ELEKTROTEHNIČKI FAKULTET
BEOGRAD

Algoritamski aspekti razvoja i implementacije Web pretraživača

Diplomski rad

iz predmeta:
Pretraživanje i Istraživanje podataka na Internetu
RI5PIP

Student : Aleksandar Bradić 231/99
Mentor : prof. dr. Veljko Milutinović

Beograd
Mart 2007.

Sadržaj

1. Uvod 5
2. Osnovni problemi i algoritmi u razvoju i implementaciji Web pretraživača 7
  2.1. Uvod 7
  2.2. Crawling 9
    2.2.1. Osnove 9
    2.2.2. Eliminacija duplikata stranica 12
    2.2.3. Case Study : Nutch crawler 14
  2.3. Indeksiranje 15
    2.3.1. Osnove 15
    2.3.2. Metode estimacije veličine indeksa 16
    2.3.3. Case Study : Lucene indexer 18
  2.4. Rankiranje 21
    2.4.1. Osnove 21
    2.4.2. PageRank 21
    2.4.3. HITS 25
  2.5. Clustering 27
    2.5.1. Osnove 27
    2.5.2. Markov Clustering (MCL) 29
    2.5.3. Iterative Conductance Cutting (ICC) 30
3. Algoritam za grupisanje rezultata pretraživanja, korišćenjem slučajnih lutanja na link grafu 31
  3.1. Uvod 31
  3.2. Postavka problema 32
  3.3. Postojeća rešenja 33
  3.4. Predloženi algoritam 34
  3.5. Analiza 36
  3.6. Dalja istraživanja 37
  3.7. Implementacija 38
  3.8. Zaključak 40
4. randomNode : Implementacija clustering web pretraživača 41
5. Zaključak 48
6. Reference 49

1. Uvod

Pretraživanje danas predstavlja ključni način pristupa sadržaju na Web-u. Usled enormnih dimenzija (preko 8 milijardi stranica), pristup informacijama na Web-u, *direktnim* posećivanjem svake od stranica korišćenjem njenog URL-a, ograničava korisnika isključivo na skup unapred poznatih stranica, što predstavlja gotovo zanemarljiv segment celokupnog Web-a. Jedan od načina koji omogućava efikasniji pristup, predstavljaju tzv. *direktorijumi* (Yahoo!, dmoz..), koji podrazumevaju kataloge web stranica, klasifikovane po tematici. Iako inicijalno popularan, ovakav pristup se ubrzo pokazao kao neefikasan, prvenstveno usled ogromne brzine ekspanzije Web-a, koja uzrokuje novim sadržajima, čije je pronalaženje i klasifikacija (koja se najčešće obavlja manuelno), izuzetno zahtevan posao. Dodatno, sa sve većim povećavanjem količine informacija, klasifikacija se pokazuje kao nedovoljno efikasan pristup za rešavanje problema *preopterećenosti informacijama* (*information overload*).

Najefikasniji vid pristupa sadržaja na Web-u, jeste korišćenje *pretraživanja*. Na ovaj način, izbegava se potreba za sistematskom organizacijom celokupnog Web-a, već se deo problema prenosi na samog *korisnika*, koji svojim *upitima*, vrši selekciju segmenata kojima želi da pristupi. Zadatak *Web pretraživača* (*web search engine*) je da pruži mehanizme koji korisniku omogućavaju *efikasno* pretraživanje i *pomoć* pri selekciji *relevantnog* sadržaja. U poslednjoj deceniji, razvijen je značajan broj izuzetno efikasnih pretraživača, od kojih neki danas predstavljaju najposećenjije lokacije na Web-u (Google, Yahoo! Search, Ask..).

Osnovni motiv ovog diplomskog rada jeste prikazivanje različitih algoritamskih problema koji se javljaju u razvoju i implementaciji web pretraživača. Osnovni cilj pretraživača – omogućavanje *efikasnog* pristupa *najrelevantnijim* informacijama na Web-u, predstavlja izuzetan algoritamski problem, kako zbog količine podataka nad kojima se navedeni algoritmi izvršavaju, tako i zbog nemogućnosti *preciznog* određivanja pojmova *relevantnosti* i *strukture*. Time shodno, postojeći algortimi koji omogućavaju rešavanje nekih od ovih problema, predstavljaju *state-of-the-art* rešenja u okviru teorije algoritama i služe kao osnova i referenca budućih istraživanja u oblasti Web search algoritama. Neki od najvaznijih algoritama iz ove oblasti, prikazani su okviru *poglavlja 2*.

Dodatno, u okviru diplomskog rada je predstavljena open-source platforma koja omogućava samostalan razvoj *in-house* pretraživača, koja bi, u zavisnosti od raspoloživih resursa, bio u stanju da omogući pretraživanje *značajnog* segmenta celokupnog Web-a. Iako danas na Web-u, postoji veliki broj pretraživača opšte namene, visokih performansi, razlozi za samostalni razvoj pretraživača mogu biti višestruki, od kojih bi najznačajniji bili *privatnost* (najveći broj komercijalnih pretraživača arhivira i analizira korisničke upite, prvenstveno u cilju pružanja relevantnih oglasa uz search rezultate), ali i *istraživanje podataka* (komercijalni pretraživači ograničavaju načine na koje se može vršiti obrada njihovog dataset-a na osnovne information-retrieval operacije). U praksi, posebno u okruženjima koja koriste i razvijaju *data-mining* analizu podataka, ili u okviru akademskih projekata, nije moguć direktan pristup i proizvoljna obrada na korpusu dokumenata komercijalnih pretraživača. Zato je od interesa kreiranje *in-house* pretraživača koji bi obezbedio samostalno generisanje dataset-a, nad kojim se zatim može vršiti proizvoljan tip obrade. Pokazujemo da se u današnjim uslovima, korišćenjem relativno skromnih mrežnih i računarskih resursa, može razviti pretraživač, koji omogućava efikasan pristup relativno značajnom segmentu Web-a. Ova platforma je prikazana u okviru *poglavlja 2,* u vidu *studije slučaja*, i to tako što svaki od segmenata rešenja prikazujemo tek nakon teorijske razrade odgovarajućih algoritamskih aspekata.

Konačno, u okviru diplomskog rada, razvijamo i algoritam koji omogućava grupisanje rezultata pretraživanja, koristeći informacije o vezama između stranica relevantnih u odnosu na zadati upit. Razvijeno rešenje poredimo sa već postojećim i pokazujemo da pruža performanse koje su poredive sa poznatim rešenjima, dok istovremeno zahteva manje resursa za implementaciju od postojećih rešenja. U okviru *poglavlja 3*, prikazan je pregled navedenog algoritma, njegova analiza i detalji implementacije.

Na kraju, u *poglavlju 4*, dat je prikaz softverskog projekta, realizovanog u okviru diplomskog rada, koji prikazuje kompletnu implementaciju Web pretraživača, korišćenjem opisane open-source platforme za prikupljanje i indeksiranje podataka sa Web-a, sa samostalno razvijenom aplikacijom za pristup indeksiranim podacima i njihovo pretraživanje. Dodatno, u okviru same aplikacije, implementiran je navedeni algoritam za grupisanje rezultata pretraživanja, tako da celokupno rešenje predstavlja jednu zaokruženu celinu koja realizuje funkciju *clustering* web pretraživača opšte namene.

Osnovna namena diplomskog rada jeste uvod u problematiku razvoja i implementacije Web pretraživača. Data oblast predstavlja stalni izvor nekih od najinteresantnijih i najkompleksnijih praktičnih problema u okviru teorijskog računarstva. Opisana teorija predstavlja samo uvod i pregled najosnovnijih problema i načina na koje su oni rešeni. Pored toga, u okviru *poglavlja 6*, dat je detaljan opis literature korišćene u pripremi i izradi diplomskog rada. Navedena literatura predstavlja iscrpan izvor referenci za dalji nastavak istraživanja u okviru ove oblasti. Pored toga, u okviru rada, načinjen je i pokušaj razvoja inovativnog algoritma, koji se obraća *search result clustering* oblasti, koja je relativno mlada u odnosu na ostale oblasti u problematici Web pretraživanja, i u okviru koje još uvek ne postoje *de facto* algortmi za rešavanje određenih problema. Dodatna inspiracija za stalno istraživanje i razvoj u oblasti Web algoritama je činjenica da veći broj *state-of-the-art* algoritama, predstavlja *patente* i kao takvi se ne mogu koristiti u okviru sopstvenih pretraživača, bez odgovarajuće naknade. Ova činjenica predstavlja stalnu motivaciju za dalje istraživanje u oblasti algoritama, uopšte.

2. Osnovni problemi i algoritmi u razvoju i implementaciji Web pretraživača

2.1. Uvod

Osnovu World Wide Web-a (koji nazivamo i samo Web), predstavlja jednostavni, otvoreni, *klijent-server* dizajn : (1) *server* komunicira sa klijentima koristeći *http* protokol (lightweight, asinhroni protokol koji omogućava prenos raznih vrsta informacija - teksta, slike, medija - kao što su audio i video fajlovi, enkodovani u okviru *html* markup jezika), (2) *klijent* (browser), vrši parsiranje i grafički prikaz dobijenih html stranica.

Sam html jezik, u okviru definisanog markup-a, podrazumeva i definisanje tzv. *hiperlinkova (hyperlinks),* koji predstavljaju referencu jednog dokumenta ka drugom, od kojih je svaki jedinstveno definisan svojim URL-om (*uniform resource locator*). Na ovaj način, Web (u najužem smislu) možemo posmatrati kao *mrežu*, URL-ova, međusobno *povezanih* hiperlinkovima.

Najefikasniji način pristupa informacijama na Web-u jeste korišćenjem *Web pretraživača*. U opštem slučaju, pretraživač se sastoji iz tri dela :

- *crawler* , koji je zadužen za automatsko prikupljanje stranica sa Web-a i njihovo smeštanje u indeks pretraživača
- *indexer ,* koji obezbeđuje kreiranje odgovarajuće strukture (*inverted index*), koja omogućava efikasnu reprezentaciju i pretraživanje arhiviranih stranica.
- *query handler,* koji prihvata korisničke upite i odgovara na njih korišćenjem indeksa pretraživača

Osnovni princip koji je uzrokovao eksplozivni rast Web-a – decentralizovano i nekontrolisano publikovanje sadržaja – se pokazuje kao najveći izazov za Web pretraživače, u njihovom nastojanju da indeksiraju i arhiviraju sadržaj na Web-u. Budući da je publikovanje dosupno praktično svakom korisniku, web stranice pokazuju heterogenost u velikom broju ključnih informacionih aspekata (istina, neistina, kontradikcije…), što dovodi do problema određivanja stranica koje sadrže relevantne i objektivne informacije, vezano za zadatu temu.

Dodatni problem, predstavlja dimenzija i struktura samog Web-a. U praksi, čak nije ni moguće dati jednostavan odgovor na pitanje "koliko je veliki Web". Najrelevantniji odgovor na ovo pitanje bi se mogao dati posmatranjem veličine indeksa nekih od najvećih Web pretraživača. (krajem 1995, Altavista je prijavljivala veličinu indeksa od 30 miliona statičkih Web stranica, dok je na jesen 2005. godine, Google prijavljivao veličinu indeksa od oko 8 milijardi stranica). Pod ovim podrazumevamo samo *statičke* Web stranice (stranice čiji se sadržaj ne menja u periodu između dva pristupa).

Usled navedenog, svaki *Web pretraživač,* se pre ili kasnije (u zavisnosti od veličine), suočava sa nekim od sledećih problema:

- Brzina rasta Web-a je znatno veća nego što je postojeća tehnologija u stanju da indeksira.
- Veliki broj Web stranica ažuriraju svoj sadržaj veoma često, što zahteva da ih pretraživači češće posećuju, da bi imali ažurne kopije u indeksu
- Dinamičke stranice se ili sporo i teško indeksiraju ili mogu rezultovati u prekomerenom broju rezultata
- Veliki broj dinamički generisanih websajtova nije uopšte moguće indeksirati korišćenjem standardnih web pretraživača (ovi sajtovi čine tzv. "*nevidljivi web*")
- Secure stranice (https), mogu predstavljati problem crawler-ima jer im ne mogu pristupiti ili iz tehničkih razloga ili ih ne indeksiraju iz privatnosnih razloga
- Relevantnost stranica, pored toga što se teško određuje, može biti i dvosmislena, odnosno korisnik i pretraživač mogu imati različita "shvatanja" relevantnosti

- Korisnički upiti su ograničeni isključivo na *ključne reči*, što može rezultovati u velikom broju *lažnih pozitivnih* rezultata (stranice koje sadrže date ključne reči, ali nisu ono što je korisnik tražio).
- Količina *relevantnih* rezultata u odnosu na zadati *upit* jeste obično veća nego što je korisnik u mogućnosti da pregleda (ovo u praksi rezultuje u činjenici da se najčešće pregleda samo prvih par stranica sa rezultatima)
- Kvalitet sadržaja na Web-u varira, tako da su neophodne tehnike koje odvajaju "*signal od šuma*", onosno stranice *niskog* kvaliteta od stranica *visokog* kvaliteta
- Veliki broj stranica na Web-u sadrži validne informacije, ali nije struktuiran u skladu sa definisanim konvencijama
- Veliki broj stranica, predstavljaju *duplikate*, odnosno stranice istog ili sličnog sadržaja, ali sa različitim url-ovima. Neophodno je eliminisati dupliranje indeksiranja ovakvih stranica
- Neki pretraživači ne rankiraju stranice po relevantnosti, već po količini novca koju oglašavači plaćaju da se nađu među rezultatima sa najvećim *skorom*
- Na Web-u, mogu postojati grupe sajtova, kreirane isključivo sa ciljem manipulacije funkcijom rankiranja stranica (linkspam)
- Takođe, mogu postojati i drugi oblici spam-ovanja web pretraživača, kao što su *cloaking* (vraćanje različitog skupa stranica u zavisnosti da li pristupa korisnik ili crawler), *doorway pages (*stranice profilisane u odnosu na određeni upit, koje pri učitavanju vrše redirekciju na stranicu potpuno različitog sadržaja)

Za neke od navedenih problema, postoje efikasna rešenja, dok neki problemi (na primer, search engine spamming), danas predstavljaju izuzetno aktivnu oblast istraživanja.

U okviru ovog poglavlja, predstavljamo osnovne komponente jednog web pretraživača (*crawling*, *indexing*, *search*, *reprezentaciju*, *rankiranje* stranica i *clustering*). Razmatramo osnovne principe realizacije i implementacije delova pretraživača koji implementraju ove funkcije i pokušavamo da razmotrimo osnovne probleme, kao i najpoznatije *state-of-the-art* algoritme za rešavanje ovih problema.

Dodatno, u vidu studije slučaja, predstavljamo open source platformu, koja se zasniva na projektima *Lucene* (*full-text document indexing biblioteka*) i *Nutch* (*crawling*, *link representation* i *search*). Navedena platforma, predstavlja celokupnu implementaciju jednog web pretraživača i (budući da je izvorni kod slobodno dostupan), omogućava dalji razvoj, implementaciju i analizu novih algoritama.

2.2. Crawling

2.2.1. Osnove

Web crawling predstavlja proces *prikupljanja* stranica sa Web-a, radi njihovog indeksiranja u okviru Web pretraživača. Cilj crawling-a jeste prikupljanje što većeg broja Web stranica, zajedno sa informacijama o njihovoj međusobnoj povezanosti, u što kraćem vremenskom periodu i na najjefikasniji mogući način.

Funkcionalnosti koje crawler *mora* da obezbedi :

- *Robustnost* – Na Web-u, postoje serveri koji kreiranju tzv. „*spider traps*", zlonamerno generisane web stranice koje imaju za cilj onemogućavanje rada crawler-a, tako što nastoje da kreiraju beskonačnu petlju unutar domena u kojoj se crawler može „zaglaviti". Crawler-i moraju biti dizajnirani tako da budu otporni na ovaj vid „*zamki*" (posebno iz razloga što mogu postojati i greške ovoga tipa koje se ne javljaju namerno, već usled grešaka)
- *Pristojnost* – Web serveri imaju implicitne i eksplicitne polise po pitanju rate-a kojim ih crawler može posećivati. Ove polise se moraju poštovati.

Funkcionalnosti koje *bi* crawler *trebalo* da obezbedi :

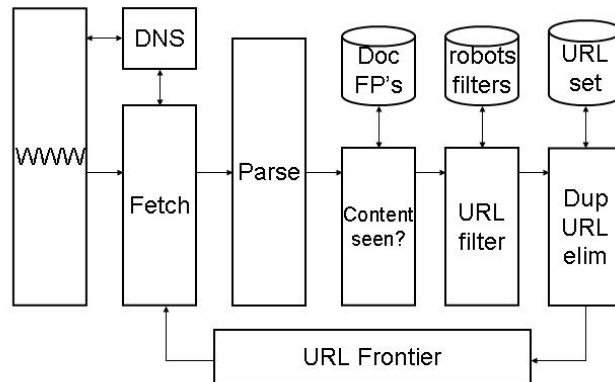
- *Distribuiranost* – Crawler bi trebao da bude u stanju da se izvršava distribuirano na više računara
- *Skalabilnost* – Arhitektura crawler-a bi trebalo da podržava povećavanje opsega crawl-rate-a dodavanjem dodatnih računara i bandwidth-a
- *Performanse i efikasnost* – Crawl sistem bi trebao da maksimalno utilizuje različite sistemske resurse, kao što su procesor, storage i mrežni bandwidth
- *Kvalitet* – Crawler bi trebao da bude orijentisan ka *fetch*-ovanju prvo "*korisnijih*" stranica
- *Ažurnost rezultata* – Crawler bi trebao da funkcioniše neprekidno, ponovno fetch-ujući sveže kopije prethodno fetch-ovanih stranica. U principu, crawler bi trebao da crawl-uje stranicu sa frekvencijom koja aproksimira frekvenciju promene date stranice
- *Proširivost* – crawler bi trebao da bude proširiv u više aspekata : da se snađe sa novim formatima podataka, novim fetch protokolima i drugo, što zahteva modularnu arhitekturu crawler-a.

Osnovna operacija svakog hypertext *crawler*-a je sledeća : Crawler počinje skupom od jednog ili više URL-ova (*seed set*), odabira URL iz ovog skupa i fetch-uje stranicu na tom URL-u. Nakon toga, parsira fetch-ovanu stranicu, da bi ekstrahovao tekst i linkove iz stranice. Tekst se predaje *indekseru*, dok se url-ovi dodaju u *URL frontier* (predstavlja strukturu zasnovanu na *redovima za čekanje*), koji se u svakom trenutku sastoji od URL-ova čije odgovarajuće stranice trebaju biti fetch-ovane od strane crawler-a. Ovaj postupak se ekvivalentno može posmatrati i kao obilazak Web Grafa.

Sam crawler se sastoji od više modula :

1. *URL frontier*, koji sadrži URL-ove koji će biti fetch-ovani u tekućem crawl-u
2. *DNS resolution* modul koji određuje adresu web servera na kome se nalazi URL koji fetch-ujemo (ovaj modul se može nalaziti i van modula, tj. može se koristiti sistemski servis za DNS rezoluciju)
3. *fetch* modul – koji retrieve-uje stranicu na datom URL-u
4. *parsing* modul – koji ekstrahuje skup linkova sa zadate web strane
5. modul koji određuje da li se ekstrahovani link već u URL frontier redu ili je nedavno fetch-ovan

Na *slici 1* je prikazana komplektna struktura crawler-a sa navedenim modulima.



Slika 1

Crawling se obično obavlja u vidu više thread-ova, od kojih svaki obavlja logički ciklus prikazan na *slici 1* . Ovi thread-ovi se mogu obavljati u okviru jednog procesa ili biti particionisani na više procesa koji se izvršavaju na distibuiranom sistemu.

Crawler *thread* počinje tako što uzima URL iz *frontier*-a i *fetch*-uje web stranu na zadatom url-u, (najčešće koristeći http protokol). *Fetch*-ovana stranica se smešta na pomoćni *storage*, gde se obavljaju operacije nad njom. Kao prvo, testira se da li stranica sa istim sadržajem već ne postoji na nekom drugom url (*eliminacija duplikata*). Dalje, stranica se parsira i ekstrahuju se tekst i linkovi. Tekst se prosleđuje indekseru, kao i informacije o linkovima koji se koriste kao dodatna informacija za rankiranje. Dodatno, svaki od linkova prolazi niz testova da bi se utvrdilo da li ide u *URL frontier*. Kao prvo, konsultuje se *URL filter*, u okviru koga se može definisati isključenje određenih domena (npr. svih *.com* url-ova – vrši crawl svih url-ova osim onih koji pripadaju .com domenu), takođe selekcija može biti i inverzna (crawluju se samo *.com* url-ovi).

Veliki broj host-ova na web-u definiše delove svojih web sajtova kao zabranjene za crawling, što se definiše *Robot Exclusion Protocol* standardom. Ovo se omogućuje postavljanjem fajla sa imenom robots.txt u korenu URL hijerarhije zadatog sajta. Crawler mora fetch-ovati *robots.txt* svakog web sajta da bi odredio da li treba da nastavi crawl-ovanje ostatka stranica i njihovo dodavanje u URL frontier.

Dat je primer robots.txt fajla koji definiše da je pristup segmentu sajta koji počinje sa */yoursite/temp*, zabranjen za sve crawler-e, osim za crawler pod imenom “*searchengine*” :

```
User-agent : *
Disallow : /yoursite/temp

User-agent : searchengine
Disallow :
```

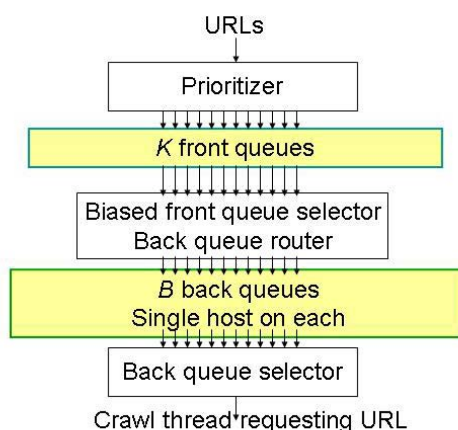
Konačno, proverava se da li već postoji crawl-ovan *duplikat* datog URL-a (korišćenjem algoritama za *duplicate detection*, opisanih u nastavku teksta). Ukoliko *duplikat* postoji, dati URL se ne dodaje u *URL frontier*. Kada se URL doda u *frontier*, dodeljuje mu se *prioritet*, na osnovu koga se određuje kada će biti uklonjen iz *frontier*-a (url-ovi sa većim prioritetima se ranije uklanjaju iz *frontier*-a, čime se ostvaruje njihovo češće fetch-ovanje).

Budući da crawl proces najvećim delom zavisi od samog *URL frontier-a*, u nastavku teksta opisujemo jedan mogući način njegove realizacije.

URL frontier

URL frontier održava listu URL-ova, preuređuje ih u zadatom redosledu i servira ih kad god crawler zahteva sledeći URL. Redosled kojim se URL-ovi serviraju diktiraju dva osnovna principa : *(1) ažurnost* - stranice koje se češće menjaju trebaju imati veće prioritete, radi češćeg crawling-a i (2) *učtivost* – potrebno je izbeći uzastopne upite jednom istom hostu u kratkom vremenskom intervalu. Ovo poslednje se često dešava upravo usled *lokalnosti reference* – veliki broj URL-ova linkuju na URL-ove na istom host-u. Kao rezultat toga, URL frontier implementiran kao jednostavni *priority queue* može rezultovati u burst-u fetch zahteva za jednim istim host-om. Najčešće korišćena heuristika je ubacivanje *razmaka* između uzastopnih zahteva ka istom host-u, koja je za red veličine veća od vremena koje je bilo potrebno za poslednji fetch sa tog host-a.

Na *slici 2* je prikazana implementacija URL frontier-a, koja poštuje navedene principe. Njeni ciljevi su obezbeđivanje : *(1)* da u svakom trenutku postoji samo jedna konekcija otvorena ka jednom host-u, *(2)* da postoji vreme čekanja od par sekundi između uzastopnih zahteva ka istom host-u, *(3)* da poštuje ograničenja nametnuta *robots exclusion* protokolom



slika 2

Dva osnovna modula *URL frontier*-a su skup *K front queue*-ova i skup *back queue*-ova, od kojih svi predstavljaju FIFO queue-ove (redove za čekanje). *Front queue* redovi služe za implementaciju prioritetizacije dok *back queue* redovi služe za implementaciju "*učtivosti*".

Kada se URL doda u *frontier*, *prioritizer* dodeljuje URL-u *prioritet* (čija vrednost može biti između *1* i *K* , i određuje se na osnovu prethodne istorije *fetch*-ovanja, uizimajući u obzir brzinu kojom se stranica menjala u prethodnim crawl-ovima). Svaki od *B back queue*-ova održava sledeće invarijante : *(1)* on je neprazan za vreme dok crawl radi, *(2)* sadrži samo URL-ove jednog host-a. Pomoćna tabela *T* se koristi za održavanje preslikavanja između host-ova i back queue-ova. Dodatno, održavamo *heap* sa po jednim ulazom za svaki *back queue*, u koji se ulazi na osnovu najranijeg vremena t_e ,u kome će host koji odgovara *queue*-u biti ponovo kontaktiran.

Crawler *thread* koji zahteva URL iz *frontier*-a ekstrahuje vrh *heap*-a i (ukoliko je potrebno), čeka odgovarajuće vreme t_e. Zatim, uzima URL u , sa vrha *back queue*-a *q* , koji odgovara vrhu heap-a i fetch-uje URL u. Nakon *fetch*-ovanja, pozivajući *thread* proverava da li je *queue* prazan. Ukoliko jeste, odabira *front queue* i sa vrha uzima URL v. Izbor front queue-a je takav da favorizuje *queue*-ove sa višim prioritetom, čime se obezbeđuje da se prioritetniji url-ovi češće fetch-uju. Posmatramo v da bi proverili da li već postoji *back queue* koji sadrži URL-ove sa ovog host-a. Ukoliko postoji , v se dodaje u taj *queue* i ponovo tražimo novog kandidata iz *front queue*-a, koji ubacujemo u prazan queue q. Ova procedura se ponavlja sve dok q ponovo ne postane neprazan, kada *thread* ubacuje ulaz u *heap* za queue q sa novim najskorijim vremenom

t_e , na osnovu karakteristika URL-a v , nakon čega se nastavlja procesiranje.

Broj *front queue*-ova, zajedno sa *polisom* dodeljivanja prioriteta i odabiranja *queue*-ova, definiše *prioritetizaciju* koju ugrađujemo u sistem. Broj *back queue*-ova definiše opseg u okviru koga možemo držati sve crawl thread-ove zauzete, istovremeno poštujući princip *učtivosti*.

Na ovaj način, realizovana je kompletna arhitektura crawler-a, koja omogućava rešavanje problema *svežine* stranica u indeksu (uvođenjem prioritetizacije), istovremeno implementirajući *učtivost* pri crawl-ing (uvođenjem *back queue*-ova).

Pored navedenog metoda, *svežina* stranica se može dodatno poboljšati, integracijom *crawl* procesa sa procesom rankiranja stranica (ovo je moguće za slučaj *online* algoritama za određivanje relevantnosti, kao što je *OPIC* algoritam)

Još jedan od faktora koji može značajno ubrzati sam *crawl* proces jeste eliminacija duplikata, koja je prikazana kao jedan od koraka u navedenoj arhitekturi *crawler*-a. U nastavku teksta, predstavljamo metode koje se mogu koristiti za efikasno određivanje *identičnih* i *približnih duplikata*, posmatrane stranice.

2.2.2. Eliminacija duplikata stranica

Web sadrži višestruke *kopije* mnogih dokumenata. Po nekim ocenama, 40% svih stranica na Web-u predstavljaju višestruke kopije, istog sadržaja. Pretraživači se trude da izbegnu indeksiranje duplikata, čime ostvaruju uštedu u zauzeću storage kapaciteta, kao i ubrzanje *crawl* procesa.

Najjednostavniji pristup detekciji duplikata je određivanje *fingerprint*-a svake web strane, koji predstavlja *digest* (*hash* proizvoljne dužine, npr. *64*. bita) sadržaja date strane. Na ovaj način, ukoliko dve stranice imaju identične *fingerprint*-e, stranice se proglašavaju za *duplikate*. Međutim, ovaj jednostavan pristup nije efikasan u detekciji *približnih duplikata* (stranice gotovo identičnog sadržaja, sa izuzetkom par karaktera, npr. datum, ime host-a...), kakvi se često javljaju na Web-u.

Shingling

Najjefikasnija tehnika za detekciju *približnih duplikata* jeste *shingling*. Ukoliko je data sekvenca tokena dokumenta d i pozitivni celi broj k , definišemo $k - shingle$ dokumenta d , kao skup svih uzastopnih sekvenci od k tokena dokumenta d . Na primer, posmatrajmo tekst : „*a rose is rose*“ . $4 - shingle$ -ovi u ovom tekstu su „*rose is a*“ , „*rose is a rose*“ i “*is a rose is*”. Primećujemo da se svaki od prva dva $4 - shingle$ -a, javlja dva puta u okviru teksta. U praksi, vrednost $k = 4$, se najčešće koristi u detekciji *približnih duplikata* web stranica. Intuitivno, dva dokumenta predstavljajju *približne dokumente*, ukoliko generišu gotovo iste skupove $shingle$ -ova.

Navedena intuitivna pretpostavka se može formalizovati na sledeći način :

Neka $S(d_i)$, označava skup *shingle*-ova u dokumentu d_i. Definišemo *Jaccard-ov koeficijent* (koji meri *stepen preklapanja* između dva skupa $S(d_1)$ i $S(d_2)$, kao :

$$J(S(d_1), S(d_2)) = \frac{|S(d_1) \cap S(d_2)|}{|S(d_1) \cup S(d_2)|}$$

Problem testiranja da li su dve stranice *približni duplikati*, možemo predstaviti kao problem

izračunavanja *Jaccard-ovog koeficijenta*. Ukoliko dobijena vrednost, prelazi prethodno definisanu granicu (npr. *0.9*) deklarišemo dokumente kao *približne duplikate* i jedan od njih eliminišemo iz procesa indeksiranja.

Za određivanje *Jaccard*-ovih koeficijenata, koristimo specijalan oblik *hesiranja* . Kao prvo, vršimo mapiranje svakog *shingle*-a u odgovarajuću *hash* vrednost (nad velikim prostorom vrednosti *hash-a*, npr. 64 bita). Za $i = 1,2$, neka je $H(d_i)$ odgovarajući skup 64-bitnih *hash* vrednosti koje su određene iz $S(d_i)$. Zatim, primenjujemo sledeću metod za određivanje parova dokumenata čiji skupovi $H()$, imaju veliki *Jaccard*-ov koeficijent : Neka je π, slučajna permutacija koja predstavlja preslikavanje iz skupa 64-bitnih celih brojeva na skup 64-bitnim celih brojeva.
Označimo sa $\Pi(d_i)$, skup permutovanih *hash* vrednosti u $H(d_i)$, pri čemu za svako $h \in H(d_i)$, postoji odgovarajuća vrednost $\pi(h) \in \Pi(d_i)$.

Neka je x_i^π, najmanji celi broj u $\Pi(d_i)$. Tada imamo da važi: $J(\Pi(d_1), \Pi(d_2)) = \Pr\left[x_1^\pi = x_2^\pi\right]$.

Prema tome, možemo izvesti jednostavan probabilistički test za *Jaccard*-ov koeficijent *shingle* skupova : poredimo vrednosti za x_i^π, određene iz različitih dokumenata. Ukoliko je određeni par identičan, dobijamo *kandidate* za *približne duplikate*. Dati test ponavljamo nezavisno za veliki broj slučajnih permutacija π (u praksi se vrednost od 200 ponavljanja pokazuje kao najoptimalnija). Nazovimo skup datih 200 vrednosti x_i^π, *skicom* $\psi(d_i)$*,* dokumenta d_i. Na osnovu poznate *skice*, moguće je aproksimirati *Jaccard*-ov koeficijent kao $\frac{|\psi_i \cap \psi_j|}{200}$. Ukoliko je dobijena vrednost veća od definisanog *praga* , deklarišemo da su d_i i d_j, *slični*.

Postavlja se pitanje, efikasnog izračunavanja opisane vrednosti aproksimacije *Jaccard*-ovog koeficijenta, za sve parove i i j . Poređenje svakog dokumenta sa svakim je prilično neefikasno, posebno u slučaju velikog broja dokumenata.

Kao prvo, koristimo *fingerprinting* , sa uklanjanje svih osim jedne kopije potpuno *identičnih* dokumenata. Nakon toga, koristimo *union-find* algoritam za kreiranje *grupa* koje sadrže dokumente koji su slični. Određujemo *skice* za svaki od dokumenata i izračunavamo broj *shingle*-ova koji su zajednički za svaki par dokumenata čije *skice,* imaju zajedničke elemente. Polazimo od liste sortiranih parova $\left\langle x_i^\pi, d_j \right\rangle$ i za svako x_i^π, generišemo sve parove i, j , za koje se x_i^π, nalazi u obe njihove *skice* . Nakon toga, vršimo njihovo spajanje u brojače za svako i, j , kod kojih se javljaju *skice* sa nenultim preklapanjem. Nakon ovog postupka, možemo odrediti koji parovi i, j , imaju *skice* koje se *jako* preklapaju. Konačno, određivanje traženog *Jaccard*-ovog koeficijenta se može dodatno ubrzati tako što eliminišemo iz razmatranja one parove i, j , čije skice imaju *mali* broj zajedničkih *shingle*-ova. Ovo se može obaviti tako što se sortira skup vrednosti x_i^π, u okviru svake *skice*, na koju ponovo primenjujemo *shingle* operaciju, kojom dobijamo skup *super-shingle*-ova za svaki dokument. Ukoliko dva dokumenta imaju zajednički *super-shingle*, vršimo određivanje vrednosti aproksimacije *Jaccard*-ovog koeficijenta, dok u slučaju da nemaju zajednički *super-shingle*, pretpostavljamo da je vrednost koeficijenta znatno manja od definisanog *praga* i kao takve ih ne uzimamo u dalje razmatranje.

2.2.3. Case Study : Nutch crawler

Nutch projekat (http://lucene.apache.org/nutch/), predstavlja skup open source komponenti, razvijenih na jeziku Java, sa ciljem implementacije celokupnog web pretraživača, na platformi otvorenog koda.

Crawler (*Nutch crawler*), razvijen u okviru projekta, pored osnovnih funkcionalnosti, podrazumeva i *modularnu arhitekturu*, koja omogućava proširivanje samog crawler-a, razvojem *plugin*-ova (na primer za parsiranje i fetch nestandardnih formata, preprocesiranje i slično..)

Osnovu crawler sistema, predstavlja *crawl* aplikacija i skup konfiguracionih fajlova u okviru kojih se definiše ponašanje i rad samog crawl procesa.

Osnovna konfiguracija definiše se u okviru fajlova *nutch-default.xml* (koji definiše globalne parametre vezane za crawl uopšte) i *nutch-site.xml,* u okviru koga se definišu osnovni parametri vezani za konkretnu implementaciju crawler-a (parametri koji se pojavljuju u zaglavlju upita koje šalje crawler – ime crawler-a, opis, url, kontakt email i sl.). Takođe, od posebnog interesa je fajl *crawl-urlfilter.txt*, u okviru koga možemo, korišćenjem regularnih izraza definisati strukturu url-ova čiji sadržaj prihvatamo ili odbijamo.

Sam crawler se može izuzetno detaljno konfigurisati, u okviru *nutch-default.xml* fajla. Neke od najznačajnijih konfiguracionih opcija su :

- *http.content.limit* – maksimalna veličina sadržaja http stranice koja se fetch-uje
- *http.redirect.max* – maksimalan broj redirekcija koje crawler prati da bi fetch-ovao stranicu
- *fetcher.server.delay* – dužina pauze koju crawler pravi između dva uzastopna zahteva ka istom host-u
- *fetcher.threads.fetch* – broj fetcher thread-ova koji se izvršavaju
- *fetcher.threads.per.host.by.ip* – binarna vrednost koja određuje da li se host-ovi broje po ip adresi ili po hostname-u (tj. da li se podrazumevaju *virtuelni* ili *stvarni* host-ovi)

Dva osnovna načina na koji se može obavljati crawl su *Intranet* (*single-shot*) i *Internet* (*multiple-segment*) crawl. *Intranet* crawl, podrazumeva definisanje skupa početnih url-ova (u okviru proizvoljnog fajla koji se predaje kao parametar pri pozivanju *crawl* aplikacije), koji predstavlja početni skup url-ova u *fetch queue*-u. Ceo crawl proces se obavlja kao jedna celina (*single-shot*), odnosno, tek nakon završetka celokupnog procesa (koji se definiše maksimalnom dubinom do koje se vrši pretraživanje, kao i maksimalnim brojem stranica na svakom od nivoa) vrši se indeksiranje stranica. Ovaj pristup je neefikasan u slučaju *crawl*-a velikih dimenzija, budući da *crawl*-ovani sadržaj postaje dostupan tek nakon završetka celokupne procedure. Ukoliko se želi omogućiti pretraživanje sadržaja paralelno sa njihovim *fetch*-ovanjem, koristimo *Internet* crawl, kod koga naizmenično obavljamo cikluse generisanja *fetch liste*, *fetch*-ovana i indeksiranja, tako da po završetku svakog ciklusa, novodobijeni sadržaj postaje dostupan u okviru indeksa i može se pretraživati.

Sam crawler imlementira arhitekturu slično prethodno navedenoj, uz korišćenje back queue-a za implementaciju *pristojnosti* u radu, koja se može proizvoljno definisati u okviru konfiguracionog fajla. Takođe, implementiran je i mehanizam *prioriteta*, koji obezbeđuje prioritet pri *fetch*-ovanju, relevantnijem sadržaju, pri čemu se relevantnost svake od stranica, određuje *online*, korišćenjem OPIC algoritma.

2.3. *Indeksiranje*

2.3.1. Osnove

Invertovani indeks, predstavlja osnovnu struktura podataka koja se korsiti u okviru *Web pretraživača* i *information retrieval* softvera uopšte. Pod invertovanim indeksom, podrazumevamo *indeks* struktru koja sadrži *presikavanja* između *ključnih reči* i njihovih lokacija u *skupu dokumenata*, i korišćenjem koje se omogućava efikasno pretraživanje posmatranog skupa.

Postoje dve osnovne varijante realizacije invertovanih indeksa : *na nivou zapisa* (*record level inverted index*), koji se još naziva *inverted file index* i koji sadrži listu referenci na *dokument* za svaku reč koja se u okviru njega javlja makar jedanput i *na nivou reči* (*word level inverted index*) koji dodatno u okviru indeksa, sadrži i informacije o poziciji svakog javljanja date reči u okviru odgovarajućeg dokumenta.

Primer invertovanog indeksa :

Ukoliko su dati dokumenti : $T_0 =$ `"it is what it is"`, $T_1 =$ `"what is it"` and $T_2 =$ `"it is a banana"`, invertovani *fajl indeks* ima sledeći oblik :

```
"a":      {2}
"banana": {2}
"is":     {0, 1, 2}
"it":     {0, 1, 2}
"what":   {0, 1}
```

Pretraživanje za reči “what” , “is” and “it” daje kao odgovor skup: $\{0,1\} \cap \{0,1,2\} \cap \{0,1,2\} = \{0,1\}$.

Ukoliko posmatramo iste dokumente, *full inverted indeks* (koji sadrži sadrži i pozicije u okviru dokumenta), bi imao sledeći oblik :

```
"a":      {(2, 2)}
"banana": {(2, 3)}
"is":     {(0, 1), (0, 4), (1, 1), (2, 1)}
"it":     {(0, 0), (0, 3), (1, 2), (2, 0)}
"what":   {(0, 2), (1, 0)}
```

Ukoliko vršimo pretraživanje za frazu “*what is it*”, dobijamo da se sve reči iz posmatrane fraze javljaju u dokumentima 0 i 1. Međutim, na osnovu brojača pozicija, možemo zaključiti da se navedne reči nalaze u zadatom poretku, samo u dokumentu 1.

Kod Web pretraživača, paralelno sa crawl operacijom, obavlja se i operacija indeksiranja fetch-ovanih stranica, upravo korišćenjem strukture *invertovanog indeksa*. Za zadati korpus dokumenata, prolazi se kroz svaki dokument i za svaki token, vrši se njegovo ažuriranje u okviru indeksa : ukoliko već postoji, dodaje se tekući dokument kao lokacija u kojoj se nalazi, dok ukoliko ne postoji, kreira se novi *ulaz* u indeksu, za zadati token i tekući dokument se postavlja za prvu lokaciju u kojoj se navedeni token nalazi. Nakon završetka ovog procesa, sve operacije pretraživanja (koje su oblika : “naći sve stranice na web-u u kojima se nalaze navedeni tokeni”), obavljaju se preko dobijenog invertovanog indeksa. Budući da je u pitanju *hash* struktura, pristup polju u indeksu koje odgovara zadatom tokenu se ostvaruje teorijski u $O(1)$. U praksi, usled ogromnih dimenzija indeksa kod web pretraživača, operacija pristupa indeksu je nešto sporija (budući da nije moguće istovremeno držati celokupan ideks u operativnoj memoriji), ali i dalje daleko brža nego u slučaju korišćenja sekvencijalnog pristupa.

2.3.2. Metode estimacije veličine indeksa

Iako je relativno jednostavno odrediti *fizičku* veličinu samog indeksa (u smislu broja dokumenata koji je indeksiran), u slučaju Web pretraživača, tako definisana veličina je od *malog* značaja, budući da ne pruža informaciju o količini informacija koje se nalaze u indeksu. U praksi je od interesa procenjivanje veličine segmenta Web-a koji indeksira jedan pretraživač. Ovo se pokazuje kao veoma teško za određivanje, budući da u principu postoji praktično beskonačan broj dinamičkih web stranica. (npr. "soft 404 errors" stranice koje Web server automatski generiše). Prema tome, pitanje određivanja veličine indeksa pretraživača, možemo jedino da reformulišemo kao pitanje određivanja relativne veličine indeksa dva zadata pretraživača.

Ipak i određivanje ove veličine se takođe pokazuje kao *teško*, iz sledećih razloga :

1. Pretraživači mogu da vrate web strane čiji sadržaj nisu (potpuno ili čak ni parcijalno) indeksirali. U principu, pretraživači indeksiraju samo prvih par hiljada reči na web strani. U nekim slučajevima, pretraživač je *svestan* stranice na koju linkuju stranice koje on indeksira, ali sama stranica nije indeksirana. Ovo ipak omogućava da pretraživač vrati smislene rezultate vezane za p u search results
2. Pretraživači u principu, organizuju indekse u vidu više *particija*, od kojih se ne razmatraju sve u svakom pretraživanju. Na primer, web strana, duboko u okviru web sajta može biti indeksirana ali se ne razmatra u slučaju generalnog web search-a, dok se razmatra u slučaju specificnih upita vezanih za posmatrani web sajt

Prema tome, pretraživači mogu sadržati višestruke klase indeksiranih strana, tako da ne postoji jedinstvena mera njihovih celokupnih indeksa. Kao odgovor na navedene probleme, razvijen je veliki broj tehnika koje omogućavaju procenu *odnosa* veličina indeksa dva pretraživača, E_1 i E_2. Osnovna hipoteza od koje se polazi jeste da svaki pretraživač indeksira onaj segment Web-a, koji se odabira *nezavisno* i *slučajno* sa uniformnom raspodelom. Uvedene pretpostavke : da postoji *konačna* veličina web-a iz koje pretraživač odabira podskup i da pretraživač odabira iz *slučajno odabranog podskupa* – su daleko od realnosti. Ipak, navedene pretpostavke predstavljaju dobru osnovu i omogućuju primenu klasične tehnike estimacije poznate kao *capture-recapture method:*

Odabiramo *slučajnu* stranicu iz E_1 i ispitujemo da li se nalazi i u indeksu pretraživača E_2, istovremeno ispitujući da li se *slučajno odabrana* stranica iz E_2 nalazi u E_1. Na osnovu navedene procedure, dobijamo vrednosti x i y takve da možemo oceniti se deo x , stranica iz E_1, koji se i u E_2, kao i daje deo y, stranica iz E_2, koji se takođe nalazi i u E_1.
Prema tome, ukoliko sa $|E_i|$ označimo veličinu indeksa, pretraživača E_i, imamo:

$$x|E_1| = y|E_2| \Rightarrow \frac{|E_1|}{|E_2|} \approx \frac{y}{x}$$

Pod uslovom da je pretpostavka o *nezavisnosti* i *uniformnosti* E_1 i E_2 tačna i da je proces slučajnog odabiranja, *nepristrasan*, ova jednačina daje objektivni estimator za odnos $\frac{|E_1|}{|E_2|}$.
Razlikujemo dva scenarija : (1) merenje obavlja uz postojanje *direktnog* pristupa indeksima oba pretraživača i (2) merenje se obavlja *indirektnim* pristupom indeksu (preko javnog interfejsa pretraživača).

Jedan od načina na koji bi mogli implementirati proces *slučajnog odabiranja*, jeste generisanje slučajne stranice iz skupa svih stranica na Web-a i testiranje da li data stranica postoji u svakom od pretraživača. Nazalost, odabiranje *slučajne* Web stranice predstavlja težak problem. Neki od mogućih pristupa ovom problemu su :

1. *Slučajno pretraživanje* : Polazimo od skupa svih upita prethodno postavljenim određenom pretraživaču. Odabiramo slučajno jedan od ovih upita i šaljemo ga pretraživaču E_1 i odabiramo slučajno jedan od dobijenih rezultata. Osnovni nedostatak ovog pristupa da početni skup upita nije statistički nezavisan, tako da i dobijeni dokumenti nisu potpuno uniformni.
2. *Slučajne IP adrese* : Generišemo skup slučajnih IP adresa i šaljemo zahtev Web serveru na svakoj od generisanih adresa i prikupljamo sve stranice. Osnovni problem ovog pristupa je činjenica da veći broj host-ova može deliti jednu istu IP adresu (virtual hosts).
3. *Slučajno lutanje* : Ukoliko pretpostavimo da se Web može predstaviti kao jako povezani graf, moguće je obaviti *slučajno lutanje* (*random walk*) na grafu, polazeći od proizvoljne web stranice, koji konvergira ka stacionarnoj raspodeli, iz koje možemo odrediti verovatnoću svakog čvora i obaviti uniformno odabiranje, tako što odabiramo stranice inverzno frekvenciji njihovog pojavljivanja u okviru walk-a. Problem kod ovog pristupa se sastoji u tome što se Web ne može predstaviti *jako* povezanim grafom (već se sastoji od većeg broja, *slabo* povezanih komponenti), kao i činjenica da vreme konvergencije *walk*-a, može biti proizvoljno dugo.
4. *Slučajni upiti* : Pretpostavljamo da je moguće dobiti *slučajnu* stranicu iz indeksa, postavljanjem *slučajnog upita* pretraživaču. Međutim, jednostavni pristup generisanja niza slučajne dužine, koji se sastoji od slučajno izabranih reči iz rečnika, se pokazuje kao neefikasan, budući da se sve reči ne javljaju sa jednakom verovatnoćom, usled čega koristimo tzv. *Web rečnik*, koji se može dobiti *crawl*-ovanjem određenog segmenta Web-a i indeksiranjem reči svakog dobijenog dokumenta. Iz dobijenog rečnika, kreiramo upit koji se sastoji od dve ili više slučajno izabrane *reči*. Verovatnoća događaja da se stranica nalazi u skupu rezultata tako kreiranog upita, indukuje raspodelu nad svim stranicama u uniji dva pretraživača. Nakon toga, ocenjujemo vrednost $|E_1|/|E_2|$, tako što uzimamo odnos odgovarajućih indukovanih raspodela. U praksi ovo možemo obaviti na sledeći način : Kreiramo slučajni upit na zadati način i postavljamo ga E_1, nakon čega odabiramo stranicu p, *slučajno* iz prvih 100 dobijenih rezultata. Zatim, testiramo da li se p, nalazi u E_2, na sledeći način : odabiramo 6-8 tokena koji se javljaju sa malom učestanošću u p i koristimo ih kao upit pretraživaću E_2. Iako predstavlja efinasniji metod od prethodno navedenih i ovaj pristup ima određene probleme, budući da je *pristrasan* ka dužim dokumentima, kao i da sama procedura odabiranja zavisi od *pristrasnosti* algoritma za rankiranje, koji određuje koje će se stranice javiti među prvih 100 najboljih rezultata.

2.3.3. Case Study : Lucene indexer

Lucene projekat (http://lucene.apache.org/), predstavlja open source *information retrieval* biblioteku, implementiranu na jeziku Java, koja omogućuje izuzetno efikasno indeksiranje i pretraživanje, uz mogućnost integracije sa drugim aplikacijama.

U okviru Lucene projekta, definisan je API koji omogućava jednostavno kreiranje i ažuriranje *invertovanog indeksa.* Da bi se dokument dodao u intvertovani indeks, prethodno se vrši njegovo skeniranje i dobija se lista podataka, koji opisuju frekvenciju ponavljanja svake reči u okviru dokumenta i sastoji se od : reči, ID dokumenta i lokacije ili frekvencije reči unutar dokumenta. Sam indeks, možemo posmatrati kao skup parova oblika <*word, document-id*>*,* sortiranih po *word* polju. Za razliku od standardnih načina reprezentacije indeksa (B-stabla, koja omogućavaju upis i pretraživanje u $O(\log n)$), *Lucene* koristi nešto drugačiji pristup, koji umesto postojanja jednog indeksa, podrazumeva postojanje više *indeks segmenata*, čije se spajanje vrši periodično. Za svaki novi indeksirani dokument, Lucene kreira novi *segment*, pri čemu ubrzo vrši spajanje malih sa velikim segmentima – što održava mali ukupan broj segmenata. Radi optimizacije operacije pretraživanja, Lucene vrši spajanje svih segmenata u jedan, pri čemu se, radi izbegavanja mogućih konflikata konkurentnih *čitanja* i *upisa* u dati indeks, segmenti nikad ne modifikuju na mestu na kome se nalaze u indeksu, već se samo kreiraju novi. Pri spajanju, u novokreirani indeks, Lucene upisuje najnoviji indeks, dok vrši brisanje starijih, nakon što više nema aktivnih procesa koji obavljaju *čitanje* datog indeksa. Ovaj pristup se jako dobro skalira u odnosu na veličinu indeksa, pri čemu omogućava fleksibilnost u proizvoljnom definisanju odnosa između brzine *indeksiranja* i brzine *pretraživanja*.

Lucene *indeks segment* se sastoji od više fajlova :

- *dictionary index* – koji sadrži po jedan ulaz za svakih 100 ulaza u *rečniku (dictionary)*
- *dictionary* – koji sadrži po jedan ulaz sa svaku reč koja se javlja u okviru dokumenta
- *postings file* – koji sadrži po jedan ulaz za svaki par oblika <*word, document-id*>

Budući da Lucene nikad ne vrši ažuriranje segmenata na mestu gde su kreirani, oni mogu biti smešteni u običnim fajlovima, umesto u *B-tree* strukturama, koje su komplikovanije za implementaciju. Radi bržeg pristupa, koristi se *dictionary index,* koji sadrži *offset*-e *dictionary* fajla, na kojima se nalaze *offset*-i ka odgovarajućim ulazima *postigns* fajla.

Rankiranje

Pored efikasne realizacije operacija *indeksiranja* i *pretraživanja*, zadatak Lucene biblioteke je i *rankiranje* dokumenata, koje definiše redosled po kome su raspoređeni dokumenti koji odgovaraju zadatom upitu. Rankiranje, koje definiše relevantnost zadatog dokumenta, u odnosu na zadati upit, u okviru Lucene-a, je realizovano kao kombinacija standardnog information-retrieval *Vector Space modela (VSM)* i *Boolean modela.* Osnovna ideja *VSM* pristupa je dokument relevatniji u odnosu na zadati upit, ukoliko se *tokeni* tog upita, češće javljaju u posmatranom dokumentu u odnosu na sve ostale dokumente u indeksu. *Boolean model* se dodatno koristi za sužavanje skupa dokumenata koji se razmatraju, na osnovu logičkih pravila (*AND*,*OR*,*NOT*…), specificiranih u okviru samog upita. U nastavku, opisujemo neke od detalja implementacije *scoring* i *ranking* operacija.

U okviru *Lucene* biblioteke, objekti koje *ocenjujemo* (*scoring*), se predstavljaju *Document* klasom. *Document* predstavlja kolekciju *Field* objekata, od kojih svaki sadrži semantičku informaciju o načinu na koji se kreira i arhivira (*tokenized, untokenized,raw data, compressed…*). Sama *scoring* operacija se obavlja nad *Field* objektima, nakon čega se rezultati ove operacije koriste za određivanje odgovarajućih Dokumenata. Ovo je važno, budući da se može desiti, da dva dokumenta, koji sadrže potpuno identični sadržaj, ali različito raspoređen u okviru *Field* objekata, mogu rezultovati u različitim *score*-ovima u odnosu na isti upit.

Dodatno, na rankiranje rezultata se može uticati korišćenjem “*boosting*” operacije, kojom se direktno utiče na vrednost dobijenog *score*-a, i to na jedan od sledećih načina :

- *Document level boosting* – u toku indeksiranja, pozivanjem *document.setBoost()* metode, pre nego što se dokument doda u indeks
- *Document's Field level boosting* – u toku indeksiranja, pozivanjem *field.setBoost()* metode, pre dodavanja *Field* objekta u dokument
- *Query Level boosting* – u toku pretraživanja, postavljanjem boost vrednosti u okviru query objekta, korišćenjem *Query.setBoost()* metode

Boost vrednosti, zadate u vreme indeksiranja se, radi efikasnosti, preprocesiraju i upisuju (za vreme upisivanja dokumenta) u vidu jednog bajta, na sledeći način : Za svako polje dokumenta, vrši se množenje svih njegovih *boost* vrednosti, nakon čega se dobijeni proizvod množi *boost* vrednošću celog dokumenta, kao i sa “*field length norm*” vrednošću, koja predstavlja dužinu datog polja u posmatranom dokumentu (čime se realizuje potenciranje *kraćih* polja). Dobijeni rezultat se dekoduje kao jedan bajt (uz mogući gubitak preciznosti) i smešta u tekući direktorijum. Samo izračunavanje *field length norm* vrednosti, obavlja *Similarity* objekat, u toku izvršavanja operacije indeksiranja.

Određivanje *score* funkcije za dokument d, u odnosu na upit q, se zasniva na ideji *kosinusne udaljenosti* i *tačkastog proizvoda* između dokumenta i vektora upita (*query vectors*) u okriru Vector Space modela. Dokument čiji je vektor *bliži* vektoru upita (u smislu kosinusne norme), dobija veću vrednost *score* funkcije. Sama *score* vrednost, u odnosu na upit q, za zadati dokument d, se određuje kao :

$$score(q,d) = coord(q,d) \cdot queryNorm(q) \cdot \sum_{t \in q} \left(tf(t \in d) \cdot idf(t)^2 \cdot t.getBoost() \cdot norm(t,d) \right)$$

pri čemu :

- $tf(t \in d)$ (*term frequency*), predstavlja broj puta koliko se *term* t, pojavljuje u dokumentu d, koji posmatramo. Dokumenti u kojima se zadati term, javlja više puta, dobijaju veću tf vrednost. Određivanje ove vrednosti se normalizuje kao :

$$tf(t \in d) = frekvencija^{1/2}$$

- $idf(t)$ (*inverse document frequency*), predstavlja inverznu vrednost broja dokumenata u kojima se term t pojavljuje (*docFreq*). Ovo utiče na to da *term*-ovi koji se ređe javljaju, imaju veći uticaj na kompletni skor. Vrednost za idf, određujemo kao :

$$idf(t) = 1 + \log\left(\frac{numDocs}{docFreq + 1}\right)$$

- $coord(q,d)$**,** predstavlja faktor koji ukazuje na to koliko se tokena iz zadatog upita, nalazi u zadatom dokumentu. Dokument koji sadrži veći broj tokena, dobija veći skor od dokumenta koji sadrži manji broj. Ova vrednost se određuje za vreme pretraživanja, na osnovu informacija iz zadatog upita
- $queryNorm(q)$, predstavlja normalizacioni faktor, koji ne utiče na rankiranje samog dokumenta, već samo omogućava poređenje score-ova iz različitih upita. Ova vrednost se takođe određuje u vreme pretraživanja, kao :

$$queryNorm(q) = queryNorm(sumOfSquaredWeights) = \frac{1}{sumOfSquaredWeights^{1/2}}$$

Funkciju sume kvadriranih težina ($sumOfSquaredWeights$) , određuje *Weigth* objekat zadatog upita. Na primer, *boolean query,* određuje ovu vrednost kao :

$$sumOfSquaredWeights = q.getBoost()^2 \cdot \sum_{t \in q} \left(idf(t) \cdot t.getBoost()\right)^2$$

$t.getBoost()$, predstavlja boost vrednost koja se određuje u vreme pretraživanja, za zadati term t , u upitu q , ili se postavlja pozivima *setBoost()* metode.

- $norm(t,d)$, predstavlja enkapsulaciju više *boost* i *length* faktora :
    - *Document boost* – koji se postavlja pozivanjem *doc.setBoost()* metode, pre dodavanja dokumenta u indeks
    - *Field boost* – pozivanjem *field.setBoost(),* pre dodavanja polja u dokument
    - *lengthNorm(field)* – koji se određuje kada se dokument dodaje u indeks, u skladu sa brojem tokena ovog polja u dokumentu, tako da kraća polja imaju veći doprinos ukupnom skoru.

Kada se dokument dodaje u indeks, vrši se množenje svih navedenih faktora. Ukoliko dokument ima više polja istog imena, vrši se množenje svih odgovarajućih *boost* vrednosti :

$$norm(t,d) = doc.getBoost() \cdot lengthNorm(field) \cdot \prod_{f \in d} f.getBoost()$$

Konačno, celokupan proces ocenjivanja i rankiranja dokumenata u okviru Lucene biblioteke, možemo predstaviti na sledeći način :

- Nakon zadatog upita, kreirani *Query* objekat se prosleđuje *Searcher* objektu
- U okviru *Searcher*, kreira se *Hits* objekat, koji čuva reference na četiri važna objekta :
    - *Weight* objekat zatatog *Query*-ja
    - *Searcher* objekat koji je inicirao poziv
    - *Filter* objekat , koji se koristi za limitiranje skupa rezultata
    - *Sort* objekat, kojim se definiše željeni način sortiranja rezultata
- Počinje se proces identifikacije dokumenata koji odgovaraju zadatom upitu, pozivanjem *getMoreDocs* metode. Nakon toga, pozivamo metodu *Searcher* objekta, koja vraća *TopDoc* objekat, koji sadrži internu kolekciju rezultata pretraživanja. *Searcher* kreira *TopDocCollector* i predaje ga, zajedno sa *Weight i Filter*, sledećoj *search* metodi.
- Ukoliko se koristi *Filter*, vrši se odlučivanje dokumenata koje uključujemo u proces rankiranja. U suprotnom, odmah pozivamo *score* metodu, *Scorer* objekta.
- U okviru *score* metode na osnovu *HitCollector* objekta, vršimo ocenjivanje svake od stranica, korišćenjem opisanog algoritma. *Scorer* koji koristimo, zavisi od tipa *Query* objekta (u praksi najčešće koristimo *BooleanScorer2*)
- Vršimo inicijalizaciju *Coordinator* objekta, koji se koristi za određivanje *coord()* faktora. Konačno, koristeći dobijeno *Scorer,* ulazimo u petlju, koja se zasniva na *Scorer#next()* metodi, pri čemu *next()* metoda vrši pomeraj na sledeći dokument koji odgovara upitu. Na primer, u slučaju jednostavnog *OR* upita, koristi se DisjunctionSum Scorer, koji vrši kombinaciju *score* vrednosti, dobijene za svaki od *tokena* koji su u OR relaciji, u okviru zadatog upita.

2.4. Rankiranje

2.4.1. Osnove

Navedeni mehanizmi određivanja relevantnosti stranica u okviru indeksa, koje odgovaraju zadatom upitu, se pokazuju kao izuzetno efikasni u okviru klasičnog *information retrieval* domena. Međutim, u okviru problematike Web pretraživača, dati metodi se često pokazuju kao nedovoljno efikasni, upravo usled specifičnosti sadržaja na Web-u, koja se ogleda u postojanju *strukture*, indukovane hiperlinkovima između stranica. Usled ovoga, celokupan informacioni sadržaj o stranicama na Web-u se ne iscrpljuje isključivo posmatranjem *sadržaja* samih stranica, već zahteva i posmatranje informacija o *odnosima* između stranica.

U cilju efikasne reprezentacije navedene *strukture*, Web se najčešće pradstavlja, u vidu *Web grafa*, $W(P,L)$, koji predstavlja *graf*, kod koga P, predstavlja skup stranica na Web-u, dok L, predstavlja skup svih hiperlinkova između stranica. Web Graf predstavlja *slabo povezani*, usmereni graf. Pod pojmom *slabe povezanosti*, podrazumevamo da postoje parovi stranica koji nisu međusobno *dostižni* (od jedne stranice se ne može stići do druge, praćenjem hiperlinkova). Prosečan broj *inlink*-ova (linkova koji ukazuju na datu stranicu) se po velikom broju studija, kreće u rasponu od *8* do *15*. Postoje empirijski dokazi da da ovi linkovi nisu slučajno raspodeljeni. Međutim broj linkova ne prati *Poisson*-ovu raspodelu, što bi se očekivalo u slučaju da svaka Web stranica odabira odredišta linkova sa slučajnom raspodelom. U praksi, raspodela koja se najčešće pokazuje u istraživanjima jeste *power-law* raspodela, koja podrazumeva da je broj stranica sa ulaznim stepenom *i* (brojem linkova koji ukazuju na stranicu) , proporcionalan $\frac{1}{i^{\alpha}}$, pri čemu se empirijski pokazuje vrednost $\alpha \approx 2.1$. (specijalan slučaj *power-law* raspodele, predstavlja *Zipf-ova raspodela*, kod koje je $\alpha = 1$ i koja se često koristi za opisivanje raspodele reči u tekstu).

Opisana struktura (Web Graph), omogućava efikasnu reprezentaciju informacija koje se sastoje u *odnosima* između stranica na Web-u, i analizom date strukture (*link analysis*), moguće je doći do niza značajnih informacija o samim stranicama, među kojima je najznačajnija informacija o *relevantnosti* zadatih stranica. U nastavku predstavljamo nekoliko najkorišćenijih metoda za određivanje relevatnosti stranica, korišćenjem analize strukture Web grafa.

2.4.2. PageRank

PageRank algoritam (koji se koristi u okviru *Google* pretraživača), ima za cilj dodeljivanje numeričke vrednosti u rasponu 0 do 1 (koja se naziva *pagerank*), svakom čvoru u Web Grafu, koja ukazuje na njegovu relevantost, pri čemu data vrednost prvenstveno zavisi od same link strukture Web Grafa.

Ukupna PageRank vrednost za svaku stranicu, se određuje za zadati upit, kao kombinacija *pagerank* vrednosti, određene strukturom Web Grafa i *score* vrednosti samog dokumenta, koja je određena samim sadržajem stranice (na način sličan opisanom određivanju *score* funkcije stranica u *Lucene* indeksu). Konačno, na osnovu dobijenih PageRank vrednosti, vrši se rankiranje liste rezultata dobijene u odnosu na zadati upit.

U nastavku teksta, razmatramo problem određivanja *pagerank* vrednosti za svaki čvor u posmatranom grafu. Posmatrani algoritam se zasniva na navedenoj činjenici da se Web Graf može predstaviti kao *povezani* graf i pretpostavlja metod *slučajnih lutanja* (*random walks*) na grafu, kao optimalni metod za određivanje i karakterizaciju same link strukture, iz koje se može dobiti informacija o relevantnosti svakog od čvorova u datom grafu (odnosno, svake od stranica na Web-u).

Posmatrajmo model "*slučajnog surfera*" koji počinje u *slučajno* odabranoj Web stranici (čvoru Web Grafa) i počinje *slučajno lutanje* (*random walk*) na *Web*-u (Web Grafu), tako što, u svakom nizu koraka, ukoliko se nalazi na stranici A, slučajno odabira jednu od stranici iz skupa stranica na koje A pokazuje i "prelazi" na nju. (na primer, ukoliko posmatrana stranica, ukazuje na *3* stranice, na svaku od datih stranica se "prelazi" sa verovatnoćom 1/3).

Kako se *surfer* kreće od čvora do čvora, u okviru *walk*-a, neke od čvorova posećuje češće nego druge – intuitivno, možemo pretpostaviti da su to su čvorovi sa mnogo ulaznih linkova, koji dolaze takođe, od drugih često posećenih čvorova. Dodatno, uvodimo i *teleport* operaciju – koja, za razliku od kretanja od čvora do čvora, podrazumeva "skok" na slučajno odabranu Web stranicu (slučajno odabran čvor Web Grafa). Drugim rečima, ukoliko Web Graf, ima n čvorova, *teleport* operacija "*vodi*" surfera u bilo koji od čvorova sa verovatnoćom $\frac{1}{n}$ (prema tome, može se vratiti i u istu poziciju, sa verovatnoćom $\frac{1}{n}$).

U praksi, *teleport* operacija se koristi u sledećim slučajevima :

1. Kada se *surfer* nađe u stranici koja ne sadrži linkove na bilo koju drugu stranicu

2. Takođe, u bilo kom trenutku, *surfer* može prestati sa praćenjem linkova i preći na proizvoljno odabranu stranicu. Pretpostavljamo da se ovo događa sa fiksnom verovatnoćom $0 < \alpha < 1$, pri čemu je verovatnoća da *surfer* produži tekući *walk*, jednaka $1-\alpha$ (tipično, pretpostavljena vrednost za α je oko 0.1).

Pokazuje se da kada *surfer* obilazi web graf koristeći opisani proces (*random walk* + *teleport*), on u svakom čvoru v Web Grafa, provodi fiksan deo vremena $\pi(v)$, koji zavisi od : *(1)* strukture web grafa i *(2)* vrednosti α . Vrednost $\pi(v)$ nazivamo *pagerank* posmatrane stranice v.

Opisana procedura se može formalizovati, korišćenjem reprezentacije navedenog procesa u vidu *Markovljevog lanca (Markov Chain)*.

Markovljev lanac predstavlja diskretan stohastički proces, koji se može predstaviti nizom koraka, od kojih se u svakom vrši slučajan izbor, tako da izbor u svakom koraku zavisi samo od tekućeg stanja u kome se nalazi proces, a ne i od prethodnih. U konkretnom slučaju, posmatramo *lanac* sa n stanja, pri čemu svako stanje odgovara jednoj stranici na Web-u (proces se nalazi u stanju $k \in (1,n)$, ukoliko se *walk* trenutno nalazi u čvoru koji odgovara stranici k).

Markovljev lanac sa n stanja se može opisati *matricom verovatnoća prelaza* P , koja je dimenzija $n \times n$. Svaki element P_{ij} se naziva *tranziciona verovatnoća* i opisuje verovatnoću da će sledeće stanje *lanca* biti j , pod uslovom da je tekuće stanje i. Imamo da važi :

$$P_{ij} \in [0,1], \forall i, j$$

$$\sum_{j=1}^{n} P_{ij} = 1, \forall i$$

Verovatnostni vektor, predstavlja vektor čiji se svi elementi nalaze u intervalu *[0,1],* i u zbiru imaju vrednost 1. *N-dimenzionalni verovatnosni vektor,* čija svaka komponenta odgovara jednom od n *stanja* Markovljevog lanca se može posmatrati kao *verovatnostna raspodela* nad njegovim stanjima.

U skladu sa opisanim modelom, s*lučajni surfer* na Web Grafu se može posmatrati kao *Markovljev lanac*, sa po jednim stanjem za svaku Web stranu, pri čemu svaka *tranziciona verovatnoća* predstavlja verovatnoću "*pomeraja*" iz jedne stranice u drugu. Opisani *Markovljev lanac* se može odrediti na osnovu matrice povezanosti Web Grafa.

Raspodela verovatnoća pozicija *sufrera* u bilo kom trenutku se može opisati *verovatnosnim vektorom* $\vec{x}$. U trenutku $t = 0$, *surfer* počinje iz stanja čiji odgovarajući ulaz u $\vec{x}$, ima vrednost *1*, dok svi ostali imaju vrednost *0*. *Raspodela verovatnoća* u trenutku $t = 1$ se određuje kao $\vec{x} \cdot P$, dok se u trenutku $t = 2$, određuje kao $(\vec{x} \cdot P) \cdot P = \vec{x}P^2$. Na ovaj način, moguće je odrediti raspodelu u proizvoljnom trenutku $t = k$, kao $\vec{x} \cdot P^k$, poznajući samo početnu raspodelu $\vec{x}$ i matricu prelaza P .

Ukoliko bi se Markovljev lanac obavljao u beskonačno mnogo koraka, svako stanje bi bilo posećeno sa (različitom) frekvencijom, koja zavisi od strukture Markovljevog lanca. Po analogiji, sa posmatranim modelom, možemo reći da će *surfer* posećivati određene stranice (na primer, popularne stranice sa vestima), češće nego druge stranice.

Ovu pretpostavku, možemo formalizovati i analizirati, korišćenjem poznatih rezultata iz teorije *Markovljevih lanaca* :

Definicija : Markovljev lanac je *ergodičan*, ukoliko važe sledeća dva uslova :

- Za bilo koja dva stanja i, j , postoji celi broj $k \geq 2$, tako da postoji sekvenca od k stanja $s_1 = i, s_2, ..., s_k = j$, tako da $\forall 1 \leq k \leq k-1$,tranziciona verovatnoća $P_{s_l, s_{l+1}} > 0$
- Postoji trenutak T_0 , takav da za sva stanja j , Markovljevog lanca, sa proizvoljno odabrano startno stanje i (u trenutku $t = 0$) i za svako $t > T_0$, verovatnoća da se lanac nalazi u stanju j , u trenutku t , je veća od nule

Teorema : Za svaki *ergodičan* Markovljev lanac, postoji jedinstvena *stacionarna raspodela*, nad stanjima, $\vec{\pi}$, tako da u slučaju da $N(i,t)$, predstavlja broj poseta stanju i , u t koraka, važi :

$$\lim_{t \to \infty} \frac{N(i,t)}{t} = \pi(i)$$

pri čemu $\pi(i) > 0$, predstavlja stacionarnu raspodelu za stanje i.

Kao posledica navedene teoreme, sledi da opisani *random walk*, rezultuje u stacionarnoj raspodeli verovatnoća nad indukovanim Markovljevim lancem. Stacionarna raspodela određenog stanja ukazuje na "važnost" stranice koja odgovara datom atom stanju i naziva se *pagerank*.

Da bi odredili vrednosti *pagerank* funkcije za svaku stranicu, posmatramo *leve sopstvene vektore*, tranzicione matrice P ,koji predstavljaju N-vektore $\vec{\pi}$, tako da važi :

$$\vec{\pi}P = \lambda\vec{\pi}$$

Elementi sopstvenog vektora $\vec{\pi}$, predstavljaju stacionarne verovatnoće *random walk*-a, odnosno, odgovaraju *pagerank* vrednostima odgovarajućih Web stranica. Date *pagerank* vrednosti možemo odrediti izračunavanjem dominantnog (sa sopstvenom vrednošću *1*) *levog sopstvenog vektora* matrice P .

Postoji veliki broj poznatih algoritama za određivanje *levog sopstvenog vektora*, koji se međusobno razlikuju u kompleksnosti i brzini konvergencije. Elementarni, i ne najefikasniji način na koji se data vrednost može odrediti je korišćenjem metoda, koji se naziva *stepena iteracija (power iteration).* Na osnovu navedene analize, imamo da ukoliko sa $\vec{x}$, predstavimo početnu raspodelu stanja, raspodela u trenutku t tima vrednost $\vec{x}P^t$. Za velike vrednosti t, očekujemo da je raspodela $\vec{x}P^t$, slična raspodeli $\vec{x}P^{t+1}$, budući da očekujemo da se Markovljev lanac nalazi u stabilnom stanju. Dodatno, na osnovu navedene teoreme, imamo da je ova vrednost nezavisna od početnog vektora $\vec{x}$. Prema tome, metod *stepene iteracije* , zapravo predstavlja simulaciju *walk*-a : počinjemo u slučajno izabranom stanju i zaustavljamo se nakon t koraka, pri čemu u svakom koraku, čuvamo informaciju o frekvenciji posećivanja svakog od stanja. Nakon velikog broja koraka, ove frekvencije počinju da konvergiraju ka ”stabilnim” vrednostima, koje proglašavamo za određene *pagerank* vrednosti.

Kao primer određivanja *pagerank* vrednosti, posmatrajmo jednostavan graf, koji se sastoji od 3 čvora (numerisana sa 1,2,3), sa linkovima : $1 \to 2{,}3 \to 2{,}2 \to 1{,}2 \to 3$, pri čemu je verovatnoća *teleport* operacije u svakom koraku jednaka $\alpha = 0.5$. Na osnovu ovih podataka, možemo odrediti matricu verovatnoća prelaza :

$$P = \begin{pmatrix} 1/6 & 2/3 & 1/6 \\ 5/12 & 1/6 & 5/12 \\ 1/6 & 2/3 & 1/6 \end{pmatrix}$$

Pretpostavimo da *surfer*, počinje u stanju *1*, koje odgovara početnom vektoru raspodele $\vec{x} = (1 \quad 0 \quad 0)$. Nakon jednog koraka, raspodela ima vrednost :

$$\vec{x} \cdot P = (1/6 \quad 2/3 \quad 1/6)$$

Nakon dva koraka, raspodela $\vec{x}P^2$, ima vrednost $\vec{x} = (1/3 \quad 1/3 \quad 1/3)$, nakon tri koraka, $\vec{x} = (7/24 \quad 5/12 \quad 7/24)$. Ponavljajući datu operaciju, uočavamo da raspodela konvergira ka stacionarnom stanju $\vec{x} = (5/18 \quad 4/9 \quad 5/18)$. Data vrednost odgovara traženoj *pagerank*, vrednosti, tako da konačno imamo :

$$\vec{\pi} = (5/18 \quad 4/9 \quad 5/18)$$

Primećujemo da dobijena vrednost, isključivo zavisi od strukture posmatranog grafa, tako da možemo reći da pagerank predstavlja *meru nezavisnu od upita* (*query-independent measure*).
U praksi, često je od interesa da se pri pretraživanju obezbedi i uticaj zadatog upita na rankiranje rezultata pretraživanja, tako da se u komercijalnim pretraživačima, dobijena vrednost koristi u kombinaciji sa drugim metodama koje vrše rankiranje u odnosu na sam *sadržaj* stranice (kao što je opisana *Lucene scoring* funkcija).

2.4.3. HITS

Za razliku od PageRank algoritma, koji dodeljuje svakoj web stranici ocenu u rasponu (0,1), koja ukazuje na njenu relevantnost, HITS (*Hypertext Induced Topic Selection*) algoritam, predstavlja shemu koja podrazumeva dodeljivanje *dve* ocene svakoj Web stranici, od kojih se jedna naziva *hub score* a druga *authority score*. Osnovna ideja je da se za svaki upit, određuju dve liste rankiranih rezultata od kojih je jedna indukovana *hub* rezultatima, a druga *authority* rezultatima,

Ovaj pristup polazi od opažanja da postoje dve vrste stranica koje mogu biti korisne u pretraživanjima vezanim za teme *opšteg* tipa – jedne su *autoritativni izvori informacija* vezani za zadatu temu (*authorities*), dok su druge, *stranice koje sadrže listu likova ka drugim autoritativnim stranicama* (*hub pages*). Osnovna ideja ovog pristupa je korišćenje *hub* stranica za otkrivanje *authority* stranica.

Pod ”dobrom” *hub* stranicom, podrazumevamo stranicu koja pokazuje na veliki broj ”dobrih” *authority* stranica. Takođe, ”dobru” *authority* stranicu, rekurzivno definišemo kao stranicu koja pokazuje veliki broj ”dobrih” *hub* stranica.

Posmatrajmo proizvoljan podskup Web grafa (podskup svih stranica, zajedno sa linkovima između njih), na kome određujemo relevatnost stranica korišćenjem HITS algoritma. Budući da je definicija *hub*-ova i *authority*-ja cirkularna, *hub* i *authority* vrednosti za svaku stranicu, nad ovim podskupom računamo iterativno, na sledeći način :

Za svaku stranicu x (čvor izabranog podskupa web grafa), sa $h(x)$ označavamo njen *hub score*, dok sa $a(x)$, označavamo njen *authority score*. Inicijalno, postavljamo vrednosti $h(x) = a(x) = 1$, za sve čvorove x. Takođe, sa $x \to y$, označavamo postojanje hiperlinka iz stranice x, ka stranici y. Osnovu iterativnog algoritma, predstavlja sledeći par funkcija koje definišu ažuriranje *hub* i *authority* vrednosti za svaku stranicu, koji se zasniva na intuitivnom opažanju da *dobri hub-ovi pokazuju na dobre authoritity stranice i da na dobre authoritity stranice pokazuju dobri hub-ovi* :

$$(1) \qquad h(x) \leftarrow \sum_{x \to y} a(y)$$

$$(2) \qquad a(x) \leftarrow \sum_{y \to x} h(y)$$

Prema tome, *(1)* postavlja *hub score* stranice x na *authority score* stranica na koje data stranica linkuje (ukoliko x linkuje na stranice koje imaju veći *authority*, njegov *hub score* se povećava), dok, na osnovu *(2)* imamo da, ukoliko na stranicu x linkuju dobri *hub*-ovi, njen *authority score* se povećava.

Gore navedene jednačine možemo zapisati u matričnoj formi. Neka $\vec{h}$ i $\vec{a}$, označavaju vektore svih *hub* i *authority* vrednosti, stranica u posmatranom podskupu Web grafa. Neka A predstavlja *matricu povezanosti* posmatranog podskupa stranica : A je kvadratna matrica sa po jednim redom i kolonom za svaku stranicu ($A_{ij} = 1$, ako postoji link od i ka j). Tada imamo da važi :

$$\vec{h} \leftarrow A\vec{a}$$
$$\vec{a} \leftarrow A^T\vec{h}$$

gde A^T predstavlja transponovanu matricu, nastalu od matrice A. Primetimo da desna strana svakog od izraza, predstavlja vektor, koji takođe predstavlja levu stranu onog drugog izraza.

Ukoliko ove vrednosti međusobno zamenimo, posmatrane izraze možemo da zapišemo kao :

$$\vec{h} \leftarrow AA^T \vec{h}$$
$$\vec{a} \leftarrow A^T A\vec{a}$$

Ukoliko zamenimo simbole $\leftarrow$ sa $=$ i uvedemo (nepoznatu) sopstvenu vrdednost, prva jednačina postaje jednačina sopstvenog vektora AA^T, dok druga postaje jednačina sopstvenog vektora $A^T A$:

$$\vec{h} = \lambda_h AA^T \vec{h}$$
$$\vec{a} = \lambda_a A^T A\vec{a}$$

U navedenim jednačinama, λ_h, predstavlja sopstvenu vrednost AA^T, dok λ_a, predstavlja sopstvenu vrednost $A^T A$.

Možemo zaključiti sledeće :

- Iterativna ažuriranja, u jednačinama *(1)* i *(2)* , skalirana odgovarajućim sopstvenim vrednostima, jesu ekvivalentna *stepenoj metodi* za određivanje sopstvenih vektora AA^T i $A^T A$. Prema tome, iterativno određene vrednosti za $\vec{h}$ i $\vec{a}$, konvergiraju ka stacionarnim vrednostima određenim elementima matrice A, odnosno strukturom link grafa.
- Pri određivanju navedenih vrednosti, nismo ograničeni na *stepenu iterativnu metodu,* koja sporo konvergira, već možemo koristiti proizvoljnu metodu za određivanje sopstvenih vrednosti matrice.

Konačno, procedura za određivanje *hub*/*authority* vrednosti ima sledeći oblik :

1. Određujemo željeni podskup Web stranica i kreiramo graf indukovan njihovim linkovima i određujemo AA^T i $A^T A$
2. Određujemo glavne sopstvene vektore AA^T i $A^T A$ i kreiramo vektor *hub* vrednosti $\vec{h}$ i *authority* vrednosti $\vec{a}$
3. Određujemo stranice sa najvećim *hub* vrednostima i sa najvećim *authority* vrednostima

Korišćenjem opisane procedure, određujemo najbolje *hub* i *authority* stranice u okviru zadatog podskupa Web grafa. U praksi, od interesa je da posmatrani podskup sadrži samo stranice vezane za posmatranu temu. Jedan od načina za određivanje ovog skupa je sledeća procedura:

- U odnosu na zadati upit, koristeći celokupni indeks, određujemo sve stranice koje odgovaraju upitu. Ovaj skup se naziva *root set* stranica.
- Kreiramo osnovni skup (*base set*) stranica, tako što uključujemo sve stranice u *root* skupu, i sve stranice na koje ukazuju stranice u *root* skupu

Konačno, koristimo dobijeni *base* skup, kao podskup Web Grafa, nad kojim vršimo određivanje *hub* i *authority* vrednosti.

2.5. Clustering

2.5.1. Osnove

Grupisanje podataka (clustering) predstavlja način obrade podataka, kojim se u samim podacima otkrivaju tzv. "grupe" (clusters) podataka koje pokazuju izvestan stepen "*prirodne bliskosti*". Pod algoritmima za grupisanje podataka, podrazumevamo algoritme koji automatski otkrivajju *grupe*("*klastere*") u zadatom skupu podataka. Ovakav vid obrade podataka u praksi ima niz primena, posebno u oblasti "istraživanja podataka" (data mining).

Dva najčešća pristupa problematici grupisanja rezultata pretraživanja poklapaju se sa pristupima u okviru data mining metodologije (supervised/unsupervised learning). Kod supervised learning-a koristimo algoritme mašinskog učenja (naïve Bayes, SVM , Neuronske mreže), koji na osnovu zadatog skupa "ručno" klasifikovanih dokumenata (learning set), nakom procesa "učenja" (težinskih faktora kod naïve bayes, hiperprostora kod svm, koeficijenata kod nn), vrše klasifikaciju stranica na osnovu njihovog sadržaja. Drugi pristup u okviru data mining metodologije predstavlja tzv. "unsupervised" learning, koji vrši klasifikaciju dokumenata bez prethodnog "znanja", koristeći samo informacije prusutne u samim podacima (k-nearest neighboor). Ova metodologija se često koristi u "istraživanju podataka", kada ne postoji prethodno definisani korpus podataka ili kada nismo sigurni šta tačno tražimo u podacima.Tome shodno, rezultate pretraživanja možemo posmatrati kao obične dokumente i primenjivati standardne metode mašinskog učenja za njihovo grupisanje (supervised – u odnosu na zadatu temu ili unsupervised – po sličnosti dokumenata). Ovakav pristup primenjuju neki komercijalni clustering pretraživači (*Alexa*, *Clusty*..).

Međutim, problematika Web pretraživanja je specifična u tome što sami dokumenti pored sopstvenog sadržaja, sadrže i informaciju o međusobnoj povezanosti (link graph). U praksi je pokazano da sam link graph sadrži količinu informacija koja omogućava njihovo efikasno rankiranje (page rank algoritam se zasniva upravo na link information). Prema tome, prirodno se pretpostavlja da se link informacija može efikasno iskoristiti i u cilju grupisanja search rezultata.
U ostatku ovog poglavlja, razmatramo problematiku i postojeće metode grupisanja rezultata korišćenjem link graph-a. Takođe, predstavljamo algoritam za grupisanje rezultata, razvijen u okviru diplomske teze. Ovaj algoritam predstavlja aproksimaciju clustering procesa, koji omogućava grupisanje rezultata kvaliteta približnom *state-of-the-art* algoritmima, ali u vremenu izračunavanja koje je znatno efikasnije od postojećih algoritama.

Graph clustering problem, možemo formulisati na sledeći način :

Posmatramo povezani, neusmereni graf $\mathrm{G(V,E)}$. Neka je $|\mathrm{V}| = \mathrm{n}$ i $|\mathrm{E}| = \mathrm{m}$ i neka je $\mathrm{C} = (C_1,...,C_k)$, particija_skupa V. Skup C, nazivamo *grupisanje* (*klastering)* grafa G, dok podskupove C_i , nazivamo *klasterima (clusters)*. C je *trivijalan klaster*, ukoliko je ili k = 1 ili ukoliko svi klasteri sadrže samo jedan element. Klaster C_i, predstavljamo kao indukovani podgraf, grafa G , tj, graf : $G[C_i] = (C_i, E(C_i))$, gde je $E(C_i) = \{\{v,w\} \in E: v,w \in C_i\}$. Tada je $E(C) = \bigcup_{i=1}^{k} E(C_i)$, skup *intra-klaster grana* (grana između klastera), dok je $E \setminus E(C)$, skup *inter-klaster grana* (grana unutar klastera). Broj intra-klaster grana se označava sa $m(C)$, dok se broj inter-klaster grana označava sa $\overline{m}(C)$. Grupisanje (*clustering*) $C = (C, V \setminus C)$ se jos naziva i presek (*cut*) , grafa G, gde $\overline{m}(C)$ označava dimenziju preseka. Presek minimalne dimenzije se naziva i *minicut*.

U nastavku teksta, definišemo neke mere, koje obezbeđuju evaluaciju performansi *graph clustering* algoritama :

Pokrivenost

Mera pokrivenosti - *coverage* (C), za zadati *clustering* C, predstavlja udeo intra-klaster grana u kompletnom skupu grana :

$$\mathrm{coverage}(C) = \frac{m(C)}{m} = \frac{m(C)}{m(C) + \overline{m}(C)}$$

Intuitivno, veća vrednost za *coverage* , ukazuje na bolji kvalitet *clustering*-a C. Primećujemo da *minicut* ima maksimalni *coverage*, i u tom smislu, predstavlja "optimalni" *klastering*. Ipak, u praksi, kao mera dobrog *clustering-a*, pored velike *coverage* vrednosti, uvode se dodatna ograničenja, kao što su veličina i broj klastera. Iako se *minicut* može odrediti u polinomijalnom vremenu, problem konstruisanja clustering-a za fiksnim brojem klastera, $k \quad (k \geq 3)$, je NP-hard, kao i problem nalaženja *minicut*-a koji zadovoljava određena ograničenja po pitanju dimenzije klastera.

Performanse

Mera performansi – *performance* (C) za zadati *clustering* C, određuje broj "pravilno interpretiranih parova čvorova" u grafu. Tačnije, ona predstavlja udeo intra-cluster grana zajedno sa nepovezanim parovima čvorova u različitim klasterima , unutar skupa svih čvorova :

$$performance(C) = \frac{m(C) + \sum_{\{v,w\} \notin E, v \in C_i, w \in C_j, i \neq j} 1}{\frac{1}{2} n(n-1)}$$

Problem minimizacije performance metrike se redukuje na problem particionisanja grafa, koji je, takođe NP-hard.

Intra-cluster i Inter-cluster *propusnost*

Propusnost (konduktansa) datog cut-a poredi veličinu cut-a i broj grana u bilo kom od dva indukovana podgrafa. Tada konduktansa $\phi(G)$, grafa G, predstavlja minimalnu vrednost konduktanse za sve cut-ove u G. Za clustering $C = (C_1, ..., C_k)$, grafa G, *intra-cluster konduktansa* $\alpha(C)$, predstavlja minimalnu vrednost konduktanse na svim indukovanim podgrafovima $G[C_i]$, dok je *inter-cluster konduktansa* $\delta(C)$, maksimalna vrednost konduktanse na svim indukovanim presecima $(C_i, V \setminus C_i)$. Posmatrajmo presek $C = (C, V \setminus C)$, grafa G, i definišimo konduktanse $\phi(C)$ i $\phi(G)$, na sledeći način :

$$\phi(C) = \begin{cases} 1, C \in \{0, V\} \\ 0, C \notin \{0, V\}, \overline{m}(C) = 0 \\ \dfrac{\overline{m}(C)}{\min(\sum_{v \in C} \deg v, \sum_{v \in V \setminus C} \deg v)}, inace \end{cases}$$

$$\phi(G) = \min_{C \subseteq V} \phi(C)$$

Presek ima malu *konduktansu* ukoliko mu je veličina mala relativno u odnosu na gustinu obe strane preseka. Minimizacija konduktanse nad svim presecima u grafu i nalaženje odgovarajućeg preseka je NP-hard problem, ali se u opštem slučaju može aproksimirati poli-logaritamskim algoritmom.

Sada definišemo intra-cluster konduktansu $\alpha(C)$ i inter-cluster konduktansu $\delta(C)$, kao :

$$\alpha(C) = \min_{i\in\{1,\dots k\}} \phi(G[C_i])$$

$$\delta(C) = 1 - \max_{i\in\{1,\dots k\}} \phi(C_i)$$

Clustering sa malom intra-cluster konduktansom, obično, dovodi do prilično “grubog” grupisanja, dok clustering sa malom inter-cluster konduktansom, dovodi do preterano “finog” grupisanja.

U nastavku teksta, predstavljamo dva poznata algoritma za clustering, koji pokazuju dobre performanse u smislu prethodno definisanih metrika. Oba algoritma koriste normalizovanu matricu povezanosti grafa G, koja se dobija kao : $M(G) = D(G)^{-1} A(G)$, gde je $A(G)$, matrica povezanosti, dok je $D(G)$, dijagonalna matrica, koja sadrži informacije o *stepenu* svakoga čvora.

2.5.2. Markov Clustering (MCL)

Osnovna ideja iza Markov Clustering (MCL) algoritma je da “*random walk koji poseti jako povezani cluster, ne napušta cluster dok ne obiđe veći deo njegovih grana*”. Umesto da simulira random walk-ove, MCL iterativno modifikuje matricu tranzicionih verovatnoća. Počevši od $M = M(G)$ (koja odgovara random walk-ovima dužine najviše jedan), sledeće dve operacije se iterativno ponavljaju :

- *ekspanzija* – u kojoj se M stepenuje stepenom $e \in N_{>1}$, čime se simulira e koraka random walk-a, koristeći tekuću tranzicionu matricu
- *inflacija* – u kojoj se M renormalizuje stepenovanjem svakog elementa na r-ti stepen, $r \in R^{+}$.

Primećujemo da za $r > 1$, inflacija ističe heterogenost verovatnoća unutar jednog reda matrice, dok se ra $r < 1$, ističe homogenost verovatnoća. Iteracija se završava kada se dostigne rekurentno stanje ili fiksna tačka. Rekurentno stanje perioda $k \in N$, predstavlja matricu koja je invarijantna u odnosu na k perioda inflacije i ekspanzije, dok se fiksna tačka može posmatrati kao rekurentno stanje periode 1. U praksi, MCL najčesće zavšava rad u fiksnoj tački.

Konačan clustering se dobija identifikacijom povezanih komponenti u matrici povezanosti, nakon što se dostigne rekurentno stanje. MCL predstavlja deterministički algoritam, čiju kompleksnost diktira operacija ekspanzije, koja u suštini predstavlja operacuju množenja matrica.

Markov Clustering (MCL) Algoritam :

Ulaz : Graf $G(V,E)$, parametar ekspanzije e, parametar inflacije r

$M \leftarrow M(G)$

dok M ne dostigne fiksnu tačku, ponavljamo:

$M \leftarrow M^e$

za svako $u \in V$:

za svako $v \in V$: $M_{uv} \leftarrow M_{uv}^r$

za svako $v \in V$: $M_{uv} \leftarrow \frac{M_{uv}}{\sum_{w \in V} M_{uw}}$

$H \leftarrow$ graf indukovan ne-nultim elementima M

$C \leftarrow$ clustering indukovano povezanim komponentama u H

U domenu prethodno definisanih metrika, možemo primetiti da MCL algoritam ističe inter-cluster connectivity.

2.5.3. Iterative Conductance Cutting (ICC)

Osnovna ideja Iterative Conductance Cutting (ICC) algoritma iterativno deli cluster-e, koristeći cut-ove minimalne konduktanse. Određivanje cut-a minimalne konduktanse je NP-hard, tako da se koristi algoritam koji omogućava poli-logaritamsku aproksimaciju. Posmatrajmo raspored čvorova, kreiran na osnovu sopstvenog vektora druge najveće sopstvene vrednosti $M(G)$. Među svim presecima koji dele ovaj raspored na dva dela, odabiramo presek sa minimalnom konduktansom. Deljenje cluster-a se završava, kada vrednost aproksimacije za konduktansu pređe zadati threshold α^*.ICC takođe predstavlja deterministički algoritam, njegovo vreme izvršavanja zavisi od broja iteracija, pri čemu je vreme izvrsavanja aproksimacije preseka, uslovljeno operacijom izračunavanja sopstvene vrednosti matrice, koja se mora obavljati u svakoj iteraciji.

Iterative Conductance Cutting (ICC)

Ulaz : $G(V,E)$, threshold vrednost konduktanse, $0 < \alpha^* < 1$

$C \leftarrow \{V\}$

sve dok postoji $c \in C$, sa $\phi(G[c]) < \alpha^*$, obavljamo sledeće operacije:

$x \leftarrow$ sopstveni vektor od $M(G[c])$,povezan sa drugom najvećom sopstvenom vrednošću

$S \leftarrow \{S \subset c : \max_{v \in S}\{x_v\} < \min_{w \in C \setminus S}\{x_w\}\}$

$c' \leftarrow \arg\min_{s \in S}\{\phi(s)\}$

$c \leftarrow (C \setminus \{c\}) \bigcup \{c', c \setminus c'\}$

3. Algoritam za grupisanje rezultata pretraživanja, korišćenjem slučajnih lutanja na link grafu

3.1. Uvod

Pored standardnih funkcija pretraživanja, koje pružaju današnji pretraživači, često je od interesa implementacija funkcionalnosti grupisanja (*clustering*) rezultata pretraživanja. Budući da najčešći upiti obično rezultuju u enormnim korpusima relevantnih stranica, potrebni su efikasni mehanizmi koji omogućavaju lakše "snalaženje" korisnika, među dobijenim rezultatima.

U praksi, ovaj način obrade rezultata se ne koristi kod "mainstream" web pretraživača, budući da, sa jedne strane, implementacija algoritama za clustering iziskuje značajne resurse, dok sa druge strane, postojanje ovakve funkcije još uvek finansijski ne opravdava troškove (u smislu potrošnje sistemskih resursa), njene implementacije. Zato se danas, ova funkcionalnost uglavnom implementira u okviru specijalnih, tzv. "clustering" web pretraživača (*Clusty*, *WebClust*, *Mooter*).

Način na koji se vrši grupisanje dokumenata, u okviru navedenih sistema, se uglavnom zasniva na algoritmima za klasifikaciju iz oblasti *mašinskog učenja*. Ovi algoritmi se zasnivaju na analizi sadržaja samih dokumenata (ekstrakcija tokena i reprezentacija tokena u okviru *vector space modela*), nakon čega se može vršiti *unsupervised* klasifikacija (koja ne podrazumeva nikakvo prethodno saznanje o dokumentima, koristi se isključivo dobijeni *prostor* token-a, u okviru koga se vrši grupisanje korišćenjem npr. k-nearest neighbor algoritma) ili *supervised* klasifikacija (kod koje se polazi od korpusa dokumenata definisanog značenja, na osnovu koga se vrši *učenje* parametara algoritma, nakon čega se vrši klasifikacija novih dokumenata datim algoritmom, npr. Naïve Bayes, Suppor Vector Machine).

Dodatni način na koji se može vršiti grupisanje rezultata je korišćenje informacije o njihovim međusobnim vezama (što predstavlja specifičnost web search rezultata). Ovaj vid informacija se u praksi efikasno koristi za određivanje relevantnosti stranica (*PageRank*, *OPIC*, *HITS*…).
Korišćenjem date reprezentacije, moguće je problem grupisanja rezultata predstaviti kao *graph clustering* problem i rešiti ga korišćenjem poznatih clustering algoritama (od kojih su neki opisani u poglavlju 2.6).

Međutim, postojeći algoritmi za clustering grafova su visoke kompleksnosti (većina, zapravo predstavlja aproksimacije NP-hard problema), i kao takvi, nisu prikladni za implemenaciju na sistemima koji vrše obradu podataka reda veličine kakva se sreće kod web pretraživača ($\sim 10^6 - 10^9$ dokumenata).

U nastavku teksta, predstavljamo jednostavan randomizovani algoritam, razvijen u okviru diplomskog rada, koji koristi *slučajna lutanja* (*random walks*) na link grafu, za grupisanje rezultata pretraživanja, u odnosu na zadati upit. Sam algoritam postiže rezultate koji su poredivi sa rezultatima postojećih determinističkih algoritama za graph clustering. Istovremeno, algoritam ima znatno nižu kompleksnost od navedenih, što omogućava njegovu jednostavnu implementaciju i u pretraživačima čija prvenstvena namena nije *clustering*. Dodatno, sam algoritam pripada familiji *randomizovanih algoritama*, usled čega je moguće proizvoljno podešavati odnos između njegove kompleksnosti i performansi, što ga upravo čini pogodnim za primenu u pretraživačima opšte namene.

Konačno, u poglavlju 4, dajemo opis *randomNode* pretraživača (razvijenog u okviru diplomskog rada), koji koristi upravo navedeni algoritam, u cilju grupisanja rezultata pretraživanja.

3.2. Postavka problema

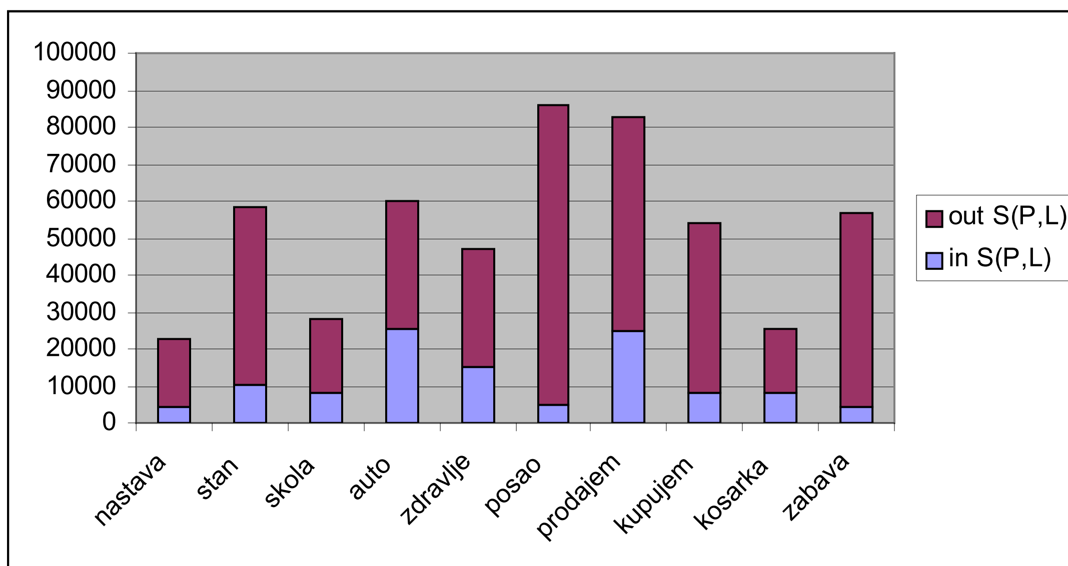
Posmatrajmo graf $G(V,E)$, gde je $|V|=n$, broj stranica u indeksu pretraživača i $|E|=k$, broj linkova između stranica. *Indukovanim grafom* $S(P,L)$, nazivamo podgraf grafa G, u odnosu nazadati upit q, takav da važi :

$$S(P,L) \mid P \subseteq V, L \subseteq E \;,\; p \in P \mid p \in q(V) \text{ i } (u,v) \in L \mid (u \in P \wedge v \in P)$$

Drugim rečima, *indukovani graf* predstavlja graf koji nastaje kada iz datog grafa izdvojimo samo čvorove koji odgovaraju stranicama koje odgovaraju datom upitu i samo one linkove čija su oba kraja u skupu stranica koje odgovaraju upitu (ostale linkove zanemarujemo).

Empirijska opažanja na Web grafu, ukazuju na postojanje *tematske lokalnosti* među stranicama na Web-u – stranice vezane za istu temu se najčešće nalaze *blizu* u Web grafu. Ovo uzrokuje činjenicu da, pri kreiranju *indukovanog grafa* , u odnosu na zadati upit, značajan deo grana koje ukazuje na selektovane čvorove *ostaje* unutar novokreiranog grafa. Na taj način, dobijamo strukturu koja zadržava značajne karakteristike originalnog grafa, što nam dozvoljava da *pretpostavimo* da se *grupisanje*, značajnog kvaliteta, može ostvariti i posmatrajući samo čvorove i grane indukovanog grafa. (Na taj način se značajno smanjuje kompleksnost procedure grupisanja, budući da posmatramo znatno manji skup čvorova).

U cilju analize navedene pretpostavke, na slici 3 je prikazan grafik dobijen analizom procenta grana koje ostaju unutar indukovanog grafa, za neke od čestih reči u okviru Srpskog jezika, na korpusu dokumenata *randomNode* pretraživača (opisanog u glavi 4) :



slika 3 : procenat link-ova koji "ostaje" u indukovanom grafu

Primećujemo, da u zavisnosti od zadatog upita, udeo linkova u indukovanom grafu varira između 5% i 40%. Ovo je za očekivati, budući da na ovaj način upravo obuhvatamo pojavu, da, u zavisnosti od teme, stranice imaju veću ili manju tendenciju ka međusobnom grupisanju.

Konačno, problem grupisanja rezultata pretraživanja možemo predstaviti kao problem *efikasnog* određivanja svih *prirodnih* grupa (*cluster*-a) čvorova u grafu $S(P, L)$, koji nastaje na osnovu grafa $G(V, E)$, kao *indukovani podgraf,* u odnosu na zadati *upit* q.

3.3. Postojeća rešenja

Problem clustering-a indukovanog podgrafa, predstavljamo kao problem određivanja *prirodnih* cluster-a u zadatom grafu, odnosno njegovo particionisanje na odgovarajući broj *jako povezanih pogdgrafa*. Osnovni način na koji se ovo može ostvariti je korišćenjem *graph clustering* algoritama, opisanih u poglavlju 2.6. Pored ovih, navodimo još neke od poznatih pristupa datom problemu :

- *Max-flow min-cut formulacija* [Flake et al. 2000] : definišemo *community*, kao skup čvorova koji sadrži više intracommunity grana (početak i kraj unutar *community*-ja), nego intercommunity grana (između *community*-ja). Problem ekstrakcije *community*-ja se definiše kao minimum-cut problem na grafu – Inicijalno, određujemo početni skup čvorova, korišćenjem HITS algoritma. Počevši od dobijenog skupa čvorova za koje je poznato da pripadaju *community*-ju, nalazimo minimalni cut-set grana koje dele graf na dve povezane komponente, tako da svi početni čvorovi ostanu u jednoj od njih. Zatim se ta komponenta koristi za ponovno otkrivanje početnih čvorova; proces se ponavlja sve dok se ne pronađe zadovoljavajuća komponenta, koja predstavlja *community* koji odgovara početno zadatim čvorovima.
- *Graph partitioning* – Osnovna ideja se zasniva na rekurzivnom particionisanju grafa : u svakom koraku, graf se deli na dve particije, od kojih se svaka particioniše u sledećem koraku, pri čemu se optimizuje odgovarajuća metrika, definisana na grafu. Popularni METIS algoritam za particionisanje grafova, pokušava da odredi najbolji separator, tako što minimizuje broj grana koje se "seku", sa ciljem dobijanja dve povezane komponente prilično bliskih dimenzija. U ovu grupu metoda, spadaju MCL i ICC algoritmi opisani u poglavlju 2.6. Dodatne metode uključuju i *spectral clustering*, koji podrazumeva particionisanje grafa korišćenjem nekoliko prvih singularnih vektora matrice povezanosti datog grafa.
- *Bipartite cores* – polazi od ideje određivanja grupa rezultata, korišćenjem HITS algoritma – za zadati upit, odabiraju se određen broj rezultata sa najvećim skorom, i uzimaju za početne čvorove. Nakon toga, nalaze se svi čvorovi koji ukazuju njih ili na koje oni ukazuju, čime se dobija podgraf celokupnog Web grafa. Nakon toga, primenjuje se HITS algoritam na dobijeni graf i 10 najboljih hub i authority čvorova se vraćaju kao predstavnici core community-ja za zadati upit.

Osnovni problem postojećih rešenja, sastoji se u njihovoj visokoj kompleksnosti izračunavanja. Čak i u slučaju, na primer, spektralnih metoda, za koje postoje efikasni polinomijalni algoritmi, vreme izračunavanja je i dalje suviše veliko za probleme na grafovima velikih dimenzija.
Dodatno, pojam *cluster*-a, može biti definisan na razne načine i ne mora neophodno odgovarati intuitivnoj *min-cut* formulaciji.

U nastavku teksta predstavljamo jednostavan randomizovani algoritam za aproksimativno grupisanje čvorova *indukovanog grafa*. U okviru algoritam, pojam *cluster*-a, se "*prirodno*" definiše u smislu random walk-a na usmerenom grafu. Performanse algoritma su poredive sa gore navedenim postojećim algoritmima, dok kompleksnost zavisi od parametra algoritma, koji diktiraju stepen aproksimacije.

3.4. Predloženi algoritam

Neka je $G(V,E)$, povezani, *neusmereni* graf, gde je $|V|=n$ i $|E|=m$. Pod *slučajnim lutanjem* (**random walk**) *na grafu*, podrazumevamo Markovljev Lanac M_G, takav da su stanja M_G, čvorovi grafa G, i da za svaka dva čvora $u,v \in V$, tranzicione verovatnoće (matrice prelaza), uzimaju vrednost :

$$P_{uv} = \begin{cases} \dfrac{1}{d(u)}, ukoliko(u,v) \in E \\ 0, inace \end{cases}$$

gde sa $d(u)$, označavamo *stepen* čvora u.

U slučajnu *usmerenog* grafa, tranzicione verovatnoće uzimaju vrednosti:

$$P_{uv} = \begin{cases} \dfrac{1}{d(u)}, ukoliko(u \to v) \in E \\ 0, inace \end{cases}$$

Intuitivno, *random walk* na usmerenom grafu G, možemo predstaviti kao *proces*, koji se obavlja u nizu diskretnih *koraka* : počevši od čvora u, slučajno odabiramo jedan od čvorova do kojih postoji usmerena putanja od u, pri čemu verovatnoća je verovatnoća odabira tačno određenog čvora v, inverzno proporcionalna broj čvorova do kojih takođe postoji usmerena putanja. Navedeni proces se ponavlja sve dok *walk* , ne stigne u *stanje zaustavljanja*, odnosno do čvora *stepena 0* (iz koga ne postoji putanja ka bilo kom drugom čvoru u grafu).

Jedan od osnovnih rezultata teorije slučajnih lutanja na grafovima, predstavlja teorema da je za slučaj *jako povezanih neusmerenih grafova*, odgovarajući *random walk*, *ergodican* (prolazi kroz svaki čvor na grafu), pri čemu je vreme potrebno da se *obiđe* svaki čvor *(cover time)* $C(G) \le 2m(n-1)$, *gde* n predstavlja broj čvorova, dok m, predstavlja broj grana zadatog grafa. Ovaj rezultat predstavlja osnovu nekih od izuzetno uspešnih algoritama za graph clustering (kao što je MCL, algoritam, opisan u *poglavlju 2.6.2*)

Međutim, u posmatranom slučaju (podgraf *indukovan* u odnosu na zadati upit), navedene rezultate nije moguće primeniti, iz više razloga : (1) zato što se radi o *usmerenom* grafu (za koji tvrdnja o ergodicnosti ne važi, budući da uvek postoje grafovi *stepena 0*) i (2) zato što posmatrani graf nije *jako povezan* (u praksi su obično u pitanju *sparse* grafovi, koji se sastoje od velikog broja *nepovezanih* komponenti). Pod *razuđenim* (*sparse*), grafovima, u opštem slučaju, podrazumevamo grafove kod kojih je broj grana reda veličine $|E| = O(|V|^k)$, gde je $1 < k < 2$.

Shodno navedenom, dajemo predlog algoritma koji omogučava otkrivanje *prirodnih* clustera, u *sparse* grafovima, koji se sastoje od više *nepovezanih* komponenti:

Osnovu predloženog algoritma predstavlja *slučanji proces* na indukovanom podgrafu koji podrazumeva odgovarajući broj *slučajnih lutanja sa skokovima (random walks with jumps)* na zadatom grafu, pri čemu se operacija *skoka* obavlja kada dati *walk*, dostigne *stanje zaustavljanja*, nakon čega se ponovo slučajno odabira sledeći čvor u grafu, iz koga se započinje novi *walk*. Broj skokova se određuje tako da obezbeđuje *pokrivanje svih čvorova* u grafu (da svaki od čvorova grafa bude *posećen* makar jedanput), kao i *pokrivanje što većeg broja cluster*-a u grafu (u zavisnosti od stepena aproksimacije).

Random Walk Clustering Algoritam:

```
Ulaz : Graf G(V,E), |V| = N i parametar aproksimacije K

WALK faza:
i ← 0
dok je i < K , obavljamo sledeće operacije :
        s ← rand(1,N)
        sve dok s ≠ 0 ponavljamo :
              ukoliko u w_i , ne postoji čvor s : w_i ← (s,1)
              s(w_i) = s(w_i) + 1
              s ← rand(adj) ;  v ∈ adj | ∃(s → v) ∈ E
              ukoliko je adj = { },  s ← 0

dobijamo vektor W = (w_1,...,w_K), koji predstavlja skup walk-ova

MERGE faza:
za svaki walk w_i ∈ W, i ∈ (1,K) :
        za svaki čvor n ∈ w_i
              ukoliko ∃s ∈ w_m | deg(s) > deg(n) , izbacujemo (cut) n iz w_i
              ukoliko ∃s ∈ w_m | deg(s) = deg(n), vršimo spajanje (join) w_i i w_m

konačno, dobijamo vektor C = (w_1....w_M), pri čemu je M ≤ K , koji
predstavlja skup cluster-a u zadatom grafu
```

Predstavljeni algoritam možemo neformalno opisati na sledeći način :

Polazimo od slučajno odabranog čvora zadatog grafa, iz koga počinjemo *random walk.* Inicijalizujemo skup čvorova datog *walk*-a, i počinjemo obilazak polazeći iz datog čvora. Pri svakom sledećem koraku, posmatramo da li se novi čvor već nalazi u skupu čvorova. Ukoliko to nije slučaj, dodajemo da u skup, nakon čega vršimo ažuriranje brojača za zadati čvor, u okviru tekućeg walk-a. Navedeni postupak obavljamo sve dok ne dođemo u čvor, koji nema *outlink*-ova (što predstavlja *stanje zaustavljanja* walk-a), kada vršimo *jump* operaciju, koja se ogleda u tome što ponovo izabiramo slučajan čvor u grafu, iz koga započinjemo novi *walk*.

Nakon *K* obavljenih skokova, gde je *K* parametar koji se zadaje kao ulaz algoritma, završavamo *walk* fazu, koja rezultuje u vektoru *W* , koji sadrži skupove čvorova posećenih u okviru svakog walk-a, kao i odgovarajuće brojače, koji opisuju koliko je puta svaki od čvorova posećen u okviru walk-a. Ova informacija ukazuje na "važnost" čvora u okviru datog walk-a.

Dobijeni skup walk-ova posmatramo kao elemente potencijalnih *cluster*-a. Međutim, postoje čvorovi koji su posećeni u okviru dva ili više walk-ova, tako da je neophodno obaviti *merge* fazu algoritma, u okviru koje se na osnovu informacija o "važnosti" svakog čvora u okviru datog walk-a, obavlja jedna od dve definisane operacije : (1) *cut* – čvor se izbacuje iz posmatranog walk-a, ukoliko postoji walk u okviru koga se dati čvor javlja sa većim indeksom i (2) *merge* – ukoliko postoji više walk-ova u okviru kojih se dati čvor javlja sa istim indeksom, vrši se njihovo spajanje po posmatranom čvoru.

Konačan rezultat izvršavanja algoritma predstavlja skup svih *preostalih* walk-ova, koji predstavljaju *cluster*-e u posmatranom grafu.

3.5. Analiza

Osnova ideja korišćenja random walk-ova za obavljanje funkcije clustering-a, zasniva se na činjenici da *random walk na grafu ima tendenciju da ostane unutar cluster*-a. Ovo se može intuituvno zaključiti, uzevši u obzir formulaciju *cluster*-a, kao jako povezanog podgrafa. Ukoliko se random walk nađe unutar cluster-a, najveća je verovatnoća da će dostići *stanje zaustavljanja* upravo unutar cluster-a, budući da se najveći broj grana upravo završava unutar cluster-a.

Empirijska analiza ukazuje da se prosečna *dužina* walk-a reda veličine $O(\log N)$, gde je N, broj čvorova posmatranog grafa. Prema tome, možemo zaključiti da minimalna vrednost parametra K, mora iznositi $\frac{N}{\log N}$. Ova vrednost parametra K, predstavlja minimalnu vrednost koja omogućava *pokrivanje* svih čvorova grafa. Međutim, ova pretpostavka bi važila samo u slučaju da imamo *povezan* graf, u kome postoji mogućnost realizacije random walk-a zadate dužine, počevši iz bilo kog slučajno izabranog čvora. U praksi, *indukovani grafovi*, nad kojima želimo da obavimo operaciju *clustering*-a, se obično sastoje od većeg broja nepovezanih komponenti (do ekstremnih slučajeva, koji se sastoje od N komponenti - kod kojih ne postoji ni jedna grana između bilo koja dva čvora).

Parametar K, koji može uzimati vrednosti iz opsega $\left(\frac{N}{\log N}, N\right)$, nazivamo *stepen aproksimacije*, upravo iz razloga što za najmanju moguću vrednost omogućava *clustering*, samo za slučaj *jako povezanih* grafova, dok za najveću vrednost, omogućava *clustering,* za slučaj bilo koje topologije grafa, uključujući i graf koji ne sadrži ni jednu granu.

Na ovaj način, promenom parametra K moguće je proizvoljno regulisati kompleksnost *WALK* faze algoritma, koja iznosi $O(K \log N)$ i u zavisnosti od vrednosti K, kompleksnost datog algoritma se kreće u opsegu između $O(N)$ i $O(N^2)$.

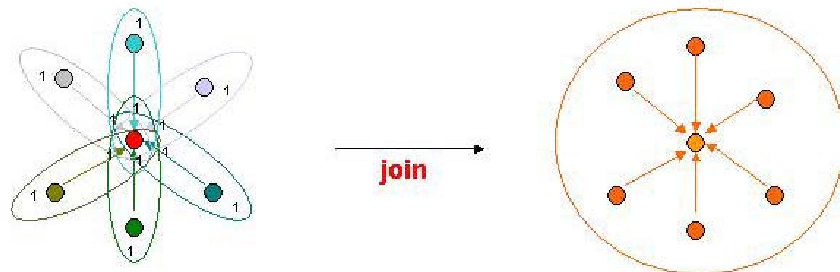
Veće vrednosti parametra K, omogućavaju veću preciznost algoritma, međutim, budući da je očekivanje broja čvorova koji se obilaze u walk fazi $N \log N$, očigledno je da će se najveći broj *walk*-ova preklapati, odnosno postojaće čvorovi, koji se nalaze u skupu čvorova više walk-ova.
Cilj *clustering* operacije jeste podela skupa čvorova posmatranog grafa, na skupove međusobno disjunktnih čvorova, koji predstavljaju *cluster*-e u datom grafu. Iz tog razloga, neophodno je implementirati *MERGE* fazu algoritma, u okviru koje se vrši *reorganizacija* dobijenih walk-ova, u cilju kreiranja međusobno disjunktnih skupova, koji predstavljaju *cluster*-e.

Osnovna ideja implementacije predstavljene *MERGE* faze algoritma se zasniva na činjenici da *random walk* ima tendenciju da *otkriva* "prirodno" dominantne elemente unutar *cluster*-a, tako što dominantnije elemente posećuje *češće*. Na taj način, informacije o broju posećivanja svakog čvora u okviru odgovarajućeg walk-a imaju tendenciju da aproksimiraju *stacionarnu raspodelu* Markovljevog lanca koji opisuje dati walk (kao kod PageRank algoritma) i samim tim, čvorovi sa većim indeksom imaju tendenciju da budu češće posećeni od strane "slučajnog posetioca", odnosno, predstavljaju dominante čvorove u odnosu na zadati cluster.

Na osnovu ove pretpostavke, formulišemo procedure *cut* i *join*, koje se zasnivaju na principu da dobijeni cluster-i, moraju da budi optimizovani po dva kriterijuma : *veličini* (favorizuju se *cluster*-i većih dimenzija nad manjim) i *relevantnosti* (svaki *cluster* treba da ima maksimalnu relevantnost, tj. da sadrži čvorove koji su najrelevantniji upravo u odnosu na zadati cluster).

Shodno ovim zahtevima, definišemo jednostavnu heuristiku koja omogućava njihovo ostvarivanje. U okviru *cut* operacije, vršimo eliminisanje čvorova iz walk-ova u kojima se javljaju sa manjim indeksom (nastojimo da ostanu u clusterima za koje su najrelevantniji), dok u okviru *join* operacije, nastojimo da grupišemo skupove čvorova u okviru kojih se posmatrani čvor javlja sa jednakim indeksom, nastojeći da dobijemo cluster većih dimenzija, koji je orijentisan oko jako relevantnog zajedničkog čvora.

Dodatno, opisana *join* operacija je izuzetno efikasna u grupisanju "*patoloskih*" grafova, kao što je npr. graf na levoj strani, *slike 4*, koji predstavlja skup čvorova, koji imaju samo jedan zajednički čvor. Ukoliko pretpostavimo vrednost parametra aproksimacije $K = N$, koliko je neophodno za slučaj potpuno nepovezanih grafova (odnosno $K = N - 1$, za posmatrani graf, budući da postoji po jedna grana), pod pretpostavkom uniformnosti generatora slučajnih brojeva, svaki od čvorova će biti odabran samo jednom, međutim usled strukture samog grafa, dužina svakog od walk-ova će biti jednaka 2. Na ovaj način, random walk bi "*otkrio*" $N - 1$, "*prirodnih*" cluster-a, što ne odgovara onome što se očekuje od clustering algoritma. Međutim, u okviru *join* faze, uzevši u obzir da se u svakom walk-u, zajednički čvor javlja sa indeksom 1, primenom navedenog principa o kreiranju *cluster*-a većih dimenzija, svi navedeni walk-ovi se spajaju oko zajedničkog čvora i formiraju jedan cluster, prikazan na desnoj strani *slike 4* .



slika 4 : generisanje cluster-a korišćenjem join operacija

3.6. Dalja istraživanja

Opisani algoritam predstavlja dobru osnovu za dalje istraživanje u oblasti primene *slučajnih lutanja na grafu*, na problem *grupisanja* čvorova grafa. Opisana analiza se zasniva na čisto empirijskim činjenicama i neophodno je razviti teorijsku osnovu za dalju analizu specifičnih parametara algoritma. Od interesa su dokazivanje matematičkog očekivanja dužine walk-a, kao i teorijsko modelovanje samog *slučajnog procesa* na kome se zasniva algoritam. Međutim, navedeni zadaci nisu nimalo trivijalni, prvenstveno iz razloga što zahtevaju dublju analizu problema slučajnih lutanja na *usmerenim*, *slabo povezanim grafovima,* kakav je indukovani podgraf koji posmatramo. U ovom slučaju, nije moguće primeniti standardne rezultate vezane za konvergencije slučajnih lutanja na *neusmerenim* grafovima, već je neophodno uključiti dublju analizu *generativnih modela* samih grafova (slučajnih procesa, koji opisuju proces nastanka samog grafa). Dodatno, neophodno je analizirati realne slučajeve indukovanih grafova, na korpusima dokumenata znatno većim od posmatranog, uočiti empirijske pravilnosti i razviti odgovarajući generativni model. Na tako dobijenom modelu, pored teorijske analize, moguće je realizovati i simulacionu analizu opisanog procesa i analizirati njegove performanse.

3.7. Implementacija

Navedeni algoritam, može biti implementiran kao *offline* ili *online* algoritam. Kao *offline*, algoritam uzima kao ulaz matricu povezanosti A, dimenzija $N \times N$, gde je N, broj čvorova u posmatranom grafu i kao rezultat daje listu walk-ova koji odgovaraju finalnim *cluster*-ima.

Dajemo primer implementacije algoritma na jeziku Java :

```
// WALK faza

Random rnd = new Random();
LinkedList clusters = new LinkedList();

for (int i=0; i<K; i++) {
      int cnt = 0;
      s = rnd.nextInt(K);
      HashMap walk = new HashMap();
      walk.put(s,1);
      while(true) {
            if (cnt++ > N) break;
            v = rnd.nextInt(K);
            if (adj[s][v] == true) {
                  if (!walk.containsKey(v)) walk.put(v);
                  else {
                        int t = (Integer)walk.remove(v);
                        walk.put(v,++t);
                  }
      }
      clusters.add(walk);
}

// MERGE faza

ListIterator it1 = clusters.listIterator(0);
while(it1.hasNext()) {
      HashMap walk = (HashMap)it1.next();
      ListIterator it2  = clusters.listIterator(0);
      while(it2.hasNext()) {
            HashMap tmp = (HashMap)it2.next();
            Iterator it3 = walk.keySet().iterator();
                  while(it3.hasNext()) {
                        int key = (Integer)it3.next();
                        int val1 = (Integer)it1.get(key);
                        int val2 = (Integer)tmp.get(key);
                        if (val2>val1)it1.remove(key);
                        else if(val2<val1)tmp.remove(key);
                        else {
                              join(walk,tmp);
                              it2.remove();
                        }
                  }
      }
}
```

U okviru opisane implementacije, *walk*-ovi, odnosno *cluster*-i se predstavljaju korišćenjem hash tabela (*HashMap*), kod kojih *key* predstavlja broj čvora, dok *value* predstavlja brojač poseta čvoru u okviru walk-a (ova reprezentacija se koristi radi optimizacije čestog izvršavanja operacije pretraživanja *walk*-a za zadatim čvorom – koja se kod *hash* tabela izvršava u vremenu približnom $O(1)$). Skup svih walk-ova, predstavlja se kao ulančana lista (*LinkedList*). Pristup elementima liste realizovan je preko *Iteratora*, koji znatno uprošćavaju samu realizaciju, budući da omogućavaju dinamičko menjanje veličine liste. U okviru merge faze, koristimo dva iteratora, jedan za predstavljanje walk-a koji trenutno ispitujemo i drugi za pristup svim ostalim elementima liste. Za svaki element walk-a, ispitujemo da li se nalazi u nekom od preostalih walk-ova i u zavisnosti od njegovog indeksa, vršimo izbacivanje elementa iz liste u kojoj se javlja sa manjim indeksom, ili u slučaju da se javlja sa jednakim indeksom, pozivamo pomoćnu proceduru *join (destination, source)*, koja u *destination* walk (HashMap), ubacuje sve elemente *source* walk-a, tako da, u slučaju da postoji jos identičnih elemenata, u novonastali walk ubacuje one sa većim indeksom. Nakon završetka ove operacije, iz liste walk-ova izbacujemo *destination* walk, budući da se svi njegovi elementi nalaze u okviru novonastalog walk-a.

Konačno, skup walk-ova koji su ostali u listi, nakon završetka *merge* operacije, predstavlja finalan skup cluster-a.

Dodatno, opisani algoritam se može implementirati i kao *online* algoritam, bez korišćenja matrice povezanosti, u okviru direktnog čitanja iz indeksa. Dajemo prikaz jednostavne implementacije ovog algoritma na jeziku *Java*, za slučaj korišćenja *Lucene* indeksa. Ulaz u algoritam predstavlja *HashMap urls*, koja predstavlja listu url-ova, koji se dobijaju kao search rezultati za zadati *upit* :

```
// WALK faza

LinkedList clusters = new LinkedList();
LinkDbReader linkdb = new LinkDbReader(fs,dir,conf);

for(int i=0; i<K; i++) {
      int cnt = 0;
      String url = getRandomUrl(urls);
      HashMap walk = new HashMap();
      walk.add(url,1);
      while(true) {
            if(cnt++ > N) break;
            inlinks = linkdb.getInlinks(url);
            url = getRandomUrl(inlinks);
            if(!walk.containsKey(url)) walk.put(url);
            else {
                  int t = (Integer)walk.remove(url);
                  walk.put(url,++t);
            }
      }
      clusters.add(walk);
}

// MERGE faza izgleda isto kao i kod offline algoritma
```

Primećujemo da, budući da koristimo HashMap za reprezentaciju walk-a, algoritam izgleda identično, bez obzira da li čvorove predstavljamo brojevima ili URL-ovima.

3.8. Zaključak

U ovom poglavlju, dat je prikaz, inovativnog algoritma za grupisanje rezultata pretraživanja. Doprinosi navedenog pristupa, ogledaju se u uvođenju pojma *indukovanog pogdrafa*, u odnosu na zadati *upit* i formulacije problema grupisanja rezultata u vidu problema grupisanja čvorova datog podgrafa. Dodatno, predložen je randomizovani algoritam koji omogućava efikasno grupisanje čvorova, korišćenjem *slučajnih lutanja* na grafu, čija se kompleksnost može menjati u zavisnosti od potrebe, korišćenjem uvedenog *parametra aproksimacije*. Za razliku od poznatih algoritama, posmatrani algoritam je orijentisan ka *sparse* grafovima, koji se sastoje od više nepovezanih komponenti (kakvi su u praksi posmatrani *indukovani* podgrafovi). Za posmatranu klasu grafova, dati algoritam pokazuje rezultate približne rezultatima *state-of-the-art* algoritama za *graph clustering*, ali je njegova kompleksnost niza i može se regulisati u zavisnosti od potrebe, što ga čini idealnim za primenu u sistemima, čija prvenstvena namena nije clustering. Pored navedenog, predstavljena su i dva načina (*offline* i *online*) implementacije datog algoritma, na jeziku Java. U sledećem poglavlju, biće prikazan pretraživač opšte namene, u okviru koga je upravo opisani algoritam implementiran u cilju pružanja funkcionalnosti grupisanja rezultata pretraživanja.

4. randomNode : Implementacija clustering web pretraživača

U okviru diplomskog rada, razvijen je i softverski projekat, pod imenom *randomNode*, koji predstavlja celokupnu implementaciju *Web pretraživača* opšte namene. Sam projekat se zasniva na open source rešenjima *Nutch* i *Lucene*, opisanim u *poglavlju 2*. , koji obezbeđuju platformu za realizaciju *crawl*, *indexing* i *ranking* funkcionalnosti. Dodatno, u okviru projekta, razvijen je aplikacija na jeziku *Java*, koja se sastoji od dve komponente : *klijentske aplikacije*, koja prihvata korisničke upite i prikazuje rezultate i *serverske aplikacije*, koja omogućava proizvoljno pretraživanje *Lucene* indeksa, i u okviru koje je implementiran opisani algoritam za grupisanje dobijenih rezultata pretraživanja.

U nastavku teksta, opisujemo neke od detalja implementacije *Nutch/Lucene* search platforme, kao i detalje razvoja i implementacije same aplikacije

Crawling

Osnovni cilj projekta predstavlja demonstracija implementacije web pretraživača, koji bi bio u stanju da efikasno pretražuje značajan segment Web-a, koristeći limitirane resurse. Time shodno, pretraživač je implementiran samo na jednom računaru (Intel Celeron 2.4 Ghz, 512MB RAM, 2x40GB HDD), koristeći jako limitirane mrežne resurse (Home ADSL konekcija, 256Kb download / 64Kb upload kapaciteta).

Budući da je nemoguće tačno odrediti broj stranica na .yu domenu, najbližu referencu predstavlja veličina indeksa najpoznatijih pretraživača Srpskog govornog područija (www.srpko.com prijavljuje veličinu indeksa od oko 8 miliona stranica, dok www.pogodak.com prijavljuje veličinu od preko 10 miliona stranica). Međutim, ove cifre predstavljaju ukupan broj *svih* stranica na Web-u, koje uključuju sadržaj na Srpskom jeziku. Po nekim procenama, broj dokumenata iz *.yu* domena se kreće između 5 i 8 miliona.

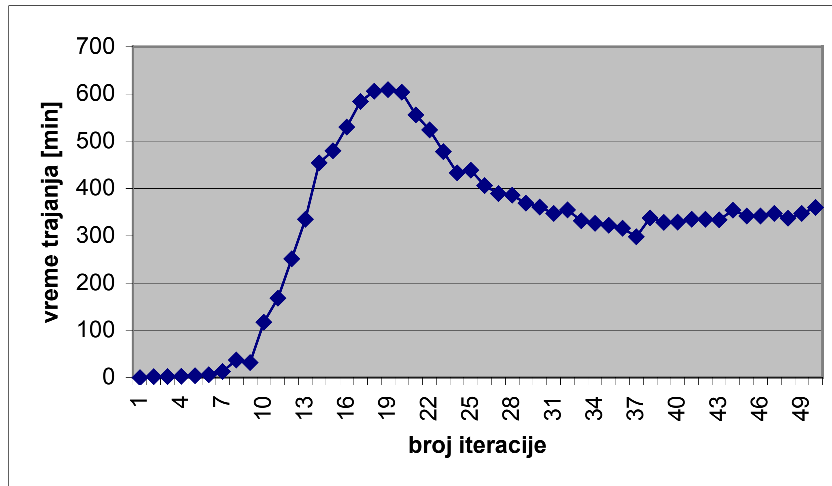
Kao cilj *crawling* procesa, postavili smo indeksiranje *1 miliona stranica*, u vremenskom periodu od 14 dana. Radi analize ponašanja crawler-a, sam proces započinjemo samo od jedne stranice : http://www.etf.bg.ac.yu. Od interesa je ponašanje crawler-a u smislu brzine rasta, konvergencije i količine fetch-ovanih novih stranica u svakoj iteraciji.

Dubina pretraživanja je ograničena na 50 (najviše se prati 50 linkova po dubini), pri čemu je maksimalan broj stranica na svakom nivou praktično neograničen (odnosno, ograničen na 100000 stranica). Maksimalna veličina stranice koja se fetch-uje je ograničena na 64K. Implementirana je *pristojnost* u ponašanju, tako što je period između dva zahteva ka istom serveru ograničen na *5 sekundi*. Sam crawler se izvršava u vidu maksimalno 40 *thread*-ova, pri čemu je u svakom trenutku dozvoljen samo jedan *thread* po istom host-u.

Navedeni parametri imaju za cilj da obezbede završavanje crawl procesa u okviru zadatog vremenskog perioda. Postojeći kapacitet link-a je u stanju da obezbedi maksimalnu količinu prenetih podataka od 36.9Gb, u toku datog perioda. Međutim, očekivana količina podataka je niža, uzevši u obzir da se sam proces obavlja na jednom računaru, usled čega se značajan deo raspoloživog vremena koristi za indeksiranje (za vreme koga bi bilo moguće vršiti fetch, ali se usled limitiranih procesorskih resursa, ovo ne radi, uzevši u obzir zahtevnost procedure indeksiranja). Uzevši u obzir srednju veličinu stranice od oko 15Kb, očekivani korpus od milion stranica bi trebao da zauzima oko 14.3 Gb.

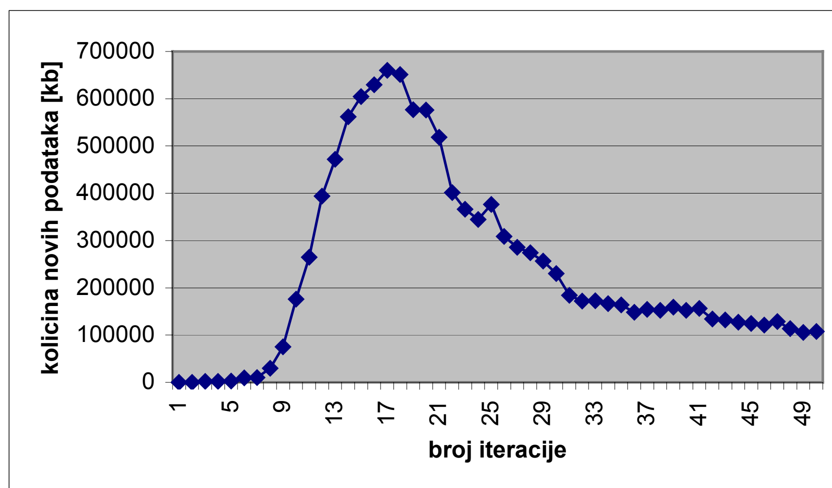
U nastavku teksta predstavljamo i analiziramo rezultate rada samog crawler-a i dimenzije dobijenog korpusa dokumenata.

Na *slici 5*, prikazano je vreme potrebno za obavljanje svakog od *koraka* pretraživanja, pri čemu svaki *korak* podrazumeva ciklus *fetch*-ovanja stranica na odgovarajućem nivou dubine linkova, dok je na grafikonu y prikazana količina novih stranica, dobijena kao rezultat obavljanja svakog od navedenih *koraka.*



slika 5 : vreme trajanja svakog od koraka crawl procedure

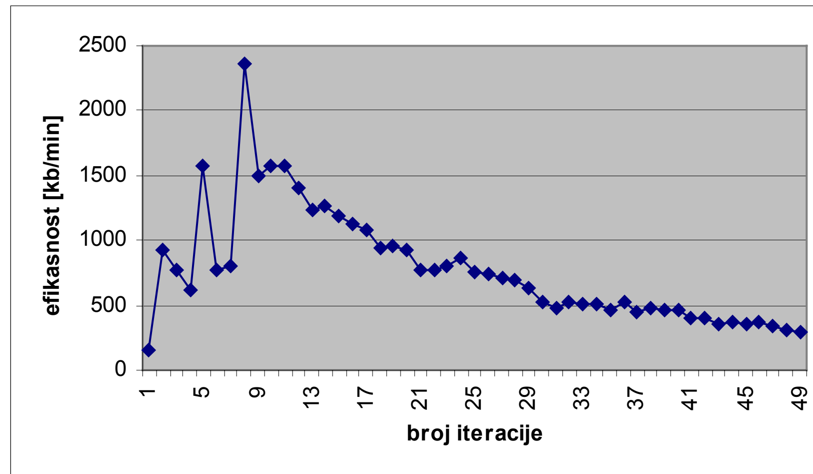
Primećujemo da je, na početku, vreme obavljanja svakog koraka, veće od prethodnog. Ovo je uzrokovano činjenicom da stranice fetch-ovane u svakom koraku imaju *izlazni stepen* veći od 1, odnosno ukazuju na više stranica, koje se smeštaju u *fetch queue,* koji na ovaj način stalno raste. Međutim, usled restrikcije *crawl* procedure samo na stranice iz *.yu* domena, nakon 20-te iteracije, stopa rasta se smanjuje, upravo usled činjenice da sve više stranica sadrži linkove ka stranicama van *.yu* domena, kao i linkove ka stranicama, ciji je sadržaj već fetch-ovan.



slika 6 : količina novih podataka fetch-ovana u svakom koraku

Na *slici 6*, primećujemo da u periodu koji odgovara segmentu rasta *fetch queue*-a, raste i količina novih stranica koje se unose u indeks. Međutim, za crawl na dubinama većim od 20, primećujemo da količina novih informacija značajno opada, kako usled smanjenja veličine queue-a, tako i usled javljanja sve većeg broja duplikata stranica koje već postoje u indeksu.

Navedeni problem, možemo posmatrati uvođenjem pojma *efikasnosti* crawl procedure, koji možemo definisati kao odnos količine novih informacija koje se unose u indeks i vremena utrošenog za njihovo *fetch*-ovanje. Za posmatrani crawl, na *slici 7* , predstavljamo vrednost funkcije *efikasnosti (izražene u kb/min)*, za svaki od koraka posmatranog *crawl*-a.



slika 7 : efikasnost crawl procedure u svakom koraku

Na osnovu dobijenog grafikona, primećujemo da već nakon nivoa dubine *10*, efikasnost crawl procedure počinje da opada. Posmatrajući ovaj i prethodne grafike, zaključujemo da se maksimalne performanse dobijaju za *crawl*-ove, koji su dubina manjih od *25*. Sve vrednosti preko ove predstavljaju loše iskorisćenje resursa. Prema tome, zaključujemo da bi se bolji rezultati (u smislu brzine i veličine indeksa), postigli ukoliko bi koristili strategiju, koja umesto što počinje od jednog url-a i crawl-uje do dubine 50, počinje od dva nezavisna (dijametralno suprotna, u smislu web grafa) url-a i vrši crawl do dubine 25.

U opštem slučaju, zaključujemo da se optimalne performanse crawler-a postižu, primenom crawl strategije u okviru koje se na osnovu nekog poznatog *direktorijuma* Web stranica (kao što je dmoz), generiše skup *k* inicijalnih stranica (*seed set*), slučajnim izborom *k* stranica iz (dmoz) indeksa, nakon čega se iz svake nezavisno počinje *crawl*, koji prati linkove najviše do dubine *10*.

Indeksiranje

Nakon završetka opisanog crawl procesa (koji je trajao oko *12 dana*), dobijen je indeks, koji sadrži **883868** URL-ova, od kojih je **629974** fetch-ovano, dok **152142** nije fetch-ovano. Ukupna veličina fetch-ovanih stranica na disku je oko *12GB*, dok je veličina celokupnog indeksa *16.6GB*.

Iako je veličina dobijenog indeksa (*880 hiljada stranica*), nešto manja od očekivane (*1 milion*), procenjujemo da je korišćenje prethodno opisane poboljšane strategije crawl-ovanja, moglo da rezultuje u indeksu veličine do *1.2 miliona stranica*, koristeći iste sistemske resurse.

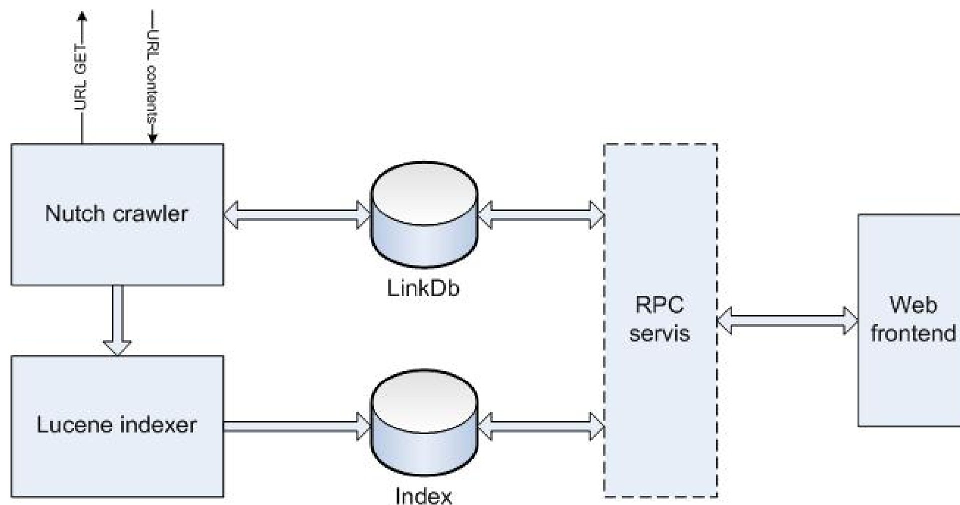
Dobijeni rezultati su izuzetno značajni, budući da ukazuju na to da je za korišćenje "*kućnih*" resursa, moguće obaviti crawl značajnog segmenta Web-a (kao što je *.yu* domen) u vremenskom periodu koji je manji od *mesec dana* (pod uslovom korišćenja 768/196 ADSL linka).

Search/Clustering

Za razliku od *crawl* i *indexing* funkcionalnosti koje su implementirane korišćenjem *open source* paketa *Nutch* i *Lucene*, funkcionalnosti pretraživanja i grupisanja rezultata su razvijene kao samostalan projekat.

Osnovna ideja projekta je realizacija inovativnog search okruženja koje bi omogućilo korisnicima pretraživanje na način koji nije moguć, korišćenjem konvencionalnih pretraživača. Prvenstveni cilj je okrenutost ka korisnicima koji su zainteresovani ka *istraživanju*, pre nego *pretraživanju* Web-a. Najveći problem konvencionalnih pretraživača je u tome što se za svaki upit, na jednoj stranici servira relativno mali (uglavnom *10*) rezultata. Glavni cilj projekta, predstavlja implementacija pretraživača koji omogućuje prikazivanje znatno većeg broj (*100* i više) na svakoj stranici sa rezultatima, na način koji omogućava efikasno snalaženje i percepciju dobijenih rezultata.

Razvijeno rešenje se sastoji od dve komponente : *Web frontend-a i Search/Clustering RPC servisa, kao što je prikazano na slici :*



slika 8 : arhitektura randomNode pretraživača

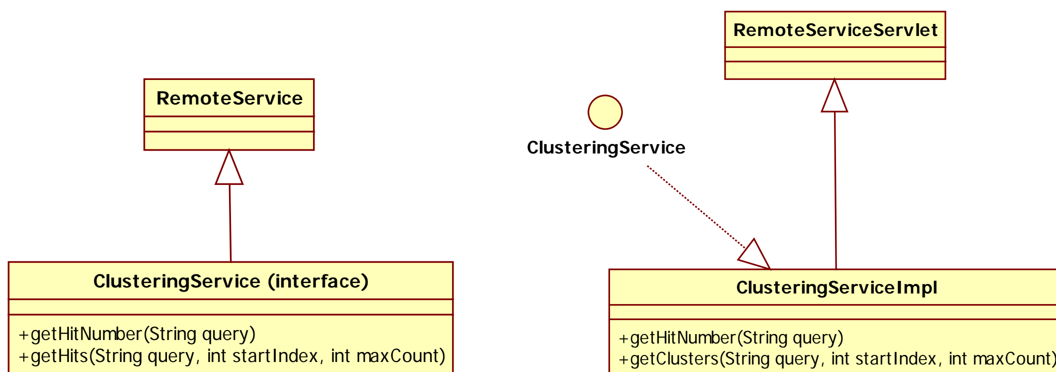
Web frontend je razvijen kao *Ajax* aplikacija, na *Java* jeziku, uz korišćenje *GWT* kompajlera. U razvoju, prednost je data ovom vidu rešenja, nad uobičajenim jsp/servlet rešenjima, upravo iz razloga kreiranja *interaktivnog* okruženja koje omogućava real-time *istraživanje* podataka. Samo okruženje je interaktivno, koje u zavisnosti od korisničkih *akcija*, *asinhrono* komunicira sa serverom i pruža dodatne informacije, bez osvežavanja stranice, čime se ostvaruje željeni cilj prikazivanja što većeg broja rezultata na jednoj search stranici (prikazuju se samo naslovi stranica, dok se detalji vezani za sadržaj stranice prikazuju *dinamički*, u vidu pop-up prozora, kada korisnik postavi fokus na zadati link).

RPC servis implementiran kao standardan *Java servlet*, implementiran na *Tomcat 5.5 servlet container-u*. Osnovne funkcije datog servisa su prihvatanje upita poslatih od strane *frontend* aplikacije, pretraživanje *Lucene* indeksa i ekstrakcija rezultata. Nakon dobijanja rezultata, dodatno se obavlja njihovo grupisanje, koristeći algoritam opisan u *poglavlju 3*. Tako grupisani rezultati, vraćaju se nazad *frontend* aplikaciji. Aplikacija dobija rezultate upita, grupisane u odgovarajući broj *cluster*-a i vrši njihovo raspoređivanje i prikazivanje.

Softverska arhitektura

RPC Servis

RPC Servis je realizovan u vidu *ClusteringServiceImpl* servlet-a (nasleđuje *RemoteServiceServlet*), u okviru koga su implementirane metode *getHitNumber,* koja na osnovu zadatog upita, pretražuje *Lucene* indeks i vraća broj stranica u indeksu koje odgovaraju upitu, kao i metoda *getClusters* , koja na osnovu zadatog upita i rezultata pretraživanja *indeksa*, vrši pretraživanje *LinkDb* baze, u cilju kreiranja *indukovanog podgrafa,* nad kojim se zatim primenjuje algoritam za grupisanje rezultata, opisan u *poglavlju 3*. Konačno, funkcija vraća kao rezultat niz rezultata, grupisan u odgovarajuće *cluster*-e.

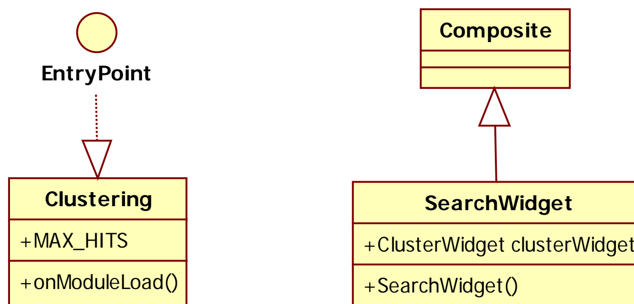


slika 9 : komponente RPC servisa

Web Frontend

Web frontend je razvijen u vidu GWT aplikacije, na jeziku *Java*, koji se zatim kompajlira u odgovarajuće HTML, JavaScript i XML komponente, dobijene *AJAX* aplikacije .

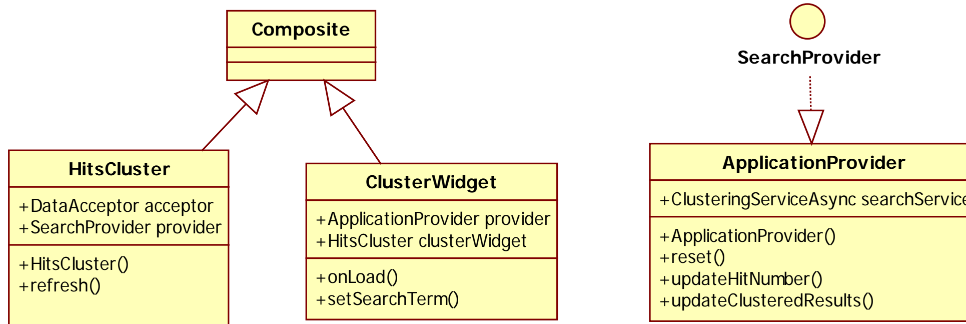
Po učitavanju projekta, prvo se instancira *Clustering* objekat (implementira *EntryPoint* interfejs), u okviru koga se vrši instanciranje *SearchWidget* i *ClusterWidget* widget-a.



slika 10 : EntryPoint web frontend-a

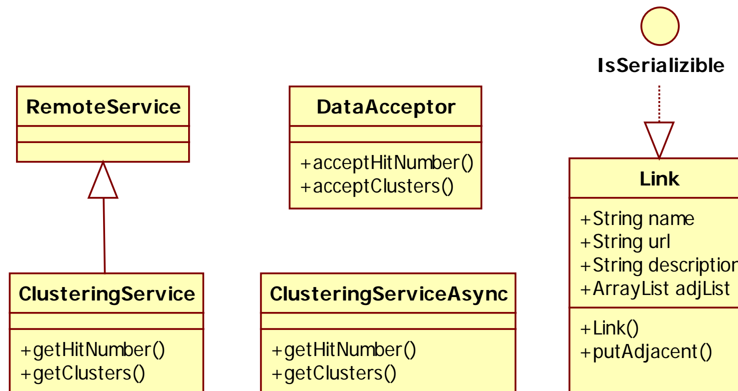
U okviru *Clustering* objekta, vrši se nalaženje odgovarajućih elemenata u *DOM* stablu odgovarajuće html stranice u okviru kojih će biti instancirani odgovarajući widget-i (*SearchWidget* u *search* tabelu, *ClusteringWidget* u clustering tabelu). U okviru *SearchWidget*-a kreiraju se odgovarajuća *searchbox* polja i dodaju *keyboard listener*-i koji nakon unosa teksta (korisničkog upita), vrše instanciranje glavnog *ClusteringWidget*-a.

ClusterWidget, pri instanciranju, vrši instanciranje *ApplicationProvider* objekta, koji ubuduće koristi za pristup definisanom RPC servisu. Dodatno, vrši instanciranje *HitsCluster* objekta, pri čemu kao parametar, predaje instancirani *ApplicationProvider* objekat. U okviru *HitsCluster* objekta, na osnovu zadatog upita, koristi se *provider* objekat sa slanje upita i primanje odgovora RPC servisa.



slika 11 : komponente za grupisanje i prikaz rezutata

Za komunikaciju sa RPC servisom, na strani klijenta, neophodno je definisati interfejs koji opisuje servise koje pruža željeni servlet (*ClusteringService*), kao i odgovarajući *callback* objekat (*ClusteringServiceAsync*), koji se predaju pozivima odgovarajućih *provider* metoda. Poruke između aplikacije i servleta se razmenjuju u vidu niza *Link* objekata, od kojih svaki sadrži sve informacije o jednom *search* rezultatu. Iz *callback* objekta, konačno se poziva *DataAcceptor* objekat, koji na osnovu dobijenih rezultata, vrši kreiranje odgovarajućih cluster-a, njihovo raspoređivanje i prikaz na ekranu.



slika 12 : komponente za pristup RPC servisu

Izgled glavnog ekrana, dela aplikacije koji implementira osnovne zahteve pretraživanja, prikazan je na *slici 13*. Za razliku od klasičnih pretraživača, radi ostvarivanja prikaza što većeg broj rezultata na jednoj stranici, prikazuju se samo naslovi, dok se ostale informacije (url, summary..), dobijaju u vidu popup prozora, kada korisnik postavi fokus na konkretni rezultat. Dodatno, pri postavljanju fokusa na određeni rezultat, vrši se *higlighting* svih ostalih rezultata, koji sadrže *hiperlink* ka posmatranoj stranici. Na ovaj način se ostvaruju osnovni zadaci projekta – prikaz što većeg broja rezultata na jednoj stranici, pri čemu *inlink higlighting,* dodatno omogućava bolji uvid u strukturu i povezanost rezutltata na ekranu.

slika 13 : randomNode pretraživač

Konačno, na *slici 14* , prikazujemo kompletnu implementaciju pretraživača, koji pored navedenih, uključuje i funkcionalnost grupisanja rezultata pretraživanja, korišćenjem algoritma razvijenog u okviru diplomskog rada.

slika 14 : randomNode pretraživač, sa grupisanjem rezultata

5. Zaključak

U okviru ovog diplomskog rada, prikazani su osnovni algoritamski aspekti razvoja i implementacije Web pretraživača. Obrađeni su osnovni problemi, specifični za Web search problematiku, analizirane su njihove posledice i predstavljena odgovarajuća rešenja. Dodatno, predstavljena je platforma za samostalnu implementaciju osnovnih funkcionalnosti jednog web pretraživača. U okviru rada, razmotrena je teza da je na današnjem stadijumu razvoja tehnologije, čak i u "*kućnim*" uslovima, moguća implementacija pretraživača koji bi bio u stanju da indeksira i pretražuje *značajan* segment Web-a. Razmotreni su potencijalni razlozi za razvoj jednog ovakvog pretraživača, od kojih je najznačajniji razvoj i analiza novih algoritama za obradu podataka sa Web-a. Za cilj celokupnog diplomskog rada, postavljena je implementacija Web pretraživača, koji bi implementirao inovativni interfejs koji omogućava lakše *istraživanje* određenih segmenata Web-a. Jedna od osnovnih metoda koja omogućava lakši pristup i istraživanje podataka, predstavlja primena *clustering* algoritama. Razmotreni su neki od postojećih algoritama za clustering i uočeni neki od nedostataka u njihovoj primeni na specifične tipove search problema. Time shodno, u *poglavlju 3*, predstavljen je inovativan algoritam, koji omogućava efikasno grupisanje rezultata pretraživanja, koji koristi strukturu *indukovanog podgrafa*. Pokazuje se da opisani algoritam ostvaruje rezultate približne poznatim *graph clustering* algoritmima, ali ostvarujući nizu kompleksnost izračunavanja i omogućujući proizvoljnu regulaciju odnosa kvalitet/performanse, usled čega se pokazuje kao izuzetno pogodan za implementaciju u pretraživačima opšte namene.

Predstavljena je celokupna implementacija jednog Web pretraživača, koji se zasniva na navedenoj crawl/indexing *open source* platformi, uz samostalnu implementaciju interfejsa za pretraživanje i clustering. Razmatramo detalje implementacije samog pretraživača, počevši od implementacije samog crawl procesa, analiziramo ponašanje i uočavamo probleme u radu, koji se odrazavaju u padu efikasnosti sa porastom opsega posla, i predlažemo strategiju za rešavanje navedenih problema. Na kraju, predstavljamo i detalje softverske arhitekture realizovanog search/clustering interfejsa i analiziramo rad pretraživača u celini.

Osnovni motiv samog rada jeste postavljanje, kako teorijske, tako i praktične osnove za dalji razvoj i istraživanje u oblasti Web algoritama. Realizovana platforma, može posluziti kao osnov za dalje istrazivanje, razvoj i testiranje inovativnih algoritama za obradu i istrazivanje podataka na Web-u, ali i za razvoj konkretnih aplikacija, koje uključuju Web search kao jednu od komponenti.

6. Reference

1. Serge Abiteboul , Mihai Preda, Gregory Cobena *: Adaptive On-Line Page importance Computation*, Proceedings of International World Wide Web Conference, Budapest, Hungary, 2003.
2. L.A. Adamic, R.M. Lukose. A.R. Puniyani, B.A. Huberman *: Search in power-law networks*, Physical Review, volume 64, 2001.
3. Ricardo Baeza-Yates *: Crawling a Country : Better Strategies than Breath-First for Web Page Ordering*, Proceedings of International World Wide Conference, Chiba, Japan, 2005.
4. Ziv Bar-Yossef, Maxim Gurevich *: Random Sampling from Search Engine's Index*, Proceedings of International World Wide Web Conference, Edinburgh, Scotland, 2006.
5. Ron Bekkerman, etf al. *: Web Page Clustering using Heuristic Search in the Web Graph*, Proceedings of International Joint Conference on Artificial Intelligence IJCAI-07, Hyderabad, India, 2007.
6. Krishna Bharat, Bay-Wei Chang, Monika Henzinger, Matthias Ruhl *: Who Links to Whom : Mining Linkage between Web Sites*, Proceedings of IEEE International Conference on Data Mining, San Jose, 2001.
7. Monica Bianchini, Marco Gori, Franco Scarselli *: Inside PageRank*, ACM Transactions on Internet Technology, Vol. 5. Issue 1., 2005.
8. Ulrik Brandes, Marco Gaertler, Dorotea Wagner *: Experiments on Graph Clustering, Algorithms*, Lecture Notes in Computer Science, Di Battisa and U. Zwich (Eds.), 2003.
9. Amit Chakrabarti *: Randomized Graph Partitioning Algorithms*, Technical Report, Indian Instute of Technology, Bombay, 1997.
10. Deepayan Chakrabarti, Christos Faloutsos, *Graph Mining : Laws, Generators and Algorithms*, ACM Computing Surveys, Vol. 38, Issue 1, 2006.
11. Colin Cooper, Alan Frieze *: Crawling on Simple Models of Web Graphs*, Internet Mathematics Vol. 1 , No. 1, 2002.
12. Biran Davidson *: Topical Locality in the Web*, Proceedigns of the 23rd Annual International Conference of Research and Development in Information Retrieval (SIGIR 2000), Athens, Greece, 2000.
13. Stijn Van Dongen *: A Cluster Algorithm for Graphs*, technica report, Centruum voor Wiskunde en Informacica, Netherlands, 2000.
14. Gary William Flake, Robert E. Tarjan, Kostas Tsioutsioulikilis *: Graph Clustering and Minnimum Cut trees*, Internet Mathematics, Vol 1. No 4 , 2004.
15. Gary Willian Flake, Steve Lawrence, Lee Giles *: Efficient identification of Web Communities*, Proceedings of sixth ACM SIGKDD international conference on Knowledge dicovery and data mininh KDD '00, Boston, 2000.
16. David Gibson, Jon Kleinberg, Prabhakar Raghavan *: Inferring Web Communities from Link Topology*, Proceedings of 9th ACM Conference on Hypertext and Hypermedia, Pittsburgh, 1998.
17. Parikshit Gopalan, Richard Lipton, Aranyak Mehta *: Randomized Time Space Tradeoffs in Directed Graph Connectivity*, Proceedings Foundations of Software Technolgy and Theoretical Computer Science (FSTTCS) Conference, Mumbai, India, 2003.
18. David Harel, Yehuda Koren *: On Clustering Using Random Walks*, Proceedings of FSTTCS 2001.
19. Monika Henzinger *: Algorithmic Challenges in Web Search Engines*, Internet Mathematics Vol. I, No. 1, 2002.
20. Monika Henzinger, Steve Lawrence *: Extracting knowledge from the World Wide Web*, Proceedings of National Academy of Sciences, April 2004.
21. Monika Henzinger : *Hyperlink Analysis on the World Wide Web* , Proceedings of the sixteen ACM conference on Hypertext and hypermedia HYPERTEXT '05, Salzburg, 2005.
22. Lazslo Lovasz *: Random Walks on Graphs : A Survey*, Combinatorics, Paul Erdos is Eighty, Vol. 2. Keszthely, Hungary, 1993.
23. Ravi Kannan, Santosh Vempala, Adrian Vetta *: On Clustering : Good, Bad and Spectral*, Journal of ACM, Vol. 51, No.3, May 2004.

24. Jon Kleinberg *: Complex Networks and Decentralized Search algorithms*, Proceedings of the International Congress of Mathematicians (ICM), Paris, 2006.
25. Christopher Manning, Prabhakar Ragahvan, Hinrich Schutze *: Introduction to Information Retrieval*, Cambridge University Press, 2007.
26. Michael Mitzenmacher, Eli Upfal : *Probability and Computing, Randomized Algorithms and Probabilistic Analysis*, Cambridge University Press, 2005.
27. Rajeev Motwani, Prabhakar Raghavan *: Randomized Algorithms*, Cambridge University Press, 1995.
28. Lary Page, Sergey Brin, Terry Winograd *: The PageRank Citation Raniking : Bringing Order to the Web*, techical report, Stanford University, 1998.
29. Tomas Sarlos, et al. *: To Randomized or Not to Randomize*, Space Optimal Summaries for Hyperlink Analysis, Proceedings of International World Wide Web Conference, Edinburgh, Scotland, 2006.
30. Satu Virtanen *: Clustering the Chilean Web*, Proceedings of the First Latin American Web Congress (LA-WEB 2003), IEEE Computer Society, 2003.