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

Diplomová práce

Zpracování časových údajů pro jejich vizualizaci

Plzeň 2015 Bc. David Hrbáček

Prohlášení

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

V Plzni dne 21. června 2015

Bc. David Hrbáček

Abstract

Visualization oriented time data processing

The historical records are usually visualized using a timeline or a graph, however a large and complex set of such data can be hardly displayed at once. This diploma thesis deals with design and implementation of historical events ranking tool that provides individual record importance for visualization applications needed to display the data in a clear and efficent way. It describes several algorithms for importance ranking of graph vertices. In detail, it analyses the PageRank algorithm and its modifications. The analysis of several existing Java libraries which enables representation of graph is a part of the diploma thesis. The next part describes implementation of own graph library and library for importance ranking of graph vertices using different PageRank algorithm types. The last part of implementation enables communication between database and visualization layers of the final application and provides its services via Java and REST interface. The thesis also deals with measuring required time and memory of implemented PageRank algorithm types and analyzing appropriate value of terminating condition used in this algorithm.

Zpracování časových údajů pro jejich vizualizaci

Při vizualizaci rozsáhlých historických záznamů pomocí časové osy či grafu nelze uživateli prezentovat všechna data najednou. Tato práce se zabývá návrhem a implementací takového nástroje pro ohodnocení historických událostí, který poskytuje vizualizačním nástrojům informace o důležitosti jednotlivých záznamů, jež povede k zpřehlednění zobrazovaných dat. Práce popisuje několik algoritmů pro důležitostní ohodnocení uzlů grafu. Podrobně pak analyzuje algoritmus PageRank a jeho modifikace. Součástí práce je analýza několika Java knihoven umožňujících reprezentaci grafu. Dále pak implementace vlastní grafové knihovny a knihovny pro ohodnocování vrcholů grafu pomocí jednotlivých typů algoritmu PageRank. Poslední součástí implementační části jsou knihovny umožňující komunikaci mezi databázovou a vizualizační vrstvou výsledné aplikace, které poskytují své funkce prostřednictvím Java a REST rozhraní. Práce se také věnuje měření potřebného času a paměti implementovaných typů algoritmu PageRank a analýze vhodné hodnoty zastavovací podmínky tohoto algoritmu.

Poděkování

Rád bych poděkoval svému vedoucímu práce Ing. Richardu Lipkovi, Ph.D., za cenné rady, výbornou spolupráci a konstruktivní poznámky, které mě vždy navedly správným směrem. Dále Michalu Kacerovskému za pomoc při testování, gramatické korekce a morální podporu. V neposlední řadě patří mé díky rodině za podporu při studiu.

Obsah

1 Úvod **1**

2 Grafové algoritmy **2**
2.1 Graf historických událostí 2
2.1.1 Význam uzlů a hran 3
2.1.2 Práce s grafem 4
2.2 Metriky ohodnocení důležitosti uzlů 5
2.2.1 Centralita grafu 5
2.2.2 Ohodnocení uzlů grafu pomocí sady kritérií 5
2.2.3 Ohodnocení uzlů grafu pomocí algoritmu PageRank 6
2.2.4 PageRank s uživatelsky přednastavenými prioritami 9

3 Knihovny implementující graf **13**
3.1 Grph 14
3.2 JUNG 15
3.3 GraphStream 16
3.4 JGraphT 17
3.5 Navrhovaná knihovna 18
3.5.1 Vlastnosti navrhované knihovny 19

4 Návrh knihovny pro reprezentaci a ohodnocení grafu **20**

5 Implementace grafové knihovny **23**
5.1 Knihovna `graph` 23
5.1.1 Entitní třídy grafu historických událostí 25
5.2 Knihovna `graph-ranking` 27

6 Implementace PageRanku **30**
6.1 Výpočet pomocí maticových počtů 30
6.2 Výpočet pomocí seznamu sousednosti 31
6.2.1 Implementace 32
6.3 PageRank s prioritními hranami 32
6.3.1 Správa uživatelských priorit 32
6.3.2 Výpočet 34

6.3.3 Implementace 35
6.4 PageRank s maticí teleportačních konstant 35
6.4.1 Sestavení matice **G** 35
6.4.2 Výpočet 37

7 Knihovna pro vnější přístup **38**

8 REST server **42**
8.1 Návrh REST rozhraní 42
8.1.1 Získání ohodnoceného grafu na základě uživatelského dotazu 42
8.1.2 Získání názvů všech vlastností 44
8.1.3 Získání uzlu podle jeho ID 44
8.1.4 Vkládání uzlů nebo hran 44
8.1.5 Zpracování chybových stavů 46
8.2 Implementace REST serveru 47
8.2.1 Nastavení serverové aplikace 47
8.2.2 Struktura implementace 48

9 Výsledky a tesování **51**
9.1 Ovlivnění důležitostí uzlů uživatelskými prioritami 51
9.2 Výpočetní složitost jednotlivých typů PageRanku 52
9.3 Paměťová náročnost jednotlivých typů PageRanku 53
9.4 Stanovení zastavovací podmínky ε 55
9.5 Testování implementovaných knihoven 55

10 Závěr **57**

11 Přehled použitých zkratek **58**

A Generátor grafu historických událostí **61**

B Historický graf Domažlic **63**

1 Úvod

V současné době má každý člověk díky internetu snadný přístup k nejrůznějším informacím. Stačí zadat do vyhledávače požadované téma a většinou se nám zobrazí vše, co s ním souvisí. Problémem však obvykle bývá zdlouhavý proces nalezení relevantních informací mezi změtí všech dat, kterÁ byla nalezena. Vyhledávače také většinou vracejí nalezené výsledky v lineární struktuře, která sice může být vhodná pro velký počet typů informací, ale některá data jsou však pro člověka více srozumitelná v jiných strukturách, například ve formě grafu.

Jedním z příkladů reprezentace dat formou grafu jsou informace o historických událostech. Pokud totiž lze mezi událostmi najít vztahy jako je příčina a následek, účast na události nebo místo ke kterému se vztahuje, můžeme reprezentovat události grafovou strukturou. Zjednodušeným příkladem takovýchto grafů mohou být královské rodokmeny, které jsou zároveň i jednou z prvních realizací prezentování informací formou grafu. Přirozeným způsobem vizualizace historických událostí je pak časová osa, v níž navíc můžeme zasadit jednotlivé události, osobnosti a místa do časového kontextu.

Problém v tomto případě může představovat vizualizace rozsáhlého grafu s velkým počtem vrcholů a hran, kdy není možné jednotlivé vrcholy přehledně uspořádat pro současné zobrazení na časové ose. Takovouto situaci řeší například postup, kdy zobrazíme jen ty vrcholy, které jsou významné, a ty jež jsou méně významné, zobrazíme až při detailnějším pohledu.

Jak ale rozpoznat významné vrcholy od nevýznamných? V odpovědi na tuto otázku nám může pomoct samotná struktura analyzovaného grafu. I znalost významu jednotlivých typů vrcholů, respektive hran a jejich od toho se odvíjející uživatelská preference, velmi často představuje výrazné usnadnění.

Tato diplomová práce popisuje několik algoritmů provádějících důležitostní ohodnocení jednotlivých uzlů grafu na základě jeho topologie a uživatelských priorit za cílem přispění k efektivnímu zobrazování grafu historických událostí. V jejím rámci bude podmnožina popsaných algoritmů také implementována a otestována na zmíněném grafu historických událostí tak, aby údaje o důležitostech jednotlivých uzlů mohly být snadno zpracovány různými vizualizačními nástroji.

2 Grafové algoritmy

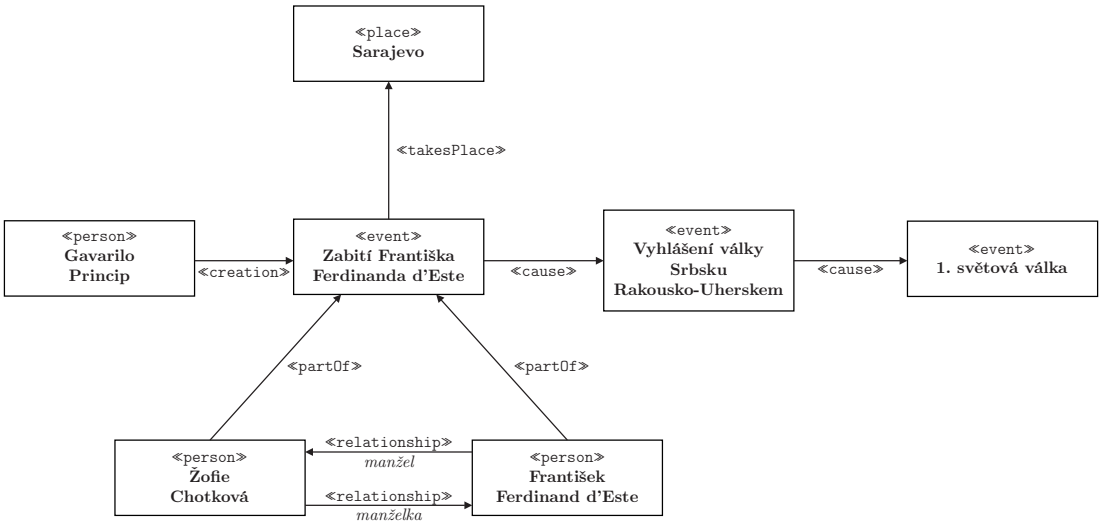
Tato kapitola popisuje různé algoritmy, pomocí nichž můžeme ohodnotit důležitost jednotlivých vrcholů v grafu.

Speciálně v případě této diplomové práce se jedná o graf historických událostí. Proto nejprve níže popíšeme strukturu a význam jednotlivých vrcholů, respektive hran tohoto grafu.

2.1 Graf historických událostí

Graf, jehož uzly mají být prioritně ohodnoceny, reprezentuje historické události, místa, osobnosti a vazby mezi nimi. Vybrané uzly tedy mohou představovat například události a hrany, které je spojují, jejich návaznosti. Hrany grafu jsou tudíž orientované. Daná struktura grafu byla při zadávání této diplomové práce již pevně stanovena.

Ukázku takovéhoto grafu lze vidět na obrázku 2.1, který popisuje základní události a osoby spojené s vypuknutím 1. světové války.



Obrázek 2.1: Historický graf zobrazující hlavní události a osoby spojené s vypuknutím 1. světové války

Jak je na obrázku vidět, jednotlivé události a hrany jsou označeny stereotypy, které určují o jaký druh vrcholu, respektive hrany, se jedná (všechny typy hran a uzlů jsou popsány níže v podkapitole 2.1.1).

Stereotypy hran také mohou určovat hierarchii uzlů, například hrana se stereotypem `partOf` udává, že vrchol ze kterého tato hrana směřuje, je součástí

(podřazeným vrcholem) jiného vrcholu. To lze vidět i na obrázku 2.1, kde je vrchol osoby Františka Ferdinanda d'Este pouze součástí vrcholu reprezentujícího událost atentátu na něj. Analogický vztah můžeme hledat mezi určitou válkou a například bitvami, které byly její součástí.

Na obrázku 2.1 lze také vidět, že i graf, který znázorňuje historické události, nemusí být acyklický, jelikož nezobrazuje pouze posloupnost na sebe navazujících událostí, ale také vztahy mezi jednotlivými vrcholy (které většinou reprezentují osoby).

2.1.1 Význam uzlů a hran

Uzly grafu jsou rozděleny do několika kategorií, které označujeme jako stereotypy:

- `person` – představující historickou osobnost (např. Karel IV.),
- `event` – reprezentující historickou událost (např. započetí stavby hradu Karlštejn),
- `place` – místo, ke kterému se mohou vztahovat události (např. Karlštejn),
- `item` – věc historického významu (např. korunovační klenoty).

Všechny uzly samozřejmě obsahují své datování a mohou nést své další specifické vlastnosti.

Hrany mají také své stereotypy:

- `relationship` – vztah mezi dvěma osobami nebo místy,
- `interaction` – spolupráce mezi dvěma osobami,
- `participation` – účast osoby nebo místa na určité události,
- `creation` – vytvoření nějaké události nebo místa (tento stereotyp by měla např. hrana mezi uzly Karel IV. a založení Univerzity Karlovy),
- `cause` – příčina jiné události,
- `partOf` – pokud je uzel součástí nějaké události,
- `takesPlace` – vztah uzlu k nějakému místu.

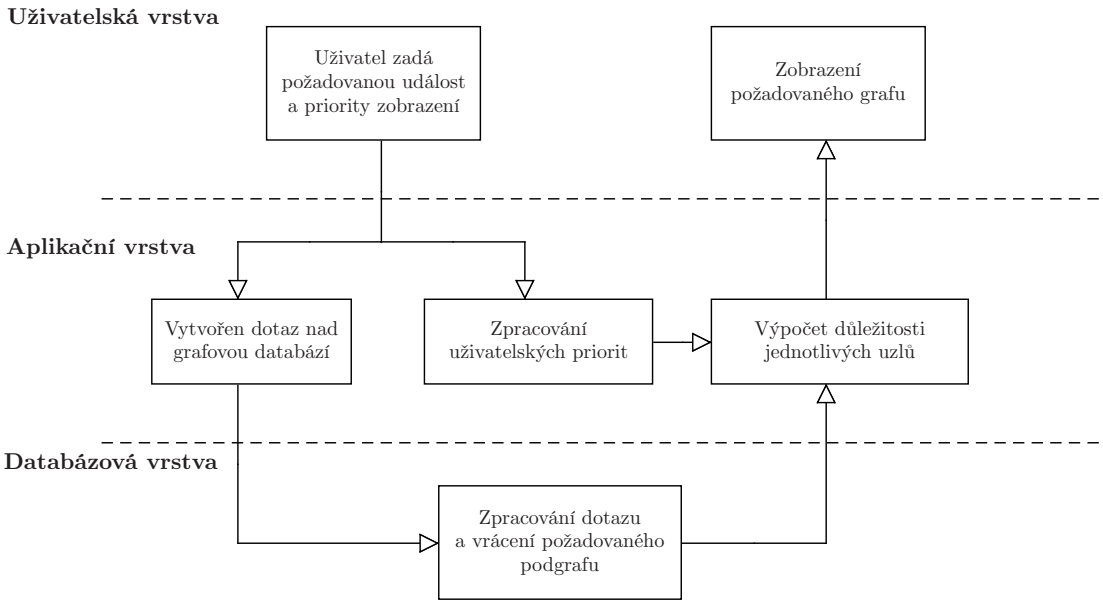
Hrany mohou také nést další údaje, například druh vztahu mezi dvěma osobami (rodičovství, manželství, ...).

2.1.2 Práce s grafem

Vzhledem k možné celkové obsáhlosti grafu historických událostí musí být nejprve vybrán podgraf, kterým se uživatel chce zabývat.

Uživatel také může určit, kterým typům uzlů a hran dává větší přednost při zobrazení nebo které chce naopak potlačit. Toho lze docílit pomocí přiřazení priorit k jednotlivým stereotypům uzlů a hran.

Celý mechanismus výběru je stručně popsán následujícím diagramem:



Obrázek 2.2: Diagram zobrazující akce jednotlivých vrstev výsledné aplikace pro zobrazování historických událostí

Jak z diagramu vyplývá, výpočet důležitosti jednotlivých uzlů je vždy prováděn jen pro podgraf, který uživatele právě zajímá. Samotný výběr požadovaného podgrafu a výpočet priorit uzlů nesmí trvat dlouho dobu, aby nebyl výpočet pro uživatele příliš zdlouhavý.

Stanovení uživatelských priorit může být zadáno ohodnocením důležitosti jednotlivých stereotypů hran a uzlů. Například pokud by uživatele zajímaly vztahy historických osobností, nastaví vysokou prioritu uzlům se stereotypem `person` a hranám `relationship`.

Je nutné také zdůraznit, že tato práce se věnuje pouze realizaci aplikační vrstvy. Ta bude komunikovat s okolními vrstvami pomocí různých rozhraní (podrobněji v kapitole 4).

Databázová a uživatelská vrstva je pak realizována v rámci jiných diplomových prací.

2.2 Metriky ohodnocení důležitosti uzlů

V této kapitole bude využíváno pojmů z teorie grafů, které jsou popsány například v [1].

2.2.1 Centralita grafu

V teorii grafů udává centralita důležitost jednotlivých uzlů, ze kterých se graf skládá. Centralita se hojně využívá například při analýzách sociálních sítí, jejichž pomocí se hledají vlivní jedinci. Existuje několik druhů centralit, mezi základní pak patří následující [5]:

Degree centrality C_D Tato centralita je nejstarší a zároveň jednoduchá na výpočet. Důležitost uzlu je dána jeho stupněm, tedy čím více má uzel sousedů, tím vyšší je jeho důležitost. Výpočetní složitost této metriky pro všechny uzly grafu je $O(n^2)$ v závislosti na způsobu jeho reprezentace, kde n udává počet uzlů.

Closeness centrality C_C V souvislém grafu udává vzdálenost pozorovaného uzlu od ostatních. C_C uzlu je dána převrácenou hodnotou součtu nejkratších vzdáleností z pozorovaného uzlu do všech ostatních.

Tato centralita má již relativně vysokou výpočetní složitost $O(nm + n^2)$ pro nevážené grafy, resp. $O(nm + n^3)$ pro vážené grafy[2], kde n udává počet uzlů a m počet hran.

Betweenness centrality C_B Udává počet nejkratších cest mezi dvěma různými uzly, na kterých leží pozorovaný uzel.

Například v síti, která reprezentuje dopravní model města, kde uzly jsou křižovatky, by uzly s vysokou C_B byly důležité spojnice města s pravděpodobně vysokou vytížeností.

Výpočet C_B má výpočetní složitost $O(n^3)$.

2.2.2 Ohodnocení uzlů grafu pomocí sady kritérií

Jednotlivé centrality se také dají zkombinovat do multikriteriálního algoritmu ohodnocení uzlů grafu, kde se podle sady pravidel vyhodnocuje jejich důležitost. Ve vědeckém článku [3] je například zmiňována následující sada:

1. Pokud nemůžeme rozlišit dva uzly v síti, můžeme říci, že mají shodnou důležitost. Pokud pozice uzlu a je rovna pozici uzlu b, pak je jejich důležitost také rovna (topologická ekvivalence uzlů – $N \setminus \{a\} \equiv N \setminus \{b\}$).

2. Uzel s větším stupněm má větší vliv na graf, tím pádem je více důležitý (degree centralita).

3. Pokud uzel spojuje dvě nebo více komunit (podgrafů), je klíčovým, protože řídí mezi těmito podgrafy komunikaci. Takovouto metriku udává betweenness. Pokud uzel a má větší betweenness než uzel b, pak je více důležitý.

4. Uzel, který je blíže středu grafu, je důležitější. Takovouto metriku udává closeness.

5. Uzel se sousedy, kteří mají vyšší vliv, je více důležitý.

Pravidlo 1 tedy neurčuje důležitost uzlu, ale pouze se snaží zmenšit množinu vyhodnocovaných uzlů.

Důležitost jednotlivých uzlů je následně ohodnocena pomocí n-rozměrné funkce, kde n je dáno počtem kritérií (pro uzel a v našem případě tedy $a(a_2, a_3, a_4, a_5)$). Poté můžeme říci, že uzel a je dominantní nad uzlem b, pokud platí:

$$\forall i \in \{1, 2, ..., m\}, a_i \leq b_i \wedge \exists i \in \{1, 2, ..., m\}, a_i < b_i \tag{2.1}$$

Pomocí této logické formule nám vzniknou množiny dominantních a nedominantních uzlů. Uzlům dominantní množiny dáme nejvyšší prioritu. Množinu nedominantních uzlů opět rozdělíme pomocí logické formule 2.1 a těm uzlům, které jsou nyní dominantní, přidělíme druhou nejvyšší prioritu. Tento postup budeme opakovat, dokud nebude množina nedominantních uzlů prázdná. Zmíněným postupem docílíme rozdělení uzlů do jednotlivých množin s různými prioritami.

2.2.3 Ohodnocení uzlů grafu pomocí algoritmu PageRank

PageRank[6] (pojmenovaný po Larrym Pageovi) je poměrně známý algoritmus, který byl navržen pro ohodnocování důležitosti jednotlivých webových stránek společností Google. Dnes je PageRank jen jednou z mnoha metrik při přiřazování priority webovým stánkám touto společností. Algoritmus je použitelný nejen pro ohodnocování webových stránek, ale prakticky pro jakýkoliv graf, kde nás zajímá důležitost jednotlivých uzlů.

Na rozdíl od centralit grafu se PageRank při vyhodnocování důležitosti nesoustředí jen na pozorovaný uzel v grafu, ale stanovuje jeho důležitost v závislosti na důležitosti okolních uzlů.

Důležitost uzlu

Máme graf reprezentovaný množinou uzlů a orientovaných hran. Důležitost uzlu k je dána hodnotou x_k. Uzel je tím důležitější, čím více důležitých uzlů se na něj odkazuje, přičemž čím více odchozích odkazů uzel má, tím menší část své priority předává uzlům, na které se odkazuje. Tento vztah je vyjádřen pomocí následující rovnice:

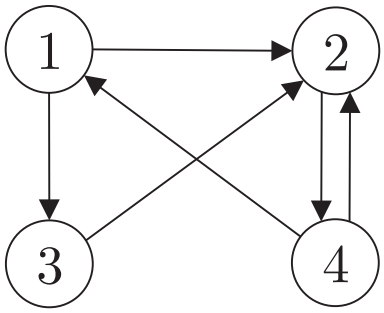
$$x_k = \sum_{j \in L_k} \frac{x_j}{n_j} \tag{2.2}$$

kde L_k je množina uzlů, pro které platí, že $(j,k) \in E(G)$ a n_j je odchozí stupeň uzlu j.

Pokud bychom tedy měli uzel události *II. světová válka*, jistě by byl tento uzel velice důležitý, jelikož se k němu váže mnoho jiných více či méně důležitých událostí, osob a míst. Zároveň by ale tento uzel měl i mnoho odchozích hran, které by reprezentovaly například následky II. světové války. Tudíž by těmto uzlům předal jen velice malou část své priority.

Vztah 2.2 můžeme aplikovat na graf na obrázku 2.3, kde nám pro jednotlivé uzly vyjde následující soustava rovnic.

$$\begin{aligned} x_1 &= \tfrac{x_4}{2} \\ x_2 &= \tfrac{x_1}{2} + x_3 + \tfrac{x_4}{2} \\ x_3 &= x_1 \\ x_4 &= x_2 \end{aligned}$$



Obrázek 2.3: Graf pro prezentaci sestavení soustavy rovnic PageRanku.

Tuto soustavu rovnic můžeme zapsat jako $\mathbf{x}^T = \mathbf{A}\mathbf{x}^T$, kde $\mathbf{x} = (x_1, x_2, x_3, x_4)$ a

$$\mathbf{A} = \begin{pmatrix} 0 & 1/2 & 1/2 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ 1/2 & 1/2 & 0 & 0 \end{pmatrix}$$

Po transponování pak dostáváme následující rovnici:

$$\mathbf{x} = \mathbf{x}\mathbf{A} \tag{2.3}$$

Přirozeným řešením této rovnice může být vektor $\mathbf{x} = (0,0,0,0)$. Takovýto výsledek by však znamenal, že důležitost všech uzlů je nulová.

Na rovnici 2.3 lze však také pohlížet jako na hledání vlastního vektoru matice $\mathbf{A}$ s vlastním číslem 1[1] . Hledání vlastního vektoru lze provést několika metodami (Gauss-Seidelova, Jacobiho, ...).

[1]Vlastní vektor matice $\mathbf{A}$ je takový nenulový vektor, pro který platí $\mathbf{x}\mathbf{A} = \lambda\mathbf{x}$, kde λ se nazývá vlastní číslo matice $\mathbf{A}$.

PageRank řeší hledání vlastního vektoru pomocí iteračního procesu. Rovnice je tedy upravena do tvaru $\mathbf{x_{k+1}} = \mathbf{x_k}\mathbf{A}$, kde vektor $\mathbf{x_0} = (1/n, 1/n, ..., 1/n)$ a n udává počet uzlů grafu (v případě grafu na obrázku 2.3 je $\mathbf{x_0} = (1/4, 1/4, 1/4, 1/4)$). V práci [4] je pak podrobně dokázáno, že tato iterační metoda konverguje k vlastnímu vektoru matice $\mathbf{A}$, pokud je $\mathbf{A}$ řádkově stochastická matice (viz níže).

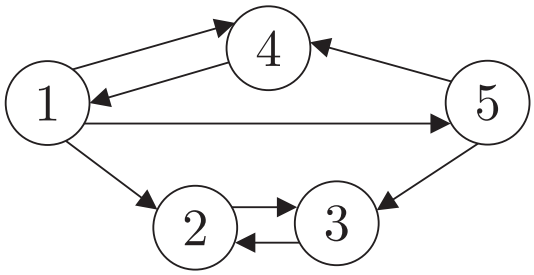
Absorpční uzly

V grafu může být i několik absorpčních uzlů, ze kterých již nevede hrana ven. Tyto uzly by v matici $\mathbf{A}$ byly reprezentovány nulovou řádkou. Proto je matice $\mathbf{A}$ vyměněna za matici $\mathbf{S}$, která nahrazuje nulové řádky řádkou $(1/n, 1/n, ..., 1/n)$. Tato řádka udává, že se z absorpčního uzlu přejde do jakéhokoli uzlu se stejnou pravděpodobností.

Součet prvků každé řádky matice $\mathbf{S}$ tak dává hodnotu 1. Matice, které splňují tuto podmínku, se nazývají *řádkově stochastické matice*.

Cykly v grafu

Pokud máme graf, který není silně souvislý, pak nastane situace, že s přibývajícími iteracemi bude stoupat důležitost uzlů uvnitř silně souvislého podgrafu (uzly 2 a 3 na obrázku 2.4), jelikož z tohoto podgrafu již nevede hrana do zbylých uzlů.



Obrázek 2.4: Ukázka grafu se silně souvislým podgrafem.

S touto situací je počítáno už v návrhu PageRanku, a to tak že k matici $\mathbf{S}$ je přičtena čtvercová matice $\mathbf{E}$ o rozměrech $n \times n$, kde $e_{ij} = 1/n$. Matice $\mathbf{E}$ tedy představuje teleportační matici (tento pojem pochází z [4]), která umožňuje teleportaci z aktuálního do libovolného uzlu, a to se stejnou prioritou pro všechny uzly.

Aby bylo zajištěno, že výsledná matice bude opět řádkově stochastická, zavedeme konstantu $\alpha \in (0, 1)$, která určuje s jakou pravděpodobností nenastane „teleportace“. Výsledná matice tedy bude mít následující tvar:

$$\mathbf{G} = \alpha\mathbf{S} + (1 - \alpha)\mathbf{E} \tag{2.4}$$

kdy je autory algoritmu doporučováno, aby $\alpha = 0,85$ [6]. Tato hodnota je kompromisem mezi rychlou konvergencí algoritmu a relevantnosti výsledků vůči topologii grafu.

PageRank uzlů grafu spočítáme pomocí následujícího iteračního algoritmu:

$$\mathbf{x_{k+1}} = \mathbf{x_k}\mathbf{G} \tag{2.5}$$

Zastavovací podmínka

Zbývá jen určit zastavovací podmínku. Výpočet zastavíme, pokud *jednotková norma* rozdílu vektorů $\mathbf{x_{k+1}}$ a $\mathbf{x_k}$ je menší než zvolené ε (volba optimální hodnoty ε je popsána v kapitole 9.4) [6]. Jednotková norma vektoru je pak dána následujícím vztahem:

$$||\mathbf{v}|| = \sum_{i=1}^{n} |v_i| \tag{2.6}$$

Kde v_i jsou jednotlivé prvky vektoru $\mathbf{v}$.

Výpočetní složitost potřebná pro jednu iteraci může být v závislosti na implementaci až $O(n)$ (viz dále v kapitole 6). Přičemž podle [6] lze dostatečné přesnosti výsledku dosáhnout již po 50 až 100 iteracích (pro grafy o stamiliónech uzlů).

2.2.4 PageRank s uživatelsky přednastavenými prioritami

V případě grafu historických událostí a jeho struktury popsané v kapitole 2.1 může uživatel požadovat specifické zobrazení grafu. Může se například zajímat jen o jeden typ uzlů a vazeb mezi nimi. Je tedy určitě vhodné umožnit uživateli, aby si vybral jím preferované typy uzlů a hran.

Pokud tedy nechceme, aby o důležitosti jednotlivých uzlů rozhodoval jen samotný algoritmus PageRanku, můžeme se pokusit o úpravu jeho jednotlivých matic, ze kterých se počítají důležitosti uzlů.

Úprava teleportační matice

Na první pohled je nejjednodušší upravit teleportační matici[7]. Jak bylo zmíněno výše, tato matice slouží primárně k potlačení akumulace důležitosti uzlů, které jsou součástí silně souvislého podgrafu. My tuto matici můžeme využít tak, že pravděpodobnosti teleportace do jednotlivých uzlů budou dány podle uživatelských priorit.

Matici $\mathbf{E}$ tedy nahradíme maticí $\mathbf{T}$ (tato matice je opět řádkově stochastická, tudíž součet prvků jednoho řádku musí dávat 1), kde jednotlivé prvky řádků budou reprezentovat pravděpodobnost teleportace do jiného uzlu.

Jak je popsáno například v [8], celý iterační vztah můžeme upravit a teleportační matici $\mathbf{E}$ nahradit vektorem $\mathbf{e} = (1/n, 1/n, ..., 1/n)$ dimenze n, tento vektor

tedy odpovídá jedné řádce matice **E**. Analogicky můžeme přistoupit i k matici **T** a vytvořit vektor **t**. Iterační proces je poté popsán pomocí následujícího vzorce:

$$\mathbf{x_{k+1}} = \alpha \mathbf{x_k} \mathbf{S} + (1 - \alpha)\mathbf{t} \tag{2.7}$$

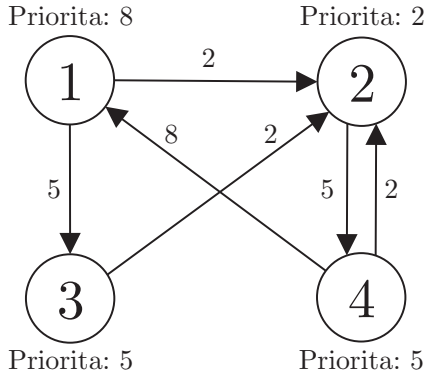
K ukázce vektoru **t** můžeme použít graf z obrázku 2.3. Vytvoříme stupně důležitosti 1 až 10, kde 1 značí nejnižší významnost. Chceme, aby uzel 1 měl důležitost 8, uzly 3 a 4 byly středně důležité, tudíž jejich důležitost je 5 a uzel 2 byl pro nás nejméně zajímavým – důležitost 2. Následně provedeme normalizaci těchto priorit, tudíž vektor **t** bude vypadat následovně:

$$\mathbf{t} = (2/5, 1/10, 1/4, 1/4)$$

Pokud ovšem bude teleportační konstanta $\alpha = 0.85$, pak touto úpravou nedosáhneme příliš výrazných výsledků, jelikož vliv tohoto vektoru na výslednou důležitost uzlů dosáhne 15%. Pokud ale hodnotu α razantně snížíme, dosáhneme sice očekávaných výsledků, ale tyto důležitosti by již výrazně méně korespondovaly s vazbami mezi uzly.

Úprava přechodové matice

Další možností může být úprava samotné přechodové matice **S**, konkrétně pravděpodobností přechodů mezi jednotlivými uzly. Při sestavování matice přechodů víme, do jakých uzlů vedou odchozí hrany (přesněji známe jejich priority). Pokud tedy ohodnotíme odchozí hrany prioritou uzlů, do kterých směřují, pomůže nám to opět k zvýraznění důležitosti uzlů, které preferujeme. Takovýto graf můžeme vidět na obrázku 2.5.



Obrázek 2.5: Graf s prioritně ohodnocenými uzly a hranami.

Jako u matice **T** i zde musíme provést normalizaci priorit hran. Výsledná matice přechodů pak bude vypadat následovně:

$$\mathbf{S} = \begin{pmatrix} 0 & {}^{2}\!/_{7} & {}^{5}\!/_{7} & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ {}^{4}\!/_{5} & {}^{1}\!/_{5} & 0 & 0 \end{pmatrix}$$

Bohužel i s takto sestrojenou maticí přechodů stále nedostaneme požadované priority uzlů. Jak lze vidět v tabulce 2.1 níže, uzel 2 má stále nejvyšší prioritu a uzel 4 je pak druhý nejlépe hodnocený uzel grafu. To je zapříčiněno tím, že je na něj přenesena všechna pritorita uzlu 2, jelikož z něj vede jen jedna odchozí hrana.

Na grafu 2.5 je také vidět, že priorita hrany $(2, 3)$ je sice jen dvě, ale v matici **S** již uzel 3 předává všechnu svoji prioritu uzlu 2, jelikož nemá jinou odchozí hranu.

Vytvoření matice teleportačních konstant

Jak bylo nastíněno výše, problém u hran s prioritou nastává v případě, kdy uzel má jen jednu odchozí hranu do uzlu s nízkou prioritou nebo pokud všechny jeho odchozí hrany mají nízkou prioritu.

Řešení této situace by opět mohlo spočívat ve využití teleportační matice, respektive teleportační konstanty α. Tuto konstantu nahradíme maticí **X**, která má následující tvar:

$$\mathbf{X} = \alpha\mathbf{I} \tag{2.8}$$

Kde **I** je jednotková matice. Tato matice tedy má zatím stejný efekt jako teleportační konstanta α. Změna spočívá v tom, že nyní má každý uzel svoji teleportační konstantu (uzlu i nyní patří α_{ii} matice **X**).

Pokud pro uzel i při vytváření matice **S** zjistíme, že má jen jednu odchozí hranu, která má nízkou prioritou, nebo že všechny jeho odchozí hrany mají nízkou prioritou, poté provedeme úpravu hodnoty α_{ii} matice **X**. A to tak, že hodnotu α_{ii} snížíme (zvýšíme pravděpodobnost teleportace z daného uzlu jinam na úkor pravděpodobnosti přechodu do uzlu s nízkou uživatelskou prioritou).

Popisovaným postupem docílíme toho, že do uzlů které mají nízkou uživatelskou prioritu se bude přecházet s menší pravděpodobností a místo toho se zvýší pravděpodobnost přechodu do uzlů, které mají nastavenou vyšší uživatelskou prioritu zadanou v teleportační matici.

Tento postup staticky neovlivňuje priority všech uzlů (jako by tomu bylo, pokud bychom si vystačili s úpravou teleportační matice a snížení hodnoty α v podkapitole 2.2.4), ale pouze těch, u kterých chceme cíleně snížit prioritu.

Zbývá určit postup výběru uzlů i, kterým zmenšíme hodnotu α_{ii} matice **X**. Nejprve zjistíme, pro jaké uzly budeme měnit jejich hodnotu α_{ii}. Nabízí se přístup, kdy nejprve spočítáme průměrnou prioritu hran, kterou označíme jako p. Pokud

bude mít kterýkoliv uzel průměrnou prioritu odchozích hran menší než p, pak jeho α_{ii} bude spočtena podle následující lienární funkce:

$$\alpha_{ii} = \frac{3}{4(p-1)}(p_i - 1) + 0,1 \tag{2.9}$$

kde p_i označuje průměrnou prioritu odchozích hran uzlu i. Jelikož jako minimální priorita bylo stanoveno číslo 1, nemůže se stát, že p_i bude menší. Konkréně pro $p_i = 1$ vrací tato funkce hodnotu $0,1$. Pokud do rovnice vložíme $p_i = p$, pak nám vrátí doporučenou hodnotu $\alpha_{ii} = 0,85$.

Graf z obrázku 2.5 má průměrnou prioritu hran $p = 4$ a uzly s podprůměrnou prioritou výchozích hran jsou uzly 1 a 3, konkrétně $p_1 = 3,5$ a $p_3 = 2$. Matice $\mathbf{X}$ tedy bude vypadat následovně:

$$\mathbf{X} = \begin{pmatrix} 0,725 & 0 & 0 & 0 \\ 0 & 0,85 & 0 & 0 \\ 0 & 0 & 0,35 & 0 \\ 0 & 0 & 0 & 0,85 \end{pmatrix}$$

Nyní tedy můžeme složit matici $\mathbf{G}$ následovně:

$$\mathbf{G} = \mathbf{XS} + (\mathbf{I} - \mathbf{X})\mathbf{T} \tag{2.10}$$

V tabulce 2.1 si lze prohlédnout výsledky při aplikování jednotlivých metod na algoritmus PageRanku v porovnání s PageRankem bez uživatelských priorit. V tabulce je také vidět, že požadovaného pořadí priorit jsme dosáhli až při použití matice teleportačních konstant $\mathbf{X}$.

	Uzel			
Algoritmus	1	2	3	4
PageRank	0,183	0,359	0,115	0,343
PageRank s $\mathbf{T}$	0,201	0,345	0,123	0,331
PageRank s prioritami hran	0,250	0,283	0,189	0,278
PageRank s $\mathbf{X}$	0,297	0,216	0,229	0,258

Tabulka 2.1: Porovnání důležitostí uzlů grafu z obrázku 2.5 při jednotlivých typech PageRanku. Výsledky po patnácti iteracích.

Cena za takto uživatelsky optimalizované důležitosti uzlů je ovšem vykoupena výpočetní složitostí $O(n^2)$, jelikož matici $(\mathbf{I} - \mathbf{X})\mathbf{T}$ již nelze převést na vektor $\mathbf{t}$, protože každý uzel má jinou hodnotu α_{ii}, a tedy každá řádka matice $(\mathbf{I} - \mathbf{X})\mathbf{T}$ obsahuje jiné hodnoty.

3 Knihovny implementující graf

Pro implementaci algoritmu, jenž provádí ohodnocení důležitosti uzlů, bylo potřeba najít adekvátní knihovnu poskytující abstrakci grafu. Idálně by takováto knihovna měla disponovat následujícími vlastnostmi:

- **Rychlé operace nad prvky grafu** – Knihovna by měla dokázat přidávat (resp. odebírat) jednotlivé uzly a hrany grafu pokud možno v co nejkratším čase. Důležitým aspektem je také rychlost procházení grafu, která má vysoký vliv na rychlost výpočtu důležitosti jednotlivých uzlů.
- **Jednoznačná identifikace uzlů a hran** – Výpočet důležitosti jednotlivých uzlů je vždy prováděn nad uživatelem zvoleným podgrafem (tento podgraf bude uchovávat vybraná knihovna). Uživatel pak může v průběhu času přidávat nebo ubírat jednotlivé uzly a hrany. Nad takto zvoleným podgrafem se poté provádí výpočet důležitosti.

  Proto je vhodné, aby knihovna dokázala pokud možno v konstantním čase určit, zda určitý prvek grafu se již nachází v pracovním podgrafu nebo ne.
- **Možnost přiřazení důležitosti uzlům** – Jednotlivé uzly by ideálně obsahovaly vlastnost `priorita`.
- **Implementované prioritní algoritmy** – Výhodou grafového balíku by byla sada algoritmů počítajících prioritu pro možnost porovnání výsledků.

Rozbor existujících knihoven se věnuje jen těm, které jsou implementovány v jazyce Java, jelikož jedním z požadavků zadání diplomové práce byla implementace v tomto jazyce. Většina grafových knihoven také nabízí možnost vizualizace grafu, která v tomto porovnání nehraje žádnou roli, jelikož účelem této práce není graf vizualizovat, ale předat jej s vypočítanými důležitostmi zobrazovací vrstvě.

Jak bylo uvedeno v kapitole 2.1, požadovaná knihovna také musí umět implementovat orientovaný graf.

3.1 Grph

webová adresa:	`http://www.i3s.unice.fr/~hogie/grph/`
licence:	GNU LGPL
aktuální verze[1]**:**	`v1.5.27` vydána dne 14. ledna 2015
autoři:	Luc Hogie a kolektiv; University of Nice-Sophia Antipolis; I3S laboratory

Tato knihovna byla navržena jako výkonný nástroj pro práci s rozsáhlými grafy a jejich vizualizaci [9]. Podporuje orientované i neorientované grafy a hypergrafy[2] (právě implementace hypergrafu byla i jedním z hlavních důvodu vzniku této knihovny).

Knihovna využívá mnoha technik ke snížení výpočetního času při provádění operací nad grafem nebo jeho vizualizace (paralelizmus, cachování, *on-the-fly* kompilace specifických částí implementace napsaných v jazyce C/C++).

Práce s knihovnou

Knihovna reprezentuje uzly i hrany pouze pomocí celočíselné hodnoty. Všechny operace nad grafem jsou prováděny prostřednictvím třídy `Grph` pomocí funkcí pracujících s těmito celočíselnými hodnotami. Vlastnosti jednotlivých uzlů nebo hran pak mohou být uloženy v pomocné mapě, jejímž klíčem je identifikátor uzlu nebo hrany.

Třída `ObjectGrph<V,E>` nabízí práci s grafem, kde jsou uzly a hrany reprezentovány objekty. Nevýhodou tohoto přístupu je ale chybějící možnost vložení objektu reprezentující uzel pod libovolným celočíselným identifikátorem, jelikož identifikátor je generován knihovnou automaticky při vkládání.

Dokumentace

Dokumentace knihovny není na moc dobré úrovni. K dispozici je uživateli vygenerovaný Javadoc, ve kterém nejsou okomentovány ani některé velice často používané metody (např. u metody `InMemoryGrph.addVertex(int)` chybí jakýkoliv popis, i například co se děje při konfliktu identifikátorů). Proto se s knihovnou nepracuje příliš dobře a často je potřeba podívat se pro zjištění významu některých metod nebo tříd i na samotnou implementaci.

Na stránkách knihovny je také k dispozici několik dokumentů, které mají uživateli vyjasnit způsob práce s knihovnou. Některé tyto dokumenty ovšem vypadají jako rozpracované.

[1] Ke dni 7. března 2015.

[2] Graf, jehož hrany mohou spojovat více než dva uzly.

Poskytované algoritmy

Knihovna disponuje rozsáhlou množinou implementovaných algoritmů rozdělenou do několika sekcí (prohledávání grafu, nejkratší cesty, shlukování, řezy, ...). Grph nabízí také rozhraní pro možnost implementace vlastního algoritmu.

3.2 JUNG

webová adresa:	`http://jung.sourceforge.net/`
licence:	Berkeley Software Distribution license
aktuální verze[3]:	`v2.0.1` vydána dne 24. ledna 2010
autoři:	Joshua O'Madadhain, Danyel Fisher, Tom Nelson

JUNG (Java Universal Network/Graph Framework) [10] je grafová knihovna, která byla vyvíjena mezi lety 2003 až 2010. Knihovna byla vytvořena z potřeby obecného, flexibilního a silného API pro manipulaci, analýzu a vizualizaci grafů a sítí. Její architektura je navržena tak, aby podporovala velké množství reprezentací entit a jejich vztahů, jako jsou orientované a neorientované grafy, hypergrafy a grafy s paralelními hranami.

Práce s knihovnou

Knihovna reprezentuje orientovaný graf prostřednictvím generického rozhraní `DirectedGraph<V,E>`. Pomocí tohoto rozhraní se přistupuje ke všem údajům jednotlivých uzlů a hran, které se nacházejí v daném grafu (přidání uzlu, hrany, zjištění vstupních a výstupních hran uzlu, ...). Jak je patrné z definice rozhraní `DirectedGraph<V,E>`, uzly i hrany mohou být reprezentovány libovolným objektem.

Nevýhodou tohoto přístupu je nutnost vlastnit odkaz na objekt třídy reprezentující uzel nebo hranu, abychom mohli například zjistit, zda se v grafu již nachází nebo ne. Pokud tedy chceme získat informace o vazbách uvnitř grafu, musíme nejprve mít k dispozici instanci zkoumaného objektu nebo jeho kopii.

Tato nevýhoda by šla potlačit pomocí uchování pomocných map, kde by byl klíčem identifikátor uzlu ,respektive hrany, a hodnotou objekt, který uzel, respektive hranu reprezentuje.

Dokumentace

Na webových stránkách knihovny je k dispozici mnoho dokumentačních souborů ve formě prezentací a tutorialových dokumentů. Některé odkazy jsou bohužel slepé, jelikož knihovna i její webové stránky jsou několik let neudržované.

Knihovna také disponuje vygenerovanou Javadoc dokumentací, která je velice kvalitně zpracována.

[3]Ke dni 7. března 2015.

Na webových stránkách je k dispozici několik Java appletů, které demonstrují funkčnost knihovny a především možnosti vizualizace.

Poskytované algoritmy

Knihovna disponuje velice rozsáhlou kolekcí algoritmů. Všechny implementované algoritmy se nacházejí v balíku `edu.uci.ics.jung.algorithms`, kde jsou dále rozděleny podle kategorií. Uvnitř kategorie `scorings` se nacházejí algoritmy, které hodnotí priority uzlů. Jsou zde implementovány mimo jiné algoritmy PageRank, betweenness centrality, closeness centrality a degree centrality, které byly popsány v kapitole 2.

Knihovna také poskytuje rozsáhlou množinu rozhraní, která dovolují implementovat vlastní algoritmy.

3.3 GraphStream

webová adresa:	`http://graphstream-project.org/`
licence:	GNU GPL
aktuální verze[4]:	`v1.2` vydána dne 24. ledna 2013
autoři:	S. Balev, J. Baudry, A. Dutot, Y. Pigné, G. Savin; University of Le Havre; LITIS computer science lab

GraphStream [11] je knihovna vyvíjená na University of Le Havre, která se soustředí především na dynamické aspekty grafů a sítí a jejich vizualizaci. Nabízí několik tříd grafů, které umožňují pracovat s různými typy orientovaných a neorientovaných grafů a multigrafů[5]. GraphStream dovoluje libovolně v čase přidávat, ubírat nebo upravovat uzly a hrany a tyto události dynamicky zobrazovat.

Knihovna samotná je rozdělena na několik balíků. Tím základním je `gs-core`, který musí být použit vždy, pokud chceme pracovat s jakýmkoli jiným balíkem knihovny GraphStream, jelikož obsahuje implementaci samotného grafu. Dalšími balíky jsou `gs-algo`, který obsahuje implementaci grafových algoritmů a `gs-ui` starající se o vizualizaci grafu.

Práce s knihovnou

Knihovna reprezentuje graf prostřednictvím rozhraní `Graph`, jehož implementace představují různé typy grafů. Uzly a hrany jsou tvořeny rozhraním `Node`, respektive `Edge`. Při vkládání nového uzlu nebo hrany do grafu musí být vždy specifikován jeho jedinečný identifikátor v podobě řetězce (což může zpomalovat hledání nebo vkládání nových uzlů, jelikož výpočet hash funkce řetězce trvá déle

[4]Ke dni 7. března 2015.

[5]Graf, ve kterém může dva uzly spojovat více než jedna hrana.

než u celočíselných identifikátorů). Každý uzel i hrana mají i svůj číselný index, který je ale přiřazován automaticky a není nijak možné jeho hodnotu ovlivnit.

Ukládání a přístup k vlastnostem uzlů a hran je řešen pomocí mapy, kde klíčem je řetězec a hodnotou libovolný objekt nebo jejich kolekce. S vlastnostmi rozhraní `Node` a `Edge` se pracuje pomocí metod `setAttribute()`, `getAttribute()`, `removeAttribute()` a dalšími.

Knihovna umožňuje vytvořit vlastní implementaci rozhraní `Node`, respektive `Edge`, které mohou nést další specifické vlastnosti. Instanci grafu se poté musí nastavit nová továrna uzlů, respektive hran.

Dokumentace

Na webových stránkách knihovny je k dispozici velké množství tutoriálů pro její jednotlivé části, které usnadňují uživateli první kroky s knihovnou.

K dispozici jsou také vygenerované Javadoc dokumentace pro jednotlivé části knihovny, které jsou velice kvalitně zpracované.

Poskytované algoritmy

Balík `gs-algo` této knihovny disponuje rozsáhlou množinou algoritmů a generátorů grafu. Většina grafů implementuje rozhraní `DynamicAlgorithm`, které vždy při změně grafu provádí znovu výpočet, což je velice efektivní právě při vizualizaci, kdy lze postupně pozorovat průběžné změny například v prioritním ohodnocení jednotlivých uzlů grafu.

Tento balík také implementuje algoritmy popsané v kapitole 2. Například právě implementace PageRanku byla vytvořena jako dynamický algoritmus.

3.4 JGraphT

webová adresa: `http://jgrapht.org/`
licence: GNU LGPL a EPL
aktuální verze[6]: `v0.9.0` vydána v prosinci 2013
autoři: Barak Naveh a přispěvatelé

Grafová knihovna aktivně vyvíjená od roku 2003, která je široce využívána v projektech mnoha různých zaměření (například i bioinformatika nebo chemie) [12]. Knihovna se soustředí na orientované a neorientované grafy a grafy, jejichž hrany jsou vážené. Poskytuje také implementaci *listenable* grafů, které umožňují automatické provádění nadefinovaných akcí po specifických úpravách struktury grafu.

Knihovna je podobná svým názvem knihovně JGraph. Obě knihovny ale pocházejí od různých autorů a mají rozdílné zaměření. JGraphT se zaměřuje především

[6]Ke dni 8. března 2015.

na reprezentaci grafu jako datové struktury a jeho algoritmické zpracování, zatímco JGraph je zaměřen na vizualizaci grafu a jeho editování prostřednictvím uživatelského rozhraní (proto nebyl zařazen mezi popisované knihovny). Nicméně JGraphT nabízí adaptér pro možnost vizualizace jeho grafových struktur pomocí knihovny JGraph.

Práce s knihovnou

Knihovna je definicí svých rozhraní velice podobná knihovně JUNG (viz kapitola 3.2 výše). S grafem se pracuje prostřednictvím generického rozhraní `Graph<V,E>`, kde `V` představuje třídu reprezentující uzly a `E` hrany grafu. Toto rozhraní implementuje několik tříd, každá zajišťuje jiný typ grafu.

Pro práci s orientovaným grafem slouží třída `DefaultDirectedGraph<V,E>`. Pomocí jejích metod pak můžeme do grafu jednotlivé uzly a hrany vkládat, mazat nebo testovat, zda se v něm již nachází.

Jako v případě knihovny JUNG je potřeba vlastnit odkaz na dotazovaný uzel nebo hranu, pokud například chceme zjistit, zda se v grafu nachází. To je velice nepříjemné kupříkladu při přidávání hrany v průběhu vytváření grafu, kdy musíme vlastnit odkaz na obě instance uzlů, které chceme spojit hranou.

Dokumentace

Na webových stránkách knihovny se nachází základní informace o knihovně spolu s ukázkami základních operací nad grafem. Stránky také obsahují odkaz na Javadoc dokumentaci, která velice detailně popisuje implementaci celé knihovny.

Knihovna také disponuje svými Wiki stránkami, kde jsou k nahlédnutí další, již konkrétněji zaměřené ukázky použití. Zároveň se zde uživatel může dozvědět, jakých návrhových vzorů je použito při její implementaci a jaké zásady by měly být dodržovány při dalším rozšiřování knihovny.

Poskytované algoritmy

JGraphT obsahuje balík `algo`, který poskytuje přes dvacet implementovaných grafových algoritmů. Většina algoritmů patří mezi prohledávací (Bellman-Fordův algoritmus, chromatické číslování, Prim-Jarnikova kostra grafu, ...).

Knihovna neposkytuje žádné rozhraní, definující algoritmus. Většina tříd implementujících algoritmy požaduje jako parametr konstruktoru graf, na kterém bude algoritmus provádět.

3.5 Navrhovaná knihovna

Z výše popsaných knihoven vyhovují nejvíce JUNG a JGraphT. První jmenovaná má výhodu rozsáhlého portfolia implementovaných grafových algoritmů. Její zásadní nevýhodou je pak ovšem vývoj zastavený v roce 2010. Obě knihovny

velice podobně reprezentují graf. Nevýhody této reprezentace (včetně možnosti jejich potlačení) byly popsány v jejich rozborech.

Vzhledem k tomu, že ani jedna z popisovaných grafových knihoven nevyhovovala všem požadavkům, které byly vyjmenovány na začátku kapitoly, bude v rámci této diplomové práce navržena nová knihovna.

Ačkoliv by se popsané nevýhody knihoven JUNG nebo JGraphT daly poměrně lehce potlačit, autorovi této práce se zdál návrh a implementace vlastní grafové knihovny jako zajímavá výzva.

3.5.1 Vlastnosti navrhované knihovny

Jednoznačná identifikace uzlů a hran

Navrhovaná knihovna bude pracovat s generickými entitami reprezentujícími uzly a hrany. Tyto entity ovšem budou muset obsahovat svůj jednoznačný identifikátor. Tím se jednoduše vyloučí možnost mít v grafu více stejných uzlů nebo hran. Díky tomu také mohou být uzly a grafy uloženy v mapách, čímž se k nim zajistí přístup v konstantním čase.

Rozdělení logické a datové části

Knihovna rozdělí reprezentaci uzlu na jeho logickou a datovou část. Logická část uzlu určuje jaké hrany do něj a z něj vedou, jaký má stupeň, sousedy, prioritu atd. Datová část pak představuje vlastnosti, které daný uzel uchovává (v našem případě například historickou osobnost a údaje o ní). Obě tyto části budou reprezentovány svým objektem.

Když pak budeme chtít z grafu vybrat uzel s určitým identifikátorem, dostaneme jeho logickou část, kterou lze následně využít k získání datové části.

Analogicky se přistoupí i k hranám grafu.

Prioritní algoritmy

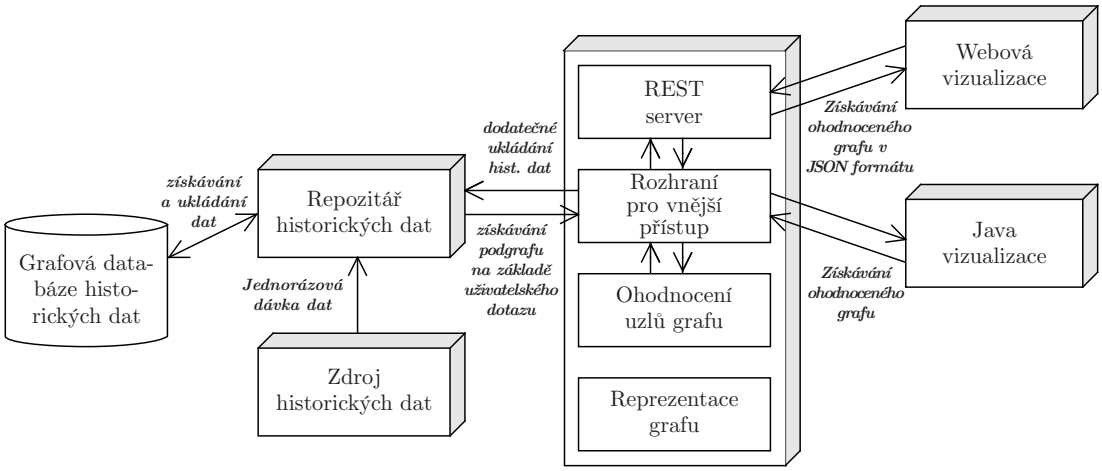
Knihovna bude obsahovat implementaci všech variant PageRanků popsaných v kapitole 2. Dále bude poskytovat rozhraní pro možnost vytvoření nových prioritních algoritmů.

4 Návrh knihovny pro reprezentaci a ohodnocení grafu

Předchozí kapitoly popisují strukturu historického grafu (viz 2.1), dále jednotlivé algoritmy pro ohodnocení uzlů grafu (viz 2.2) a existující knihovny implementující reprezentaci grafu (přičemž autor práce se rozhodl implementovat vlastní grafovou knihovnu, viz 3.5.1).

Tato kapitola se věnuje návrhu knihovny pro reprezentaci grafu a ohodnocení důležitosti jeho uzlů. Dále popisuje její integraci do výsledného nástroje pro zobrazování historických událostí na časové ose. Navazuje tak na stručné nastínění zasazení této práce do kontextu ostatních diplomových prací popsané v kapitole 2.1.2.

Na obrázku 4.1 lze vidět jednotlivé části výsledné aplikace, jejichž význam popisují podkapitoly níže.



Obrázek 4.1: Jednotlivé části, ze kterých se skládá výsledná aplikace pro vizualizaci historických událostí na časové ose

Grafová databáze historických dat

V této databázi jsou uchovávány všechny historické události, jejich vazby, stereotypy a další vlastnosti. Konkrétně se jedná o grafovou NoSQL databázi neo4j[1].

[1]Dostupné na `http://neo4j.com/`.

Repozitář historických dat

Tato část zajišťuje[2] mapování dat uložených v databázi na objekty tříd historických uzlů a jejich vazeb, které se v dalších částech následně ohodnocují a vizualizují. Dále tato část zprostředkovává dotazování nad databází a vkládání nových vrcholů a hran.

Zdroj historických dat

Část, která jednorázovou dávkou nahrála data historických událostí a osob do databáze[3].

Reprezentace grafu

Knihovna implementující reprezentaci grafu. Tato knihovna je využívána v ostatních částech výsledné aplikace (repozitář historických událostí, ohodnocení uzlů grafu, rozhraní pro vnější přístup, Java vizualizace). Implementaci této knihovny podrobně popisuje kapitola 5.

Ohodnocení uzlů grafu

Knihovna implementující algoritmy, které hodnotí důležitost jednotlivých uzlů grafu. Její realizace je podrobně popsána v kapitole 5.2, popis implementovaných algoritmů se pak nachází v kapitole 6.

Rozhraní pro vnější přístup

Knihovna poskytující Java rozhraní pro vizualizační části. Rozhraní disponuje následujícími službami:

- Získání ohodnoceného grafu historických událostí na základě uživatelského dotazu,
- získání vrcholu grafu (a případně jeho okolí specifikované hloubky) podle jeho ID,
- získání názvů všech vlastností uzlů,
- vložení uzlu nebo hrany do grafu historických událostí.

Rozhraní podrobněji popisuje kapitola 7. Knihovna také zajišťuje komunikaci s databází prostřednictvím části objektového mapování (viz obrázek 4.1).

[2]Části **Grafová databáze historických dat** a **Repozitář historických dat** byly vypracovány Davidem Merunkem v rámci paralelní diplomové práce s názvem *Generování a vizualizace časové osy.*

[3]Část **Zdroj historických dat** byla vypracována Gabrielou Hessovou v rámci paralelní bakalářské práce s názvem *Automatické získání historických údajů z webových zdrojů.*

REST server

Jelikož má výsledná aplikace dva různé typy vizualizace (jeden prostřednictvím desktopové Java aplikace, druhý pomocí webového rozhraní), je potřeba vytvořit paralelní rozhraní, které je nezávislé na programovacím jazyce a poskytuje stejné služby jako výše popsané Java rozhraní. Jako ideální řešení pro tento úkol bylo zvoleno rozhraní REST[4].

Zatímco lze rozhraní tvořené Java knihovnou přímo použít v libovolné aplikaci, pro zpřístupnění téže funkcionality prostřednictvím REST služeb je nutné jej koncipovat jako webovou službu.

Java vizualizace

Jak bylo zmíněno výše, tato část vizualizuje ohodnocený graf historických událostí prostřednictvím desktopové Java aplikace[5]. V tomto kontextu tak zbylé části výsledné aplikace představují aplikační a datovou úroveň.

Webová vizualizace

Na rozdíl od Java vizualizace přistupuje tato část k REST rozhraní, které je koncipováno jako serverová aplikace. Výsledná aplikace je pak tedy rozdělena na dvě fyzické části:

- Server poskytující prioritně ohodnocená data o historických událostech na základě klientských dotazů,
- klient zajišťující zobrazení přijatých dat na základě zaslaného požadavku[6].

[4]**REST** (Representational state transfer) – architektura rozhraní navržená pro distribuované prostředí. Typicky využívá protokolu HTTP/1.1 (dotazy `GET`, `POST`, `DELETE`, atd.) pro komunikaci a identifikátoru URI pro identifikaci jednotlivých zdrojů.

[5]Část věnující se **Java vizualizaci** byla vypracována Janem Moulisem v rámci paralelní diplomové práce s názvem *Vizualizace časové osy*.

[6]Část věnující se **Webové vizualizaci** byla vypracována Michalem Kacerovským v rámci paralelní diplomové práce s názvem *Vizuální reprezentace precedenčního grafu*.

5 Implementace grafové knihovny

Jak bylo popsáno v předchozí kapitole, součástí této diplomové práce je i implementace vlastní grafové knihovny. Základní rozdělení této knihovny se inspiruje knihovnou GraphStream, a sice jejím rozdělením na dva základní Maven projekty:

- **graph** – Tato část je samotnou implementaci orientovaného grafu. Umožňuje vytvářet instance grafu, do kterého lze přidávat uzly a hrany nebo je mazat. Balík také obsahuje implementaci dvou základních entit časového grafu, a to tříd `Node` a `Bond`, jež představují historický uzel, respektive vazbu mezi mezi nimi.
- **graph-ranking** – Část sloužící k algoritmickému ohodnocení grafu. Zde jsou implementovány všechny používané typy algoritmu PageRank, popsané v kapitole 2. Také poskytuje rozhraní pro implementaci dalších ohodnocovacích algoritmů (`VertexImportanceComputer`), rozhraní pro přepočet důležitostí jednotlivých uzlů ze základní hodnoty (součet důležitostí přes všechny uzly je roven jedné) na jinak specifikovanou (například zvolený počet úrovní důležitostí) nebo také třídy pro zpracování uživatelských priorit.

5.1 Knihovna `graph`

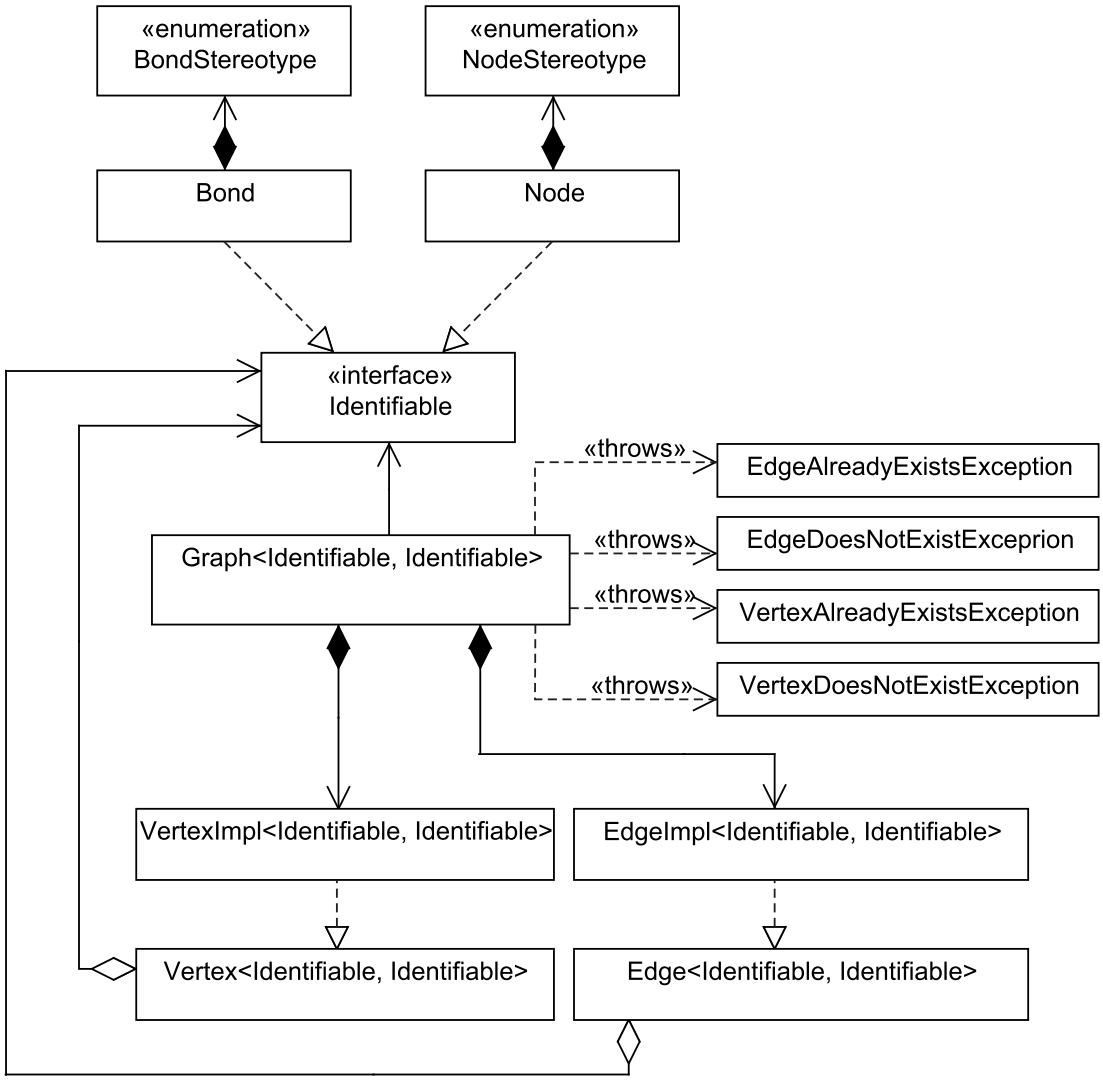
Na obrázku 5.1 lze vidět UML diagram tříd této knihovny. Z diagramu plyne, že hlavní třídou grafové knihovny je třída `Graph<Identifiable, Identifiable>` (první typ vždy určuje třídu reprezentující vrcholy, druhý typ určuje třídu představující hrany), která v sobě zapouzdřuje jednotlivé vrcholy (`Vertex`) a hrany (`Edge`). V diagramu jsou také zahrnuty entitní třídy `Node` a `Bond`, jež představují historické události, osobnosti, místa, předměty a jejich vazby. Obě třídy také nesou svůj stereotyp daný výčtovou hodnotou `NodeStereotype`, respektive `BondStereotype`.

Rozhraní Identifiable

Rozhraní, které obsahuje pouze metodu `getId()`. Ta zaručuje, že každý objekt implementující dané rozhraní obsahuje jednoznačný identifikátor.

Třída `Graph<Identifiable, Identifiable>`

Tato třída představuje graf, jehož vrcholy, respektive hrany, mohou být reprezentovány libovolnými třídami implementující rozhraní `Identifiable`. Díky tomu není možné, aby se v grafu nacházely dva stejné vrcholy nebo hrany se stejným



Obrázek 5.1: UML diagram knihovny graph

ID. Zároveň tak graf může obsahovat mapu, kde je klíčem ID vrcholu, respektive hrany, a hodnotou je objekt představující daný vrchol, respektive hranu, který vlastní tento identifikátor, což zaručuje přístup k vrcholům a hranám grafu v konstantním čase.

Třída `Graph<Identifiable, Identifiable>` poskytuje metody pro vložení nebo smazání uzlu, respektive hrany z grafu. Pokud je do grafu vkládán vrchol, který v něm již leží (shodující se ID), pak metoda `insertVertex()` vyhodí výjimku `VertexAlreadyExistsException`, jež zabrání jeho vložení.

Analogicky se pak postupuje při vkládání hran, kdy se navíc ověřuje, zda v grafu existují vrcholy, mezi nimiž má být daná hrana vytvořena. Testování existence vrcholů nebo hran se provádí také při jejich mazání.

Metoda `insertVertex()` přijímá jako parametr instanci třídy, která reprezentuje vrchol grafu. Výstupem této metody je instance třídy `Vertex<Identifiable, Identifiable>`, která uchovává vstupní a výstupní hrany vrcholu.

Analogicky opět pracuje metoda `insertEdge()`, která vrací instanci třídy `Edge<Identifiable, Identifiable>`

Rozhraní `Vertex<Identifiable, Identifiable>`

Rozhraní reprezentující vrchol grafu. Vrchol grafu tato knihovna rozděluje na dvě pomyslné části:

- **Logická**, pomocí níž můžeme přistupovat ke vstupním a výstupním hranám nebo vrcholům – určuje strukturu okolí.
- **Datová**, která uchovává data reprezentující daný vrchol (v našem kontextu jsou to například informace o určité historické události). K instanci entity nesoucí data se přistupuje pomocí metody `getValue()`.

Toto rozhraní implementuje třída `VertexImpl<Identifiable, Identifiable>`, jejíž instance jsou vytvářeny třídou `Graph<Identifiable, Identifiable>` při vkládání nového vrcholu do grafu.

Rozhraní `Edge<Identifiable, Identifiable>`

Rozhraní představující hranu grafu, které má analogickou funkčnost jako rozhraní `Vertex` popisované výše. Rozhraní `Edge` rozděluje reprezentaci hrany na logickou a datovou část stejně jako `Vertex`. Jediný rozdíl je v poskytování logických metod, kdy poskytuje metody pro získání odkazu na zdrojový, nebo cílový vrchol dané hrany.

Rozhraní implementuje třída `EdgeImpl<Identifiable, Identifiable>`, která je vytvářena třídou `Graph<Identifiable, Identifiable>` při vkládání hrany do grafu.

5.1.1 Entitní třídy grafu historických událostí

Předchozí odstavce popisují strukturu samotného grafu. K vytvoření grafu je nutné, aby jeho vrcholy, respektive hrany, byly reprezentovány třídami implementující rozhraní `Identifiable`. Níže popsané třídy představují uzly grafu historických událostí a jejich vazby, které toto rozhraní implementují.

Třída `Node`

Tato třída představuje historickou osobu, událost, místo nebo předmět. Stereotyp uzlu je určuje výčtový typ `NodeStereotype` (jednotlivé stereotypy popisuje kapitola 2.1). Uzel dále obsahuje následující atributy:

- `id` – celočíselný identifikátor uzlu,
- `name` – jméno události, místa, věci nebo historické osobnosti,
- `description` – základní popis daného uzlu,
- `stereotype` – stereotyp uzlu,
- `begin`, `end` – instance třídy `DateTime`[1] nesoucí informace o čase zahájení (například narození historické osobnosti) a konci (smrt) platnosti daného uzlu,
- `tags` – pole řetězců, které reprezentuje štítky patřící k uzlu (nepovinný atribut),
- `properties` – mapa, kde klíčem je řetězec a hodnotou objekt třídy `Object`. Tato mapa může obsahovat dodatečné údaje o historickém uzlu, jimiž mohou být například specifické vlastnosti pro jednotlivé stereotypy uzlů.

Třída `Bond`

Třída představuje vazbu mezi historickými uzly. Stejně jako uzly mají i jejich vazby různé stereotypy reprezentované výčtovým typem `BondStereotype` (stereotypů je celkem osm, jejich názvy a významy popisuje kapitola 2.1). Každá hrana pak disponuje následujícími atributy:

- `id` – celočíselný identifikátor hrany,
- `name` – jméno vazby,
- `description` – základní popis vazby mezi uzly (nepovinný atribut),
- `tags` – pole řetězců, které reprezentuje štítky patřící k vazbě (nepovinný atribut),
- `properties` – mapa, kde klíčem je řetězec a hodnotou objekt třídy `Object`. Jako u historických uzlů může obsahovat dodatečné informace o dané vazbě.

[1] Třída knihovny Joda-Time reprezentující čas.

5.2 Knihovna `graph-ranking`

Tato knihovna poskytuje implementaci několika variant algoritmu PageRank. Pomocí implementace rozhraní `VertexImportanceComputer` je navíc možné přidávat další ohodnocovací algoritmy.

Knihovnu tvoří samostatný Maven projektem, který je závislý na knihovně `graph`. Obrázek 5.2 zobrazuje její UML diagram tříd, ze kterého lze vidět, že středobodem knihovny je rozhraní `VertexImportanceComputer`, které implementují všechny ohodnocovací algoritmy.

Význam jednotlivých implementací algoritmu PageRank a jejich pomocných tříd podrobně popisuje kapitola 6. Níže se tak nachází popis jednotlivých tříd, které nejsou přímo spojeny s tímto algoritmem, ale lze je využít při implementaci jiných hodnotících algoritmů.

Rozhraní `VertexImportanceComputer`

Jak již bylo několikrát zmíněno, toto rozhraní implementují třídy, které provádějí ohodnocení významnosti vrcholů grafu na základě libovolného algoritmu nebo sady pravidel. Rozhraní obsahuje pouze metodu `computeVertexImportances()` s jediným parametrem, a to grafem, nad kterým bude prováděn výpočet.

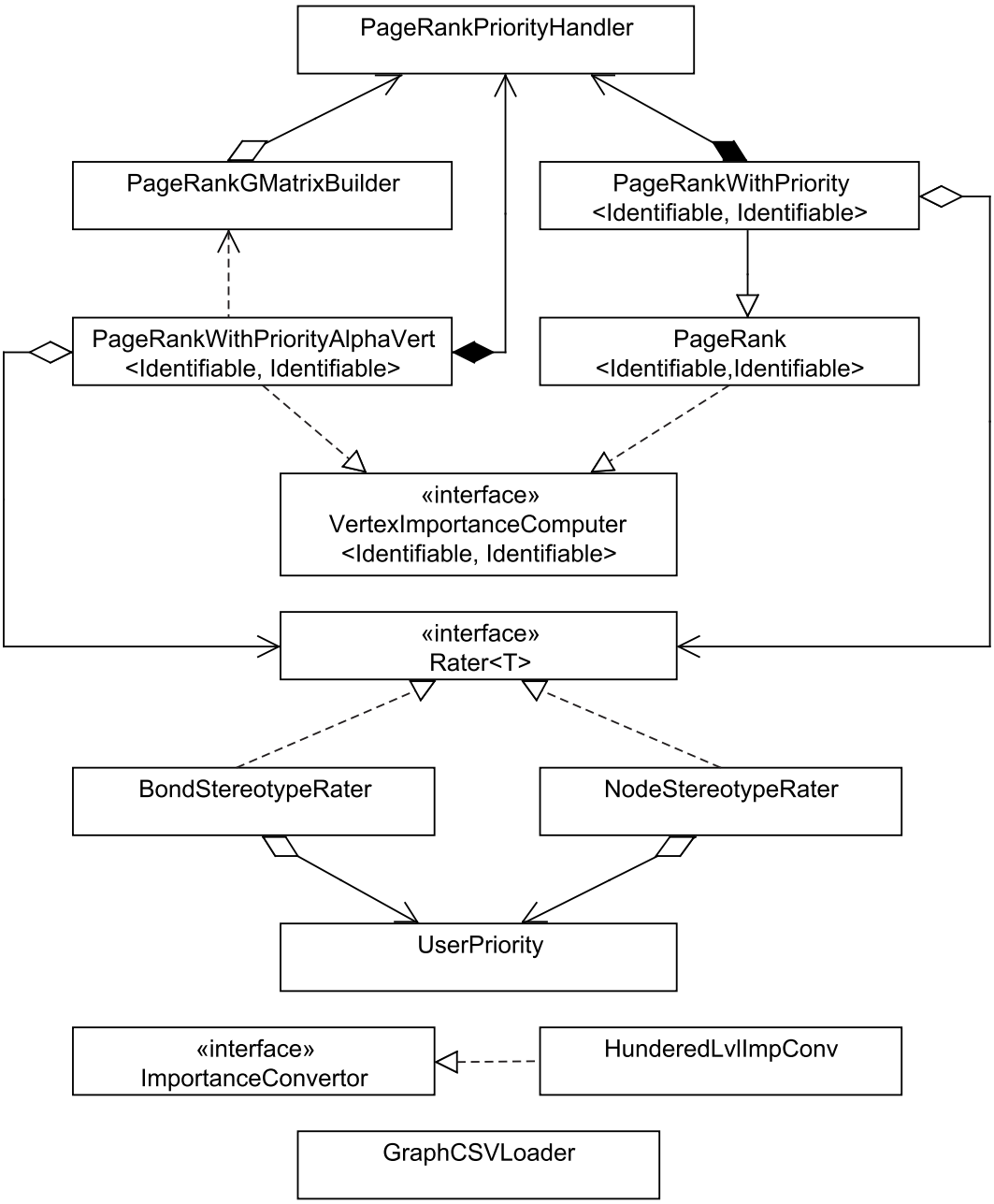
Třída `UserPriority`

Třída, která zapouzdřuje uživatelské priority jednotlivých stereotypů vrcholů a hran. Před výpočtem PageRanku, jenž bere v potaz předdefinované priority, přiřadí uživatel jednotlivým stereotypům uzlů a vazeb celočíselnou hodnotu od jedné do deseti (deset znamená nejvyšší prioritu). Třída `UserPriority` tyto hodnoty uchovává a na jejich základě pak ovlivňuje výslednou důležitost jednotlivých uzlů.

Rozhraní `Rater<T>`

Pomocí tohoto rozhraní se určuje priorita libovolného typu `T` na základě pravidel stanovených ve třídě, která `Rater<T>` implementuje. Rozhraní obsahuje jedinou metodu `rate(T)`, jež vrací hodnotu typu `double`.

Rozhraní tak využívají například varianty PageRanku, které berou v potaz předem nastavené uživatelské priority. Na základě implementace metody `rate(T)` tak lze pro jeden algoritmus vytvořit několik variant, podle kterých se bude vyhodnocovat priorita uzlů nebo hran.



Obrázek 5.2: UML diagram tříd knihovny graphRanking

Třída `NodeStereotypeRater`

Třída implementuje rozhraní `Rater<Vertex<Node,Bond>>`. Jejím prostřednictvím se provádí určení uživatelské důležitosti uzlu na základě jeho stereotypu.

Aby bylo možné určit uživatelskou prioritu daného uzlu podle jeho stereotypu, musí tato třída vlastnit odkaz na objekt třídy `UserPriority`. Tuto vazbu znázorňuje diagram tříd na obrázku 5.2.

Třída `BondStereotypeRater`

Tato třída plní stejnou roli jako `NodeStereotypeRater` s tím rozdílem, že určuje prioritu hrany. Ta se určuje nejen na základě jejího stereotypu, ale také ze stereotypu uzlu, do kterého vede.

Výsledná priorita hrany je pak vypočítána jako aritmetický průměr z priority hrany a cílového uzlu.

Rozhraní `ImportanceConvertor`

Po výpočtu libovolného ohodnocujícího algoritmu mají vrcholy stanovenou důležitost na základě metrik daného algoritmu (v případě PageRanku je suma důležitostí všech uzlů rovna jedné). Třídy implementující toto rozhraní provádějí po výpočtu převod priorit na klientem požadovaný formát.

Třída `HunderedLvlImpConv`

Tato třída reprezentuje implementaci rozhraní `ImportanceConvertor`, která rozděluje důležitost uzlů do sta úrovní – 1 pro nejnižší, 100 pro nejvyšší.

Třída `GraphCSVLoader`

Třída zajišťující načítání grafů z CSV[2] souboru. Tato třída byla vytvořena pro možnost testování algoritmů ohodnocující uzly v době, kdy ještě nebyl zajištěn přístup k grafové databázi.

Graf vždy sestává ze dvou CSV souborů, kdy první načítaný obsahoval uzly historického významu a druhý nesl jejich vzájemné vazby.

[2]CSV (Comma Separated Value) – jednoduchý souborový formát určený pro výměnu tabulkových dat.

6 Implementace PageRanku

K výpočtu důležitostí jednotlivých uzlů z grafu historických událostí byl vybrán algoritmus PageRank (popisovaný v části 2.2.3). Důvodem byla především jeho nízká časová náročnost (pro výpočet jedné iterace $O(n)$ – $O(n^2)$ v závislosti na implementaci, při 50 – 100 iteracích pro velice rozsáhlé grafy [6]) a celkem snadné uživatelské úpravě priorit na základě definovaných pravidel.

Existují dvě základní možnosti jak implementovat PageRank, a to pomocí maticových výpočtů (uvádí část 2.2.3) nebo prostřednictvím seznamů sousednosti jednotlivých uzlů. V rámci této diplomové práce byly vyzkoušeny oba dva typy. Z těchto dvou variant se následně vybrala verze využívající spojových seznamů, jelikož potřebuje na výpočet méně času.

6.1 Výpočet pomocí maticových počtů

Jak bylo popsáno v kapitole 2.2.3, jedna iterace odpovídá operaci násobení vektoru maticí, přesněji $\mathbf{x_{k+1}} = \mathbf{x_k}\mathbf{G}$. Tato operace má výpočetní složitost $O(n^2)$. Připomeneme, že matici $\mathbf{G}$ určuje vztah 2.4:

$$\mathbf{G} = \alpha\mathbf{S} + (1 - \alpha)\mathbf{E}$$

kde matice $\mathbf{S}$ je upravená matice přechodů $\mathbf{A}$ tak, že nahrazuje řádky slepých uzlů řádkami $(1/n, 1/n, ..., 1/n)$.

Před samotným výpočtem se nejprve musí sestavit matice $\mathbf{G}$. Časová složitost této operace je opět $O(n^2)$, a to především vzhledem k fyzické reprezentaci grafu (spojovým seznamem sousednosti, jak bylo popsáno v kapitole 5.1), ze které se matice $\mathbf{G}$ vytváří, jelikož je potřeba pro všechny uzly grafu provést následující operace:

1. Přiřazení indexu (1 až n) všem vrcholům ohodnocovaného grafu. Tento index udává řádku, respektive sloupec, daného vrcholu v budoucí matici $\mathbf{G}$,

2. vytvoření základu matice $\mathbf{G}$, která je naplněna hodnotami $(1 - a) \cdot 1/n$ (odpovídá matici $(1 - \alpha)\mathbf{E}$),

3. pro každou hranu směřující z vrcholu u do vrcholu v je k prvku $g_{u,v}$ matice $\mathbf{G}$ přičtena hodnota $1/deg^-(u)$[1] (krok odpovídá vytvoření matice $\mathbf{A}$).

4. Pokud je vrchol u slepý (nevede z něj žádná odchozí hrana), pak je k prvkům $g_{u,1}$ až $g_{u,n}$ matice $\mathbf{G}$ přičtena hodnota $1/n$ (tento krok odpovídá vytvoření matice $\mathbf{S}$ z matice $\mathbf{A}$).

[1] $deg^-(u)$ udává odchozí stupeň vrcholu u.

Po sestavení matice $\mathbf{G}$ je vytvořen počáteční vektor důležitostí $\mathbf{x_0} = (1/n, 1/n, ..., 1/n)$. Nyní se mohou počítat jednotlivé iterace.

Zastavovací podmínka je dána jednotkovou normou (viz vzorec 2.6) rozdílu vektorů důležitostí vrcholů dvou posledních iterací ($\mathbf{x_{k+1}}$ a $\mathbf{x_k}$). Pokud je jednotková norma menší než hodnota ε, pak zastavíme výpočet (volbu hodnoty ε popisuje kapitola 9.4).

6.2 Výpočet pomocí seznamu sousednosti

Jak lze vidět výše, sestavení matice $\mathbf{G}$ je poměrně zdlouhavý proces, navíc i následný výpočet iterace má složitost $O(n^2)$.

Uvážíme-li fakt, že matice $\mathbf{A}$ je velice řídká (každý historický uzel je spojen jen s několika málo dalšími), bude pro nás výhodnější počítat důležitost zvlášť pro jednotlivé uzly.

Nejprve však upravíme maticový iterační vzorec pro PageRank analogicky k vzorci 2.7 podle zdroje [8]:

$$\begin{aligned} \mathbf{x_{k+1}} &= \mathbf{x_k}\mathbf{G} \\ &= \mathbf{x_k}(\alpha\mathbf{S} + (1-\alpha)\mathbf{E}) \\ &= \alpha\mathbf{x_k}\mathbf{S} + (1-\alpha)\mathbf{e} \end{aligned} \tag{6.1}$$

kde $\mathbf{e}$ je n-dimenzionální vektor, jehož všechny složky obsahují hodnotu $1/n$.

Touto úpravou jsme se zbavili matice $\mathbf{E}$, a můžeme tak snadno vypočítat iteraci jednoho vrcholu pomocí následujícího vzorce:

$$R(u)_{k+1} = \alpha\Big(\sum_{v \in B_u} \frac{R(v)_k}{deg^-(v)} + \sum_{q \in D} \frac{R(q)_k}{n}\Big) + \frac{1-\alpha}{n} \tag{6.2}$$

Kde $R(u)_k$ udává důležitost vrcholu u v k-té iteraci. B_u je množina vrcholů, jež mají odchozí hranu vedoucí do vrcholu u. Odchozí stupeň vrcholu v udává $deg^-(v)$ a D je množina slepých vrcholů (vrchol x je slepý, pokud je jeho $deg^-(x) = 0$).

Při bližším pohledu na vzorec 6.2 zjistíme, že $R(u)_k$ reprezentuje jednu složku vektoru $\mathbf{x_k}$. Uzávorkovaná část vzorce je pak ekvivalentem násobení vektoru $\mathbf{x_k}$ se sloupcem matice $\mathbf{S}$, kdy ovšem toto násobení provádíme jen v případě nenulovosti daného prvku sloupce matice $\mathbf{S}$ (existence hrany (v, u) nebo pokud je vrchol v slepý). Část $\frac{1-\alpha}{n}$ pak už jen přičítá prvek vektoru $(1-\alpha)\mathbf{e}$.

Pokud tedy aplikujeme vzorec 6.2 na všechny vrcholy grafu, dostaneme ekvivalentní výsledek jako při maticovém výpočtu podle vzorce 6.1.

Rozdíl, kterého tímto přístupem dosáhneme, je v počtu operací potřebných k výpočtu jedné iterace. Jak bylo zmíněno výše, většina vrcholů grafu historických událostí nemá mnoho sousedních vrcholů. Pokud budeme uvažovat, že průměrný vrchol má 10 sousedů a dalších 10 vrcholů je slepých, budeme potřebovat k výpočtu jedné iterace nad jedním uzlem 20 kroků. Výpočet iterace celého grafu si

pak vyžádá $n \cdot 20$ kroků, tím dosáhneme kýžené výpočetní složitosti $O(n)$ pro výpočet jedné iterace PageRanku.

Samozřejmě, pokud bude testovaný graf velice hustě protkán hranami, dosáhne nastíněný postup složitosti $O(n^2)$ jako v případě maticových počtů.

6.2.1 Implementace

Výše popsaný postup při výpočtu jedné iterace PageRanku implementuje třída `PageRank` balíku `graph-ranking`. Jak lze vidět na obrázku 5.2 znázorňující diagram tohoto balíku, tato třída implementuje rozhraní `VertexImportanceComputer`.

Pro inicializaci objektu této třídy je zapotřebí stanovit hodnoty α (teleportační konstanta v intervalu $(0, 1)$) a ε (zastavovací podmínka v intervalu $(0, \infty)$). Po inicializaci již stačí zavolat metodu `computeVertexImportances()`, kde se jako parametr předá graf, nad jehož vrcholy chceme vypočítat důležitost. Diagram na obrázku 6.1 znázorňuje jednotlivé kroky výpočtu.

6.3 PageRank s prioritními hranami

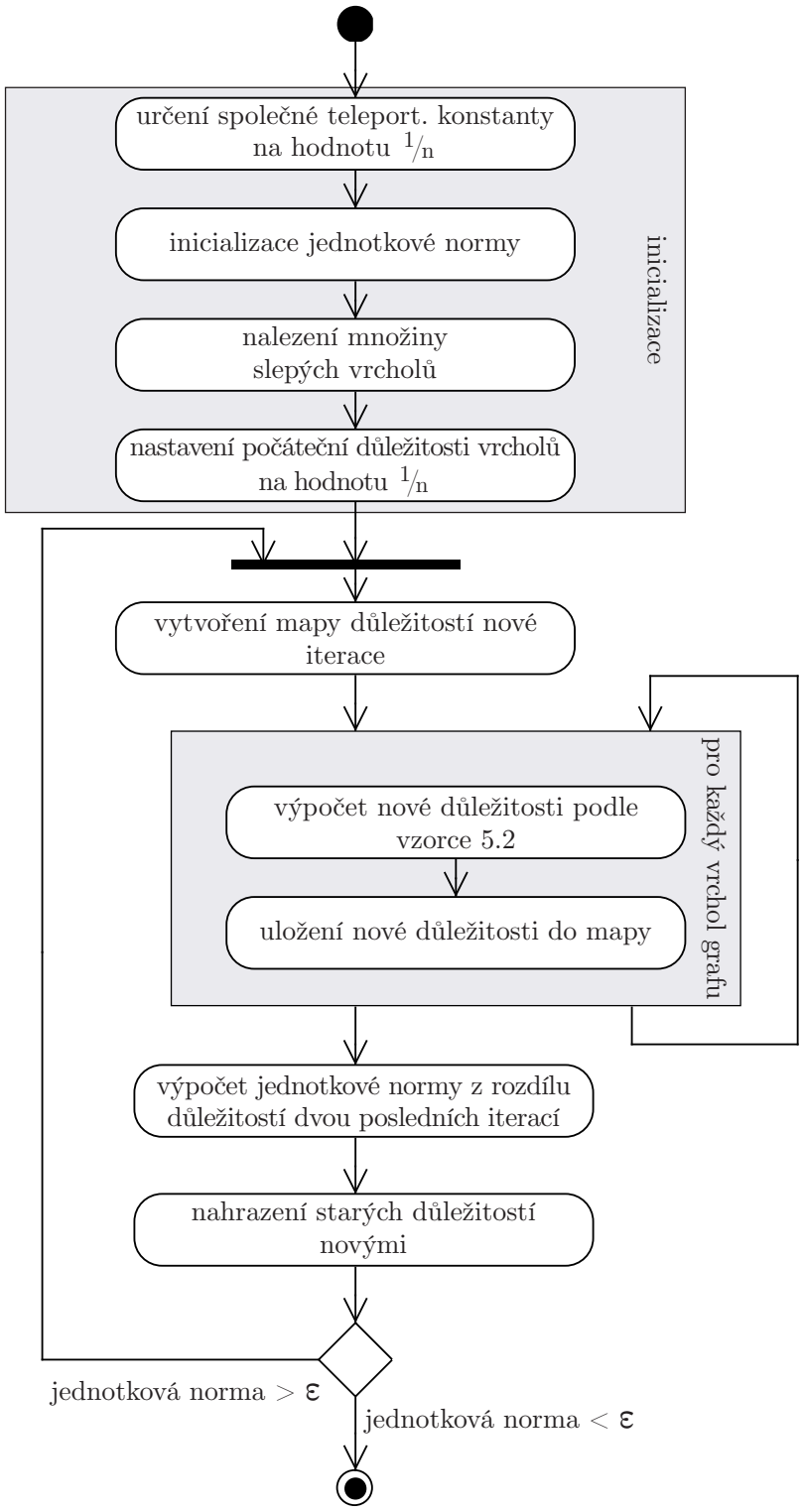
Implementace této modifikace PageRanku odpovídá postupu popsanému v kapitole 2.2.4 (část Úprava přechodové matice), kde jsme jednotlivým hranám přiřadili jejich priority podle aktuální potřeby.

Priority můžeme využít u vazeb grafu historických událostí, jelikož každá vazba obsahuje svůj stereotyp (popis jednotlivých stereotypů viz 2.1). Uživatel pak jako součást dotazu stanoví každému stereotypu prioritu na základě svých aktuálních potřeb a v průběhu výpočtu PageRanku bude na tyto priority brán zřetel. Priority jsou zadány zároveň i pro jednotlivé typy historických uzlů.

6.3.1 Správa uživatelských priorit

Uživatelem zadané priority historických uzlů a jejich vazeb je potřeba před výpočtem zpracovat a následně uchovávat v snadno a rychle dostupné podobě. Za tímto účelem vznikla třída `PageRankPriorityHandler` (znázorněna v diagramu tříd knihovny `graph-ranking` na obrázku 5.2). Tato třída poskytuje veřejné metody `getVertexPriority()` pro získání uživatelské priority vrcholu, `getEdgePriority()` vracející uživatelskou prioritu hrany a `getVertexAlpha()`, která vrací teleportační konstantu vrcholu (tato metoda je využívána v rámci modifikace PageRanku pomocí matice teleportačních konstant, jejíž implementaci popisuje kapitola 6.4).

Aby bylo možné znovu použít všechny implementace PageRanku i pro grafy, které nereprezentují historické události, nepracuje třída `PageRankPriorityHand-`



Obrázek 6.1: Diagram jednotlivých kroků při výpočtu PageRanku třídou `PageRank`

`ler` přímo s uživatelskými prioritami, ale využívá tříd implementujících rozhraní `Rater`, přesněji `NodeStereotypeRater` a `BondStereotypeRater`, které zprostředkovávají uživatelské priority pomocí své metody (více o těchto třídách viz 5.2).

Uživatelské priority se zpracovávají již při inicializaci této třídy, a to v následujících krocích:

1. Vytvoření mapy, kde klíčem je uzel a hodnotou jeho uživatelská priorita. Tato mapa obsahuje všechny uzly grafu, nad kterým proběhne výpočet PageRanku. Při vytváření této mapy se počítá suma všech priorit, aby následně mohly být všechny priority uzlů normalizovány tak, že jejich suma přes všechny uzly dá hodnotu jedna.

2. Analogicky se vytvoří mapa, kde klíčem je hrana a hodnotou její normalizovaná priorita (suma priorit přes odchozí hrany každého uzlu je rovna jedné).

3. Pokud připravujeme priority uzlů a hran pro výpočet PageRanku s maticí teleportačních konstant (viz 6.4), vytvoříme mapu, kde klíčem je uzel a hodnotu tvoří jeho teleportační konstanta. Postup pro vytváření teleportačních konstant jednotlivých uzlů popisuje kapitola 2.2.4, konkrétně část Úprava teleportační matice.

Po inicializaci již instance třídy `PageRankPriorityHandler` poskytuje přístup k normalizovaným prioritám vrcholů a hran prostřednictvím metod zmíněných výše, kdy se jako parametr se předává vrchol, respektive hrana, jejíž prioritu chceme zjistit.

6.3.2 Výpočet

Pro výpočet PageRanku s prioritně ohodnocenými hranami musíme upravit vzorec 6.2 tak, aby do výsledné důležitosti byly započítány právě priority jednotlivých uzlů a hran.

Upravený vzorec pro výpočet jedné iterace PageRanku nad jedním vrcholem pak vypadá následovně:

$$R(u)_{k+1} = \alpha\Big(\sum_{v \in B_u} R(v)_k \cdot P_E(v,u) + \sum_{q \in D} R(q)_k \cdot P_V(u)\Big) + (1-\alpha) \cdot P_V(u) \quad (6.3)$$

Vzorec výše používá dvě nové funkce – $P_E(v,u)$ a $P_V(u)$. První jmenovaná vrací prioritu hrany mezi vrcholy u a v, tato funkce odpovídá metodě `getEdgePriority()` třídy `PageRankPriorityHandler`. Druhá pak analogicky vrací prioritu vrcholu u (metoda `getVertexPriority()`).

Jako v případě vzorce 6.2, i zde představuje uzávorkovaná část násobení vektoru $\mathbf{x_k}$ se sloupcem matice $\mathbf{S}$, kdy jako výsledek získáme jednu složku vektoru $\mathbf{x_{k+1}}$. Nyní však nedělíme důležitost vstupního uzlu počtem jeho odchozích hran, ale

násobíme ji normalizovanou prioritou hrany, která vede mezi vstupním vrcholem a vrcholem, pro který počítáme jeho důležitost.

Ani *virtuální* hrany vedoucí ze slepých vrcholů do všech již neobsahují stejnou prioritu, ale řídí se prioritou uzlů, do nichž vedou. Totožně je naloženo i s teleportační částí vzorce – $(1 - \alpha) \cdot P_V(u)$.

6.3.3 Implementace

Výše zmíněný postup pro výpočet jedné iterace prioritního PageRanku implementuje třída `PageRankWithPriority`, která provádí výpočet analogicky jako třída `PageRank` (jak lze vidět v diagramu tříd na obrázku 2.2, `PageRankWithPriority` třídu `PageRank` ve skutečnosti rozšiřuje).

Obrázek 6.1 tedy vlastně popisuje i výpočet prioritního PageRanku s tím rozdílem, že v inicializační části výpočtu navíc probíhá vytváření třídy `PageRankPriorityHandler` nesoucí uživatelské priority a při samotném výpočtu důležitosti jednotlivých uzlů se místo vzorce 6.2 používá vzorec 6.3.

6.4 PageRank s maticí teleportačních konstant

Tato část popisuje modifikaci PageRanku využívající matici teleportačních konstant. Teoretický úvod k této části popisuje kapitola 2.2.4 (část Vytvoření matice teleportačních konstant).

Stručně připomeneme, že daná modifikace vytváří matici teleportačních konstant $\mathbf{X}$, ve které má každý uzel svoji teleportační konstantu. Jednotlivým vrcholům se pak přiřazují jejich teleportační konstanty na základě priorit odchozích hran (čím menší průměrná priorita odchozích hran, tím menší teleportační konstanta vrcholu).

Nahradíme tedy matici teleportačních konstant $\mathbf{X}$ za obecnou teleportační konstantu α, čímž následně dostaneme výslednou matici $\mathbf{G}$ (viz vzorec 2.10 z kapitoly 2.2.4) :

$$\mathbf{G} = \mathbf{XS} + (\mathbf{I} - \mathbf{X})\mathbf{T}$$

Jak již bylo zmíněno v kapitole 2.2.4, nevýhodou této modifikace je fakt, že část $(\mathbf{I} - \mathbf{X})\mathbf{T}$ už nemůžeme převést na vektor jako ve vzorci 6.1. Nyní tedy dostaneme matici, která není řídká, tudíž výpočetní složitost jedné iterace této modifikace PageRanku je $O(n^2)$ (násobení vektoru o dimenzi n s $n \times n$ maticí).

6.4.1 Sestavení matice G

Před samotným výpočtem tedy musíme sestavit matici $\mathbf{G}$, tento úkol obstarává třída `PageRankGMatrixBuilder`. Ta navenek vystupuje jako knihovní, pouze s jedinou statickou metodou `createGMatrix()`, které se jako parametry předávají graf `Graph`, správce uživatelských priorit `UserPriorityHandler` a mapa, kde

klíčem je vrchol `Vertex` a hodnotu tvoří celé číslo. Tato mapa reprezentuje řádky, respektive sloupce matice **G**, na kterých budou dané vrcholy ležet.

Knihovna lineární algebry

Jelikož tato verze PageRanku již pracuje s maticemi a vektory, bylo potřeba vyhledat Java knihovnu implementující tyto matematické objekty a operace nad nimi.

Jako nejlepší možné řešení byla vybrána knihovna ojAlgo [13] (publikovaná pod MIT licencí[2]), a to především na základě výsledků benchmarku Java knihoven implementujících lineární algebru [14], ze kterého tato knihovna vychází jako nejrychlejší při operaci násobení matic (respektive vektoru s maticí; vektor je touto knihovnou reprezentován jako jednořádková nebo jednosloupcová matice), které se využívá v jednotlivých iteracích PageRanku.

Knihovna sestavuje matice prostřednictvím takzvaných *builderů*, což jsou třídy usnadňující naplňování matice hodnotami. Po sestavení matice do požadovaného tvaru se pak nad tímto builderem zavolá metoda `build()`, která vrací instanci matice (rozhraní `BasicMatrix` v našem případě).

Životní cyklus třídy `PageRankGMatrixBuilder`

Jak bylo výše zmíněno, třída `PageRankGMatrixBuilder` mající na starosti sestavení matice **G** navenek vystupuje jako knihovní. Ve skutečnosti se při zavolání její metody `createGMatrix()` vytvoří její instance, která po vytvoření matice zaniká (respektive zaniká až ji Java garbage collector odstraní).

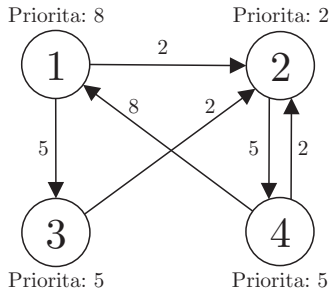
Životní cyklus instance této třídy vypadá následovně (postup vytváření matice **G**, respektive její jedné řádky ilustruje obrázek 6.2):

1. Vytvoření instance a uložení odkazů na graf, správce uživatelských priorit a mapu indexovaných vrcholů. Dále se vytvoří instance *builderu* matice **G**.

2. Pro každý vrchol grafu se vytvoří jeho řádka a vloží se do matice **G**. Tento postup je názorně předveden na obrázku 6.2, kde hodnoty matice **X**, matice **S** a vektoru **t** nese správce uživatelských priorit (`UserPriorityHandler`). Také je třeba zdůraznit, že získání vektoru **t** (viz bod 1 obrázku 6.2) se provádí jen jednou v průběhu celého vytváření matice **G**.

3. Sestavení matice **G** a její navrácení jako výstup z metody `createGMatrix()`.

[2]Znění licenčních podmínek jsou k nalezení na `http://opensource.org/licenses/MIT`.



$$\mathbf{X}=\begin{pmatrix} 0{,}725 & 0 & 0 & 0 \\ 0 & 0{,}85 & 0 & 0 \\ 0 & 0 & 0{,}35 & 0 \\ 0 & 0 & 0 & 0{,}85 \end{pmatrix}$$

$$\mathbf{S}=\begin{pmatrix} 0 & {}^{2}\!/_{7} & {}^{5}\!/_{7} & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 \\ {}^{4}\!/_{5} & {}^{1}\!/_{5} & 0 & 0 \end{pmatrix}$$

$$\mathbf{t} = \left({}^{2}\!/_{5}, {}^{1}\!/_{10}, {}^{1}\!/_{4}, {}^{1}\!/_{4}\right)$$

Postup vytváření 1. řádky $\mathbf{g_1}$ matice $\mathbf{G}$

1. získání vektoru $\mathbf{t}$
$\mathbf{t}= \left({}^{2}\!/_{5}, {}^{1}\!/_{10}, {}^{1}\!/_{4}, {}^{1}\!/_{4}\right)$

2. nalezení teleport. konstanty uzlu z matice $\mathbf{X}$
$\alpha_1 = 0{,}725$

3. vynásobení priorit uzlů hodnotou (1-α)
$\mathbf{s_1} = (0, {}^{2}\!/_{7}, {}^{5}\!/_{7}, 0)$

4. vytvoření řádky $\mathbf{g_1}$
$\mathbf{g_1} = \alpha_1 \cdot \mathbf{s_1} + (1-\alpha_1) \cdot \mathbf{t}$
$= \left(\frac{11}{100}, \frac{657}{2800}, \frac{657}{1120}, \frac{11}{160}\right)$

5. vložení do matice $\mathbf{G}$

Obrázek 6.2: Postup třídy `PageRankGMatrixBuilder` *při vytváření jedné řádky matice* $\mathbf{G}$ *na ukázkovém grafu z obrázku 2.5*

6.4.2 Výpočet

Celý výpočet této modifikace PageRanku zajišťuje třída `PageRankWithPriority-AlphaVert` implementující známé rozhraní `VertexImportanceComputer`. Tato třída má na starosti vytvoření výše zmiňované mapy určující čísla řádek jednotlivých vrcholů v matici $\mathbf{G}$. Dále pak inicializuje instanci třídy pro správu uživatelských priorit a následně volá metodu `createGMatrix()` třídy `PageRank-GMatrixBuilder` vytvářející matici $\mathbf{G}$ (viz výše).

Po vytvoření matice $\mathbf{G}$ se již začínají počítat jednotlivé iterace PageRanku podle vzorce 2.5:

$$\mathbf{x_{k+1}} = \mathbf{x_k}\mathbf{G}$$

Zastavovací podmínka je stejná jako u předchozích variant, tedy opět využijeme jednotkové normy rozdílu vektorů $\mathbf{x_{k+1}}$ a $\mathbf{x_k}$. Pokud bude tato norma menší než zvolené ε, zastavíme výpočet (podrobněji v části Zastavovací podmínka kapitoly 2.2.3).

7 Knihovna pro vnější přístup

Předchozí dvě kapitoly popisovaly implementaci grafu a jednotlivých ohodnocovacích algoritmů.

Tato se věnuje knihovně, která zajišťuje komunikaci se zdrojem grafových dat a disponuje jednoduchým klientským rozhraním, jež poskytuje službu zajišťující získání požadovaného podgrafu na základě uživatelského dotazu a dalšími službami (viz kapitola 4 věnující se návrhu jednotlivých částí této práce).

Tato knihovna je vytvořena pouze pro práci s grafem historických událostí na rozdíl od dvou předešlých, které mohou být aplikovány na graf, jehož uzly a hrany jsou reprezentovány libovolnými entitními třídami (pomocí generických tříd).

Na obrázku 7.1 je k nahlédnutí diagram tříd této knihovny. Následující podkapitoly se věnují popisu významu jednotlivých tříd a rozhraní, které jsou na tomto diagramu zobrazeny.

Rozhraní `GraphDataSource`

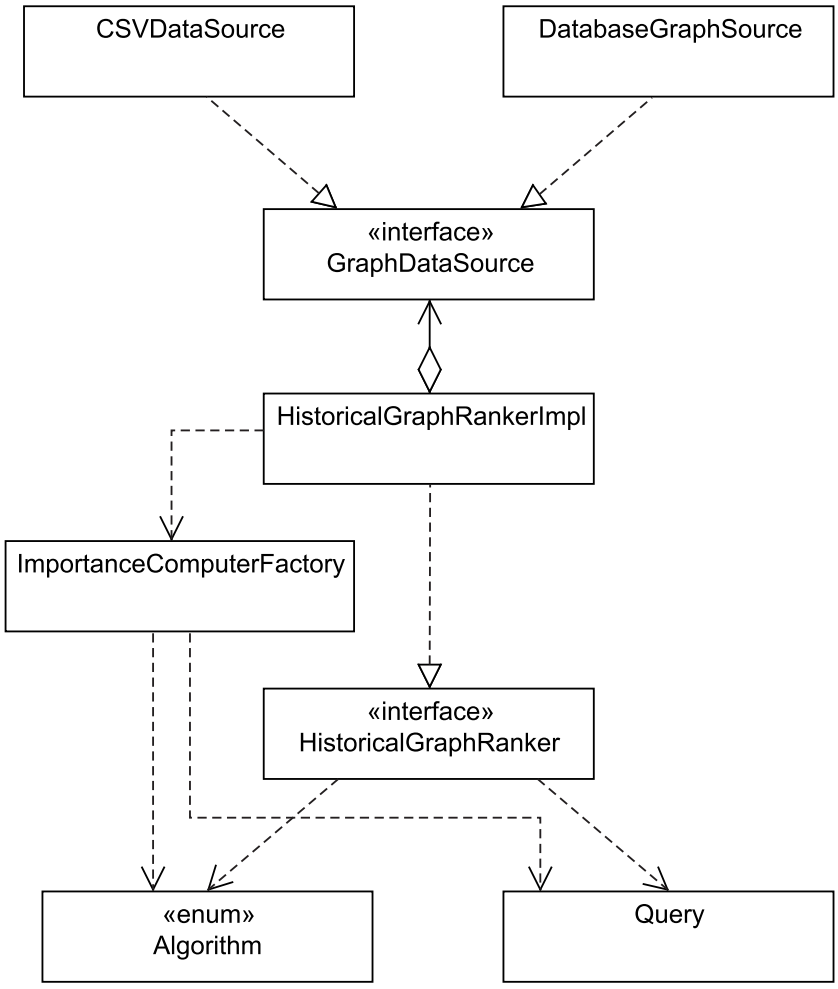
Rozhraní definující metody, kterými musí disponovat třída zajišťující získání grafových dat s historickými událostmi z libovolného datového zdroje. Význam jednotlivých metod popisuje následující seznam:

- `loadGraph()` – Hlavní metoda, jejíž návratovou hodnotou je graf historických událostí s vrcholy splňujícími podmínky, které byly zadány jako parametry této metody. Těmito parametry jsou pak časy začátku a konce období, jež uživatele právě zajímá. Dále uživatelem specifikovaná mapa vlastností, kterými mají hledané vrcholy disponovat. Poslední parametr udává, jak velké okolí vyhovujících vrcholů bude součástí vráceného grafu.

- `getNodeNeighbourhood()` – Metoda s parametry ID uzlu a hloubku okolí. Na základě těchto parametrů vrací požadovaný graf.

- `insertNode()` a `insertBond()` – Metody pro vložení nového vrcholu, respektive hrany. Návratové hodnoty udávají ID nově vložených prvků grafu. Rozhraní obsahuje také přetížené verze těchto metod, jež obsahují navíc `boolean` parametr `checkDuplicity` pro případ, že třída implementující toto rozhraní obsahuje logiku na rozpoznání duplicitních vrcholů, respektive hran. Pokud by se pak vkládal uzel nebo hrana, které jsou duplicitní, nebyly by do zdroje historických dat vloženy.

- `getNodesProperties()` – Tato metoda vrací seznam všech názvů vlastností, které obsahují uzly grafu.



Obrázek 7.1: UML diagram tříd zobrazující jednotlivé třídy a rozhraní, ze kterých se skládá knihovna, jež zajišťuje komunikaci se zdrojem grafových dat a poskytuje veřejné uživatelské rozhraní pro získání uživatelem požadovaného podgrafu

Třída `DatabaseGraphSource`

Třída implementující rozhraní `GraphDataSource`. Tato třída poskytuje spojení s grafovou databází, respektive jejím rozhraním `IDatabase` provádějícím objektové mapování.

Třída `CSVDataSource`

Tato třída sloužila jako dočasný zdroj grafových dat pro testovací účely v době, kdy ještě nebyla realizovatelná komunikace s databází. Jako parametry konstruktoru této třídy je možné zadat CSV soubory s vrcholy a hranami, z nichž se sestaví graf historických událostí. `CSVDataSource` nad takovýmto grafem následně poskytuje služby (s omezenou funkčností) rozhraní `GraphDataSource`.

Rozhraní `HistoricalGraphRanker`

Rozhraní představující fasádu pro klientskou část výsledné aplikace pro zobrazování historických událostí na časové ose.

Většina poskytovaných služeb pouze překrývá metody rozhraní `GraphDataSource`. Rozdíl mezi těmito dvěma rozhraními spočívá v tom, že `HistoricalGraphRanker` vrací v případě hledání grafu podle dotazu graf s prioritně ohodnocenými vrcholy a klientská část si dokonce může zvolit, jakým algoritmem bude vrácený graf ohodnocen.

Rozhraní `HistoricalGraphRanker` poskytuje následující metody:

- `getNodesProperties()` – Metoda se stejnou funkčností jako u rozhraní `GraphDataSource`.
- `getGraphFromQuery()` – Hlavní metoda celého rozhraní. Jejím parametrem je instance třídy `Query` (bude popsána níže), která představuje uživatelský dotaz se všemi jeho atributy.

  Na základě tohoto uživatelského dotazu se nalezne požadovaný graf, který je prioritně ohodnocen algoritmem, jejž specifikuje daná implementace rozhraní `HistoricalGraphRanker`.

  Tato metoda má také svoji přetíženou verzi, kdy se jako parametr explicitně stanoví algoritmus (pomocí výčtového typu `Algorithm`), který má být na ohodnocení grafu použit.
- `getNodeById()` – Metoda pracuje analogicky jako `getNodeNeighbourhood()`, jež je součástí rozhraní `GraphDataSource`. Jediným rozdílem může být, že v rozhraní `HistoricalGraphRanker` existují dvě přetížené verze – první, která vrací pouze daný vrchol podle jeho ID, a druhá u níž můžeme stanovit hloubku okolí.
- `insertNode()` a `insertBond()` – Obě metody mají stejnou funkcionalitu jako v rozhraní `GraphDataSource` a to včetně jejich přetížených variant.
- `getDefaultQueryDepth()` a `setQueryNeighbourDepth()` – Tyto metody vracejí (respektive nastavují) hloubku okolí vrcholů, které vyhovují uživatelskému dotazu zadanému v metodě `getGraphFromQuery()`. Vrcholy, které jsou v hloubce, jež je menší nebo rovna nadefinované hodnotě, jsou vraceny jako součást grafu při volání zmíněné metody.

Třída `HistoricalGraphRankerImpl`

Tato třída implementuje výše popsané rozhraní. Její konstruktor pak potřebuje jako parametr instanci třídy, jež implementuje rozhraní `GraphDataSource`, ze které čerpá grafová data. Druhým parametrem je instance implementující rozhraní `ImportanceConvertor` (viz kapitola 5.2), jež udává formátování důležitostí výstupních ohodnocených grafů.

Třída `Query`

Třída, která reprezentuje uživatelský dotaz na zobrazení specifického podgrafu. Na základě údajů obsažených v `Query` se z grafové databáze získá hledaný podgraf, který je následně ohodnocen s využitím uživatelských priorit, jež tato třída také obsahuje. Třída `Query` může nést dva typy dotazů:

- Dotaz specifikovaný pomocí požadovaných vlastností, jenž musí obsahovat hledané vrcholy.

  V tomto případě uživatel dostane seznam vlastností jednotlivých vrcholů (pravděpodobně v určité formě formuláře). Vlastnosti, které chce specifikovat vyplní a odešle ke zpracování. Třída `Query` pak tyto údaje nese ve formě mapy, kde klíčem je název vlastnosti a hodnotou uživatelem vyplněný údaj.

- Dotaz specifikovaný seznamem identifikátorů vrcholů a hran. Tento typ dotazu je použit ve chvíli, kdy klientská část chce například přepočítat důležitosti vrcholů nad již zobrazovaným grafem. Specifikuje tedy přesně vrcholy a hrany (pomocí jejich identifikátorů), nad kterými bude spuštěn ohodnocovací algoritmus.

Přičemž typ dotazu je závislý pouze na použitém konstruktoru této třídy (konstruktor s mapou vlastností vs. konstruktor s kolekcemi identifikátorů vrcholů a hran).

Dále nese tato třída interval (rozmezí mezi dvěma časy), ve kterém chce uživatel provádět vyhledávání historických událostí.

Poslední částí dotazu jsou uživatelem přiřazené priority jednotlivých stereotypů (v rozmezí 1 až 10, blíže viz 2.2.4) vrcholů a hran. Třída `Query` uchovává tyto údaje také v mapě, kde klíčem jsou jednotlivé výčtové typy a hodnotou přiřazená priorita.

Výčtový typ `Algorithm`

Výčtový typ vnořený ve třídě `ImportanceComputerFactory`, který umožňuje klientské straně specifikovat, jaký algoritmus chce použít při ohodnocování grafu. Tento výčtový typ je používán například ve výše popsané metodě `getGraphFromQuery()` rozhraní `HistoricalGraphRanker`.

Třída `ImportanceComputerFactory`

Tato třída představuje továrnu na ohodnocovací algoritmy, která využívá `HistoricalGraphRankerImpl`. Třída poskytuje pouze jednu veřejnou (statickou) metodu `createVertexImpComp()`, jež na základě jména algoritmu (výčtový typ `Algorithm`) a uživatelského dotazu `Query` vrací instanci požadovaného algoritmu připravenou k použití. `ImportanceComputerFactory` nabízí také přetíženou verzi této metody, ve které můžeme stanovit podrobnější nastavení požadovaného algoritmu (pomocí mapy vlastnost – hodnota).

8 REST server

Jak popisuje kapitola 4, nad aplikační vrstvou jsou vytvořeny dva typy vizualizace. První je implementována v jazyce Java. Té jsou poskytnuty služby pro získání prioritně ohodnoceného grafu prostřednictvím knihovny popsané v předchozí kapitole. Druhý typ vizualizace je implementován technologiemi HTML, JavaScript a CSS. Pro tento typ jsou data zpřístupněna pomocí serveru poskytující REST služby analogické k knihovně popsané v předešlé kapitole.

Většina REST serverů používá k výměně dat formát XML[1] nebo JSON[2]. XML má oproti JSON výhodu možnosti definovat svoji strukturu pomocí XSD[3] souboru, čímž je pak zajištěna snadná replikace požadované struktury i v klientské aplikaci – stačí, aby se řídila XSD souborem. Formát JSON má na druhou stranu mnohem lepší poměr užitečných dat ku režijním [15](strana 399). Pokud bychom tedy stejná data posílali pomocí formátu JSON a XML, JSON se přenese mezi serverem a klientem rychleji. To je vzhledem k možné rozsáhlosti odesílaných dat podstatným hlediskem, a proto jsou veškerá posílaná data mezi klientem a serverem ve formátu JSON.

8.1 Návrh REST rozhraní

Kapitoly níže popisují, jak byly navrženy jednotlivé poskytované služby. Všechny tyto služby jsou analogické k metodám třídy `HistoricalGraphRanker`, jež byla popsána v předchozí kapitole.

8.1.1 Získání ohodnoceného grafu na základě uživatelského dotazu

Tato služba poskytuje stejnou funkčnost jako metoda `getGraphFromQuery()` třídy `HistoricalGraphRanker`. Vrací tedy prioritně ohodnocený graf na základě uživatelského dotazu. I všechny potřebné parametry jsou se zmiňovanou metodou shodné. Strukturu požadavku i odpovědi lze vidět na obrázku 8.1. Pro větší přehlednost dotazu byla zvolena forma požadavku pomocí HTTP metody `POST`, jelikož pokud by byla použita metoda `GET`, musely by všechny parametry být součástí URI[4] dotazu.

[1] **XML** (eXtensible Markup Language) – Obecný značkovací jazyk vyvinutý konsorciem W3C. Umožňuje vytváření značkovacích jazyků určených k nejrůznějším typům užití.

[2] **JSON** (JavaScript Object Notation) – Způsob zápisu dat (datový formát) nezávislý na počítačové platformě, určený pro přenos dat, která mohou být organizována v polích nebo agregována v objektech.

[3] **XSD** (XML Schema Definition) – Soubor, který popisuje strukturu XML dokumentu.

[4] **URI** (Uniform Resource Identifier) – Textový řetězec s definovanou strukturou, který slouží k přesné specifikaci zdroje informací (ve smyslu dokument nebo služba), hlavně za účelem jejich použití pomocí počítačové sítě, zejména Internetu.

Požadavek

relativní URI: `/importances`
metoda: `POST`
tělo:

```
{
    "query": {
       "properties": {
            "name": "Karel IV.",
            "place": "České království"
        },
       "from": "1300-01-01T00:00:00",
       "to": "1400-01-01T00:00:00",
       "nodeImportance": {
            "person": 10,
            "event": 7,
            "place": 5,
            "item": 1
        },
        "edgeImportance": {
            "relationship": 7,
            "interaction": 10,
            "participation": 5,
            "creation": 5,
            "cause": 8,
            "part_of": 3,
            "takes_place": 1
        }
    }
}
```

Odpověď

status: `200 OK`
tělo:

```
{
    "nodes" : [
        {
            "id": 1,
            "importance": 0.21,
            "outEdges": [1,3,8],
            "inEdges": [2,5,15],
            "name": "Karel IV.",
            "stereotype": "person",
            "begin": "1316-05-14T00:00:00"
            "end": "1378-29-11T00:00:00"
            "properties" : {
                ...
            }
        },
        ...
    ],
    "edges": [
        {
            "id": 3,
            "from": 1,
            "to": 12,
            "stereotype": "partOf",
            "properties": {
                ...
            }
        },
        ...
    ]
}
```

Obrázek 8.1: Ukázka komunikace mezi klientem a REST serverem při dotazování na ohodnocený graf

Stejně jako u metody `getGraphFromQuery()` lze poslat alternativní dotaz, kdy klientská strana má již vytvořený graf, na kterém chce pouze přepočítat důležitosti jednotlivých uzlů. Ten se liší od dotazu znázorněného na obrázku 8.1 tak, že klíče `properties`, `from` a `to`, jsou nahrazeny klíči `nodes` a `edges`, které obsahují pole identifikátorů uzlů a hran grafu, nad nímž se mají důležitosti vypočítat.

Klientské straně se vrací odpověď v podobě polí `nodes` a `edges` (viz pravá část obrázku 8.1) nesoucí uzly a hrany grafu. Jednotlivé uzly a hrany obsahují základní údaje. Další, pro jednotlivé uzly specifické vlastnosti jsou pak obsaženy pod klíčem `properties`.

Hrany mají své vlastnosti uloženy totožně. Navíc uchovávají identifikátory zdrojového a cílového uzlu, aby bylo možné po přenosu opět graf sestavit.

8.1.2 Získání názvů všech vlastností

Služba analogická k metodě `getNodesProperties()` třídy `HistoricalGraphRanker`. Poskytuje tedy názvy všech vlastností uzlů. Na obrázku 8.2 lze vidět strukturu požadavku a odpovědi této služby.

Požadavek
relativní URI: `/properties`
metoda: `GET`

Odpověď
status: `200 OK`
tělo: `["propertyName1", "propertyName2", "propertyName3"]`

Obrázek 8.2: Ukázka komunikace mezi klientem a REST serverem při dotazování na názvy všech vlastností uzlů

8.1.3 Získání uzlu podle jeho ID

Služba umožňující získání uzlu podle jeho ID s možností vracení jeho okolí do určité hloubky. Opět se jedná o analogii k metodě třídy `HistoricalGraphRanker`. Tentokrát se jedná o metodu `getNodeById()`. Obrázek 8.3 pak zobrazuje strukturu požadavku a odpovědi této služby.

Jak lze na obrázku vidět, klient si může zvolit libovolně velké okolí pomocí dotazu uvnitř URI, kde klíčem je `depth` a hodnotou požadovaná hloubka. Tento klíč je nepovinný, pokud tedy klient chce získat jen uzel s daným ID, nemusí do dotazu URI vkládat žádné další informace.

Struktura odpovědi je identická se službou poskytující ohodnocený graf na základě uživatelského dotazu, vrací se tedy pole uzlů a hran. Jediný rozdíl představuje fakt, že tato služba nevrací důležitosti jednotlivých uzlů.

8.1.4 Vkládání uzlů nebo hran

Vkládání nových uzlů nebo hran jsou posledními službami, které toto REST rozhraní nabízí. I tyto služby jsou dvojníky metod třídy `HistoricalGraphRanker`, nyní `insertNode()` a `insertEdge()`.

Obrázek 8.4 zobrazuje komunikaci mezi klientem a serverem při vkládání uzlu (levá část) nebo hrany (pravá část).

Levá část obrázku, zobrazující vkládání nového uzlu, uvádí všechny potřebné údaje, které musí mít vkládaný uzel (kromě vlastnosti `end`, jelikož žijící osobnosti

Požadavek

relativní URI: `/node/18?depth=2`
metoda: `GET`

Odpověď

status: `200 OK`
tělo:

```
{
        "nodes": [
            {
                "id": 1,
                "outEdges": [1,3,8],
                "inEdges": [2,5,15],
                "name": "2. světová válka",
                "stereotype": "event",
                "begin": "1939-01-09T00:00:00"
                "end": "1945-02-09T00:00:00"
                "properties": {
                    ...
                }
            },
            ...
        ],
        "edges": [
            {
                "id": 3,
                "from": 1,
                "to": 12,
                "stereotype": "partOf",
                "properties": {
                    ...
                }
            },
            ...
        ]
    }
```

Obrázek 8.3: Ukázka komunikace mezi klientem a REST serverem při získávání uzlu podle jeho ID; v tomto konkrétním případě chce klient získat uzel s ID 18 a jeho okolí hloubky 2

nebo existující místa nemohou mít stanovený konec). Pod klíčem `properties` se pak nacházejí další doplňující volitelné vlastnosti. Pokud uzlu nechceme vkládat žádné doplňující vlastnosti, pak klíč `properties` bude obsahovat prázdný objekt {} (jak je předvedeno u vkládání hrany). Pokud proběhne vložení nového uzlu úspěšně, server vrátí odpověď obsahující ID vytvořeného uzlu. To může být užitečné při následném vkládání hran, které vedou z/do vloženého uzlu.

Při vkládání hrany je potřeba znát ID obou uzlů, mezi kterými nová hrana povede. Tyto identifikátory se vkládají do samotné URI požadavku (viz relativní

Požadavek

Vkládání uzlu

relativní URI: `/node`
metoda: `POST`
tělo:

```
{
   "name": "Stavba Karlova mostu",
   "description": "Náhrada za zničený
                   Juditin most",
   "begin": "1357-06-15T00:00:00",
   "end": "1402-01-01T00:00:00",
   "stereotype": "event",
   "properties": {
      "precisionStart": "month",
      "precisionEnd": "year"
   }
}
```

Vkládání hrany

relativní URI: `/edge/14/28`
metoda: `POST`
tělo:

```
{
   "name": "manželka",
   "description": "",
   "stereotype": "relationship",
   "properties": {}
}
```

Odpověď

status: `200 OK`
tělo:

```
{
   "node-id": 16
}
```

status: `200 OK`
tělo:

```
{
   "edge-id": 35
}
```

Obrázek 8.4: Ukázka komunikace mezi klientem a REST serverem při vkládání uzlu nebo hrany

URI při vkládání hrany na obrázku 8.4, kdy do grafu vkládáme hranu vedoucí mezi uzly `14` a `28`). Vložená hrana posléze vede z prvního zadaného uzlu do druhého. Stejně jako při vkládání uzlu server pošle klientovi odpověď, ve které je ID vložené hrany.

8.1.5 Zpracování chybových stavů

Jedním z důležitých aspektů REST serveru je schopnost informovat klienta o chybách nastalých při zpracovávání jeho požadavku. Například při nevalidním zápisu JSON těla požadavku. V takovýchto případech server vrátí odpověď s chybovým HTTP kódem, který nejlépe vystihuje daný typ chyby. V těle této odpovědi se pak nachází text ve formátu JSON s klíčem `error`, jenž obsahuje řetězec popisující příčinu chyby.

8.2 Implementace REST serveru

K implementaci REST serveru byl zvolen framework Spring MVC 4, a to z následujících důvodů:

- Framework poskytuje snadnou implementaci REST serveru s využitím anotací,
- poskytuje hotové řešení mapování objektů do JSON formátu a zpět, které umožňuje další modifikace,
- využívá návrhového vzoru IoC[5], jenž programátorovi ulehčuje získávání potřebných objektů,
- je velice oblíbený, má širokou uživatelskou základnu a z toho vyplývající velké množství různých návodů a tutoriálů.

8.2.1 Nastavení serverové aplikace

Základním místem inicializace a nastavení Java webových aplikací bývá většinou XML soubor `web.xml` (takzvaný deployment descriptor), kde se provádí mapování jednotlivých servletů k URI adresám, na kterých budou poslouchat HTTP požadavky; nastavování filtrů atd. Ani Spring MVC není v tomto směru výjimkou (respektive jedna z variant inicializace aplikace využívá soubor `web.xml`), nicméně framework zde uvádí jen jeden servlet (`DispatcherServlet`), který poslouchá na všech URI patřících webové aplikaci.

Při zpracovávání HTTP požadavku framework podle analýzy URI zjistí, jakému *controlleru* jej přiřadí ke zpracování. *Controller* je speciální typ tříd, které zajišťují chování celé aplikace. Z obyčejné třídy se vytvoří controller pomocí anotace `@Controller`, respektive `@RestController` pro třídy poskytující REST služby. To, na kterých URI budou jednotlivé controllery poslouchat požadavky, se nastavuje anotací `@RequestMapping(some/path)`.

Všechny zmíněné anotace lze spojovat s třídami, `@RequestMapping` se může spojit i s metodami jednotlivých controllerů, což umožňuje, aby jeden controller přijímal HTTP požadavky s URI mající stejný kořen (třída opatřená anotací `@RequestMapping`) a různé konce (jednotlivé metody daného controlleru opatřené anotací `@RequestMapping`). U metod se touto anotací nastavuje i metoda HTTP požadavku, jeho formát těla a další vlastnosti, které musí požadavek splňovat, aby byl danou metodou zpracován.

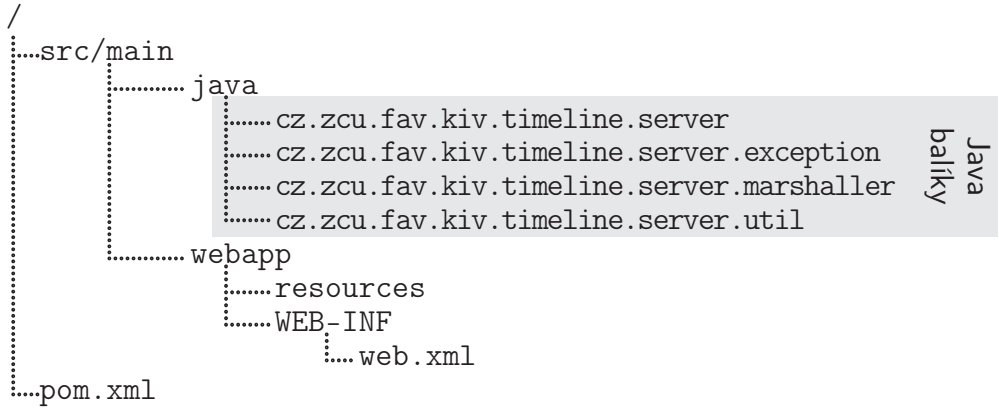
Místem, kde se inicializují jednotlivé objekty používané napříč celou aplikací (například spojení s databází), je třída s anotací `@Configuration`. Tato třída obsahuje metody označené anotací `@Bean`, které vracející instance tříd injektovaných

[5] **IoC** (Inversion of Control) – Návrhový vzor, který umožňuje uvolnit vztahy mezi jinak těsně svázanými komponentami.

na různých místech aplikace (například v samotných controllerech). Takto vytvořené objekty pak v ostatních třídách získáváme pomocí proměnných (mohou být i privátní) s anotací `@Autowired`.

8.2.2 Struktura implementace

Adresářovou hierarchii serverové aplikace zobrazuje obrázek 8.5. Jednotlivé části serveru, které jsou na obrázku vyjmenovány popisují následující podkapitoly.



Obrázek 8.5: Adresářová hierarchie serverové aplikace

Adresář `java`

V tomto adresáři se nachází veškerá implementace aplikace, která se skládá z níže popsaných balíků.

Balík `server` Tento balík obsahuje základní třídy aplikace, především třídu `ServerConfig` zajišťující nastavení celé aplikace (má anotaci `@Configuration`), jež inicializuje databázové spojení, vytváří instanci třídy implementující rozhraní `HistoricalGraphRanker` (viz kapitola 7 popisující implementaci balíku s tímto rozhraním) a provádí další akce potřebné k běhu celé aplikace.

Dále třída `DefaultRestController`, která je jediným controllerem aplikace. Zajišťuje tedy veškerou komunikaci přesně podle návrhu popsaného výše v podkapitole 8.1. Jinými slovy poskytuje metody totožné s rozhraním `HistoricalGraphRanker` ve formě REST služeb.

Popisovaná třída dědí od abstraktní třídy `AbstractController`, jejíž metody zajišťují transformaci výjimek vyhozených při zpracovávání dotazu na odpověď s popisem nastalé chyby ve formátu JSON.

Další třídou balíku je `DispatcherServlet404Throwable` rozšiřující třídu `DispatcherServlet`, jejíž účel byl popsán v podkapitole 8.2.1. `DispatcherServlet404Throwable` upravuje rodičovské třídě funkčnost tak, že v případě nenalezení controlleru pro určitý příchozí požadavek vyhodí výjimku, kterou následně zpracuje třída `AbstractController` tak, že klientské straně přijde JSON odpověď s informací o nenalezení požadované služby.

Poslední třídou tohoto balíku je knihovní třída `URIs` nesoucí konstanty reprezentující URI jednotlivých služeb.

Balík `server.exception` V tomto balíku se nachází jediná třída – `ResourceNotFoundException`, výjimka, která je vyhozena například v případě, kdy chce klient vrátit uzel s neexistujícím ID. Tuto výjimku opět zpracovává třída `AbstractController`, která ji převádí na HTTP odpověď s kódem 404 a popisem chyby uvnitř těla v JSON formátu.

Balík `server.marshaller` Tento balík obsahuje všechny třídy, které provádějí převod uzlů, hran a uživatelského dotazu z JSON formátu do `Java` objektů nebo obráceně. Následující výčet uvádí všechny třídy tohoto balíku a jejich stručný popis:

- `BondDeserializer` – Třída provádějící převod hrany z JSON formátu (hranu v JSON formátu zobrazuje pravá strana obrázku 8.4) do objektu `Bond`.
- `GraphSerializer` – Třída pro převod instance třídy `Graph<Node, Bond>` do JSON formátu (tvar JSON formátu lze vidět na obrázku 8.1 v pravé části).
- `NodeDeserializer` – Provádí převod uzlu z JSON formátu (viz obrázek 8.4 vlevo) do objektu třídy `Node`.
- `QueryDeserializer` – Třída provádějící převod uživatelského dotazu z JSON formátu (viz obrázek 8.1 vlevo) do objektu `Query`.
- `JsonDeserializationUtils` – Pomocná knihovní třída poskytující statické metody, které se využívají při serializaci/deserializaci výše popisovaných objektů.

Balík `server.util` Balík obsahující pouze knihovní třídu `TimeUtils`, která poskytuje metody pro převod času z třídy `DateTime` do řetězce ve formátu dle ISO 8601 [16] a obráceně.

Adresář `webapp`

Adresář obsahující podadresáře `resources` a `WEB-INF`. V prvním jmenovaném se mohou nacházet CSV soubory `nodes.csv` a `edges.csv`, které mohou sloužit k naplnění grafové databáze při prvním spuštění aplikace.

Druhý podadresář obsahuje soubor `web.xml`, jenž byl popsán v podkapitole 8.2.1.

Soubor `pom.xml`

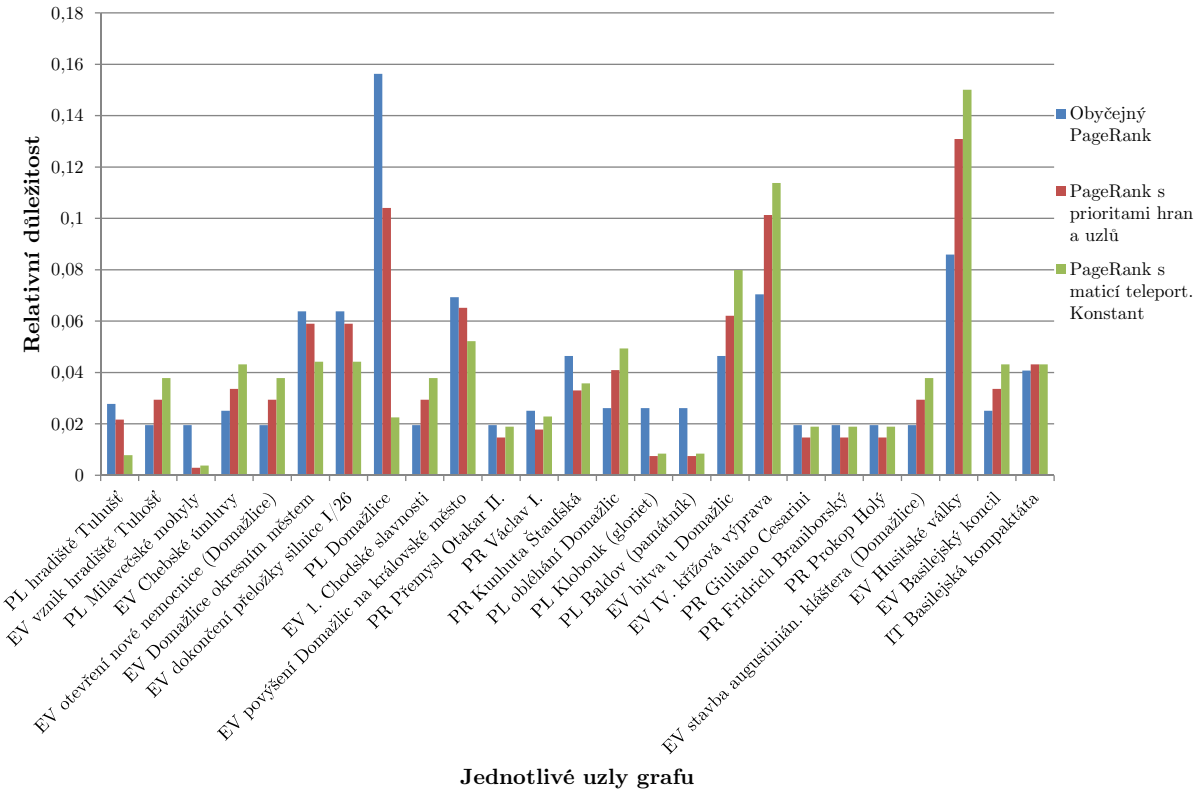
Soubor spravující závislosti této aplikace, nastavení sestavení, název a verzi. I tato aplikace je tedy vedena jako samostatný Maven projekt.

9 Výsledky a tesování

Tato kapitola se věnuje výkonnostním charakteristikám jednotlivých typů algoritmu PageRank, jejichž principy byly popsány v kapitole 2.2.3 a implementace následně v kapitole 6. Dále se tato kapitola zaměřuje na popis testování implementační části této práce.

9.1 Ovlivnění důležitostí uzlů uživatelskými prioritami

Graf na obrázku 9.1 zobrazuje relativní důležitost jednotlivých uzlů historického grafu Domažlic (viz příloha B) při použití různých typů algoritmu PageRank.



Obrázek 9.1: Graf důležitostí jednotlivých uzlů historického grafu Domažlic při použití různých typů algoritmu PageRank; každý uzel grafu udává pomocí zkratky svůj stereotyp: EV – `event`*, IT –* `item`*, PL –* `place`*, PR –* `person`

Při výpočtu obou prioritních PageRanků (s prioritami hran a uzlů, respektive s maticí teleportačních konstant) byly zvoleny uživatelské priority, které se soustředily na vysoké zvýraznění událostí při nejvyšším možném potlačení

důležitosti míst. Jednotlivým stereotypům tedy byly přiřazeny následující priority (1 pro nejnižší prioritu, 10 pro nejvyšší):

- stereotypy uzlů:
  - `event` = 10,
  - `place` = 1,
  - `item, person` = 5,
- stereotypy hran:
  - `participation, partOf` = 10,
  - `takesPlace` = 1,
  - `cause, creation, interaction, relationship` = 5.

Z grafu na obrázku 9.1 je patrné, že při aplikaci obyčejného PageRanku nad tímto grafem jsou nejvýše ohodnoceným uzlem samotné Domažlice, jelikož mají nejvíce vazeb s ostatními uzly. I v kontextu logického uvažování se zdá být tento uzel nejdůležitější, jelikož jde o samotné místo, ke kterému se vztahuje většina okolních událostí. To samé se dá říci o ostatních historických uzlech, které byly ohodnoceny vysokou důležitostí: Husitské války, IV. křížová výprava nebo povýšení Domažlic na královské město. PageRank tedy přiřadil jednotlivým historickým uzlům vcelku správné důležitosti.

Při použití prioritních typů PageRanku je důležitost Domažlic snížena. A to vzhledem k tomu, že je to uzel stereotypu `place`, do kterého navíc směřuje většina hran se stereotypem `takesPlace` a oba tyto stereotypy mají přiřazenou nejnižší možnou prioritu. Zároveň lze pozorovat zvýšení důležitostí všech událostí (což byl záměr), hlavně pak Husitských válek, IV. křížové výpravy a bitvy u Domažlic.

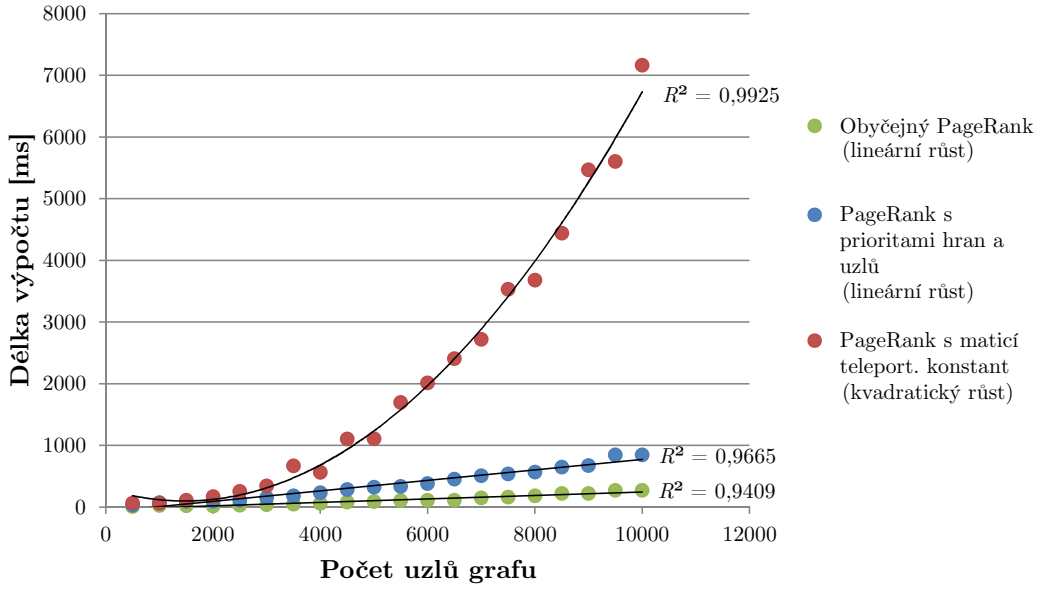
Mezi ohodnocením uzlů jednotlivými PageRanky využívajícími uživatelských priorit je také vidět rozdíl. Z grafu jasně plyne, že PageRank s maticí teleportačních konstant je více ovlivněn přiřazenými prioritami než PageRank, který využívá jen prioritně ohodnocených uzlů a hran. Důvody tohoto rozdílu již byly popsány v kapitole 2.2.3.

9.2 Výpočetní složitost jednotlivých typů PageRanku

Kapitoly 2.2.3 a 6 již zmiňovaly výpočetní složitost jedné iterace PageRanku, která závisí na použité implementaci. V případě této práce byla využita varianta výpočtu pomocí seznamu sousednosti jednotlivých uzlů, jež má výpočetní složitost $O(n)$. Tuto variantu bylo možné použít pro výpočet obyčejného PageRanku a PageRanku s prioritními uzly a hranami. Pro PageRank s maticí teleportačních konstant bylo potřeba k výpočtu jedné iterace použít násobení vektoru s maticí, kdy výpočetní složitost této operace je $O(n^2)$.

Graf na obrázku 9.2 zobrazuje časy výpočtů jednotlivých typů PageRanku při různě velkých grafech. Z grafu lze vidět, že časy potřebné k výpočtu jednotlivých typů PageRanku odpovídají výpočetním složitostem, které byly stanoveny v předchozích kapitolách. Tedy že čas potřebný k výpočtu obyčejného PageRanku, respektive PageRanku s ohodnocením uzlů a hran roste lineárně s velikostí grafu, zatímco čas PageRanku s maticí teleportačních konstant roste přibližně kvadraticky v závislosti na velikosti grafu.

Rozdíl mezi časy PageRanků s lineární výpočetní složitostí je dán především faktem, že u typu s prioritním ohodnocením uzlů a hran je potřeba před samotným spuštěním algoritmu předzpracovat celý graf podle zvolených priorit.



Obrázek 9.2: Graf udávající vztah mezi časem potřebným k výpočtu jednotlivých typů PageRanku v závislosti na velikosti grafu; jednotlivé časy jsou průměrem ze čtyř měření

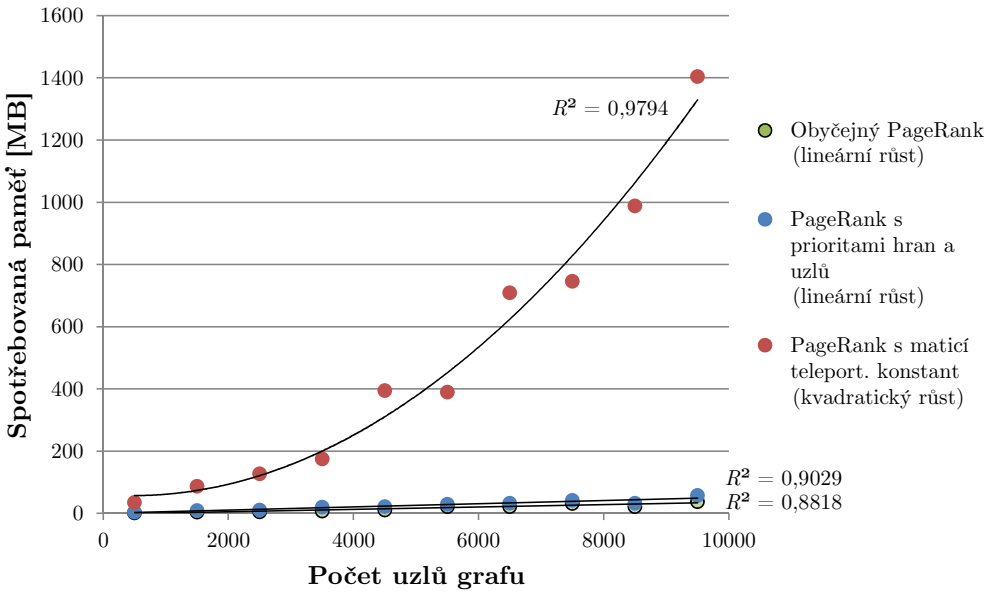
Měření času výpočtu probíhalo na grafu vytvořeném generátorem (implementaci balíku pro generování grafu popisuje příloha A).

9.3 Paměťová náročnost jednotlivých typů PageRanku

Analogicky jako výpočetní složitost byla měřena i paměťová náročnost jednotlivých typů PageRanku. K jejímu měření bylo využito nástroje JConsole, který

využívá instrumentace JVM[1] k poskytnutí informací o výkonu a spotřebovávání zdrojů běžících Java aplikací.

Graf na obrázku 9.3 zobrazuje paměť potřebnou k výpočtu důležitostí uzlů nad různě velkými grafy. Z grafu vyplývá, že zatímco u PageRanku s prioritními uzly a hranami, respektive obyčejného, roste paměťová náročnost opět lineárně (a s minimálními vzájemnými rozdíly). U PageRanku s maticí teleportačních konstant je růst kvadratický. To je samozřejmě dáno pamětí potřebnou pro uložení matice $\mathbf{G}$ o rozměrech $n \times n$, která musí být pro tento výpočet vytvořena.



Obrázek 9.3: Graf udávající vztah mezi pamětí potřebnou k výpočtu jednotlivých typů PageRanku v závislosti na velikosti grafu

Program, na kterém probíhalo měření využívané paměti, měl při spouštění nastavený JVM argument `-Xmx6G` (nastavení maximální velikosti heap paměti na 6GB). Zdrojový text programu obsahoval smyčku, uvnitř které se vždy vygeneroval graf, nad kterým byl spočítán daný typ PageRanku. Měření paměti při jednom průběhu smyčky pak probíhalo následovně:

1. Po vygenerování grafu byl proces uspán, aby mohla být provedena takzvaná *garbage kolekce* (smazání nevyužívané paměti) prostřednictvím nástroje

[1] **JVM** (Java Virtual Machine) – Sada počítačových programů a datových struktur, která využívá modul virtuálního stroje ke spuštění dalších počítačových programů a skriptů vytvořených v jazyce Java. Úkolem tohoto modulu je zpracovat pouze tzv. mezikód, který je v Javě označován jako Java bytecode.

JConsole a následné zaznamenání hladiny využívané paměti (JConsole zjišťuje hladinu využívané paměti jen jednou za několik sekund, naměřenou hodnotu pak zobrazuje prostřednictvím grafu).

2. Po probuzení byl vykonán algoritmus samotného PageRanku.

3. Následně byl proces znovu uspán, aby mohlo dojít k zaznamenání nárůstu paměti spotřebované algoritmem. Po probuzení procesu byla zavolána jedna z metod instance algoritmu. A to z toho důvodu, aby nedošlo k smazání samotné instance algoritmu z paměti při samovolném provedení garbage kolekce (na instanci algoritmu by totiž již nevedla žádná reference).

4. Po změření nejvyšší hladiny paměti byl proces opět uspán, aby mohlo dojít k vynucené garbage kolekci pomocí JConsole. To umožnilo rozpoznání paměti využité algoritmem jako vrcholu v grafu využívané paměti. Nakonec byla zvýšena proměnná udávající velikost generovaného grafu a celý postup se opakoval od bodu 1.

Po měření byl graf využívané paměti exportován do formátu CSV a s pomocí aplikace Microsoft Excel 2010 bylo provedeno odečtení hodnot před a po vykonání algoritmu při jednotlivých velikostech generovaného grafu.

9.4 Stanovení zastavovací podmínky ε

PageRank má stanovenou zastavovací podmínku jako jednotkovou normu (vzorec 2.6) rozdílu vektorů důležitostí dvou posledních iterací ($\mathbf{x_{k+1}}$ a $\mathbf{x_k}$), která musí být menší než hodnota ε, tedy:

$$||\mathbf{x_{k+1}} - \mathbf{x_k}|| < \varepsilon$$

Otázkou tedy zůstává, jakou hodnotu ε zvolit. Tabulka 2.2 níže zobrazuje různě velké hodnoty ε, které jsou aplikovány na výpočet PageRanku nad grafem o pěti stech uzlech.

V tabulce je vidět, že s postupným zmenšováním hodnoty ε se přibližně lineárně zvyšuje počet iterací. Dále lze vidět, že jednotková norma vektoru z rozdílu vektorů důležitosti postupně klesajícího ε klesá exponenciálně.

Tabulka také ukazuje, že mezi hodnotami $\varepsilon = 10^{-3}$ a $\varepsilon = 10^{-4}$ dochází ke zpřesnění výsledných důležitostí již jen v řádu desetitisícin. Z tohoto důvodu je v diplomové práci vždy nastavena zastavovací podmínka na hodnotu $\varepsilon = 10^{-3}$.

9.5 Testování implementovaných knihoven

Všechny implementované knihovny jsou opatřeny jednotkovými testy, jež v průběhu vývoje umožňovaly ověřovat správné chování nových funkcionalit a zároveň

i	ε_i	n	$\|\|\mathbf{x_{i-1}} - \mathbf{x_i}\|\|$
1	10^{-1}	3	–
2	10^{-2}	6	$3,98 \cdot 10^{-2}$
3	10^{-3}	8	$2,78 \cdot 10^{-3}$
4	10^{-4}	11	$5,52 \cdot 10^{-4}$
5	10^{-5}	14	$4.38 \cdot 10^{-5}$
6	10^{-6}	17	$3,44 \cdot 10^{-6}$
7	10^{-7}	19	$3.22 \cdot 10^{-7}$
8	10^{-8}	22	$6,46 \cdot 10^{-8}$
9	10^{-9}	25	$5,15 \cdot 10^{-9}$
10	10^{-10}	28	$4,46 \cdot 10^{-10}$

Tabulka 9.1: Tabulka zobrazující počet iterací (n) potřebných k dosažení specifikovaných zastavovacích podmínek ε_i. Dále zobrazuje postupné zpřesnění vektoru důležitosti, ke kterému dochází při zmenšení hodnoty ε_i o jeden řád.

prováděly regresní testování stávající funkcionality. K testování bylo využito knihovny JUnit [17].

Jednotlivé knihovny jsou pokryty testy, které ověřují správnou funkčnost hlavních poskytovaných služeb:

- Knihovna `graph` – Funkční vytváření grafu, vkládání a mazání vrcholů, respektive hran, nemožnost opětovného vložení stejného vrcholu či hrany do grafu atd.
- Knihovna `graph-ranking` – Ověření správného ověření výpočtu jednotlivých typů PageRanku na testovacím grafu, testování konvertoru důležitosti a zpracování uživatelských priorit.
- Knihovna `db-graph-ranking` – Testování spojení s grafovou databází, ověření funkčního vkládání uzlů a hran do databáze, získání seznamu vlastností atd.
- Knihovna `timelineVizServer` – Testy správného převádění jednotlivých Java objektů (graf, uživatelský dotaz, vrchol a hrana) do JSON formátu a obráceně. Pro testování této knihovny bylo využito i frameworku Mockito [18].

Všechny zmíněné testy jsou vždy automaticky prováděny při kompilaci jednotlivých knihoven (pomocí příkazu `mvn package`).

10 Závěr

Cílem diplomové práce bylo vytvořit nástroj pro ohodnocování důležitosti uzlů grafu historických událostí, který získává data z grafové databáze a ohodnocený graf poskytuje nástrojům umožňující jeho vizualizaci prostřednictvím časové osy.

Práce popisuje několik algoritmů pro důležitostní ohodnocení grafu. Blíže se pak věnuje analýze algoritmu PageRank a jeho modifikacím, jež umožňují ovlivnění výsledných důležitostí jednotlivých uzlů uživatelskými prioritami, kdy je vysvětlen princip tohoto algoritmu, jeho vlastnosti a odvozena výpočetní složitost.

Pro implementaci reprezentace grafu bylo analyzováno několik existujících knihoven. Ani jedna z nich bohužel přímo nevyhovovala požadavkům. Byla proto vytvořena vlastní grafová knihovna umožňující snadnou znovupoužitelnost pro graf reprezentovaný libovolnými entitami. Dále byly implementovány tři typy algoritmu PageRank v rámci knihovny poskytující algoritmy (dva typy s lineární výpočetní složitostí a jeden s kvadratickou). Tato knihovna umožňuje snadné rozšíření o další algoritmy.

Pro zajištění komunikace mezi vizualizačními nástroji a grafovou databází (které byly vytvořeny v rámci souběžných diplomových prací) byly implementovány knihovny poskytující Java, respektive REST rozhraní umožňující snadné získání ohodnoceného grafu, získávání jednotlivých vrcholů nebo vkládání uzlů a hran.

Poslední kapitola se pak věnuje měření potřebného času a paměti k výpočtu jednotlivých typů PageRanku v závislosti na velikosti grafu, kde byly experimentálně potvrzeny zmíněné výpočetní složitosti jednotlivých typů tohoto algoritmu. Dále se věnuje volbě hodnoty udávající zastavovací podmínku PageRanku a v neposlední řadě testování celé implementační části této práce.

Další vývoj implementovaných nástrojů by se mohl ubírat směrem k urychlení jednotlivých algoritmů pomocí paralelizace výpočtu, implementaci nových algoritmů nebo případně vytváření nových sad pravidel pro prioritní ohodnocení uzlů a hran.

11 Přehled použitých zkratek

CSV (Comma Separated Value) jednoduchý souborový formát určený pro výměnu tabulkových dat.

IoC (Inversion of Control) Návrhový vzor, který umožňuje uvolnit vztahy mezi jinak těsně svázanými komponentami.

JSON (JavaScript Object Notation) – Způsob zápisu dat (datový formát) nezávislý na počítačové platformě, určený pro přenos dat, která mohou být organizována v polích nebo agregována v objektech.

JVM (Java Virtual Machine) Sada počítačových programů a datových struktur, která využívá modul virtuálního stroje ke spuštění dalších počítačových programů a skriptů vytvořených v jazyce Java. Úkolem tohoto modulu je zpracovat pouze tzv. mezikód, který je v Javě označován jako Java bytecode.

REST (Representational state transfer) architektura rozhraní navržená pro distribuované prostředí. Typicky využívá protokolu HTTP/1.1 (dotazy `GET`, `POST`, `DELETE`, atd.) pro komunikaci a identifikátoru URI pro identifikaci jednotlivých zdrojů.

XML (eXtensible Markup Language) Obecný značkovací jazyk vyvinutý konsorciem W3C. Umožňuje vytváření značkovacích jazyků určených k nejrůznějším typům užití.

XSD (XML Schema Definition) Soubor, který popisuje strukturu XML dokumentu.

URI (Uniform Resource Identifier) Textový řetězec s definovanou strukturou, který slouží k přesné specifikaci zdroje informací (ve smyslu dokument nebo služba), hlavně za účelem jejich použití pomocí počítačové sítě, zejména Internetu.

Literatura

[1] RYJÁČEK, Z. *Teorie grafů, diskrétní optimalizace a výpočetní složitost 1.* Západočeská univerzita, 2007.

[2] BADER, D. A. — MADDURI, K. Parallel algorithms for evaluating centrality indices in real-world networks. In *Parallel Processing, 2006. ICPP 2006. International Conference on*, s. 539–550. IEEE, 2006.

[3] JUN, H. – WANG, B. – LEE, D. Evaluating Node Importance with Multi-criteria. In *Proc. of IEEE/ACM International Conference on Gree Computing and Communications & IEEE/ACM International Conference on Cyber*, Physical and Social Computing, pp.792-797, 2010.

[4] MIKULÁŠ, 0. *Pagerank algoritmus.* Univerzita Komenského v Bratislave, 2010.

[5] HANNEMAN, R. – RIDDLE, M. *Introduction to social network methods.* University of California Riverside, 2005.

[6] BRIN, P. – PAGE, L. – MOTWAMI, R. – WINOGRAD, T. *The PageRank citation ranking: Bringing order to the Web.* Stanford InfoLab, 1999.

[7] WHITE, S. – SMITH, P. Algorithms for estimating relative importance in networks *In Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining.* ACM, New York, NY, USA, 2003.

[8] HAVELIWALA T. Topic-sensitive PageRank *In Proceeding WWW '02 Proceedings of the 11th international conference on World Wide Web*, ACM, New York, NY, USA, 2002.

[9] *Grph, The high performance graph library for Java* [online]. poslední změna: březen 2015 [cit. 7.3.2015]. Dostupné z: http://www.i3s.unice.fr/ hogie/grph/

[10] *Java Universal Network/Graph Framework* [online]. poslední změna: leden 2010 [cit. 7.3.2015]. Dostupné z: http://jung.sourceforge.net/

[11] *GraphStream, A Dynamic Graph Library* [online].
©2010-2013, GraphStream Team [cit. 7.3.2015].
Dostupné z: http://graphstream-project.org

[12] NAVEH B. a přispěvatelé, *JGraphT* [online].
© 2003-2015, Barak Naveh [cit. 7.3.2015].
Dostupné z: http://jgrapht.org/

[13] *oj! Algorithms* [online].
©2000–2014,oj! Algorithms [cit. 4.4.2015]
Dostupné z: http://ojalgo.org/

[14] ABELES, P. *Java Matrix Benchmark* [online].
poslední změna: říjen 2013 [cit. 4.4.2015].
Dostupné z: https://code.google.com/p/java-matrix-benchmark/

[15] Kogent Solutions Inc. *Ajax Black Book, New Edition.*
Dreamtech Press, Nové Dillí, Indie, 2008.
ISBN10: 8177228382, ISBN13: 9788177228380.

[16] *ISO 8601:2004 – Date and time format*
©ISO, 3.12.2004.

[17] *JUnit* [online].
© 2002-2014 JUnit, poslední změna: 4. prosince 2014 [cit. 7.6.2015].
Dostupné z: http://junit.org/

[18] *Mockito* [online].
© 2007 Mockito contributors, poslední změna: 31. prosince 2014 [cit. 7.6.2015].
Dostupné z: http://mockito.org/

A Generátor grafu historických událostí

Pro testování jednotlivých typů PageRanků bylo potřeba vytvořit generátor grafu historických událostí. Graf vytvořený tímto generátorem musí splňovat následující požadavky:

- Procentuální zastoupení vrcholů s jednotlivými stereotypy nesmí být rovnoměrné, ale musí přibližně odpovídat reálným historickým grafům.
- Stupeň vrcholu musí přibližně odpovídat jeho stereotypu.
- Hrany mezi vrcholy s určitými stereotypy musí mít logický stereotyp (zamezení situaci, aby mezi hranami se stereotypy `item` a `item` vedla např. hrana se stereotypem `colaboration`).

Na základě těchto požadavků byla vytvořena knihovna `graph-generator` umožňující prostřednictvím třídy `HistoricalGraphGenerator` generovat graf historických událostí. Pokud se tato třída inicializuje pomocí konstruktoru bez parametrů, pak má generátor následující předdefinované vlastnosti:

- `event` – 40 %
- `item` – 10%
- `person` – 30 %
- `place` – 20 %

Jednotlivé stereotypy vrcholů mají také přednastavený stupeň, který je dán normálním rozložením s následujícími parametry:

- `event` – $\mu = 12$, $\sigma = 3$
- `item` – $\mu = 3$, $\sigma = 1$
- `person` – $\mu = 7$, $\sigma = 1$
- `place` – μ =velikost grafu/8, maximálně však 200, $\sigma = \mu/4$

Stereotypy hran jsou stanoveny na základě stereotypů vrcholů, mezi kterými vede. Jejich orientace je náhodně generována. Pro obě orientace může mít hrana různé stereotypy. Stereotyp hrany se generuje na základě nadefinovaných možností, které mají stanovené procentuální zastoupení (blíže viz `javadoc` třídy `VertexConnectionGenerator`).

Třída `HistoricalGraphGenerator` obsahuje také konstruktor, jehož prostřednictvím může uživatel nastavit všechny zmíněné parametry libovolně dle své

potřeby. Význam jednotlivých parametrů je podrobně popsán v javadoc dokumentaci této knihovny.

Samotné generování grafu se spouští zavoláním metody `generateGraph()`, která obsahuje parametr typu integer, jenž udává počet vrcholů generovaného grafu.

Zdrojový text níže zobrazuje vygenerování historického grafu o pěti stech uzlech:

```
HistoricalGraphGenerator graphGen = new HistoricalGraphGenerator();
Graph<Node, Bond> generatedGraph = graphGen.generateGraph(500);
```

B Historický graf Domažlic