Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Niko Colnerič

Gradnja in pregledovanje mreže znanstvenih člankov

diplomsko delo
na univerzitetnem študiju

Mentor: izr. prof. dr. Janez Demšar

Ljubljana, 2013

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Univerza *v Ljubljani*
Fakulteta *za računalništvo in informatiko*

Tržaška 25
1001 Ljubljana, Slovenija
telefon: 01 476 84 11
01 476 83 87
faks: 01 426 46 47
01 476 87 11
www.fri.uni-lj.si
e-mail: dekanat@fri.uni-lj.si

Št. naloge: 01910/2013
Datum: 02.04.2013

Univerza v Ljubljani, Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Kandidat: **NIKO COLNERIČ**

Naslov: **GRADNJA IN PREGLEDOVANJE MREŽE ZNANSTVENIH ČLANKOV**
CONSTRUCTION AND EXPLORATION OF NETWORKS OF SCIENTIFIC PAPERS

Vrsta naloge: Diplomsko delo univerzitetnega študija

Tematika naloge:

V znanstvenem delu pogosto porabimo veliko časa za pregledovanje obstoječe literature z določenega področja. Pri tem si pomagamo s spletnimi repozitoriji strokovnih in znanstvenih publikacij. Pregledovanje teh repozitorijev je "linearno", saj temelji na pregledovanju seznamov. Veliko boljše bi bilo članke pregledovati v obliki mreže, kjer bi vozlišča predstavljala članke, povezave med njimi podobnosti.

Cilj diplome je sprogramirati spletni pregledovalnik člankov iz določenega repozitorija (npr. Pubmed ali CiteULike). Repozitorij uporabimo za pridobivanje informacij o člankih z želenega področja, na primer glede na uporabnikovo poizvedbo, aplikacija pa pokaže mrežo, ki temelji na podobnosti člankov in omogoča uporabniku interaktivno "raziskovanje" te mreže.

Mentor:
izr. prof. dr.Janez Demšar

Dekan:
prof. dr. Nikolaj Zimic

Fakulteta za računalništvo in informatiko
2
UNIVERZA V LJUBLJANI

Izjava o avtorstvu diplomskega dela

Spodaj podpisani Niko Colnerič, z vpisno številko **63080007**, sem avtor diplomskega dela z naslovom:

Gradnja in pregledovanje mreže znanstvenih člankov

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom izr. prof. dr. Janeza Demšarja,
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela
- soglašam z javno objavo elektronske oblike diplomskega dela v zbirki "Dela FRI".

V Ljubljani, dne 30. avgusta 2013 Podpis avtorja:

Zahvaljujem se mentorju dr. Janezu Demšarju za nasvete, ideje ter napotke, ki jih je delil z menoj tekom nastajanja tega diplomskega dela.

Zahvaljujem se profesorju dr. Blažu Zupanu za izvirno idejo, ki je bila glavno vodilo pri izdelavi diplomskega dela.

Zahvaljujem se mami, očetu ter sestri za motiviranje in podporo tekom študija.

Zahvaljujem se profesorjem, asistentom in prijateljem, ki so z menoj delili izkušnje ter v meni vedno znova in znova znali vzbuditi zanimanje za razne izzive.

Mami Moniki, očetu Janezu ter sestri Blanki!

Kazalo

Seznam uporabljenih kratic in simbolov

ACM DL	digitalna knjižnica združenja ACM (*Association for Computing Machinery*)
AJAX	asinhroni JavaScript in XML (*asynchronous JavaScript and XML*)
API	aplikacijski programski vmesnik (*Application programming interface*)
CSS	kaskadne stilske podloge (*Cascading Style Sheets*)
D3	*Data-Driven Documents*
DOM	dokumentni objektni model (*Document Object Model*)
HTML	jezik za označevanje nadbesedila (*Hyper Text Markup Language*)
HTTP	komunikacijski protokol (*HyperText Transfer Protocol*)
JSON	objektna notacija jezika JavaScript (*JavaScript Object Notation*)
MeSH	*Medical Subject Headings*
MVC	model-pogled-krmilnik (*Model–view–controller*)
NLTK	knjižnica v Pythonu za obdelavo naravnega jezika (*Natural Language Toolkit*)
SVG	umerljiva vektorska grafika (*Scalable Vector Graphics*)
TF-IDF	*Term Frequency - Inverse Document Frequency*
URL	enolični krajevnik vira (*Uniform Resource Locator*)

W3C	konzorcij za svetovni splet (*World Wide Web Consortium*)
XML	razširljiv označevalni jezik (*Extensible Markup Language*)

Povzetek

V pričujočem delu predstavljamo inovativen pristop k pregledovanju znanstvenih člankov. Namesto v seznamu, sorodna dela nekega članka prikazujemo v mreži. Debelina povezave ponazarja vsebinsko podobnost dveh člankov. Podobnost dveh člankov definiramo s kosinusno podobnostjo njunih vektorjev. Povzetke v vektorje pretvorimo s TF-IDF transformacijo, predhodno pa jih tudi lematiziramo. Podatke o člankih pridobivamo s spletnih repozitorijev PubMed ter CiteULike. Dostop do podatkov smo pohitrili z lokalnim shranjevanjem podatkov. Predstavljamo tudi način odstranjevanja manj pomembnih povezav iz mreže ter postopek vizualizacije dobljene mreže sorodnih člankov s pomočjo knjižnice D3. Aplikacija je javno dostopna na naslovu `http://butler.fri.uni-lj.si/articles`.

Ključne besede:

podobnost besedil, *vizualizacija*, *mreže znanstvenih člankov*

Abstract

We present an innovative approach to reviewing related scientific articles. Rather than showing articles in a list, we show related articles in a network. Edge width represents semantic similarity of connected articles. We define similarity between two articles as cosine similarity between vectors, which represents this two articles. Vectors are obtained by TF-IDF transformation of article abstracts. Before transformation abstracts are first lemmatized. Information about articles are gathered from online repositories PubMed and CiteULike. We use caching to speed up the access. We also present a method of removing less important edges and techniques used to visualize this networks with D3 library. Our application is publicly accessible on website `http://butler.fri.uni-lj.si/articles`.

Keywords:

text similarity, *visualization*, *network of scientific articles*

Poglavje 1

Uvod

Kadar se kot inženir spopademo z reševanjem določenega problema ali pa se želimo zgolj bolje spoznati z nekim področjem, je eden izmed prvih korakov pregled sorodnega dela ter obstoječih raziskav tega področja. To početje je s pomočjo svetovnega spletna nedvomno enostavnejše, uporabniku bolj prijazno ter hitrejše kot nekoč. Obstaja namreč mnogo spletnih strani oziroma repozitorijev, katerih glavni namen je skladiščenje enormnih količin strokovnih in znanstvenih publikacij. Med njimi izpostavimo PubMed [20] in CiteULike [14], s katerima se bomo tudi podrobneje ukvarjali v tem diplomskem delu. Popularni so tudi ACM DL [13], Google Učenjak [16] ter mnogi drugi. Nekateri repozitoriji vsebujejo publikacije s širšega področja, spet drugi so fokusirani na izbrano ožje področje. PubMed večinoma vsebuje članke z področja medicine, ACM DL s področja računalništva in informatike, CiteULike ter Google Učenjak pa sta bolj splošna. Na nekaterih so rezultati iskanja prosto dostopni, drugi pokažejo samo del vsebine, za dostopanje do celotne vsebine pa je potrebno plačilo. Večina jih omogočajo iskanje vsebin po ključnih besedah. Rezultati iskanja so pri vseh repozitorijih predstavljeni linearno, dobimo namreč seznam publikacij, ki ustrezajo našemu iskalnemu nizu. Običajno pa nobeden od repozitorijev ne prikazuje nikakršne medsebojne informacije med najdenimi publikacijami. Sicer vemo, da vse publikacije zadoščajo iskalnemu nizu, o medsebojnih odnosih publikacij pa ne izvemo

ničesar. Dobro bi namreč bilo poznati medsebojno povezanost publikacij na podlagi njihove semantične podobnosti oziroma tematike, ki jo obravnavajo. Dobljene rezultate bi iskalnik tako namesto v seznamu raje prikazoval v mreži. Vsako vozlišče bi predstavljalo neko publikacijo, vsaka povezava med dvema publikacijama pa bi temeljila na njuni podobnosti. Pustimo pojem podobnosti zaenkrat še nedefiniran. Tak prikaz omogoča hitrejše iskanje publikacij s podobno vsebino, kar je koristno kadar želimo izvedeti čim več o nekem zelo ozkem področju. Hkrati pa nam omogoča, da lahko najdemo publikacije, ki so sicer z istega področja, vendar si niso preveč vsebinsko podobne, s čimer lažje dobimo pregled nad določenim področjem. Pregledovanje člankov bi tako bilo uporabniku prijaznejše ter atraktivnejše, saj bi mu poleg samih rezultatov iskanja poizkušali čim več povedati tudi o medsebojnih odnosih med publikacijami.

V diplomskem delu se osredotočimo zgolj na akademske članke, ostale publikacije (knjige, magazine, izročke s konferenc ...) zaenkrat izpustimo. Cilj diplomskega dela je izdelati spletno aplikacijo, ki bo predstavljala vmesnik pri dostopu do spletnih repozitorijev PubMed ter CiteULike. Definirali bomo nekakšno mero podobnosti, s katero bomo na podlagi povzetkov določili podobnost dveh člankov. Poleg tega se bo potrebno spopasti z odstranjevanjem povezav med članki, ki imajo premajhno podobnost. Zadnji izziv pa bo dobljene rezultate na čim bolj uporabniku prijazen in interaktiven način prikazati v mreži.

V naslednjih poglavjih bomo najprej na kratko opisali pomembnejše algoritme, kasneje pa se bomo posvetili uporabljenim tehnologijam ter sami implementaciji. Le to delimo na tri ključne korake. Prvi je pridobivanje podatkov iz tujih repozitorijev, naslednji analiza podatkov, določanje podobnosti, grajenje mreže ter odstranjevanje manj pomembnih povezav, zadnji pa vizualizacija pridobljenih ugotovitev. Na koncu sledi še prikaz primera uporabe spletne aplikacije ter sklepne ugotovitve.

Poglavje 2

Algoritmi

2.1 Jaccardov indeks

Ena preprostejših mer za določanje podobnosti dveh množic je Jaccardov indeks [8].

$$Jaccard \quad index = \frac{|A \cap B|}{|A \cup B|} \tag{2.1}$$

V enačbi 2.1 opazimo, da je definiran kot razmerje moči dveh množic. V števcu je presek množic A in B, v imenovalcu pa njuna unija. Intuitivno Jaccardov indeks šteje, koliko je takšnih elementov, ki so hkrati v množici A in množici B. Več kot jih je, večja je vrednost Jaccardovega indeksa. Vrednosti, ki jih zavzema, so na intervalu $[0, 1]$. Vrednost 1 pomeni, da sta množici enaki, vrednost 0 pa, da ne vsebujeta niti enega enakega elementa.

Jaccardov indeks je prva mera, ki smo jo preizkusili za računanje podobnosti med članki. Denimo, da imamo dva članka, med katerima želimo izračunati podobnost na podlagi njunih povzetkov (*abstract*). Povzetka sta podana kot besedilo, v definiciji 2.1 pa uporabljamo dve množici, ne dveh besedil. Zato je potrebno povzetek najprej preslikati v množico. To naredimo preprosto tako, da besedilo razrežemo na vseh mestih, kjer se pojavi presledek. Iz dobljenih žetonov nato odstranimo duplikate in tako dobimo množico besed, ki predstavlja dani članek.

Ena očitnejših pomanjkljivosti Jaccardovega indeksa pri njegovi uporabi nad besedili je ta, da je definiran za množice. Tako zanj ni pomembno, kolikokrat se neka beseda pojavi v danem besedilu, temveč upošteva le, ali se pojavi ali ne. To pa za naravo našega problema ni najboljša rešitev, saj podatek, kolikokrat se neka beseda pojavi v povzetku, lahko nosi ogromno informacije.

2.2 Model vektorskega prostora

Predstavitev povzetkov z množico besed opisana v poglavju 2.1 ni najbolj primerna. Zato iščemo predstavitev povzetkov ter nadalje mero podobnosti, ki bi v obzir jemala tudi, kolikokrat se dana beseda v povzetku pojavi. Ena takšnih predstavitev je model vektorskega prostora. Ta vsak povzetek preslika v visoko dimenzionalen vektor. Komponente vektorja so bodisi besede bodisi fraze. Če za komponente izberemo besede, potem vsaka beseda, ki se pojavi v kakšnem od dokumentov, predstavlja svojo neodvisno dimenzijo v skupnem vektorskem prostoru [11]. Če dokument vsebuje neko besedo, potem vektor za ta dokument v dimenziji te besede vsebuje neničelno vrednost. Ker vsak dokument vsebuje končno mnogo besed, dokumentov pa je lahko zelo veliko (visoko dimenzionalen vektorski prostor), vektorji večinoma vsebujejo veliko število komponent z vrednostjo nič (*sparse vector*). Nad takšnimi vektorji lahko podobnost računamo kar kot kot med dvema vektorjema.

Najprej si poglejmo predstavitev TF-IDF (*Term Frequency - Inverse Document Frequency*), ki jo bomo uporabili za predstavitev povzetkov. Nato bomo opisali še kosinusno podobnost, to je mera, s katero računamo podobnost med vektorji dobljenimi s transformacijo TF-IDF.

2.2.1 TF-IDF

Najprej opišimo nekaj osnovnih pojmov, ki jih bomo uporabljali pri definicijah. t je oznaka za nek izraz oziroma besedo, d je oznaka za dokument (povzetek),

D je oznaka za množico vseh dokumentov (povzetkov), ki jih želimo predstaviti v vektorskem prostoru, N pa predstavlja število dokumentov v D.

TF-IDF je mera, ki je za nek izraz t v dokumentu d, definirana kot produkt dveh vrednosti.

$$TF\text{-}IDF_{t,d} = TF_{t,d} * IDF_t \tag{2.2}$$

$TF_{t,d}$ predstavlja pogostost izraza t v dokumentu d (*Term Frequency*). Definirana je preprosto kot število pojavitev izraza t v dokumentu d. Sicer bi lahko uporabili tudi kakšno drugo obtežitveno funkcijo, ki število pojavitev izraza t v dokumentu d preslika v pozitivno realno število [8], vendar se bomo zadovoljili kar s številom pojavitev.

Nadalje pojasnimo frekvenco pojavitve izraza v dokumentih (*document frequency*) DF_t. Ta je definirana kot število dokumentov, ki vsebujejo izraz t (število pojavitev znotraj dokumenta nas ne zanima). IDF_t predstavlja inverzno frekvenco pojavitve izraza v dokumentih (*inverse document frequency*) ter je zgolj izpeljanka mere DF_t.

$$IDF_t = \log \frac{N}{DF_t} \tag{2.3}$$

Z utežjo IDF_t tako vsaki besedi določimo pomembnost v množici dokumentov D. Za izraze, ki se pojavljajo v skoraj vseh dokumentih dobimo nizko vrednost IDF. Za tiste, ki se pojavljajo v zelo majhnem številu dokumentov pa dobimo visoko vrednost IDF. Naprimer, kadar iščemo članke o pljučnem raku, bo verjetno večina dokumentov vsebovala besedo rak, zato ji dodelimo nizek IDF.

Za vsak izraz t, ki se pojavi znotraj množice dokumentov D, lahko glede na nek dokument d sedaj izračunamo utež $TF\text{-}IDF_{t,d}$ tako, da zanjo veljajo naslednje trditve:

- Je najvišja, če se t velikokrat pojavi v majhnem številu dokumentov.
- Je nižja, če se t v dokumentu pojavi manjkrat oziroma če se pojavi v več dokumentih.

- Je najnižja, če se t pojavi v praktično vseh dokumentih.

Preslikovanje množice dokumentov v vektorski prostor poteka po sledečem postopku. Najprej poiščemo množico vseh besed, ki se pojavijo v množici dokumentov D. Vsak element te množice predstavlja eno dimenzijo v vektorskem prostoru, v katerem bomo predstavili dokumente. Nekemu dokumentu d nato določimo vrednost vsake komponente vektorja kot utež $TF\text{-}IDF_{t,d}$. Predstavitveni vektorji tipično vsebujejo veliko število komponent z vrednostjo nič.

Argumentirajmo še zakaj je bolje uporabljati mero DF, kot mero CF, ki bi preprosto preštela kolikokrat se izraz pojavi v množici dokumentov D. Ideja glavnega argumenta je, da ne smemo dopustiti, da bi nek dokument z ogromno pojavitvami nekega izraza preveč vplival na utež IDF. Če bi namesto DF uporabljali CF, bi bilo popolnoma identično, če se nek izraz pojavi desetkrat v enem dokumentu ali pa če se pojavi enkrat v desetih dokumentih. Intuitivno pa želimo, da ima izraz v prvem primeru visok IDF, v drugem pa nižjega. To je glavni razlog za uporabo DF napram CF.

Omenimo še slabost predstavitve TF-IDF. Ta dokument preslika v vektor, katerega vrednosti neke komponente so odvisne zgolj od dveh stvari. Od števila pojavitev tega izraza v dokumentu ter od relativne pomembnosti izraza v množici vseh dokumentov. V taki predstavitvi sta angleški povedi *John is quicker than Mary* ter *Mary is quicker than John* predstavljeni enako, kljub njuni očitni razliki v pomenih.

2.2.2 Kosinusna podobnost

V poglavju 2.2.1 smo opisali kako množico dokumentov predstavimo v skupnem vektorskem prostoru. S takšno predstavitvijo lahko računanje podobnosti dveh besedil prevedemo na računanje kota med dvema vektorjema. Temu pravimo kosinusna podobnost. Njeno enačbo izpeljemo iz skalarnega produkta dveh vektorjev.

$$kosinusna\ podobnost = cos\,(\theta) = \frac{\vec{A} \cdot \vec{B}}{||A|| \cdot ||B||} \tag{2.4}$$

$$= \frac{\sum_{i=1}^{n} A_i \times B_i}{\sqrt{\sum_{i=1}^{n} (A_i)^2} \times \sqrt{\sum_{i=1}^{n} (B_i)^2}} \tag{2.5}$$

Kosinusna podobnost vrača vrednosti na intervalu $[0, 1]$, saj so vrednosti komponent v prostoru TF-IDF omejene na pozitivna števila. Tako je kot θ med dvema vektorjema omejen z intervalom $[0°, 90°]$, posledično pa kosinus kota lahko zavzema vrednosti $[0, 1]$. Vrednost 0 pomeni, da sta vektorja pravokotna, oziroma drugače, da si dokumenta sploh nista podobna. Vrednost 1 pomeni, da je kot med vektorjema 0°, oziroma da sta množica besed ter število pojavitev vsake besede v dokumentih natančno enaki. Splošno velja, večja kot je vrednost, bolj sta si dokumenta podobna.

2.3 Izris mreže

Za določanje pozicij vozlišč oziroma postavitve mreže poznamo več algoritmov. Uporabili bomo inačico *Force-directed* algoritma za risanje grafa, ki pozicije določa z Verletjevo integracijo (*Verlet integration*) ter hkrati upošteva omejitve. V družino *Force-directed* algoritmov uvrščamo več algoritmov, katerim skupno je to, da jih uporabljamo za izrisovanje mrež. So eni izmed najbolj učinkovitih metod za računanje postavitve preprostih neusmerjenih grafov [7]. Ti algoritmi ne uporabljajo nobenega predznanja o domeni (*domain-specific knowledge*) temveč za izračunanje postavitve vozlišč uporabljajo samo strukturo podanega grafa. Tako narisani grafi so estetski, razkrivajo simetrije in so za ravninske grafe praviloma brez povezav, ki bi se križale [7]. Tipično algoritmi delujejo tako, da točke grafa prvotno postavijo na naključne pozicije ter nato iterativno izboljšujejo izgled grafa. Iz starih lahko nove pozicije računamo na več načinov. Algoritem Fruchterman-Reingold tako definira odbojne sile med vsakim parom vozlišč v grafu ter hkrati privlačne sile med

parom vozlišč, ki so povezani. Naš algoritem deluje nekoliko drugače. Graf si predstavljamo kot sistem delcev. Pri sistemu delcev ima vsak delec v trenutku t dve glavni spremenljivki: pozicijo $\vec{x}$ ter hitrost $\vec{v}$ [6]. Iz teh dveh spremenljivk lahko novo pozicijo izračunamo na dva načina, opisana v spodnjih poglavjih.

2.3.1 Eulerjeva integracija

Eulerjeva integracija (*Euler integration*) novo pozicijo $\vec{x}'$ ter hitrost $\vec{v}'$ v času $t + \Delta t$ izračuna po sledečih formulah:

$$\begin{aligned} \vec{x}' &= \vec{x} + \vec{v} \cdot \Delta t \\ \vec{v}' &= \vec{v} + \vec{a} \cdot \Delta t \end{aligned} \tag{2.6}$$

V enačbah 2.6 Δt predstavlja časovni korak, pospešek $\vec{a}$ pa se izračuna po Newtonovem zakonu $\vec{f} = m \cdot \vec{a}$ [6]. Sila $\vec{f}$ je seštevek vseh sil, ki na delec delujejo. Tipično so to odbojne sile ostalih vozlišč.

2.3.2 Verletjeva integracija

Verletjeva integracija (*Verlet integration*) je nekoliko drugačen pristop k računanju novih pozicij, ki direktno ne uporablja hitrosti. Je numerična metoda, ki se uporablja za integracijo Newtonove enačbe gibanja. Enačbo Verletjeve integracije lahko izpeljemo, če enačbo pozicije x razvijemo z Taylorjevo vrsto.

$$f(x) = f(a) + \frac{f'(a)}{1!}(x-a) + \frac{f''(a)}{2!}(x-a)^2 + \frac{f'''(a)}{3!}(x-a)^3 + \ldots \tag{2.7}$$

Enačba 2.7 je enačba za razvoj funkcije v Taylorjevo vrsto. Vrednosti $f'(a)$, $f''(a)$ ter $f'''(a)$ predstavljajo vrednost prvega, drugega ter tretjega odvoda funkcije f v točki a. Enačbo 2.7 bomo uporabili za razvoj funkcije pozicije v času $t + \Delta t$ ter v času $t - \Delta t$. Pozicijo v času $t + \Delta t$ označimo z $\vec{x}'$, v času $t - \Delta t$ pa z $\vec{x}''$. Za razvoj v času $t + \Delta t$ v enačbo 2.7 namesto x

vstavimo $t + \Delta t$, namesto a pa vstavimo t. Naj še omenim, da je prvi odvod pozicije $\vec{x}$ hitrost $\vec{v}$, drugi odvod je pospešek $\vec{a}$, tretji odvod pa označimo z $\vec{b}$. Ker integriramo Newtonovo enačbo, je pospešek $\vec{a}$ dejansko količnik med silo $\vec{f}$, ki deluje na delec, ter njegovo maso m. Silo dobimo kot seštevek sil drugih delcev, ki na delec delujejo, zato lahko na silo gledamo tudi kot na funkcijo pozicije [4]. Po vstavljanju vrednosti dobimo enačbo 2.8. Na enak način dobimo tudi enačbo 2.9, le da namesto x vstavimo $t - \Delta t$.

$$\vec{x}' = \vec{x} + \vec{v} \cdot \Delta t + \frac{\vec{a}\Delta t^2}{2} + \frac{\vec{b}\Delta t^3}{6} + O\left(\Delta t^4\right) \tag{2.8}$$

$$\vec{x}'' = \vec{x} - \vec{v} \cdot \Delta t + \frac{\vec{a}\Delta t^2}{2} - \frac{\vec{b}\Delta t^3}{6} + O\left(\Delta t^4\right) \tag{2.9}$$

Enačbi 2.8 ter 2.9 seštejmo ter dobimo:

$$\vec{x}' = 2 \cdot \vec{x} - \vec{x}'' + \vec{a}\Delta t^2 + O\left(\Delta t^4\right) \tag{2.10}$$

Napaka algoritma je v odvisnosti $O\left(\Delta t^4\right)$ od koraka časa Δt [4]. Če upoštevamo zgolj razvoj po Taylorjevi vrsti do tretje stopnje, torej zanemarimo člene od četrte stopnje naprej, dobimo končni enačbi, ki ju uporabljamo v iterativnem koraku algoritma:

$$\vec{x}' = 2 \cdot \vec{x} - \vec{x}'' + \vec{a} \cdot \Delta t^2 \tag{2.11}$$

$$\vec{x}'' = \vec{x} \tag{2.12}$$

Tako si namesto hitrosti hranimo trenutno pozicijo $\vec{x}$ ter prejšnjo pozicijo $\vec{x}''$. Razlika $\vec{x}'' - \vec{x}$ pa je dejansko približek hitrosti. Takšna integracija je zelo stabilna, saj se hitrost pojavlja zgolj implicitno in se zato težje zgodi, da hitrost ne bi bila usklajena s pozicijo [6].

Poglavje 3

Implementacija

3.1 Uporabljene tehnologije

V tem odseku bomo na kratko opisali tehnologije, ki smo jih uporabljali pri implementaciji.

3.1.1 Python

Python je visokonivojski skriptni programski jezik, ki ga je leta 1991 ustvaril Guido van Rossum. Eden glavnih ciljev njegove zasnove je berljivost kode, sintaksa pa programerjem dopušča, da želeno dosežejo z relativno majhnim številom vrstic programske kode. Python tako, napram nekaterim ostalim programskim jezikom, odpravlja nepotrebne oklepaje, saj gnezdi s pomočjo zamikov. Objekti so tipizirani, imena spremenljivk pa ne. Tako je lahko spremenljivka a v nekem trenutku objekt, malo kasneje pa niz, seznam ali število v plavajoči vejici. Podpira tudi paradigmo objektno usmerjenega, imperativnega in funkcijskega programiranja [21]. Trenutno sta v uporabi dve inačici Pythona, verzija 2 ter novejša verzija 3, ki z prejšnjo ni združljiva. Za diplomsko delo smo izbrali verzijo 2.7, predvsem zaradi večjega števila dostopnih knjižnic. Tudi verzija Djanga, ki jo uporabljamo, je s podpiranjem Pythona verzije 3 šele v začetni fazi.

Django

Django je visokonivojsko odprtokodno okolje (*framework*) za razvijanje spletnih aplikacij v jeziku Python, ki je tudi samo razvito v Pythonu. Za njegov razvoj skrbi fundacija *Django Software Foundation* [15].

Django sledi konceptu MVC (model-pogled-krmilnik), ki programsko kodo aplikacije razdeli na tri čim bolj medsebojno neodvisne module. Programska koda, ki definira in dostopa do podatkovnih modelov (*model*), je tako ločena od kode, ki skrbi za usmerjanje zahtevkov (*krmilnik*), ki je nadalje ločena od kode, ki skrbi za uporabniški vmesnik (*pogled*). Prednosti takšne zasnove je veliko. Ena poglavitnejših je, da lahko spreminjamo del aplikacije, ne da bi s tem vplivali na delovanje celote [5]. Takšni zasnovi pravimo tudi *loose coupling*. Tako lahko popravimo URL, ne da bi se morali dotikali same implementacije. Dizajner lahko spreminja izgled strani, ne da bi rabil poznati Python. Sistemski administrator lahko spreminja imena tabel podatkovne baze v eni datoteki brez strahu, da bi s tem povzročil težave v delovanju ostalih delov aplikacije.

V diplomskem delu uporabljamo verzijo Djanga 1.5.1.

NLTK

NLTK je knjižnica v programskem jeziku Python, ki se uporablja za obdelavo naravnega jezika. Knjižnica trenutno prehaja na Python verzijo 3, vendar je šele v začetni fazi (*alpha*). Uporabljamo jo lahko brezplačno ter se razvija s strani skupnosti (*community-driven project*).

Ponuja preko 50 tekstovnih korpusov, orodja za označevanje, klasifikacijo, iskanje korenov besed, lematizacijo, prikazovanje razčlenitvenega drevesa povedi ter mnogo drugega. Omogoča nam naprimer izločevanje manj pomenskih besed (*stop words*) oziroma besed brez semantične vrednosti. To so na primer angleške besede I, you, a, the, all, some, we, this, to, only ...

Lxml

Lxml [19] je še ena izmed Pythonovskih knjižnic, ki smo jo uporabili tekom izdelave diplomskega dela. Uporablja se za procesiranje dokumentov formata HTML ter XML. Temelji na Cjevskih knjižnicah *libxml2* in *libxslt* ter odlično združuje hitrost procesiranja ter enostavnost uporabe. Prvotno smo za procesiranje sicer uporabljali knjižnico Beautiful Soup, vendar se je ta izkazala za prepočasno. Po menjavi knjižnice z lxml smo pri enaki velikosti dokumentov opazili drastično pohitritev.

Scikit learn

Scikit learn [9] je Pythonovska knjižnica za potrebe strojnega učenja. Interno uporablja NumPy, SciPy ter Matplotlib. Vsebuje algoritme za klasifikacijo, regresijo, gručenje (*clustering*), metoda za zniževanje dimenzionalnosti podatkov, predprocesiranje ... Uporabljali smo jo, saj vsebuje implementacijo transformacije TF-IDF (glej poglavje 2.2.1) ter kosinusno podobnost (glej poglavje 2.2.2).

3.1.2 JavaScript

JavaScript je objektni skriptni programski jezik, ki se uporablja za izdelavo interaktivnih spletnih strani. Uporabljajo ga tudi razvijalci iger ter namiznih aplikacij. Razvit je bil leta 1995 pri podjetju Netscape. Primarno je implementiran kot del spletnega brskalnika ter se izvaja na strani odjemalca (*client-side*). Omogoča interakcijo z uporabnikom ter nadziranje brskalnika po tem, ko je vsebina dokumenta HTML že naložena. Podpira več programskih paradigem, od objektno usmerjenega programiranja pa vse do imperativnega in funkcijskega. Njegova sintaksa je bila sicer razvita pod vplivom programskega jezika C, vendar je v več pogledih zelo podoben Javi [17]. Danes predstavlja enega vodilih jezikov na tem področju, o čemer poleg široke uporabe priča tudi prisotnost ogromnega števila prosto dostopnih razširitvenih knjižnic.

jQuery

jQuery je hitra, majhna ter funkcionalnosti polna knjižnica v jeziku JavaScript. Je med najbolj popularnimi knjižnicami za omenjeni jezik, saj jo uporablja kar 65% izmed 10,000 najbolj obiskanih spletnih strani [18]. Obstaja od leta 2006. Omogoča enostavno sprehajanje po dokumentu, izbiranje elementov DOM, ustvarjanje animacij ter nadzor nad dogodki.

jQuery mousewheel

Mousewheel [1] je dodatek za knjižnico jQuery, ki v spletnih brskalnikih doda podporo dogodkov, ki jih sproža kolešček na miški. Omogoča, da spremenimo privzeto obnašanje aplikacije ob vrtenju koleščka. Uporabili smo ga za horizontalno premikanje seznama člankov. Avtor dodatka Brandon Aaron, ga je izdal pod licenco MIT.

D3

D3 (*Data-Driven Documents*) [2] je kvalitetno dokumentirana knjižnica za JavaScript, namenjena manipulaciji dokumentov na podlagi podatkov. Njen glavni namen je atraktiven prikaz podatkov s pomočjo tehnologij HTML, SVG ter CSS. Omogoča povezovanje podatkov poljubne oblike z objekti tipa DOM, ki jih lahko nato preoblikujemo na podlagi prej podanih podatkov. Za izbiranje objektov tipa DOM knjižnica uporablja deklarativen pristop. Tako lahko spreminjamo kar množice objektov hkrati. Za izbiranje objektov uporablja selekcije (*selections*), ki so definirane s strani konzorcija W3C in so tako podprte v vseh sodobnejših brskalnikih. Na selekciji lahko uporabimo ogromno različnih efektov. Od nastavljanja stilov, dodajanja poslušalcev dogodkov, dodajanja oziroma odstranjevanja objektov v DOM, spreminjanja vsebine objektov, animacije ... Vsebuje razne algoritme za določanje postavitev, med drugim tudi algoritem, ki temelji na fizikalni simulaciji. Ta je bil uporabljen za vizualizacijo grafa člankov.

3.2 Viri podatkov

Prvi korak izdelave aplikacije je dostopanje do tujih repozitorijev. Aplikacija lahko dostopa do podatkov o člankih s spletnega repozitorija PubMed [20] in CiteULike [14]. Za lažjo nadaljnjo obravnavo se informacije posameznega članka, ne glede na repozitorij s katerega izvirajo, shranijo v objekt tipa *Article*. Ta ima naslednje atribute:

- **article_id** - enolična identifikacijska številka članka na spletni strani PubMed oziroma CiteULike
- **title** - naslov članka
- **abstract** - povzetek članka
- **url** - URL do članka na tujem repozitoriju
- **authors** - seznam avtorjev članka
- **keywords** - seznam ključnih besed
- **journal** - revija, v kateri je bil članek objavljen
- **site** - spletna stran, s katere smo pridobili podatke o članku
- **fetched** - datum in ura, kdaj smo podatke o članku prenesli s spletnega repozitorija

Do podatkov o člankih dostopamo na dva načina. Prvi način je iskanje člankov po iskalnem nizu, kjer rezultate prikazujemo v seznamu. Ker je število rezultatov preveliko za prikaz na eni strani, jih prikazujemo po stranem. Vsaka stran vsebuje maksimalno petindvajset zadetkov. Drugi je iskanje nekemu članku sorodnih člankov, kjer rezultate prikazujemo v mreži. Tukaj se zaradi preglednosti mreže omejimo na prvih sto člankov. Rezultat vsake faze je seznam objektov tipa *Article*, ki so pripravljeni za nadaljnjo analizo. V naslednjih dveh podpoglavjih bomo podrobneje opisali značilnost dostopa do repozitorijev PubMed ter CiteULike ter način za pohitritev dostopa z lokalnim shranjevanjem podatkov (*caching*).

3.2.1 PubMed

Do spletnega repozitorija PubMed dostopamo preko njihovega vmesnika API z imenom *E-utilities* [10], ki omogoča iskanje na sledeč način. Na strežnik pošljemo zahtevek HTTP z ustrezno oblikovanim URLjem, nazaj pa dobimo dokument XML z rezultati iskanja. Pri zahtevkih HTTP uporabljamo metodo GET. Za iskanje podatkov znotraj dokumenta XML pa uporabljamo knjižnico lxml (glej poglavje 3.1.1). Pri obeh načinih dostopa poteka iskanje podatkov o člankih v dveh korakih. V prvem koraku bodisi poiščemo identifikacijske številke člankov, ki ustrezajo nekemu iskalnemu nizu, bodisi poiščemo identifikacijske številke člankov, ki so danemu članku sorodni. Rezultatov tega koraka lokalno ne shranjujemo, saj so prihranki zanemarljivi. V drugem koraku nato pridobimo vse podatke o člankih z danimi identifikacijskimi številkami. Rezultate tega koraka shranjujemo tudi lokalno.

Iskanje identifikacijskih številk člankov po iskalnem nizu

Za pridobitev identifikacijskih številk člankov, ki ustrezajo iskalnem nizu, mora biti URL sledeče oblike[1]:

`http://eutils.ncbi.nlm.nih.gov/entrez/eutils/esearch.fcgi?db=pubmed&RetMax=<RM>&RetStart=<RS>&term=<T>`,

kjer RM predstavlja število koliko člankov želimo dobiti kot rezultat iskanja. RS označuje zaporedno številko od katere najprej želimo dobiti članke. Argumenta omogočata prenašanje podatkov po straneh. Z vrednostma $RM = 25$ ter $RS = 0$ dobimo podatke o prvih petindvajsetih člankih, z vrednostma $RM = 25$ ter $RS = 25$ pa dobimo podatke o naslednjih petindvajsetih člankih. Zadnji argument T predstavlja iskalni niz. V datoteki XML, ki jo strežnik vrne, so v elementu *IdList* vozlišča z imenom *Id*. Ta vsebujejo iskane identifikacijske številke.

[1]Pri tem ter vseh naslednjih URLjih velja, da je vrednosti obdane z znakoma večji in manjši, recimo `<T>`, potrebno zamenjati s podatki.

Iskanje identifikacijskih številk sorodnih člankov

Kadar iščemo identifikacijske številke nekemu članku sorodnih člankov, posredujemo zahtevek HTTP z URLjem naslednje oblike:

`http://eutils.ncbi.nlm.nih.gov/entrez/eutils/elink.fcgi?dbfrom=pubmed&db=pubmed&id=<ID>`,

kjer kot ID podamo identifikacijsko številko članka, kateremu iščemo sorodne članke. V dobljeni datoteki XML nato poiščemo vsa vozlišča z imenom *Id*, v katerih so shranjene iskane identifikacijske številke. Pri tem moramo paziti na podvajanje identifikacijskih številk. Podatki so namreč strukturirani v več kategorij, zato se lahko zgodi, da se nekatere identifikacijske številke pojavijo v več kategorijah hkrati.

Pridobivanje podatkov o člankih z danimi identifikacijskimi številkami

Ko imamo identifikacijske številke, je potrebno le še pridobiti vse podatke o teh člankih. To storimo s klicem lokalne metode *fetch_cache_or_web_pubmed(ids)*, ki ji podamo seznam identifikacijskih številk. Metoda deluje kot nekakšen vmesnik pri dostopanju do podatkov o člankih. Za podan seznam identifikacijskih številk preveri, podatke o katerih člankih imamo že shranjene lokalno v podatkovni bazi. Te identifikacijske številke odstranimo iz seznama *ids* ter tako pridemo do seznama identifikacijskih številk *missing_ids*. To je seznam identifikacijskih številk člankov, za katere podatkov nimamo shranjenih lokalno. Če seznam ni prazen, to nakazuje, da moramo dostopati do strani PubMed. V tem primeru pošljemo zahtevek HTTP, sedaj z URLjem oblike:

`http://eutils.ncbi.nlm.nih.gov/entrez/eutils/efetch.fcgi?db=pubmed&rettype=abstract&id=<MIDS>`,

kjer namesto *MIDS* podamo z vejicami ločen seznam identifikacijskih številk shranjenih v *missing_ids*. V datoteki XML, ki jo vrne strežnik, se najprej sprehodimo čez vsa vozlišča z imenom *PubmedArticle*. Vsako tako vozlišče namreč vsebuje podatke o enem članku. Za vsak članek nato poiščemo vse

atribute naštete v poglavju 3.2 ter ustvarimo objekt tipa *Article*. Objektu še pripnemo trenutni datum in čas ter ga shranimo v podatkovno bazo. V podatkovni bazi so sedaj shranjeni vsi potrebni podatki, ki jih v pravilnem vrstnem redu le še preberemo ter vrnemo.

Naj še omenim zakaj smo se odločili za takšno implementacijo. Dostopanje do podatkov tujih repozitorijev je počasno, hkrati pa ni smiselno, da naprimer ob pritisku gumba za osvežitev strani, vse podatke o člankih vsakič znova prenesemo s spletne strani PubMed. Naša metoda *fetch_cache_or_web_pubmed(ids)* si tako, kadar prvič dostopa do podatkov nekega članka, le te shrani v podatkovno bazo. Ko do podatkov istega članka dostopamo drugič, dostopanje do PubMeda ni potrebno, saj imamo vse potrebne podatke že shranjene lokalno.

Razmišljali smo tudi o shranjevanju identifikacijskih številk v zvezi z iskalnim nizom oziroma nekemu članku sorodnih člankov. Idejo smo opustili iz dveh razlogov. Prvič, zaradi pojavljanja novih člankov, bi naši podatki zastarali. Drug razlog pa je velikost datoteke za prenos. Datoteka s stotimi identifikacijskimi številkami je reda velikost do 10kB, medtem ko je datoteka s podatki o sto člankih reda velikosti 1MB. Shranjevanje podatkov o člankih je zatorej racionalno ter koristno, saj prinaša velike pohitritve, s hranjenjem identifikacijskih številk pa bi bili prihranki bistveno manjši, posledice zastarelih podatkov pa nekoliko hujše.

3.2.2 CiteULike

Postopek pridobivanja podatkov z repozitorija CiteULike je nekoliko drugačen. Delo nam olajša njihov vmesnik API, saj omogoča dostop do podatkov o člankih v formatu JSON. Ta je Pythonu dosti bolj pisan na kožo kakor XML. Za razliko od PubMeda prenašanje podatkov ni dvostopenjsko, saj lahko pridobimo vse potrebne informacije zgolj z enim zahtevkom HTTP. To sicer na prvi pogled deluje kot prednost, vendar se izkaže za slabost. Uporaba načina pohitritve z lokalnim shranjevanjem podatkov se tako zelo zaplete. Omenimo še tehnično zanimivost na katero naletimo tekom dostopanja do strani CiteU-

Like. Konfiguracija njihovega strežnik je takšna, da le ta avtomatsko zavrača vse zahtevke HTTP (napaka 403), ki v glavi nimajo nastavljenega atributa *User-Agent*. Dočim sama vrednost atributa sploh ni pomembna, saj deluje tudi za naključno izmišljene nize.

Iskanje podatkov o člankih po iskalnem nizu

Predno podrobneje opišemo postopek, povejmo najprej nekaj o načinu za pohitritev. Pri PubMedu smo ta del enostavno pohitrili zgolj z hranjenjem podatkov o člankih. Identifikacijske številke smo lahko prenesli vsakič znova, saj je takšen postopek za uporabnika dovolj hiter. Pri CiteULikeu zaradi enostopenjskega načina dostopa takšen pristop ni mogoč. Tehnike pohitritve bi sicer lahko izpustili, vendar bi v tem primeru sprehajanje po predhodno obiskanih straneh po nepotrebnem trajalo dlje. Odločili smo se za implementacijo tehnik za pohitritev, vendar na način primeren enostopenjskemu dostopu. Poleg podatkov o člankih hranimo še podatke o rezultatih iskanja v tabeli *CiteULikeSearch*. Vsaka zapis v tej tabeli ima tri atribute. Iskalni niz, številko strani ter seznam pripadajočih člankov. Kadar vtipkamo nek iskalni niz, nam aplikacija vrne prvih petindvajset zadetkov hkrati pa si rezultate iskanja shrani v tabelo *CiteULikeSearch*.

Poglejmo sedaj, kako deluje postopek pridobivanja podatkov. Najprej preverimo, če imamo v tabeli *CiteULikeSearch* za podani iskalni niz ter številko strani že shranjen seznam člankov. Vendar zgolj informacija, da ga imamo ni dovolj. Zanima nas tudi, kdaj smo zapis shranili v bazo. Več časa kot je poteklo od shranitve zapisa, večja je verjetnost, da so naši podatki zastareli. Namreč lahko se zgodi, da je po shranitvi izšel nov članek, ki je s tem iskalnim nizom močno povezan. Če bi zgolj uporabili lokalno shranjene podatke, bi nov članek med rezultati iskanja seveda manjkal. Problem smo zmanjšali s časovno omejitvijo, po kateri naši zapisi zastarajo. Dolžino omejitve smo nastavili na eno uro. Če torej v bazi najdemo iskani seznam člankov preverimo sledeče. Če je od vnosa zapisa potekla manj kot ena ura, vrnemo pripadajoč seznam člankov ter s tem zaključimo iskanje. Sicer, zapisa

ne upoštevamo. Vsak novo objavljen članek, bo tako na naši strani viden z maksimalno eno urno zamudo.

Kadar zapisa v tabeli *CiteULikeSearch* ni oziroma je prestar, podatke prenesemo s spletnega repozitorija tako, da pošljemo zahtevek HTTP z URLjem oblike:

`http://www.citeulike.org/json/search/all?q=<T>&per_page=<PP>&page=<P>`.

Enako kot pri PubMedu je T naš iskalni niz. Malo drugače se uporabljata argumenta PP ter P, ki sta prav tako namenjena dostopanju do podatkov po straneh. PP poda število koliko člankih želimo dobiti, P pa poda številko strani. Z vrednostma $PP = 25$ in $P = 1$ tako dobimo podatke o prvih petindvajsetih člankih, z $PP = 25$ in $P = 2$ o naslednjih petindvajsetih. Dobljena datoteka formata JSON se v Pythonu prevede v seznam slovarjev, po katerem lahko podatke iščemo brez dodatnih knjižnic. S podatki nato kreiramo seznam objektov tipa *Article*.

Pridobljene rezultate moramo sedaj še shraniti oziroma posodobiti v lokalni bazi. Najprej shranimo podatke o člankih. Za vsak članek v seznamu preverimo, če imamo zanj podatke že shranjene. Če jih imamo, jih posodobimo s trenutno pridobljenimi vrednostmi. Če jih nimamo, jih zgolj shranimo. Posodabljanje se mogoče zdi nesmiselno. Naj argumentiramo, zakaj ga vseeno uporabljamo. Čas, ki je potreben za posodobitev podatkov o članku, je napram času, potrebnem za pridobitev podatkov ter preverjanje ali so podatki za nek članke že shranjeni, zanemarljiv. Hkrati pa zagotovimo, da se morebitne spremembe vsebin na spletnem repozitoriju prenesejo tudi v lokalno bazo.

Na koncu shranimo oziroma posodobimo še zapis v tabeli *CiteULikeSearch*. Temu pripnemo tudi trenutni datum ter uro, da bomo kasneje lahko preverjali ali je zastarel.

Iskanje podatkov sorodnih člankov

Ta postopek je še nekoliko bolj zapleten. V osnovi gre tudi tukaj za iskanje po iskalnem nizu, vendar je konstrukcija samega iskalnega niza dokaj kompleksna. Zaradi enostopenjskega dostopa, bomo za pohitritev tudi tukaj uporabljali podoben pristop kot zgoraj. Kadar za nek članek pridobimo seznam njemu sorodnih člankov, si to shranimo v tabelo *CiteULikeRelated*. Vsak zapis te tabele vsebuje identifikacijsko številko članka ter seznam njemu sorodnih člankov. Enako kot v prejšnjem odseku, podatki v tabeli *CiteULikeRelated* zastarajo po eni uri, zaradi možnosti pojavitve novih sorodnih člankov. Zavoljo lažje razlage z X označimo članek, kateremu sorodne članke iščemo.

V prvem koraku tako preverimo obstoj zapisa v tabeli *CiteULikeRelated*. Če ta obstaja ter ni zastarel, vrnemo pripadajoč seznam sorodnih člankov ter smo zaključili. Sicer moramo seznam sorodnih člankov prenesti s spletnega repozitorija. Ker tudi do sorodnih člankov dostopamo s pomočjo iskanja po iskalnem nizu moramo najprej najti le tega. Pridobivanje tega iskalnega niza traja nekaj časa, zato tudi te podatke lokalno shranjujemo v tabeli *YahooAPIQueries*. Vsak zapis tabele vsebuje identifikacijsko številko članka ter iskalni niz, s katerim lahko najdemo sorodne članke. Podatki v tej tabeli ne zastarajo, saj so pridobljeni iz naslova ter povzetka članka. Le ta pa se po objavi načeloma ne spreminjata.

Če imamo za članek X iskalni niz shranjen v tabeli *YahooAPIQueries*, ga uporabimo. Sicer ga moramo še pridobiti, kar naredimo tako, da najprej pridobimo naslov ter povzetek članka X. Z malo sreče imamo podatke o članku že shranjene v bazi ter jih le preberemo. Če jih v bazi ni, dostopamo do repozitorija na naslovu URL:

`http://www.citeulike.org/json/article/<ID>`,

kjer *ID* predstavlja identifikacijsko številko članka X. Z dobljene datoteke JSON izluščimo podatke o članku, ki ga med drugim tudi shranimo v bazo. Sedaj zagotovo imamo naslov in povzetek članka X in lahko pripravimo po-

datke, ki jih bomo posredovali Yahoojevemu vmesniku API. Le ta nam bo vrnil niz, s katerim bomo lahko našli sorodne članke. Uporabimo metodo *urlencode*, ki podatke zakodira tako, da jih lahko posredujemo kot URL (glej izvorno kodo 3.1).

Izvorna koda 3.1: Priprava podatkov za Yahoojev vmesnik API

```
1 DATA = urllib.urlencode({
2     'appid': u'66ZibHHV34HmhPKjhYiT2870Hn9tcLQD9KbZ.' +
3              u'VgwFI47gVe286SogncUuiBgDn6ojnBTKQ--',
4     'output': u'json',
5     'context': u'{}␣{}'.format(<TITLE>,
6                                <ABSTRACT>
7                                ).encode('utf-8', 'ignore')
8 })
```

V kodi 3.1 *TITLE* ter *ABSTRACT* nadomestimo z naslovom ter povzetkom članka *X*. Konstanta *appid* predstavlja identifikator aplikacije CiteULike. Koda pripravi spremenljivko *DATA*, ki jo posredujemo Yahoojevemu vmesniku API z klicem:

`http://search.yahooapis.com/ContentAnalysisService/V1/termExtraction?<DATA>&callback=?`.

Ta nam vrne podatke v JSON formatu. Iz slovarja vzamemo vrednost, ki je pripeta ključu *Result* in tako pridemo do seznama ključnih besed. Iskalni niz, s katerim bomo iskali sorodne članke dobimo tako, da naslovu članka prilepimo niz, v katerem so s presledki ločene ključne besede. Označimo ga z *Q*. Tako konstruiran iskalni niz shranimo za nadaljnjo uporabo v tabelo *YahooAPIQueries*.

Sedaj lahko zares poiščemo sorodne članke. Repozitoriju posredujemo zahtevek z URLjem oblike:

`http://www.citeulike.org/json/search/all?q=<Q>&per_page=100`

V dobljenem JSON formatu poiščemo podatke o člankih ter kreiramo seznam objektov tipa *Articles*. Enako kot v prejšnjem poglavju tudi tukaj shranimo vse podatke o člankih v bazo. Hkrati poskrbimo tudi za zapis v tabeli *CiteULikeRelated*. Če le ta že obstaja, mu posodobimo seznam sorodnih člankov, sicer ga zgolj shranimo.

3.2.3 Analiza uspešnosti shranjevanja podatkov

Metode za pohitritev dostopa z lokalnim shranjevanjem podatkov so se izkazale za dokaj uspešne. Vse meritve so povprečje desetih poizkusov. Hkrati naj poudarim, da je čas merjen na strežniku. Čas potovanja zahtevka od uporabnika do našega strežnika ter nazaj torej ni vsebovan.

Poglejmo najprej meritve za repozitorij PubMed, ki so prikazane v tabeli 3.1. Polje *številke ID* vsebuje meritev časa potrebnega za prenos identifikacijskih številk z PubMeda. Polje *Podatki* vsebuje čas, ki ga porabi lokalna metoda *fetch_cache_or_web_pubmed(ids)*.

Tabela 3.1: Meritve za repozitorij PubMed

	Skupni čas brez shranjevanja		Skupni čas s shranjevanjem	
	številke ID	Podatki	številke ID	Podatki
Iskanje po iskalnem nizu	1.752s		0.352s	
	0.314s	1.436s	0.316s	0.035s
Iskanje sorodnih člankov	4.121s		0.965s	
	0.298s	3.237s	0.379s	0.110s

Skupni čas za iskanje petindvajsetih člankov smo tako znižali za 1.4 sekunde. Pri iskanju stotih sorodnih člankov je prihranek še večji, približno 3 sekunde. Opomnimo, da prihranek opazimo zgolj pri drugem dostopu do istih podatkov. Vsak prvi dostop do podatkov traja enako, kot da lokalnega shranjevanja ne bi implementirali. Obstaja tudi vmesna možnost, kjer imamo le del iskanih člankov shranjenih lokalno. Čas je v tem primeru odvisen od števila manjkajočih člankov, njegova vrednost pa je nekje med vrednostmi navedenimi v tabeli 3.1.

Opazimo tudi, da so časi za *številke ID* skoraj enaki, ne glede na to, ali gre za prenos petindvajsetih oziroma stotih številk.

Zaključimo lahko, da so metode pohitritve za PubMed dokaj uspešne, hkrati pa njihova uporaba ne prinaša slabosti.

Meritve za repozitorij CiteULike so prikazane v tabeli 3.2.

Tabela 3.2: Meritve za repozitorij CiteULike

	Skupni čas brez shranjevanja	Skupni čas s shranjevanjem
Iskanje po iskalnem nizu	1.634s	0.010s
Iskanje sorodnih člankov	11.620s	0.323s

Tukaj so prihranki bistveno večji, saj je tudi način shranjevanja podatkov drugačen. Pri iskanju po iskalnem nizu pridobimo približno 1.6 sekunde, pri iskanju sorodnih člankov pa nekaj čez 11 sekund. Poudariti je potrebno, da meritve dejansko predstavljajo mejne vrednosti. Stolpec *Skupni čas s shranjevanjem* prikazuje čas za najhitrejšo možnost, kadar imamo podatke shranjene v tabeli *CiteULikeSearch* oziroma *CiteULikeRelated* ter jih zgolj preberemo. To se zgodi zgolj v primeru, kadar do istih podatkov dostopamo drugič v roku ene ure. Prihranek tukaj je ogromen, vendar za ceno enournega zamika pri pojavitvi novih člankov. Pri iskanju sorodnih člankov nato še izračunamo podobnosti, kar je vzrok za nekoliko daljši čas. Stolpec *Skupni čas brez shranjevanja* pa predstavlja najbolj pesimističen scenarij. To je takrat, kadar je lokalna baza popolnoma prazna ter moramo zato najprej dostopati do podatkov o članku, nato do Yahoojevega vmesnika API ter na koncu do seznama sorodnih člankov. Pri normalni uporabi, se tipično zgodi, da imamo podatke o članku že shranjene, kar nekoliko pohitri dostop. Lahko imamo shranjen tudi iskalni niz, ki ga vrne Yahoojev vmesnik API, kar iskanje še dodatno pohitri.

Čas prikazan v tabelah 3.1 ter 3.2 moramo razumeti kot skrajne vrednosti. Pri tipični uporabi spletne aplikacije, bo porabljen čas nekje vmes. Upoštevati moramo tudi, da dostopamo to tujih repozitorijev preko spleta. To lahko občasno povzroči nepričakovano podaljšanje. Z podanimi meritvami tako zgolj skušamo prikazati prednosti lokalnega shranjevanja podatkov ter okvirno podati, koliko časa bi naj porabili za določeno iskanje.

3.3 Mere podobnosti

Zaradi pomanjkljivosti Jaccardovega indeksa, smo za mero podobnosti izbrali kosinusno podobnost na vektorjih, ki jih dobimo kot rezultat transformacije TF-IDF. Najprej smo iz množice člankov izločili tiste, ki nimajo podanih povzetkov, saj za njih ne bo mogoče računati podobnosti. Nato smo povzetke posredovali metodi *fit_transform* razreda *TfidfVectorizer*, ki se nahaja v paketu *sklearn* (glej poglavje 3.1.1). Metoda naredi TF-IDF transformacijo, sposobna je odstraniti manj pomenske besede, delati predprocesiranje besed, normalizacijo vektorjev ...

Knjižnica Scikit learn predstavlja ključen del pri računanju podobnosti, zato si bomo njeno uporabo pogledali malo pobližje.

Izvorna koda 3.2: Transformacija TF-IDF ter računanje kosinusne podobnosti na povzetkih člankov.

```
from sklearn.feature_extraction.text import TfidfVectorizer
from sklearn.metrics.pairwise import linear_kernel
from nltk.stem.wordnet import WordNetLemmatizer
from nltk.corpus import stopwords

lmtzr = WordNetLemmatizer()

def lema(str):
    return '␣'.join([
                     lmtzr.lemmatize(i, 'v')
                     if lmtzr.lemmatize(i, 'v') != i
                     else lmtzr.lemmatize(i)
                     for i in str.split('␣')
                    ])

stopWords = stopwords.words('english')

tfidf = TfidfVectorizer(stop_words=stopWords,
                        min_df=1,
                        preprocessor=lema,
                        norm='l2',
                        use_idf=True,
                        smooth_idf=True
                        )\
    .fit_transform(list_of_documents)
cosine_similarities = linear_kernel(tfidf, tfidf)
```

Ob klicu konstruktorja razreda *TfidfVectorizer* smo nastavili nekaj pomembnih vrednosti (glej izvorno kodo 3.2):

- **stop_words**: seznam manj pomenskih besed, ki jih želimo izločiti. S tem tudi zmanjšamo dimenzionalnost vektorskega prostora. V spremenljivki *stopWords* je seznam besed za angleščino iz paketa NLTK.

- **min_df**: meja vsaj kolikokrat se mora beseda pojaviti v množici dokumentov, da je bomo upoštevali kot dimenzijo v skupnem vektorskem prostoru. Vse besede, ki se pojavijo manjkrat v predstavitvi v vektorskem prostoru izpustimo. Mi bomo za dimenzije v vektorskem prostoru upoštevali vse besede, ki se pojavijo v množici dokumentov (z izjemo manj pomenskih besed).

- **preprocessor**: funkcija, ki lahko pred TD-IDF transformacijo poljubno preoblikuje vsak dokument. Definirali smo funkcijo *lema*, ki opravlja lematizacijo, čeprav je konsenz, da za angleški jezik lematizacija v povprečju prinaša zgolj majhne izboljšave [11]. Lematizacija je pretvarjanje besede v njeno osnovno slovarsko obliko. Naprimer lematizator pretvori angleško besedo *goes* v besedo *go*, *walking* v *walk*, *cars* v *car* ... Uporabili smo lematizator *WordNetLemmatizer*, ki je del paketa NLTK. Temu lematizatorju je potrebno ob klicu funkcije *lemmatize* podati drug argument *'v'*, kadar želimo lematizirati glagol. Če mu tega argumenta ne podamo, lematizator glagolov ni sposoben lematizirati. V funkciji *lema* vsako besedo najprej poizkusimo lematizirati kot glagol, če se ne spremeni pa poizkusimo še brez argumenta *'v'*. Če lematizator besed v svojem slovarju ne najde jih ne spreminja.

- **norm**: uporabili smo Evklidsko (*l2*) normalizacijo vektorjev. Ta priredi vrednosti komponent vektorja tako, da je dolžina vektorja enaka 1. Smeri vektorja seveda ne spreminja.

- **use_idf**: kadar je vrednost True, uporabi uteži IDF. Sicer uporabi samo vrednosti TF.

- **smooth_idf**: vse IDF uteži zgladi tako, da frekvencam prišteje enko, kot da bi obstajal dokument, ki bi vseboval vse izraze skupnega vektorskega prostora natančno enkrat [9].

Na tako kreiranem objektu nato pokličemo metodo *fit_transform*, ki ji podamo seznam povzetkov naših člankov. Ta naredi TF-IDF transformacijo ter vrne vektorje, ki predstavljajo dane povzetke v skupnem vektorskem prostoru, kot smo opisali v poglavju 2.2.1. Nad tako transformiranimi povzetki lahko računamo podobnost povzetkov s kosinusno podobnostjo. Ta je ekvivalentna klicu metode *linear_kernel*, saj so vektorji, ki jih vrne *fit_transform* že Evklidsko normalizirani [9]. Spremenljivka *cosine_similarities* je tako dvodimenzionalna tabela tipa numpy, ki hrani izračunano kosinusno podobnost med vsakim parom podanih člankov. Matrika *cosine_similarities* je seveda simetrična, saj je relacija podobnosti dveh povzetkov simetrična. Članek a je podoben članku b točno enako, kot je članek b podoben članku a. Naj še omenim, da do podobnosti med dvema člankoma dostopamo z njunima zaporednima številkama v seznamu *list_of_documents*. Podobnost med prvim ter drugim člankom v seznamu *list_of_documents* lahko tako dobimo s klicem *cosine_similarities[0][1]*.

3.4 Vizualizacija mreže

Zadnji korak implementacije je podobnosti, izračunane v prejšnjem koraku, uporabniku čim bolj pregledno prikazati. V poglavju 3.3 smo opisali postopek, s katerim smo izračunali podobnosti med vsemi pari podanih člankov. Naj še omenim, da smo število sorodnih člankov omejili na sto, saj so večji grafi bistveno manj pregledni. Če bi na mreži prikazali vse izračunane podobnosti, bi bil graf poln - torej obstajala bi povezava med vsakim parom vozlišč. Tak prikaz ne bi bil niti pregleden, zaradi ogromnega števila križajočih se povezav, pa tudi strukturnih zakonitosti mreže ne bi mogli opazovati. Zato smo se odločili, da na grafu prikažemo zgolj nekaj najpomembnejših povezav. Število povezav smo omejili na 1.5 krat število vozlišč. V našem primeru,

ko v mreži prikažemo največ sto člankov, to predstavlja največ sto petdeset najpomembnejših povezav. Najpomembnejše povezave so tiste, ki povezujejo članke, ki so si najbolj podobni.

Poglejmo si postopek prikazovanja mreže malo pobližje. V tem koraku imamo na strežniku podatke o člankih ter podatke o sto petdesetih najpomembnejših povezavah. Podobnosti pripete tem povezavam, so v razponu $[0, 1]$. Predno podatke za prikaz pošljemo odjemalcu, se na strežniku te podobnosti še zaokrožijo na deset razredov, glede na pomembnost povezave. Zaokrožujemo po naslednji enačbi:

$$max\left(ceil\left(\frac{v - minS}{maxS - minS} * 10\right), 1\right), \tag{3.1}$$

kjer $minS$ predstavlja minimalno podobnost med vsemi sto petdesetimi podobnostmi, $maxS$ pa maksimalno podobnost. Vrednost v predstavlja podobnost neke povezave, katere vrednost zaokrožujemo. Podobnosti so sicer res z intervala $[0, 1]$, vendar podobnosti z vrednostjo 0 oziroma 1 tipično nimamo. Zato vse vrednosti z uporabo $minS$ ter $maxS$ skaliramo na interval $[0, 1]$. Te vrednosti potem množimo z deset, ter uporabimo funkcijo strop (*ceil*), ki vrednosti navzgor zaokroži na celo število. Naprimer 0.1 ter 0.9 se s funkcijo *ceil* tako zaokrožita na vrednost 1. Funkcijo max uporabimo zato, da najmanjšo vrednost med podobnostmi spremenimo v 1 . Namreč $ceil(0) = 0$, za podobnost pa vrednosti 0 ne želimo, saj bodo te vrednosti predstavljale debelino povezav na grafu. Tako spremenjene podobnosti zavzemajo vrednosti z intervala $[1, 10]$, pri čemer sta vrednosti 1 in 10 vedno prisotni.

Podatke o povezavah (podobnostih) in vozliščih grafa (člankih) sedaj zapakiramo v objekt JSON, ki ga nato posredujemo odjemalcu za prikaz.

Na odjemalcu za prikaz skrbi skripta v JavaScriptu. Ta najprej podatke razčleni (*parse*), predenj jih lahko uporablja. Naj poudarimo, da se na strežniku ne ukvarjamo z računanjem pozicij točk. Za to skrbi knjižnica D3 (glej poglavje 3.1.2), ki uporablja Verletjevo integracijo (glej poglavje 2.3.2). Prednosti je več. Prvič, na strežniku po nepotrebnem ne zapravljamo časa

za izračun pozicij. Drugič, namesto golega izrisa grafa, lahko na odjemalcu prikažemo animacijo torej vsak korak iterativnega algoritma, ki izračuna pozicije točk. Takšen prikaz je z uporabniškega stališča bolj atraktiven. Knjižnica hkrati tudi skrbi na kasnejše animirano premikanje vozlišč.

D3 ima zanimiv pristop k prikazovanju oziroma izrisovanju, ki temelji na podatkih (*data-driven*). Za risanje nekaj točk, povezav ter ostalih elementov tako ne uporabljamo zank. Poglejmo si naprimer kodo za risanje grafa:

Izvorna koda 3.3: Koda za izris grafa s knjižnico D3

```
var svgGraph = d3.select("#svgGraph");
var myNodes = $.parseJSON(nodes);
var myLinks = $.parseJSON(links);
var color = d3.scale.category20();
var scaleLength = d3.scale.linear()
        .domain([1, 10])    /* in values*/
        .rangeRound([150, 30]) /* out values */
var scChrg = d3.scale.linear()
        .domain([300, 850])    /* in values*/
        .rangeRound([-30, -150]); /* out values */
var scaleCharge = function(d){
        return Math.min(scChrg(d), -30)
}

var force = d3.layout.force()
        .charge(scaleCharge(Math.min(graphSize[0],
                                     graphSize[1])))
        .linkDistance(function(d){
                return scaleLength(d.value);
        })
        .size(graphSize);

force
        .nodes(myNodes)
        .links(myLinks)
        .start();

var link = svgGraph.selectAll(".link")
        .data(myLinks)
        .enter().append("line")
        .attr("class", "link")
        .style("stroke-width", function(d) {
                return d.value;
        });


```

```
37 var node = svgGraph.selectAll(".node")
38         .data(myNodes)
39         .enter().append('circle')
40         .attr("class", function(d){
41                 return "node␣"+ journalToClass(d.journal )}
42         )
43         .attr("r", function(d, i) {
44                 return d.article_id == id ?
45                         radSel : rad
46         })
47         .style("fill", function(d) {
48                 return d.article_id == id ?
49                         "white": color(d.journal);
50         })
51         .style("stroke", function(d) {
52                 return d.article_id == id ?
53                         color(d.journal): "#fff";
54         })
55         .call(force.drag);
```

S spremenljivko *svgGraph* lahko dostopamo do SVG elementa na katerega bomo risali graf. Spremenljivki *myNodes* ter *myLinks* vsebujeta vse podatke o vozliščih ter povezavah. Funkcija *color* vhodni spremenljivki priredi eno izmed dvajsetih najprej definiranih barv. Funkcija *scaleLength* pa vhodno vrednost z intervala $[1, 10]$ preslika v vrednost na intervalu $[150, 30]$. S spremenljivko *force* lahko dostopamo do grafa. S klici kot sta *charge*, *linkDistance* ter ostali spreminjamo nastavitve našega grafa. Z *linkDistance* smo tako nastavili dolžino vsake povezave kot vrednost, ki jo vrne funkcija *scaleLength*. Z klicem *charge* pa smo vsem vozliščem nastavili naboj na vrednost, ki jo vrača funkcija *scaleCharge*. Le ta vrednost določi glede na velikost okna tako, da se mu graf čim bolje prilega. Naboj negativne vrednosti pomeni, da se vozlišča odbijajo. Spremenljivka *id* nosi identifikacijsko številko članka, okrog katerega rišemo mrežo sorodnih člankov. Vrednosti spremenljivk *rad*, *radSel*, *nodes*, *links* ter *graphSize* so v skripti predhodno definirane. Zanimivo je to, da lahko kot nastavitev podamo funkcijo, ki glede na vrednosti podatkov izračuna in vrne neko vrednost.

Ključna stvar, za razumevanje izvorne kode 3.3 je, da knjižnica D3 deluje primarno s podatki. V vrstici 37 tako v selekcijo *node* shranimo vse objekte razreda *node*. Vendar objektov razreda *node* nismo nikjer kreirali, zato tudi

ne morejo obstajati. V tem trenutku je v spremenljivki *node* dejansko prazen seznam, oziroma drugače selekcija *node* je prazna. Nato selekciji pripnemo podatke *myNodes* (gre za seznam naših vozlišč) z klicem *data*. Povezovanje primarno deluje tako, da se prvi element s seznam podatkov poveže z prvim elementom selekcije, drugi element s seznama podatkov z drugim elementom selekcije itd. Način povezovanja lahko definiramo tudi drugače, vendar nam ustreza privzeti način. Če je število elementov selekcije enako številu podatkov, se povezovanje lepo izide. V primeru, kadar število elementov selekcije ne ustreza številu podatkov, pa moramo knjižnici povedati, kaj naj naredi s preostalimi elementi. Imamo dva možna primera. Prvi, da je v selekciji manj elementov, kot je podatkov. To je tudi primer v naši izvorni kodi, saj je naša selekcija prazna. Takrat z klicem *enter()* knjižnici povemo, kaj naj stori s podatki, ki nimajo para v selekciji. Mi smo tem podatkom ustvarili nove krogce s klicem *.append('circle')*. Na novo kreiranih krogcih smo nato s klici *attr* nastavili še razred, polmer ter barvo. V drugem primeru, torej kadar je v selekciji več elementov kot imamo podatkov, pa bi klicali selekcijo *exit()*. Recimo, da bi se preko obstoječega odločili narisati nek drug, manjši graf. To bi storili tako, da bi izbrali vsa vozlišča razreda *node*, jim pripeli podatke o novih vozliščih, z selekcijo *exit()* pa bi odstranili vsa odvečna vozlišča s platna SVG.

Obrazložimo še kako lahko pri nastavljanju vrednosti dostopamo do podatkov, ki so povezani s tem elementom. Izvorna koda 3.3 v vrstici 40 s funkcijo nastavi vrednost razreda. Ko knjižnica kliče to funkcijo, ji poda dva argumenta. Prvi so podatki pripeti temu elementu selekcije, kot drug argument pa poda indeks elementa selekcije. Mi potrebujemo samo dostop do podatkov, zato smo funkcijo spisali tako, da sprejema samo en argument *d*. Znotraj funkcije lahko potem do podatkov dostopamo preko spremenljivke *d*. V našem primeru smo v vrstici 38 kot podatke pripeli seznam objektov tipa *Article* (za opis objekta glej poglavje 3.2). Zato lahko spremenljivko *d* obravnavamo kar kot objekt *Article* ter preko nje dostopamo do atributov objekta, kot so *journal*, *url*, *title*, *abstract* ...

Funkcija *journalToClass*, uporabljena v vrstici 41, ime revije, v kateri je bil članek objavljen, spremeni v ime, ki je varno za uporabo kot ime razreda. To stori tako, da vse znake, ki niso mala oziroma velika črka, odstrani iz imena revije. Z vrsticami 40-42 tako vsakemu krogcu, ki ga narišemo na platnu SVG hkrati priredimo vrednost razreda, glede na ime revije. To bomo potrebovali kasneje za dodajanje interaktivnosti.

Krogci se odzivajo tudi na dogodke miške. Če se z miško postavimo na nek članek, se nam v desnem delu ekrana prikažejo vsi podatki o tem članku (naslov, avtorji, revija, povzetek ter povezava na izvorno spletno stran). Ob dvojnem kliku na neko vozlišče nas spletna stran preusmeri na novo stran, na kateri se izriše mreža okrog članka, na katerega smo kliknili.

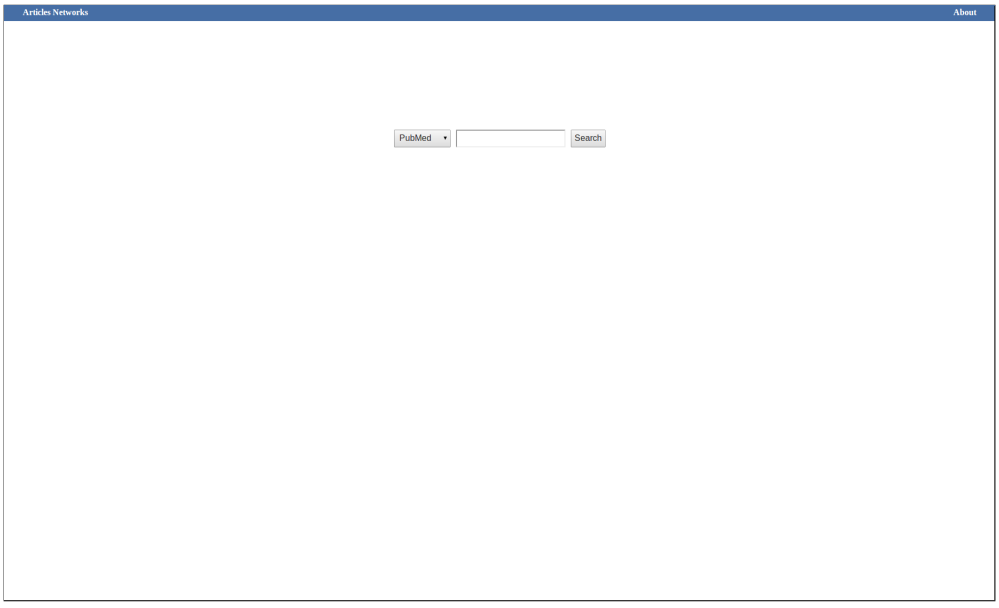
Poleg risanja grafa, naša skripta skrbi tudi za prikaz množice revij na dnu ekrana. Najprej s seznama *myNodes* konstruiramo množico vseh revij. Le to nato na zelo podoben način, kot v kodi 3.3 izrišemo na dnu zaslona. Implementirali smo tudi nekaj efektov, ki poudarijo članke določene revije. Tako se naprimer povečajo vsa vozlišča, ki predstavljajo članke določene revije, kadar se z miško postavimo nad naslov neke revije. To je tudi razlog zakaj smo vsem vozliščem grafa nastavljali vrednosti razreda.

Pri implementaciji nam je bil v ogromno pomoč primer s spletne strani [3]. Animacija, ki se pojavi po dvojnem kliku vozlišča, pa je povzeta po primeru [12].

Poglavje 4

Scenarij uporabe

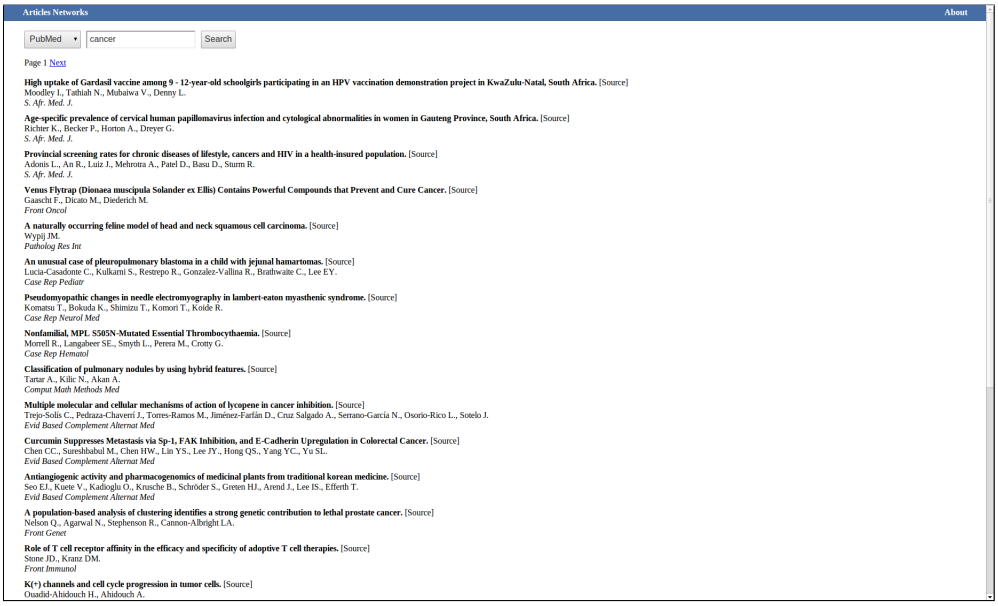
Za konec pa si poglejmo še tipičen primer uporabe naše spletne aplikacije. Ko v brskalnik vnesemo spletni naslov `http://butler.fri.uni-lj.si/articles`, se nam prikaže domača stran, ki jo lahko vidimo tudi na sliki 4.1.



Slika 4.1: Domača stran spletne aplikacije.

Zgoraj levo je povezava *Articles Networks* s katero se lahko tekom brskanja po spletni strani kadarkoli vrnemo na to domačo stran. Zgoraj desno je

povezava *About* do strani, na kateri si lahko preberemo nekaj o uporabi naše spletne aplikacije. Recimo, da iščemo članke o razredu bolezni rak (*cancer*). Najprej s spustnega seznama izberemo spletni repozitorij, po katerem bomo iskali. Na voljo imamo PubMed ter CiteULike, odločimo se za prvega. V vnosno polje nato vnesemo želeni izraz, v našem primeru *cancer*, ter kliknemo gumb *Search*. Aplikacija nam prikaže rezultate iskanja v seznamu, kot prikazuje slika 4.2.



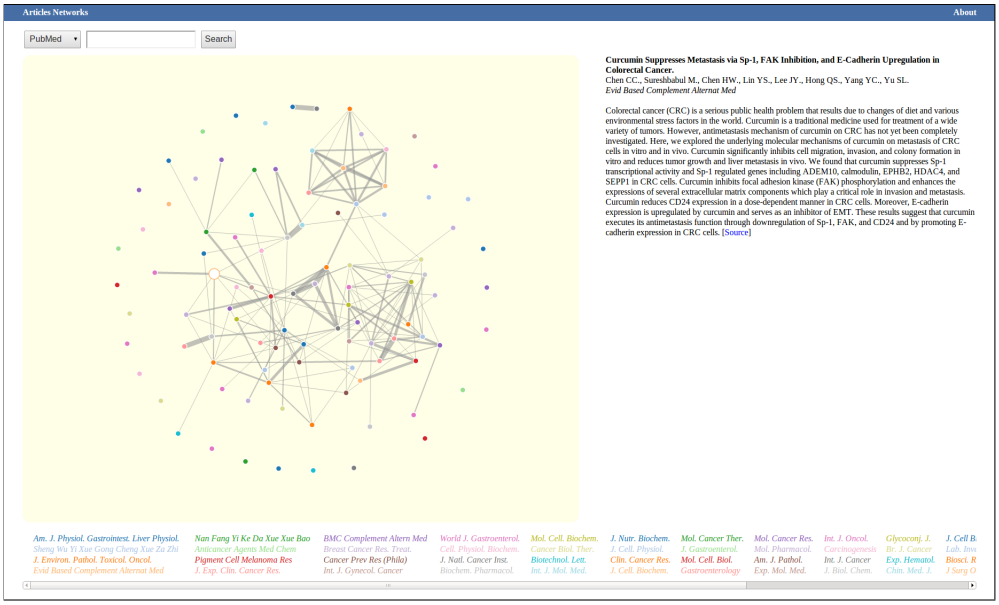
Slika 4.2: Seznam zadetkov za iskalni niz *cancer* iz repozitorija PubMed.

V seznamu je prikazanih prvih petindvajset zadetkov. Da bi videli več rezultatov uporabimo povezavo *Next*, ki se nahaja takoj pod vnosnim poljem zraven napisa *Page 1*, ki ponazarja, da se trenutno nahajamo na prvi strani. Kadar raziskujemo naslednje strani, se poleg povezave *Next*, pojavita še povezavi *Previous* ter *Home*, ki nas vrneta na prejšnjo oziroma prvo stran. Enake povezave se nahajajo tudi na koncu seznama zadetkov.

V seznamu je za vsak članek najprej prikazan njegov naslov v odebeljeni pisavi. V drugi vrstici je seznam avtorjev članka, v tretji pa revija v kateri je članek izšel. Poleg naslova je tudi povezava z imenom *Source*, ki nas vodi

na spletno stran, s katere smo prenesli podatke o članku.

Denimo, da smo se po pregledu naslovov odločili podrobneje prebrati enajsti članek s seznama, zato kliknemo na njegov naslov. Prikaže se jedrna stran naše aplikacije, kjer je okrog izbranega članka izrisana mreža podobnih člankov. Izgled mreže prikazuje slika 4.3.



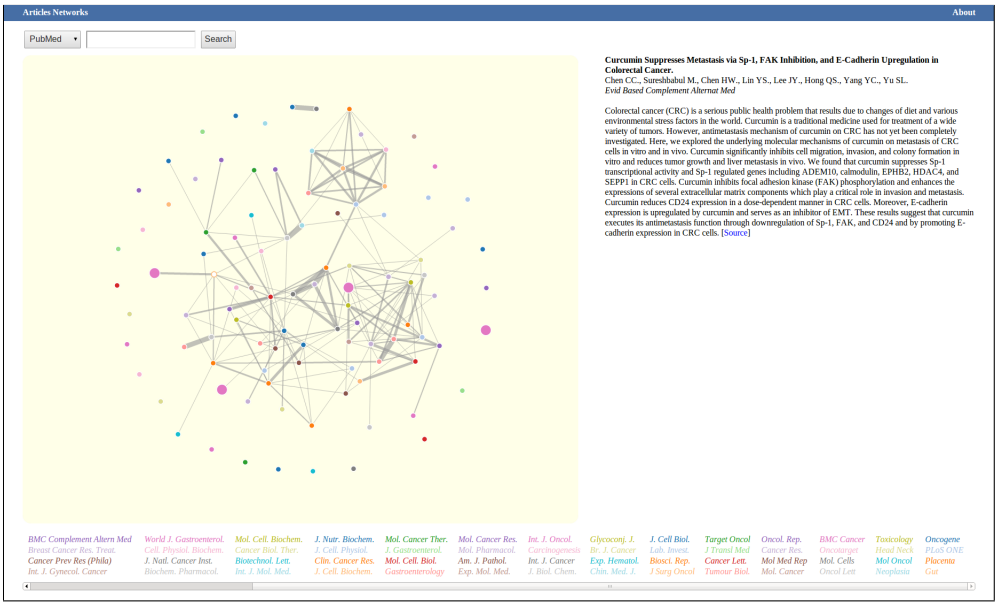
Slika 4.3: Mreža članku *Curcumin Suppresses Metastasis via Sp-1, FAK Inhibition, and E-Cadherin Upregulation in Colorectal Cancer.* sorodnih člankov.

Informacije prikazujemo v treh poljih. Levo je na svetlo rumenem ozadju prikazan graf sorodnih člankov okrog izbranega članka. Povezave predstavljajo sorodnost med dvema povezanima člankoma. Debelejša kot je povezava, bolj sta si članka sorodna.

Desno je prostor, kjer prikazujemo informacije o nekem članku. Takoj ko se stran naloži, bodo tukaj vidne informacije o članku, okrog katerega rišemo mrežo. Ta je na grafu pobarvan z belo barvo ter obrobljen z barvo revije. Vsi ostali članki so pobarvani z barvo revije. Članki, ki jim podatek o reviji manjka, so pobarvani z črno barvo. Če želimo videti informacije o katerem drugem članku v grafu, se z miško preprosto postavimo nanj.

Vozlišču, ki predstavlja članek, katerega podatke trenutno prikazujemo, se poveča polmer. V oknu za prikaz informacij je na vrhu z odebeljeno pisavo prikazan naslov članka. Sledita mu seznam avtorjev ter ime revije, v kateri je bil članek objavljen. Spodaj je prikazan še povzetek ter povezava na spletno stran, s katere smo prenesli podatke o članku. Čisto na dnu vidimo še seznam ključnih besed tega članka, kadar so te podane.

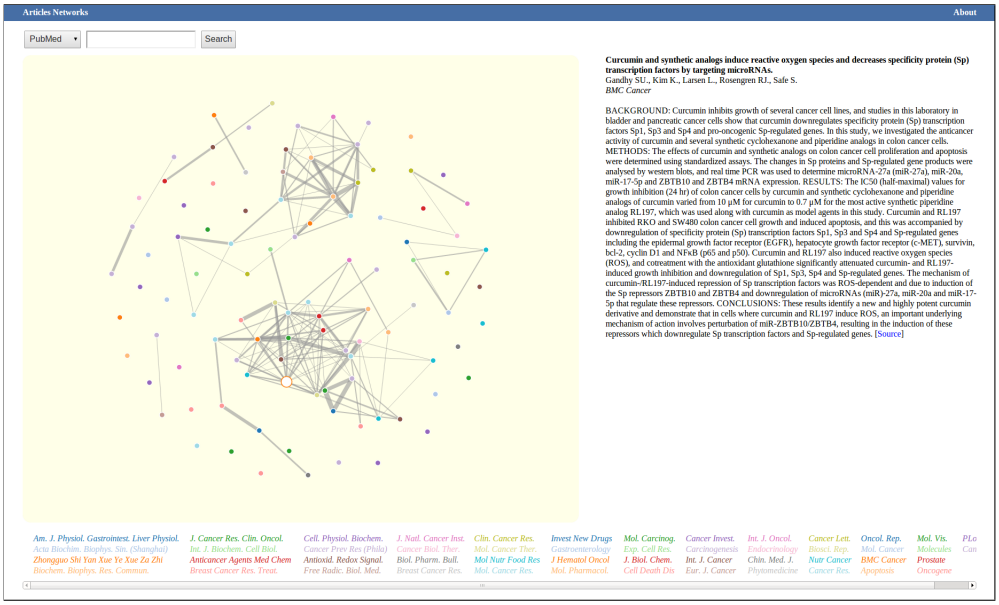
Zadnje polje za prikaz informacij je vidno na dnu ekrana. Tukaj je prikazana množica revij, v katerih so bili članki objavljeni. Naslovi revij ter pripadajoči članki v grafu so pobarvani z enako barvo. Za barvanje uporabljamo samo dvajset barv, zato smo implementirali poudarjanje člankov po revijah. Kadar se z miško postavimo nad naslov neke revije, se bodo krogci pripadajočih člankov v grafu povečali. Poudarjanje člankov po revijah prikazuje slika 4.4.



Slika 4.4: Graf s slike 4.3 s poudarjenimi članki iz revije *BMC Cancer*.

Po raziskovanju grafa ter branju nekaj povzetkov, smo se odločili, da želim videti še graf sorodnih člankov za članek, ki je po mreži sodeč trenutnemu članku najbolj soroden. To je članek, pobarvan z svetlo vijolično barvo, ki

se nahaja levo od trenutnega članka. Z dvojnim klikom na ta članek nam aplikacija nariše želeni graf, kot prikazuje slika 4.5.



Slika 4.5: Mreža članku *Curcumin and synthetic analogs induce reactive oxygen species and decreases specificity protein (Sp) transcription factors by targeting microRNAs.* sorodnih člankov.

Poglavje 5

Sklepne ugotovitve

V diplomskem delu smo se najprej spopadli s pridobivanjem podatkov iz tujih repozitorijev. PubMed ima napram repozitoriju CiteULike veliko bolj dodelan vmesnik za dostop. Z željo po pohitritvi dostopa do repozitorijev smo implementirali lokalno shranjevanje podatkov. Za PubMed je bila implementacija pohitritve enostavnejša ter zelo uspešna. Za CiteULike je predstavljala večji izziv, hkrati pa se nismo mogli izogniti ene slabosti. Novo objavljeni članki, se na naši strani pojavijo z maksimalno enourno zamudo. Aplikacijo bi lahko izboljšali z dodajanjem novih spletnih repozitorijev. Tako bi lahko implementirali še dostop do člankov iz digitalne knjižnice ACM, Googlovega Učenjaka ter ostalih.

Med članki smo nato definirali mero sorodnosti. Zaradi pomanjkljivosti Jaccardovega indeksa, smo sorodnost raje računali s kosinusno podobnostjo med TF-IDF transformiranimi vektorji. Če bi želeli uspešnost naše mere sorodnosti dodobra oceniti, bi potrebovali množico člankov, med katerim bi strokovnjaki s področja tematike člankov ročno določili sorodnosti. S temi podatki žal ne razpolagamo, naša rešitev zato spada v družino algoritmov nenadzorovanega učenja. Kljub temu menimo, da je naša mera uspešna. Kot možnost za izboljšavo mere sorodnosti, predlagamo upoštevanje ključnih besed članka. Še bolje, za članke iz repozitorija PubMed bi lahko uporabili izraze *MeSH*. Gre za enoten kontroliran seznam šestindvajset tisočih hie-

rarhično urejenih biomedicinskih izrazov. Strokovnjaki te izraze člankom dodeljujejo ročno, tipično jih vsakemu članku dodelijo deset do dvanajst. Kako izraze *MeSH* vključiti v računanje podobnosti, je področje, ki bi ga vsekakor bilo vredno raziskati.

Za vizualizacijo smo sprva uporabljali knjižnico *Raphaël*. Ta se je izkazala za preokorno, saj ni vsebovala podpore za risanje mrež. Vsako vozlišče in povezavo je bilo tako potrebno narisati ročno. Njena druga pomanjkljivost je, da ne vsebuje algoritmov za razporejanje točk grafa. Zaradi tega smo raje prešli na uporabo knjižnice D3, ki se je za naše potrebe izkazala za veliko bolj primerno. Računanje postavitve točk, smo tako s strežnika prenesli na knjižnico D3.

S tehničnega stališča, pošiljanje podatkov o člankih v grafu vsekakor ni dovršeno. Vse podatke sedaj pošljemo zakodirane v objektu formata JSON. Veliko bolje bi bilo, če bi poslali zgolj podatke o vozliščih in povezavah, podatke o samem članku pa bi po potrebi s strežnika prenesli z uporabo tehnologije AJAX. S tem bi se skrajšal tudi čas prenosa podatkov o grafu od strežnika do nas ter bi tako nekoliko pohitrili uporabo aplikacije.

Aplikacija omogoča pregledovanje akademskih člankov na nov, inovativen način, saj poizkuša čim več povedati o medsebojnih odnosih med članki. Njena uporaba je enostavna ter intuitivna. Menimo, da smo s tem dosegli zastavljene cilje ter upamo, da bo aplikacija služila čim več nadebudnim raziskovalcem.

Literatura

[1] B. Aaron. (2013). *jQuery Mouse Wheel Plugin* [Spletni vir]. Dostopno na: `https://github.com/brandonaaron/jquery-mousewheel`

[2] M. Bostock. (2012). *Data-Driven Documents* [Spletni vir]. Dostopno na: `http://d3js.org/`

[3] M. Bostock. (2013). *Force-Directed Graph* [Spletni vir]. Dostopno na: `http://bl.ocks.org/mbostock/4062045`

[4] F. Ercolessi. (1997). *A molecular dynamics primer* [Spletni vir]. Dostopno na: `http://www.fisica.uniud.it/~ercolessi/md/md/`

[5] A. Holovaty, J. Kaplan-Moss. (2013). *The Definitive Guide to Django: Web Development Done Right* [Spletni vir]. Dostopno na: `http://www.djangobook.com/en/2.0`

[6] T. Jakobsen. (2012). Advanced Character Physics [Spletni vir]. Dostopno na: `ftp://ftp.archlinuxcn.org/advanced_character_physics.pdf`

[7] S. G. Kobourov. (2012). Spring Embedders and Force Directed Graph Drawing Algorithms. *CoRR* [Spletni vir]. (abs/1201.3011). Dostopno na: `http://arxiv.org/pdf/1201.3011v1.pdf`

[8] C. D. Manning, P. Raghavan, H. Schütze. (2008). *Introduction to Information Retrieval* [Spletni vir]. Dostopno na: `http://www-nlp.stanford.edu/IR-book/`

[9] F. Pedregosa et al. (2011). Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* [Spletni vir]. (12), str. 2825-2830. Dostopno na: `http://jmlr.org/papers/v12/pedregosa11a.html`

[10] E. Sayers. (2013). *Entrez Programming Utilities Help [Internet]* [Spletni vir]. Dostopna na: `http://www.ncbi.nlm.nih.gov/books/NBK25500/`

[11] A. Singhal, Google Inc. (2001). Modern Information Retrieval: A Brief Overview. *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering* [Spletni vir]. (24). Dostopno na: `http://singhal.info/ieee2001.pdf`

[12] M. Woelk. (2013). *d3.js Loading Spinner Icon* [Spletni vir]. Dostopno na: `http://bl.ocks.org/Mattwoelk/6132258`

[13] (2013). *ACM Digital Library* [Spletni vir]. Dostopno na: `http://dl.acm.org`

[14] (2013). *CiteULike* [Spletni vir]. Dostopno na: `http://www.citeulike.org`

[15] (2013). *Django web framework* [Spletni vir]. Dostopno na: `http://en.wikipedia.org/wiki/Django_(web_framework)`

[16] (2013). *Google Učenjak* [Spletni vir]. Dostopno na: `http://scholar.google.si`

[17] (2013). *JavaScript* [Spletni vir]. Dostopno na: `https://en.wikipedia.org/wiki/JavaScript`

[18] (2013). *jQuery* [Spletni vir]. Dostopno na: `http://en.wikipedia.org/wiki/JQuery`

[19] (2013). *Lxml - XML and HTML with Python* [Spletni vir]. Dostopno na: `http://lxml.de/index.html`

[20] (2013). *PubMed* [Spletni vir]. Dostopno na: `http://www.ncbi.nlm.nih.gov/pubmed`

[21] (2013). *Python programming language* [Spletni vir]. Dostopno na: `http://en.wikipedia.org/wiki/Python_(programming_language)`