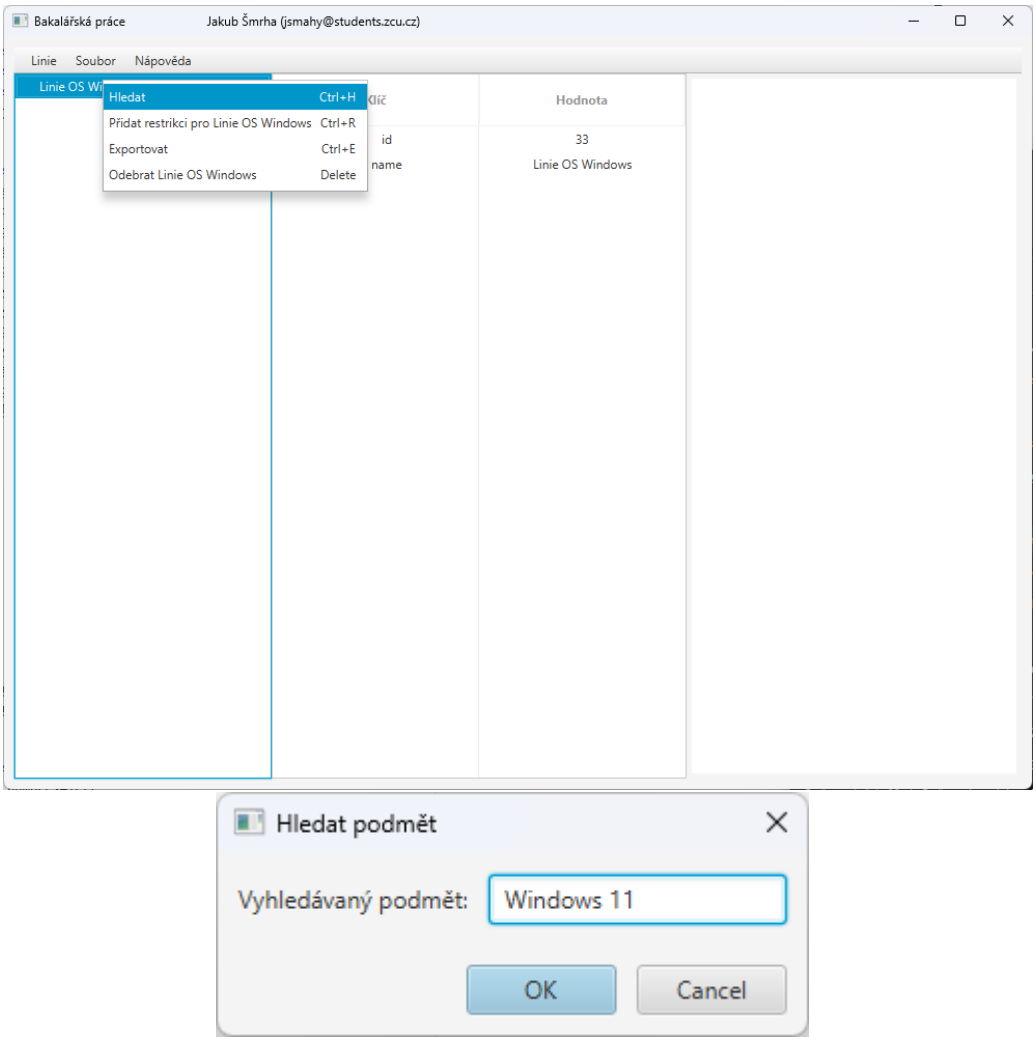
E.2 Tvorba linie

Uvedeme si modelový případ tvorby linie. V tomto příkladu budeme tvořit linii operačního systému Windows. Na tomto příkladu si ukážeme všechny problémy při tvorbě linie, mezi které patří řešení nejednoznačnosti linie a nevalidní data v dané bázi znalostí, konkrétně bázi znalostí DBpedia, která nástroj v této verzi nedokáže filtrovat.



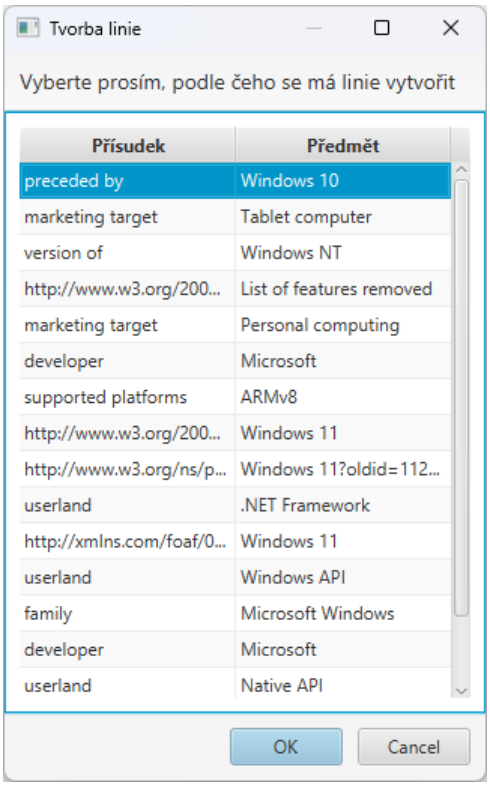
Obrázek E.2: Vytvoření linie

Obrázek E.2 zobrazuje kořen stromu „Linie OS Windows“ – název linie, který jsme vytvořili pomocí klávesové zkratky „CTRL + N“, které nám zobrazí vyskakovací okno, a následné napsání „Linie OS Windows“ do textového pole. Tento název lze kdykoliv modifikovat v tabulce napravo dvojím kliknutím na „Linie OS Windows“.



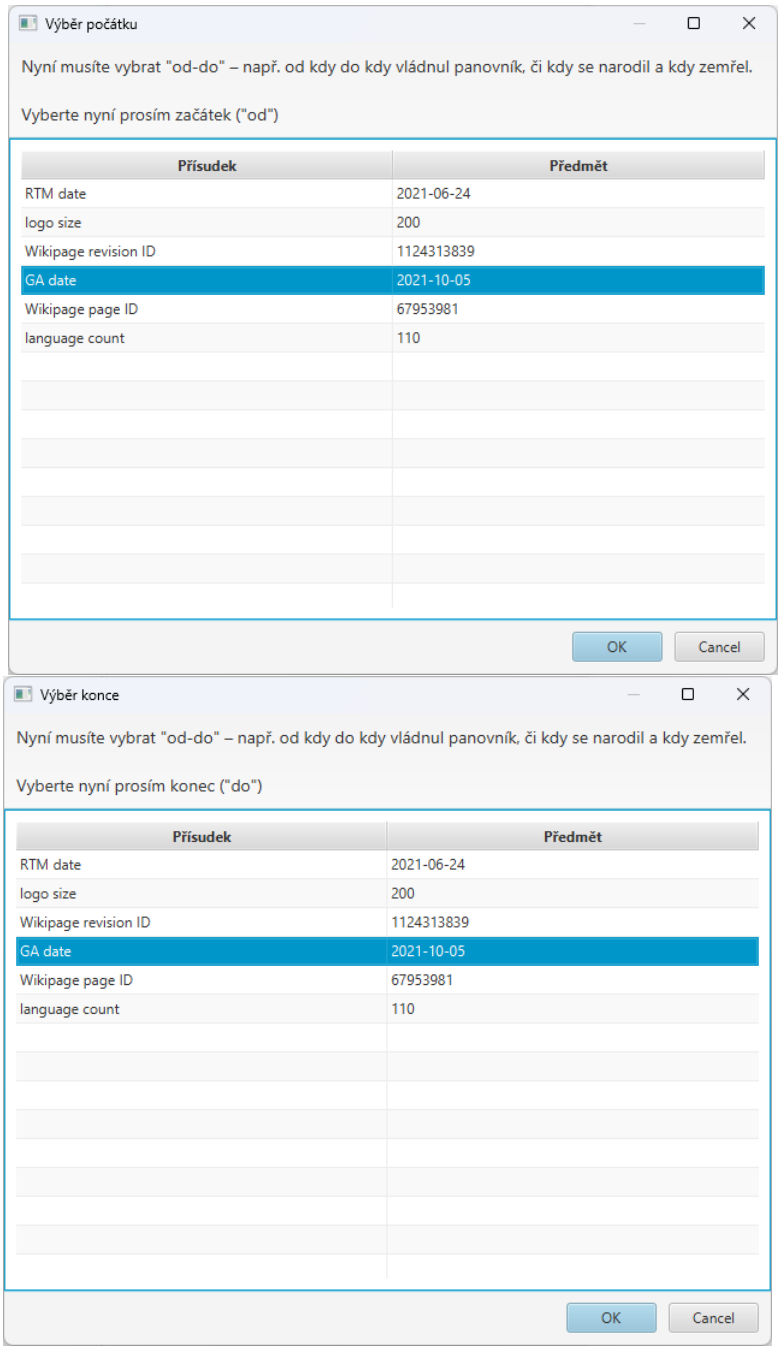
Obrázek E.3: Tvorba linie: 1. krok

Obrázek E.3 zobrazuje počátek tvorby linie. Pomocí klávesové zkratky „CTRL + H“, nebo pravým kliknutím na název linie v stromové struktuře a kliknutím na „Hledat“ se zobrazí vyskakovací okno s titulkem „Hledat podmět“. V tomto okně se nachází textové pole, do kterého zadáme počátek linie. Jelikož nás zajímá linie OS Windows, začneme nejnovější verzí „Windows 11“.



Obrázek E.4: Tvorba linie: 2. krok

Poté musíme zvolit, jak se bude linie tvořit – podle jakého přísudku. Na obrázku E.4 máme na výběr mnoho přísudků, přičemž velké množství bylo již vyfiltrováno. Zde se nabízí hledat podle předešlé verze, tj. „preceded by“. Předmět „Windows 10“ napovídá, že dalším prvkem v linii bude „Windows 10“, tj. naše linie bude v minimální podobě vypadat „Windows 11 — Windows 10“. Předpokládáme, že po tomto prvku bude následovat verze OS „Windows 8.1“ nebo „Windows 8.0“. Přísudek „preceded by“ se zdá být nejlepším kandidátem.



Obrázek E.5: Tvorba linie: 3. krok

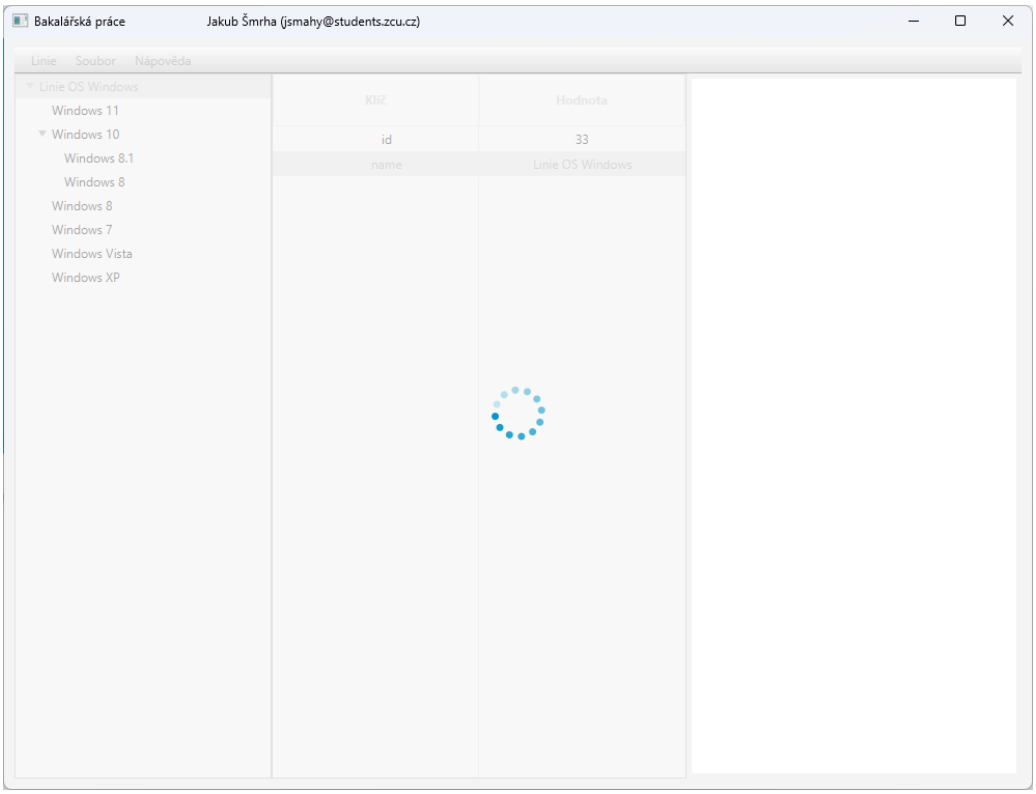
Následně musíme vybrat časové údaje o počátku a konci jednotlivých podmětů, viz obrázek E.5. Např. bychom chtěli znát, kdy byla verze OS vydaná. Přísudek „GA date“ (datum obecné dostupnosti) vypadá jako nejlepší kandidát. Dále bychom chtěli znát, kdy byla tato verze nahrazena novější verzí. Protože se jedná o nejnovější verzi, nemáme o tomto datu žádné údaje. Spokojíme se tedy pouze se začátkem časového údaje. Tento

případ, kdy začátek a konec mají totožný přísudek, resp. nemáme zvolený konec, nazýváme tzv. „momentem“. Front-end knihovně předáváme pouze jeden časový údaj, může tedy tento případ zobrazit jinak, než čárou mezi dvěma časovými údaji – např. bodem. V tomto případě zvolíme jako konec stejný přísudek, abychom ukázali, jak přibližně vypadá čára mezi začátkem a koncem.

Obrázek E.6: Tvorba linie: řešení nejednoznačnosti

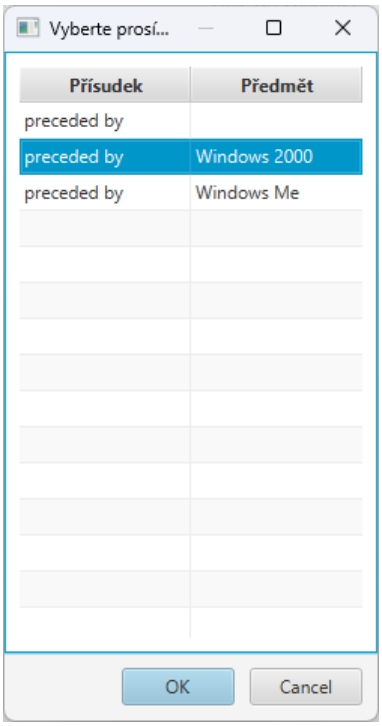
Nyní máme všechno nastavení tvorby linie hotové, nástroj se může pustit do vyhledávání. Nástroj úspěšně vytvoří část linie „Windows 11 — Windows 10“. Nicméně, u „Windows 10“ neví, jak má dále pokračovat, jelikož má na výběr z dvou možností – „Windows 8.1“ a „Windows 8“. V tomto kroku se dotáže uživatele, viz obrázek E.6. Tomu jsou představeny všechny možnosti, podle kterých může linie pokračovat. Nicméně se může také stát, že linie tímto prvkem skončí, jelikož se už nejedná o odkaz, ale literál. Vybereme verzi „Windows 8“[1].

[1]Pokud bychom zvolili verzi „Windows 8.1“, linie by skončila, jelikož DBpedia má tuto verzi uloženou jako literál. Nicméně, mohli bychom dále pokračovat verzí „Windows 8.0“ poté, co zvolíme „Windows 8.1“. Později si toto u obrázku E.8 opět ukážeme.



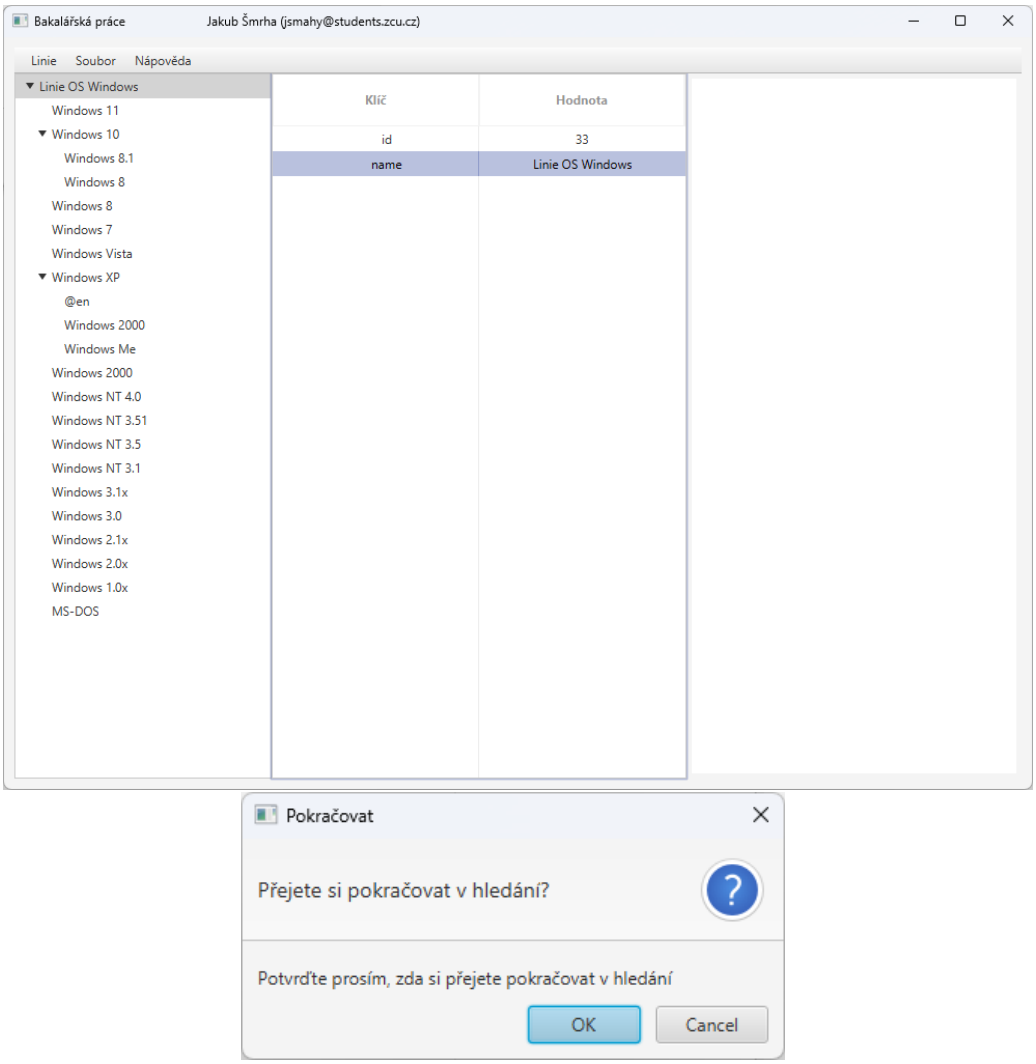
Obrázek E.7: Tvorba linie: vyhledávání

Nástroj pokračuje ve vyhledávání jako na obrázku E.7. Můžeme si také všimnout dvou předmětů pod prvkem „Windows 10“. Jedná se o všechny možnosti, z kterých jsme měli na výběr. Pokud se uživatel splete, může se kdykoliv rozhodnout linii vytvořit odznova z daného bodu – tj. „Windows 8.1“ nebo „Windows 8“.



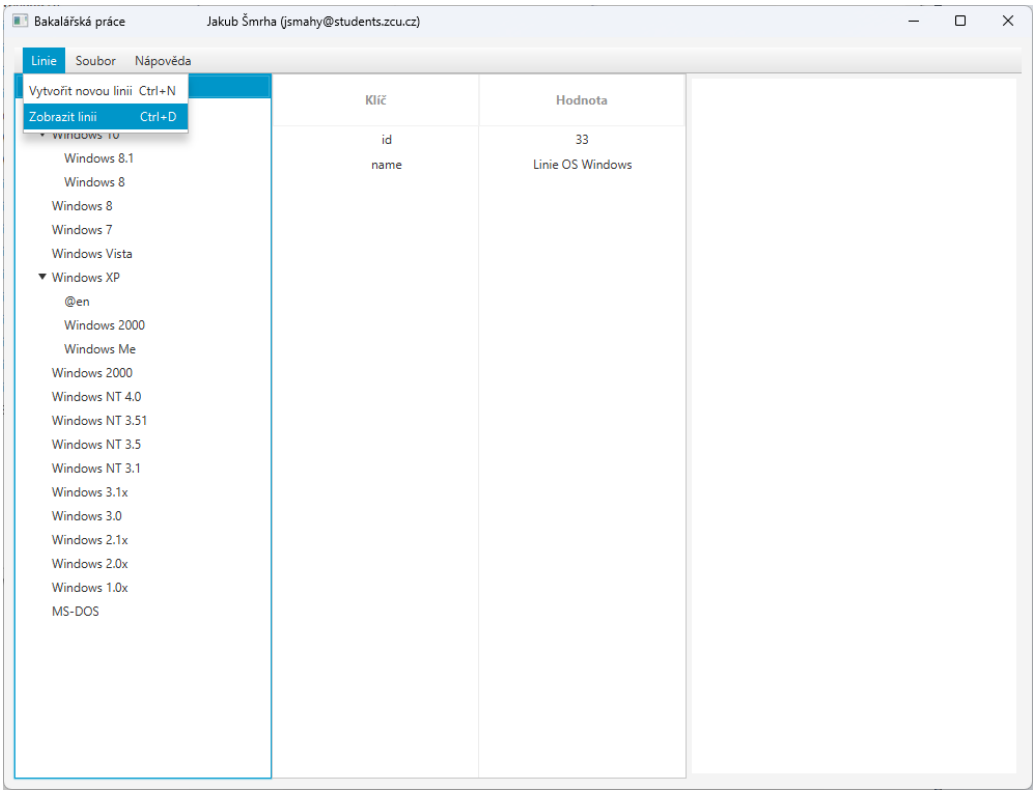
Obrázek E.8: Tvorba linie: řešení nejednoznačnosti 2

Nástroj si opět není nejistý, jak má pokračovat, jakmile narazí na „Windows XP“. Nyní máme tři možnosti – prázdný řetězec, „Windows 2000“ a „Windows Me“. Bohužel, zbavit se nesprávných dat, jako je prázdný řetězec, je obtížné. Nástroj raději zobrazuje všechna nalezená data. Pokud bychom zvolili prázdný řetězec, linie samozřejmě skončí. Zvolíme nyní „Windows 2000“. Později si ukážeme na obrázku E.12, jak by linie pokračovala, pokud bychom zvolili „Windows Me“.



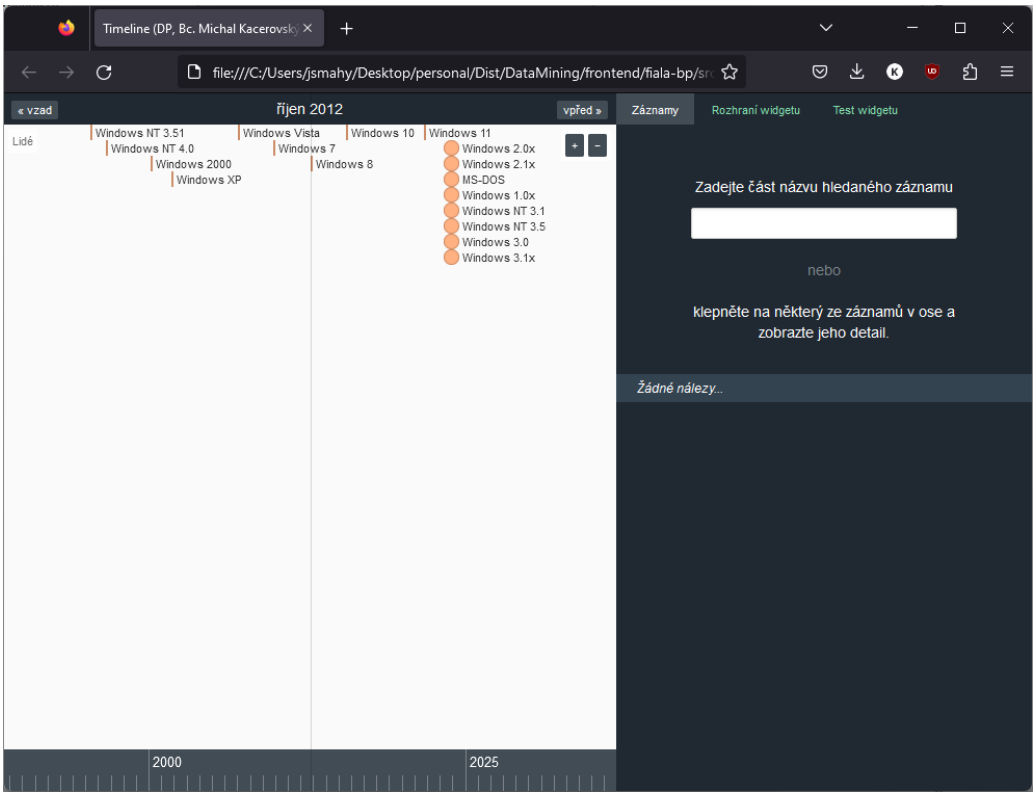
Obrázek E.9: Tvorba linie: konec

Nástroj úspěšně dokončil linii a zobrazil vyskakovací okno, zda chceme pokračovat ve vyhledávání. Nástroj se nás táže proto, že si není jistý, zda se opravdu jedná o konec linie. Pokud bychom zvolili, že chceme pokračovat, zobrazí se nám vyskakovací okno podobné na obrázku E.3 na straně 67. Poté bychom opět napsali podmět, zvolili přísudek a linie by pokračovala. Ve stromové struktuře můžeme vidět celou linii od „Windows 11“ po „MS-DOS“. Každá entita má v sobě časové údaje „od–do“ a další metadata pro zjednodušení zobrazení.



Obrázek E.10: Tvorba linie: zobrazení

Nyní bychom rádi linii zobrazili na časové linii. Můžeme tak udělat pomocí klávesové zkratky „CTRL + D“, nebo kliknutím na „Linie — Zobrazit linii“ jako na obrázku E.10.

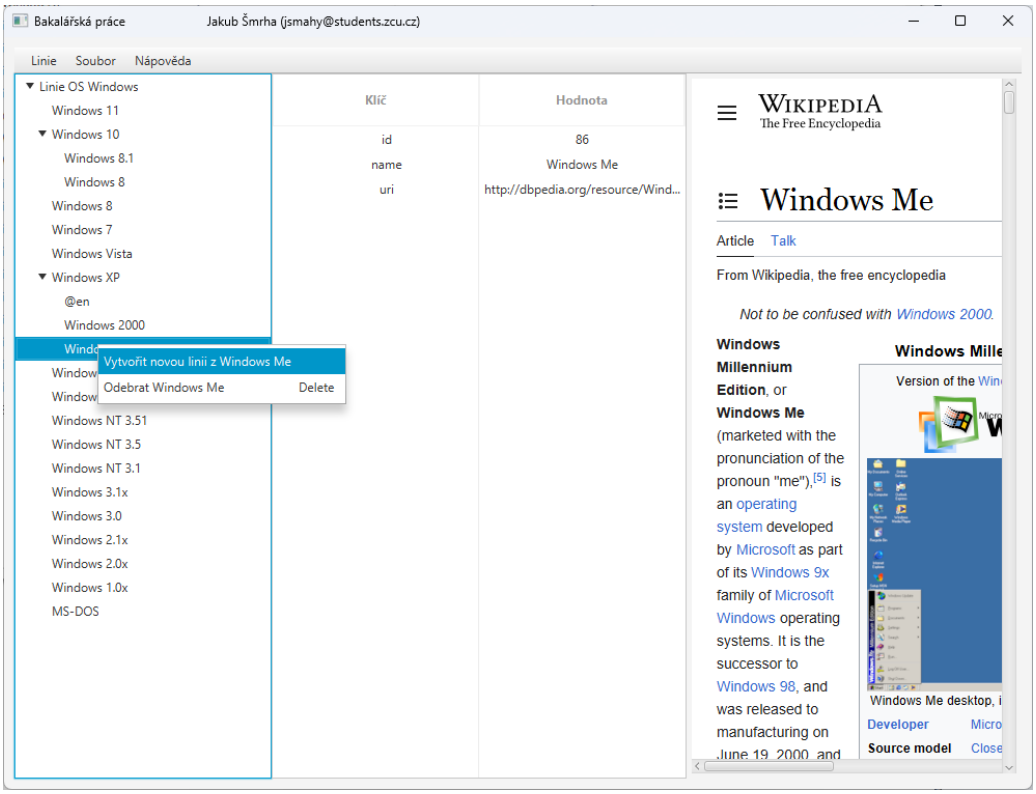


Obrázek E.11: Tvorba linie: zobrazení 2

Poté, co klikneme na „Zobrazit linii“, se nám otevře výchozí prohlížeč, viz obrázek E.11. V prohlížeči se nám zobrazí časová linie. Na časové linii můžeme vidět, že „Windows 2.0x“ až „Windows 3.1x“ jsou zobrazeny úplně napravo místo toho, aby se zobrazili správně na časové linii. Tento jev nastane tehdy, když nástroj nedokáže najít časové údaje pro danou entitu. Nicméně, ostatní OS mají správné časové údaje a jsou zobrazeny na časové linii[2].

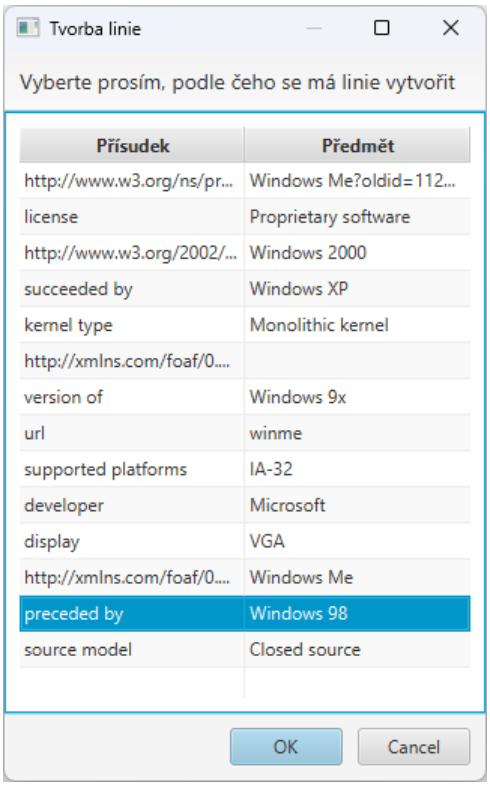
Kvůli kompresi velikosti vypadá obrázek E.11 zkresleně, proto jednotlivé verze OS nejsou úplně zobrazeny v jedné linii, jelikož by se překrýval jejich název. Po zvětšení okna prohlížeče by jednotlivé OS byly zobrazeny na jedné lince. Nyní si ještě ukážeme slíbenou možnost tvorby linie z „Windows Me“.

[2]Pokud byste chtěli tyto časové údaje přidat, jste v momentální verzi nástroje nuceni zasahovat do souboru data.js a přidat tato data podle formátu v ostatních entitách v časové linii. Nástroj nepodporuje přidání jakýchkoliv dat, nicméně je umožňuje modifikovat.



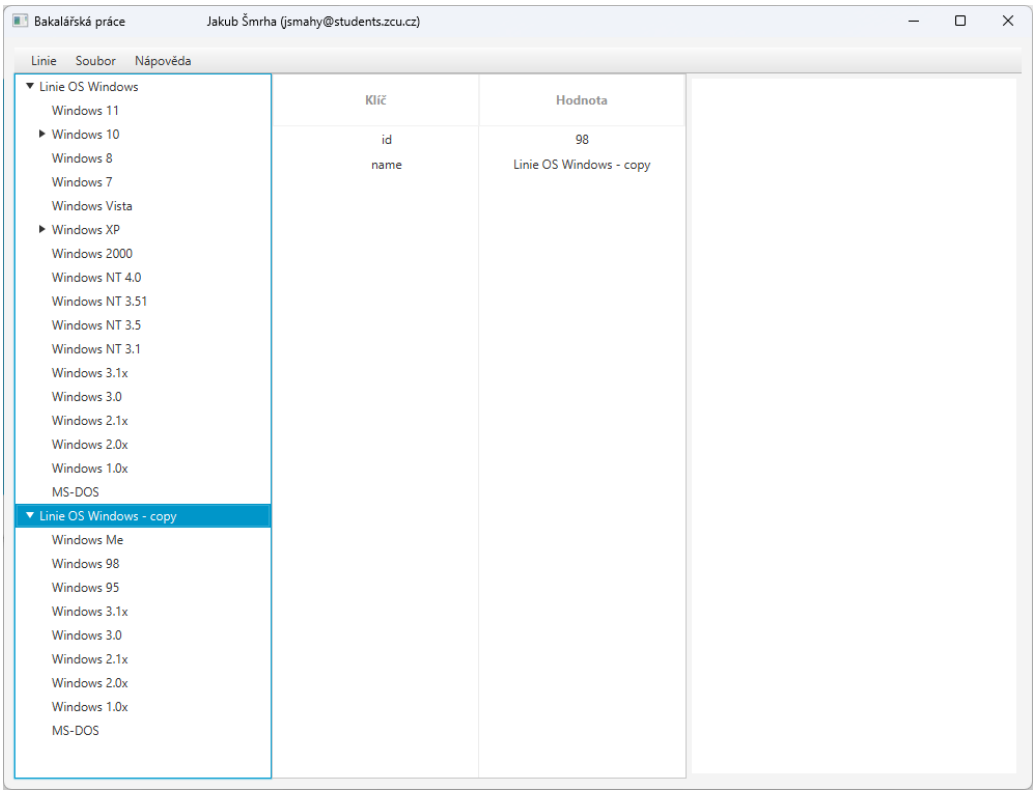
Obrázek E.12: Tvorba linie: „Windows Me“

Klikneme pravým tlačítkem v stromové struktuře na „Windows Me“ jako na obrázku E.12. Poté zvolíme „Vytvořit novou linii z Windows Me“.



Obrázek E.13: Tvorba linie: „Windows Me“ 2

Opět budeme muset zvolit přísudek, viz obrázek E.13, a časové údaje jako na obrázku E.5 na straně 69.



Obrázek E.14: Tvorba linie: konec 2

Nástroj úspěšně nalezne všechny předcházející verze Windows a dokončí linii. Na obrázku E.14 můžeme vidět dvě paralelní linie – první linie, která začíná „Windows 11“ a druhá linie vytvořená před chvílí, která začíná „Windows Me“. Obě linie končí na správné verzi Windows – „MS-DOS“.

F Programátorská příručka

Jak bylo v textu zmíněno, nástroj disponuje aplikačním rozhraním. Pokusili jsme se vytvořit API tak, aby bylo co nejjednodušší jej rozšířit. Nicméně, API není zcela hotové a musí se stále zasahovat do zdrojového kódu, jak si ukážeme později.

```
1 public class MyCustomModule extends AbstractModule {
2     @Override
3     protected void configure() {
4         // TODO: Tuto metodu pozdeji implementujeme
5     }
6 }
```

Zdrojový kód F.1: Příklad implementace `AbstractModule`

Prvně, co musíme udělat, je rozšířit třídu `AbstractModule` jako ve zdrojovém kódu F.1. Tato třída leží v balíku `com.google.inject`. Do metody `configure` můžete vkládat jakékoliv své implementace tříd, které budou později použity pro injekci závislostí. Do této metody později přidáme jednu závislost, kterou nástroj požaduje, aby správně fungoval.

```
1 (...)
2 final class SceneManager {
3     public static final String FXML_PATH = "/fxml/";
4     private static final Logger LOGGER = LogManager.getLogger(
          SceneManager.class);
5     private static final String FXML_SUFFIX = ".fxml";
6     private static final Module[] MODULES = new Module[] {
7             new DataMiningModule(),
8             new DBPediaModule(),
9             new MyCustomModule() // Add our custom module
10    };
11    (...)
12 }
```

Zdrojový kód F.2: Přidání vlastního modulu

Tuto třídu musíme nyní, bohužel, manuálně přidat do třídy `SceneManager`, která leží v balíku `cz.zcu.jsmahy.datamining`, viz zdrojový kód F.2 na řádce 9.

```
public interface RequestProgressListener {
    // Properties
    ObjectProperty<Property> ontologyPathPredicateProperty();
    ObjectProperty<Property> startDateProperty();
    ObjectProperty<Property> endDateProperty();
    ObjectProperty<TreeItem<DataNode>> treeRootProperty();
    ObjectProperty<QueryData> queryDataProperty();

    // Callbacks
    void onAddRelationship(DataNode previousDataNode, DataNode
        currentDataNode);
    void onAddNewDataNodes(List<DataNode> dataNodes);
    void onDeleteDataNodes(Collection<DataNode> deletedNodes);
    void onInvalidQuery(String query, InitialSearchResult result);
    void onSearchDone(DataNode dataNodeRoot);
    void onCreateNewRoot(DataNode newDataNodeRoot);
    void onDisplayRequest(DataNode dataNodeRoot, WebView webView, File
        topLevelFrontendDirectory);
}
```

Zdrojový kód F.3: Rozhraní posluchače

```
public class MyCustomRequestProgressListener implements
    RequestProgressListener {
    // TODO: Implement methods
}
```

Zdrojový kód F.4: Implementace posluchače

Dále musíme implementovat posluchač na události během tvorby linie. Posluchač událostí vyžaduje implementace ve zdrojovém kódu F.3, přičemž musí jednak obsahovat různé vlastnosti `javafx.beans.property.ObjectProperty`, jednak implementovat callbacky, které budou volány během tvorby linie. Všechny tyto metody jsou zdokumentovány tak, aby bylo jasné, kdy se každý callback použije. Uvnitř této třídy byste měli přidávat vytvořené instance `DataNode` do stromové struktury, anebo jakékoliv jiné vaší zvolené struktury, a zobrazit je uživateli. Nástroj ve výchozím stavu používá stromovou strukturu, nicméně jádro nástroje pracuje pouze s daty a uživatelským vstupem, není nijak na vizualizaci ani GUI závislé.

```
1 public class MyCustomModule extends AbstractModule {
2     @Override
3     protected void configure() {
4         bind(RequestProgressListener.class)
5             .to(MyCustomRequestProgressListener.class)
6             .asEagerSingleton();
7     }
8 }
```

Zdrojový kód F.5: Přidání závislosti

Nyní jako ve zdrojovém kódu F.5 přidáme tuto třídu do již zmíněné metody `configure` uvnitř `MyCustomModule` jako závislost. Implementace `RequestProgressListener` je převážně použita v `SparqlEndpointTask`, konkrétně `DBPediaEndpointTask`. Na řádce 6 říkáme, že chceme tuto třídu registrovat jako tzv. singleton. Guice by jinak při každé injekci závislostí vytvářel novou instanci posluchače a to je nežádoucí, jelikož uvnitř posluchače si uchováváme určitý stav (viz metody, které vrací instanci `ObjectProperty`).

To je vše! Pokud si nevíte rady s implementací, v (ne zrovna dobře pojmenovaném) balíku `export` je napojení na již zmíněnou front-end knihovnu pro tvorbu časových linií, ve které se nachází různé implementace, mezi které patří `RequestProgressListener`, `AbstractModule` a další věci ohledně exportování dat.