S V E U Č I L I Š T E U S P L I T U
PRIRODOSLOVNO-MATEMATIČKI FAKULTET

METODE ZA UZORKOVANJE PODRUČNOG ZNANJA U INTELIGENTNIM TUTORSKIM SUSTAVIMA

Marija Vištica

Mentor:

doc.dr.sc. Ani Grubišić

Split, rujan 2014

Sažetak

Područno znanje u inteligentnim tutorskim sustavima prezentirano je usmjerenim acikličkim grafom. Da bi se iz tog grafa izdvojio uzorak koji reprezentira područno znanje koriste se algoritmi za uzorkovanje grafa. U ovom radu prezentiramo pet algoritama za uzorkovanje (slučajna šetnja, Metropolis-Hastings slučajna šetnja, šumski požar, snježna gruda te Represent algoritam) te je ispitano koja su strukturna svojstva grafa sačuvana prilikom uzorkovanja. Tako za uzorke dobivene ovim algoritmima uspoređujemo kumulativne distribucije stupnjeva čvorova, koeficijenata grupiranja te duljina najkraćih puteva u grafu.

Ključne riječi:

inteligentni tutorski sustavi, uzorkovanje, grafovi područnog znanja, područno znanje

Abstract

A domain knowledge in the inteligent tutoring systems is presented with a directed acyclic graph. If you want to seperate sample which represent domain knowlege from that graph, you have to use sampling algorithms. In this master degree we represent five sampling algorithms (Random Walk, Metropolis-Hastings Random Walk, Forest Fire, Snowball and Represent algoritam) and investigate structural properties of the graph which are saved while sampling. For that samples we got with these algorithms, we are comparing cumulative distributions of nodes's degrees, clustering coefficients and the length of the shortest paths in a graph.

Ključne riječi:

Inteligent tutoring systems, sampling, domain knowledge graph, domain knowledge

S a d r ž a j

Popis slika

Popis tablica

1. Uvod

Kroz povijest se mnogo vremena posvetilo ideji o uporabi računala kao osobnog učitelja. Složenost i mogućnost izvršavanja te ideje je porasla nakon što se pojavilo inteligentno poučavanje, odnosno umjetna inteligencija. Tako možemo reći da se inteligentni tutorski sustavi počinju razvijati otprilike kad i umjetna inteligencija, 1950-ih godina, a njihovo poučavanje je jednako zanimljivo sada kao i prije.

Inteligentni tutorski sustavi (ITS) su računalni sustavi zasnovani na tehnikama umjetne inteligencije kako bi simulirali ljudske tutore koji znaju što podučavaju, koga podučavaju i kako podučavaju pa se za njih i kaže kako spadaju u kategoriju sustava temeljenih na znanju.

Znanje koje nosi informacije potrebne za oblikovanje strukture i plana izlaganja nastavnog sadržaja, dijagnosticiranje modela učenika i komunikaciju s učenikom se naziva područno znanje. Problem koji se javlja u ITS-ovima je problem početka učenja. Naime, pitanje je kako odabrati početni skup zadataka od kojeg će učenik nastaviti učenje odnosno problem je inicijalizirati model učenika. Obično se to radi nekakvim inicijalnim testom ili upitnikom koji bi trebao biti reprezentativni primjerak znanja koji učenik treba savladati i koji se prilagođava njegovom dosadašnjem znanju.

U nekim inteligentnim tutorskim sustavima područno znanje je opisano usmjerenim acikličkim grafom, a odabrani skup pitanja ili zadataka je podskup područnog znanja. Znajući to, uzorak možemo generirati poznatim algoritmima za uzorkovanje. U ovom radu ćemo usporediti nekoliko algoritama za uzorkovanje.

Usporediti uzorke iz algoritma znači da ćemo dati „mjeru za dobar uzorak“. U ovom radu uspoređivat ćemo prosječni stupanj čvora, prosječni koeficijent grupiranja, prosječnu duljinu najkraćeg puta između dva čvora, gustoću mreže i broj povezanih i jako povezanih komponenti. Osim toga obratit ćemo pozornost i na distribuciju stupnjeva, koeficijenata grupiranosti i najkraćih puteva te ćemo iste distribucije prikazati grafovima. Zbog jednostavnosti prikazivanja, grafom ćemo prikazati tzv. kumulativnu distribuciju.

U sljedećem poglavlju objasnit ćemo inteligentne tutorske sustave i njihovu arhitekturu. U trećem poglavlju objasnit ćemo pristupe određivanja reprezentativnog podskupa područnog znanja, tj. uzorkovanje te ćemo posebnu pažnju obratiti na kompleksne mreže i na algoritme tj. metode koje ćemo koristiti pri uzorkovanju. U četvrtom poglavlju navest ćemo tehnologiju kojom smo implementitrali zadane algoritme i detaljnije objasniti implementaciju da bih u zadnjem, petom poglavlju usporedili iste.

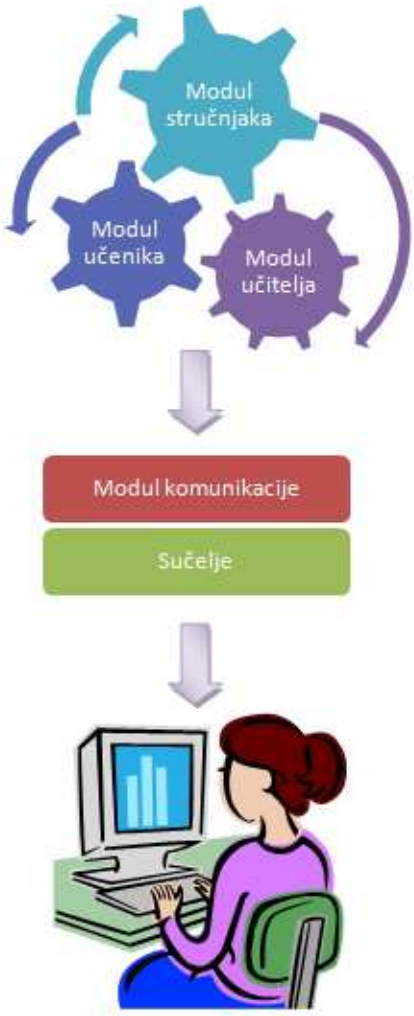
2. Inteligentni tutorski sustavi

Računalo je konstruirano kako bi obrađivalo podatke na osnovu programski opisanih algoritama. Rezultati računalne obrade podataka su informacije koje se razlikuju od pukih podataka po tome što imaju značenje. Uvođenjem značenja podaci postaju informacije, a grupiranjem specijaliziranih informacija se podaci izdižu do razine znanja. Na sličan način, ako bi algoritme koji obrađuju znanje izdigli do razine zaključivanja, onda za takav računalni sustav možemo reći kao implementira jedan od vidova umjetne inteligencije jer oponaša ljudsko biće koje posjeduje znanje te je u stanju provoditi zaključivanje nad tim znanjem.

Inteligentni tutorski sustavi (ITS) su računalni sustavi zasnovani na tehnikama umjetne inteligencije kako bi simulirali ljudske tutore koji znaju što poučavaju, koga poučavaju i kako poučavaju. Za ITS-ove se kaže kako spadaju u kategoriju **sustava temeljenih na znanju** jer prikazuju znanje o područnom znanju kojeg poučavaju, znanje o učeniku koga poučavaju i znanje o tutorskim metodama koje koriste za poučavanje učenika (SELF, 1974). Ovakav pristup je uvjetovao dekompoziciju ITS-a na modul stručnjaka, modul učenika i modul učitelja. Važnost komunikacije s učenikom je uvjetovala dodavanje komponente korisničkog sučelja u tradicionalnu arhitekturu ITS-a, zbog čega se smatraju i **sustavima za komunikaciju znanjem** (WENGER, 1987). U standardnu arhitekturu se osim modula stručnjaka, modula učenika i modula učitelja uvodi i komunikacijski modul.

2.1 Arhitektura inteligentnog tutorskog sustava

Standardna arhitektura inteligentnog tutorskog sustava uključuje komponente koje imaju znanje o područnom znanju, znanju učenika i znanje o vođenju procesa učenja, poučavanja i testiranja (SELF, 1990). Zasnovanost komponenti inteligentnog tutorskog sustava na navedenim vrstama znanja je proizašla iz njegovog tradicionalnog modela koji uključuje već navedene module: modul stručnjaka, modul učenika i modul učitelja, te korisničko sučelje prema učeniku (BURNS, 1998). U tradicionalnom modelu modul stručnjaka sadrži područno znanje, modul učenika dijagnosticira znanje učenika i stvara model učenika, a modul učitelja identificira znanje koje učeniku nedostaje i provodi strategije za učenikovo usvajanje znanja. Korisničkim sučeljem se ostvaruje komunikacijski kanal između modula učitelja, odnosno računalnog tutora i učenika. Ova se struktura realizira komponentama koje su temeljene na spoznajama iz brojnih područja umjetne inteligencije. Te tehnike su zastupljene i u modulima modela inteligentnog tutorskog sustava.



Slika 2.1. Komponente ITS-a

Kad je u pitanju pogled prema učeniku, tada izražajnost komunikacije ovisi o izražajnosti formalizma prikaza područnog znanja. Područno znanje nosi informacije potrebne za oblikovanje strukture i plana izlaganja nastavnog sadržaja, dijagnosticiranje modela učenika i komunikaciju s učenikom.

2.2 Modul stručnjaka i područno znanje

Modul stručnjaka je okosnica svakog inteligentnog tutorskog sustava jer kvaliteta učenja, poučavanja i testiranja znanja direktno ovisi o kvaliteti područnog znanja čiju bazu posjeduje, a znanje predstavlja ključ za inteligentno ponašanje. U modulu stručnjaka se relacijama unutar područnog znanja pokazuje povezanost koncepata, što je bitna informacija koja modulu učitelja omogućava generiranje i prezentiranje računalom oblikovanog nastavnog sadržaja koji je opisan nizom elemenata pri čemu svaki element sadrži povezani podskup koncepata područnog znanja.

Formalni opis područnog znanja je ulazni skup podataka modela stručnjaka. U modulu stručnjaka se analizira opis područnog znanja radi određivanja međusobne povezanosti koncepata. Najčešća su sljedeća tri pristupa modeliranja i prikazivanja područnog znanja u modulu stručnjaka (ANDERSON, 1988): model crne kutije, model prozirne kutije i kognitivno modeliranje.

Prvi pristup koristi **model crne kutije** kojim se interno funkcioniranje modela sakriva od vanjskog promatrača. Ulazno-izlazne informacije modela crne kutije jednostavno nisu pogodne za učenje i poučavanje jer nisu u stanju izložiti proces zaključivanja učeniku.

U drugom pristupu upotrebljava se **model prozirne kutije**. Prilikom izgradnje ovakvog modela koristi se neka od metoda inženjerstva znanja kako bi se ljudsko znanje prenijelo i prikazalo uz pomoć računala.

Za razliku od modela prozirne kutije kojim se simulira ljudsko znanje, u trećem pristupu se simulira i način korištenja ljudskog znanja. Ovaj pristup se referencira kao **kognitivno modeliranje** jer se njim opisuje proces rješavanja problema i ljudskog razmišljanja.

Na oblikovanje modula stručnjaka utječe i vrsta znanja koja se prikazuje. **Proceduralno** i **deklarativno znanje** su osnovne vrste znanja kojim se opisuje određeno područno znanje. Prilikom formalizacije proceduralnog znanja koriste se produkcijska pravila, dok za formalizaciju deklarativnog znanja pogoduje formalni jezik simbola ili neka druga shema prikazivanja znanja.

Tradicionalno prikazivanje znanja zasniva se na izvjesnom analitičkom načinu korištenja formalnog logičkog jezika, međutim ljudsko zaključivanje posjeduje heurističke osobine, pa se načini prikazivanja znanja općenito mogu podijeliti na analitički i heuristički (HONKELA, 2007).

Analitički način prikazivanja znanja omogućuje logičko zaključivanje. Formalni logički jezici, produkcijska pravila i semantičke mreže su neke od osnovnih tehnika analitičkog prikazivanja znanja.

Heuristički način se razlikuje od analitičkog po tome što ne koristi strogo definirana pravila u funkciji prikazivanja znanja. Ovaj način prikazivanja znanja se temelji na neodređenosti podataka. Za razliku od analitičkog načina, algoritmi kod heurističkog načina prikazivanja znanja ne nude precizna rješenja, već proizvode zadovoljavajuća rješenja ili rješavaju manje probleme koji su povezani s većim problemom. Neodređenost heurističkog načina prikazivanja znanja se ističe kod modela kao što su neizrazita logika, neuronske mreže, stabla odluke, Bayesova mreža i ostalih. Bitne osobine heurističkog načina prikazivanja znanja su brza i paralelna obrada velikih količina podataka, te pragmatičnost.

Iz navedenih karakteristika prikazivanja znanja i modela sustava zasnovanih na znanju, postavljena su sljedeća pitanja koja se mogu vezati za modeliranje područnog znanja u modulu stručnjaka (WAY, 1991):

1. Koja je priroda znanja i kako se ono prikazuje?
2. Hoće li prikaz znanja biti proceduralan ili deklarativan?
3. Koliko je izražajan model prikaza znanja?

Odgovorom na prvo pitanje se određuje pristup modelu prikazivanja znanja. Većina ekspertnih sustava, nad kojima se zasnivaju moduli stručnjaka koristi kombinaciju crnog i prozirnog modela kutije, čime se djelomično otkriva područno znanje. Kognitivno modeliranje je najsloženiji pristup kojim se ostvaruje vjerno simuliranje ljudskog znanja i razmišljanja.

Drugo pitanje određuje što se sa znanjem želi učenika naučiti. Proceduralnim znanjem se opisuje kako se problemi rješavaju ili kako nešto radi. Većinski se prikazivanje proceduralnog znanja zasnivan na pravilima.

Treće pitanje, odnosno pitanje izražajnosti prikaza ovisi o mogućnostima opisivanja vanjskog svijeta. Što je mogućnost opisivanja detalja vanjskog svijeta veća, to je jezik prikazivanja znanja izražajniji. Međutim, kod izražajnih jezika je teže vršiti automatizirano zaključivanje, jer su oni često nekonzistentni i nepotpuni, za razliku od manje izražajnih jezika kao što je propozicijska logika. Upravo izražajnost prikaza znanja znatno utječe na komunikaciju između inteligentnog tutorskog sustava i učenika.

2.3 Struktura područnog znanja temeljenog na ontologiji

Pošto se svako područje ljudskog djelovanja može se prikazati skupom pravilno povezanih koncepata koji odgovara područnom znanju, promatramo model inteligentnog tutorskog sustava koji se zasniva na formalnom ontološkom opisu područnog znanja (GRUBIŠIĆ, 2012). Ontologijom se opisuje konceptualni model nekog područja, odnosno objekti, koncepti i drugi entiteti za koje se smatra da postoje te relacije među njima (GRUBER, 1993). Osnovni strukturni elementi konceptualnog modela su koncepti i relacije. Najpoznatija definicija ontologije je: „*An ontology is an explicit specification of a conceptualization*“, odnosno, **ontologija** je eksplicitna specifikacija konceptualizacije (GRUBER, 1993).

Ontološki pristup u opisivanju područnog znanja omogućava jednostavnu formalizaciju deklarativnog znanja korištenjem različitih alata koji podržavaju rad s konceptima i relacijama. Formalni ontološki opis područnog znanja je, kao što smo i prije naglasili, ulazni

skup podataka modela stručnjaka. Na osnovu ovog opisa modul stručnjaka formira područno znanje. U modulu stručnjaka se analizira ontološki opis područnog znanja radi određivanja međusobne povezanosti koncepata. Nakon postavljanja područnog znanja, inteligentni tutorski sustav je spreman za vođenje procesa učenja, poučavanja i testiranja znanja (GRUBIŠIĆ, 2012).

Dakle, promatramo područno znanje koje je prikazano konceptima i relacijama među njima. Pošto trebamo naznačiti smjer povezanosti koncepata, koristimo pojmove podkoncept i nadkoncept. Da bismo za svaku relaciju u ontologiji nedvosmisleno naznačili koje koncepte povezuje i kakav je odnos između tih koncepata bitno je naglasiti definiciju (definicije su preuzete iz (GRUBIŠIĆ, 2012)):

Definicija 2.1

> *Neka je E_{KCP} ={K_1,…,K_n }, n≥0,skup koncepata,E_{REL}={r_1,…,r_m}, m≥0, skup relacija i $\emptyset_E$ prazni element područnog znanja.*
>
> ***Područno znanje** je **skup uređenih trojki** (K_1, r, K_2) koje definiraju da su koncepti K_1 i K_2 povezani relacijom r. U ovako definiranoj uređenoj trojci koncept K_1 je nadkoncept koncepta K_2, tj. koncept K_2 je podkoncept koncepta K_1.*

Pošto su osnovni elementi trojki područnog znanja koncepti i relacije među njima, koristimo **teoriju grafova** kao matematičku podlogu koja omogućava upravljanje podskupovima i elementima područnog znanja, kao i vizualizaciju područnog znanja (VELJAN, 1989). Zato definiramo usmjereni graf područnog znanja (Definicija 2.2) za kojeg vrijede sve zakonitosti iz teorije grafova.

Definicija 2.2

> *Za područno znanje PZ definiramo **usmjereni graf područnog znanja** GPZ = (V, A) gdje je skup vrhova V = E_{KCP}, a skup bridova A = {(K_1, K_2)|∃(K_1, r, K_2) ∈ PZ, r ≠ $\emptyset_E$, K_1 ≠ K_2} jednak skupu svih uređenih parova onih koncepata iz područnog znanja koji su povezani nekom relacijom.*

Slika 2.2. Od područnog znanja do grafa područnog znanja

Ako postoji brid br=(K_1, K_2) između koncepata K_1 i K_2, onda to kraće pišemo K_1K_2 ili samo br. Navodimo nekoliko definicija kojima pobliže određujemo graf područnog znanja.

Definicija 2.3

> ***Skup nadkoncepata koncepta*** K_x *je* $\mathbf{NadK_x} = \{K \in E_{KCP} | \exists(K, r, K_x) \in PZ, K \neq K_x, r \neq slot, filler, \emptyset_E \} = \{ K \in V | \exists(K, K_x) \in A, K \neq K_x\}$. ***Broj nadkoncepata koncepta*** K_x $\mathbf{nK_x}$ *elemenata skupa* $NadK_x$.

Definicija 2.4

> ***Skup podkoncepata koncepta*** K_x *je* $\mathbf{PodK_x} = \{K \in E_{KCP} | \exists(K, r, K_x) \in PZ, K \neq K_x, r \neq slot, filler, \emptyset_E \} = \{ K \in V | \exists(K_x, K) \in A, K \neq K_x\}$. ***Broj podkoncepata koncepta*** K_x $\mathbf{pK_x}$ *elemenata skupa* $PodK_x$.

Definicija 2.5

> *Vrh grafa područnog znanja nazivamo* ***korijen*** *ako nema nadkoncepata, a ima podkocepata.*

2.4 Graf područnog znanja temeljenog na ontologiji

Već smo naglasili da su osnovni elementi trojki područnog znanja koncepti i relacije među njima te da ćemo koristiti teoriju grafova kao matematičku podlogu koja omogućava upravljanje podskupovima i elementima područnog znanja, kao i vizualizaciju područnog znanja. Zato ćemo u ovom poglavlju definirati graf područnoga znanja (Definicija 2.2).

Pojmove nadkoncept i podkoncept, kao i oznake nK_X i pK_X smo već definirali (Definicija 2.3, Definicija 2.4). Broj pK_x se naziva i **stupanj vrha**.

Usmjerenost bridova u grafu područnog znanja omogućava definiranje relacija „neposredni prethodnik“ i „neposredni sljedbenik“ samo na temelju strelice na bridu koji povezuje dva vrha.

Definicija 2.6

> *Na skupu V definiramo relacije:*
>
> K_y *je* ***neposredni sljedbenik*** *od* $K_x \Leftrightarrow K_x$ *je* ***neposredni prethodnik*** *od* $K_y \Leftrightarrow K_y \exists (K_x, K_Y) \in E_{KCP} \setminus E_A, K_Y \in E_{KCP} \setminus E_V$

Jedan koncept može biti neposredni sljedbenik ili neposredni prethodnik drugog koncepta jedino ako nije atribut ili vrijednost atributa. Neposredni sljedbenik ili neposredni prethodnik ne može biti atribut ili vrijednost.

Ove relacije nam omogućavaju definiranje posebne vrste ulančanog podgrafa, tzv. šetnje:

Definicija 2.7

Šetnja od K_1 do K_P u grafu GPZ je podgraf SetnjaK_1K_p = (V' = {K_1, …, K_p}, A') gdje je K_{i+1} neposredni sljedbenik od K_i, $\forall_i$ = 1, …, p - 1

(analogno, K_i neposredni prethodnik od K_{i+1} , ∀, $\forall_i$ = 1, …, p – 1)

U šetnji nema koncepata koji su atributi ili vrijednosti nekog koncepta, kao ni bridova prema atributima ili vrijednostima. U šetnji svaki vrh osim K_p ima neposrednog sljedbenika, odnosno svaki vrh osim K_1 ima svog neposrednog prethodnika. Šetnju kraće označavamo sa (K_1, br_1, K_2, …, br_p, K_p), gdje je br_i brid (K_{i-1}, K_i), za i = 2, …, n. duljina šetnje l_{KxKy} je broj bridova u podgrafu koji je definira, tj. $|A'|$. Šetnja je zatvorena ako vrijedi $K_1 = K_P$.

Put je šetnja u kojoj su svi vrhovi različiti, tj. PutK_1K_p = (V' = {K_1, …, K_n}, A') gdje je K_{i+1} neposredni sljedbenik od K_i, $\forall_i$ = 1, …, p – 1, $\forall K_x, K_y \in V', K_x \neq K_Y$. Zatvoreni put zovemo **ciklus**. Duljina puta (engl. *lenth of a path*) je zbroj bridova koji se nalaze u putu. Duljina najkraćeg puta između čvorova u grafu naziva se udaljenost (engl. *distance*) između čvorova. Prosječna duljina puta je prosječni najkraći put između dva slučajno odabrana vrha.

Staza je šetnja u kojoj su svi bridovi međusobno različiti. Zatvorena staza zove se **tura**. Staza je Eulerova ako se u njoj pojavljuju svi bridovi u grafu, i to točno jedanput. Svaki put je staza.

Udaljenost između vrhova K_x i K_y d(K_x, K_y) je duljina najkraćeg puta među njima.

Pošto stalno nastojimo održati **izomorfizam između područnog znanja i grafa područnog znanja**, potrebno je vidjeti kakvom podskupu područnog znanja odgovara podgraf kojeg čini šetnja između dva vrha.

Definicija 2.8

***Povezani niz koncepata** koji započinje konceptom K_x, a završava konceptom K_y*

$PNZK_xK_y = \{ (Ki, r, K_j) \in PZ|^{K_x=K_1, K_y=K_p, r\neq \emptyset_E, K_i\neq K_j,}_{K_{i+1} neposredni\ sljedbenik\ od\ K_i, \forall_i=1,\ldots,p-1}\}$ je podskup područnog znanja PZ u kojem je podkoncept jednog koncepta ujedno i nadkoncept drugog.

Usmjereni graf od $PNZK_xK_y$ je upravo SetnjaK_xK_y. broj elemenata u $PNZK_xK_y$ je jednak broju bridova u SetnjaK_xK_y.

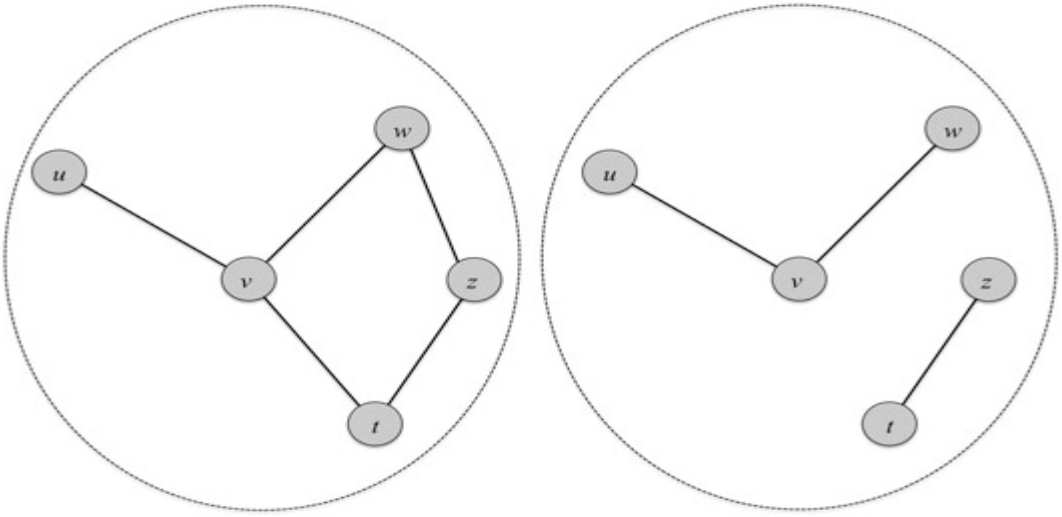
2.4.1 Komponente grafa područnog znanja

U grafu područnog znanja mogu postojati nezavisni podgrafovi čiji vrhovi nisu povezani. Zato definiramo relaciju ekvivalencije koja omogućava definiranje takvih podgrafova.

Definicija 2.9

Na skupu V definiramo relaciju ekvivalencije $\equiv$ *sa:*
$K_x \equiv K_y \Leftrightarrow \exists\, PutK_xK_y$

Ova relacija ekvivalencije definira jednu particiju skupa V, pa su komponente grafa GPZ podgrafovi inducirani klasama ekvivalencije. Kažemo da je **graf povezan** ako postoji samo jedna komponenta, u suprotnome graf je nepovezan.



Slika 2.3. Povezan i nepovezan graf

Komponente grafa su povezani podgrafovi. Pošto je put jedna vrsta šetnje, a u šetnji nema koncepata koji su atributi ili vrijednosti nekog koncepta, zaključujemo da u komponentama nema koncepata koji su atributi ili vrijednosti nekog koncepta.

2.4.2 Cjeline grafa područnog znanja

Navodimo definiciju jednog vrlo važnog pojma na kojem se temelji model stručnjaka, a to je cjelina.

Definicija 2.10

Definirano cjelinu C_i *u grafu GPZ kao najveći podgraf koji ima samo jedan korijen, tj. cjelina je graf čiji skup vrhova sadrži samo jedan korijen.* $PutK_{Ci}MaxVrh_{Ci}$ *je najveći put u nekoj cjelini, a* $MaxLen_{Ci}$ *je duljina tog puta.* $MaxVrh_{Ci}$ *je korijen cjeline koji se naziva centralni vrh cjeline* C_i

Centralni vrh nema prethodnika i od njega vodi put do svih ostalih vrhova. Bilo koji vrh u cjelini, osim centralnog vrha, ima jednog ili više neposrednih prethodnika. Cjelina je, dakle, povezani podgraf u kojem je svaki vrh povezan s centralnim vrhom (osim njega samog) najmanje jednim putem. U cjelini nema izoliranih vrhova. Dakle, najmanji podskup područnog znanja je cjelina koja je definirana korijenom. Odnosno, svaki korijen u područnom znanju definira jednu cjelinu.

Sljedeće relacije „sljedbenik od“ i „prethodnik od“ omogućavaju identificiranje korijena podgrafa koji odgovara cjelini grafa područnog znanja.

Definicija 2.11

Na skupu vrhova cjelina $C_i = (V_{Ci}, A_{Ci})$ definiramo relacije parcijalnog uređaja:
K_y ***je sljedbenik*** *od $K_x \Leftrightarrow \exists Put K_x K_y \subset C_i$*
K_x ***je prethodnik*** *od $K_y \Leftrightarrow \exists Put K_x K_y \subset C_i$*

Definicija 2.12

Centralni vrh K_{Ci} cjeline C_i je vrh za kojeg vrijedi $\forall K_x \in V_{Ci}$ vrijedi K_x je sljedbenik od K_{Ci}, odnosno K_{Ci} je prethodnik od K_x.

Centralni vrh cjeline nema prethodnika i od njega vodi put do svih ostalih vrhova. Bilo koji vrh u cjelini, osim centralnog vrha, ima jednog ili više neposrednih prethodnika.

Cjelina je, dakle, povezani podgraf u kojem je svaki vrh povezan s centralnim vrhom (osim njega samog) najmanje jednim putem. U cjelini nema izoliranih vrhova.

3. Pristupi određivanja reprezentativnog podskupa područnog znanja - uzorkovanje

Pošto područno znanje može biti obimno, potrebno je definirati takav podskup područnog znanja koji će dostojno reprezentirati cijelo područno znanje. Dakle, reprezentacijom područnog znanja obuhvaćamo sve koncepte i relacije koji su relevantni za neko područno znanje. Nadalje ćemo pristupiti definiranju reprezentacije područnog znanja kao skupa relevantnih koncepata i relacija (GRUBIŠIĆ 2012).

Reprezentacija područnog znanja nam služi da bi odredili koncepte područnog znanja na temelju kojih će se generirati ulazni test ili upitnik kojim se provjerava predznanje učenika o nekom područnom znanju. Ovako određeni ulazni test garantira da će ispitivanje znanja učenika obuhvatiti sve koncepte i relacije koji su relevantni za neko područno znanje tj. da će nam dati reprezentativni primjerak.

U ovom radu nećemo promatrati ni semantiku koncepata i relacije, već ćemo koristiti isključivo matematičke metode iz teorije grafova. U našem pristupu definiranja reprezentacije područnog znanja nećemo uzimati položaj u obzir semantiku koncepata već isključivo njihov položaj u grafu područnog znanja o kojem ovisi da li će oni pripadati reprezentaciji ili ne te ćemo određenim matematičkim metodama dobiti takvu reprezentaciju.

Upravo ova ideja je u posljednjim desetljećima, zahvaljujući velikim koracima informatičke tehnologije, dovela do povećanog interesa za mreže. One reprezentiraju odnose u i među živim bićima, računalnim sustavima, porukama, kemijskim elementima, itd. Razne grane ljudskog djelovanja sada proučavaju svojstva mreža ne bi li bolje razumjeli neki problem.

Kod jako velikih mreža vremenski je i prostorno zahtjevno izvršavati algoritme nad svim podacima. Dobra ideja je odabrati određene čvorove i veze u mreži koji bi dobro reprezentirali kompletnu mrežu. Takav odabir zove se uzorkovanje (engl. *sampling*).

Vremenom su se razvile razne metode za uzorkovanje mreža. Metode su došle iz raznih područja (fizike, fizikalne kemije, sociologije, matematike), a u teoriji grafova su se zadržale kao na jednom općem mjestu modeliranja različitih tipova mreža. Kada jednom utvrdimo kojem tipu grafa pripada naša mreža, možemo modelirati i analizirati mreže iz bilo kojeg područja, stoga za naš konkretan problem, uzorkovanje znanja, trebamo odrediti kojem tipu grafa graf područnog znanja pripada. Već smo u prethodnim poglavljima objasnili da je graf područnog znanja usmjereni aciklički graf. Zbog toga možemo koristiti skupove podataka (engl. *dataset*) za bilo kakvu mrežu koja se također modelira usmjerenim acikličkim grafom.

Stranice http://snap.stanford.edu/data/ služe kao repozitorij raznih skupova podataka i među njima možemo naći i kompleksne usmjerene acikličke mreže.

Algoritmi za uzorkovanje mogu se podijeliti na više tipova. Leskovec i drugi dijele metode za uzorkovanje na metode u kojima se u uzorak dodaju slučajno odabrani vrhovi (*Random Node, Random PageRank* i *Random Degree Node*), metode u kojima se dodaju slučajno odabrani bridovi (*Random Edge* i *Random Node-Edge*) i metode istraživanja (engl. *exploration methods*) u kojima se odabire prvi slučajno uniformno odabrani vrh, te se potom ispituju vrhovi u njegovom susjedstvu (LESKOVEC, FALOUTSOS, 2006). Takve metode su metode susjedstvo slučajnog vrha (engl. *Random Node Neighbor* (RNN)), slučajna šetnja (engl. *Random Walk* (RW)), slučajni skok (engl. *Random Jump* (RJ)) i šumski požar (engl. *Forest Fire* (FF)). Osim navedenih, za prikupljanje podataka u društvenim znanostima, pogotovo kada se želi doći do "skrivenih" dijelova društva, koristi se algoritam snježne grude (engl. *Snowball*) koji isto pripada metodi istraživanja.

U ovom radu uspoređujemo pet algoritama, od čega četiri spadaju u metode istraživanja, a peti je algoritam kojeg ne bi mogli smjestiti u navedenu podjelu. U Represent algoritmu ne postoji slučajan odabir, već se uzorak dobiva unijom najkraćih puteva najveće duljine u komponenti grafa. To je novi pristup u kojem ne možemo unaprijed odrediti veličinu uzorka, složenost algoritma prati složenost traženja najkraćih puteva u grafu, i to su mane algoritma, ali će uzorak pri svakom uzorkovanju biti isti i u specifičnom problemu reprezentacije područnog znanja dobro će reprezentirati područno znanje.

3.1 Kompleksne mreže

Mreža (engl. *network*) je skup čvorova (engl. *nodes*) sa vezama (engl. *links*) između sebe. Matematički se prikazuje pomoću grafa u kojem su čvorovi vrhovi (engl. *vertices*), a veze bridovi (engl. *edges*). Proučavanjem grafova bavi se matematička disciplina teorija grafova. Proučavanjem mreža bave se razne znanstvene discipline pritom koristeći znanja dobivena iz teorije grafova. Može se reći da pojam mreže dolazi iz stvarnog svijeta, a da je pojam grafa apstrakcija mreže.

Teorija grafova u svojim počecima bavila se uglavnom statičnim, malim grafovima, kasnije i slučajnim grafovima sa naglaskom na svojstvima vrhova i bridova [BILGIN, YENER, 2008]. Mreže u stvarnom svijetu nisu statične, ni male, a često i ne odgovaraju modelima teorije grafova kakvi su, recimo, regularni grafovi ili slučajni grafovi. U ovom stoljeću fokus istraživanja mreža je na topologiji, rastu i primjenama dinamičkih mreža koje se često još i nazivaju kompleksne mreže ili mreže stvarnog svijeta (engl. *real-world network*).

Razni kompleksni sustavi u stvarnom svijetu formiraju mreže. Možda najistaknutiji primjeri takvih mreža su društvene mreže. Međutim, slično njima ponašaju se i metaboličke mreže, transportne mreže itd. Newman u [NEWMAN, 2003] dijeli kompleksne mreže na:

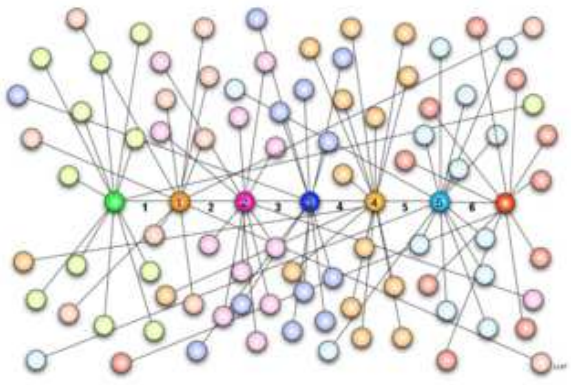
1. socijalne mreže,
2. informacijske mreže,
3. tehnološke mreže,
4. mreže u biologiji.

Samu strukturu mreže određuju njena strukturna svojstva od kojih su važniji stupnjevi čvorova, distribucija stupnjeva, koeficijent grupiranosti i distribucija koeficijenata grupiranosti, prosječna duljina najkraćeg puta i distribucija duljina najkraćeg puta, gustoća, dijametar, međupoloženost (engl. *betweenness*), itd. Topološka svojstva su *scale-free* distribucija stupnjeva, *small-world* svojstvo i slično. Dakle, u ovom radu proučavaju se strukturna svojstva mreže, a na temelju njih se zaključuje nešto o topologiji.

Kompleksne mreže nam ponekad nisu cijele dostupne, pa proučavanje i analiziranje svojstava mreže može dovesti do razumijevanja njene strukture bez da se prouči cijela mreža. U nastavku ćemo ukratko objasniti što znače svojstva koja ćemo proučavati u ovom radu.

3.1.1 Efekt malog svijeta i najkraći putevi u mreži

Efekt malog svijeta (engl. *small world effect*) je svojstvo kompleksne mreže prema kojem je prosječna najkraća udaljenost između dva čvora mala. Najkraći put između dva čvora *u* i *v* je svojstvo koje nam govori u koliko se koraka može doći od čvora *u* do čvora *v*. Potječe iz poznatog eksperimenta američkog socijalnog psihologa Stanley Milgrama iz 1960-tih. u kojem je od slučajno odabranih ljudi u Nebraski tražio da pošalju pismo slučajno odabranim osobama u Bostonu kojima su bila poznata puna imena, zanimanje i lokacija [BOCCALETTI, LATORA, et al., 2006]. Ideja je bila da ljudi pošalju pismo ljudima za koje pretpostavljaju da bi mogli poznavati ciljne ljude. Broj prosječnih koraka u kojem je pismo stiglo od pošiljatelja do primatelja bio je šest.

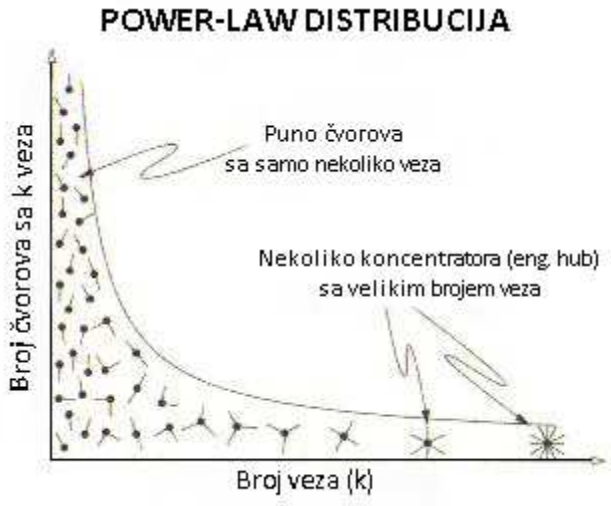


Slika 3.1. Šest stupnjeva odvojenosti prikazano grafom
[dostupno 21.06.2014. na https://commons.wikimedia.org/wiki/File:Six_degrees_of_separation_01.png]

Mreže malog svijeta (engl. *small-world networks*) opisali su Watts i Strogatz uočivši da su realne mreže negdje između regularnih i slučajnih mreža [WATTS, STROGATZ, 1998]. Oni su pokazali da krenuvši od regularnog grafa i prespajajući veze u tom grafu na slučajan način smanjuju prosječnu duljinu najkraćeg puta između dva čvora.

3.1.2 Distribucija stupnjeva u mreži i nerazmjerne mreže

S obzirom na stupnjeve čvorova u mreži, kompleksne mreže još zadovoljavaju i svojstvo nerazmjernosti (engl. *scale-free*). To je svojstvo mreže da stupnjevi čvorova prate krivulju potencije (engl. *power law*). Drugim riječima malo čvorova ima veliki stupanj, dok veliki broj čvorova ima mali stupanj. Čvorovi koji imaju veliki stupanj nazivaju se koncentratori (engl. *hub*), a njihovo se postojanje može objasniti rastom mreže i svojstvom preferencijalnog povezivanja (engl. *preferential attachment*). Naime, kako mreža raste veća je vjerojatnost da će se novi čvor povezati sa čvorovima koji imaju veći broj veza.



Slika 3.2. Power-law distribucija (preuzeto s?)

Jedan od načina na koji možemo prikazati distribuciju stupnjeva je pomoću funkcije kumulativne distribucije

$$P_k = \sum_{k=k_1}^{\infty} p_{k_1}$$

koja predstavlja vjerojatnost da je stupanj čvora veći ili jednak k.

3.1.3 Koeficijent grupiranja

Koeficijent grupiranja u grafu predstavlja vjerojatnost da su dva čvora koja su povezana sa istim čvorom u mreži i sami povezani. Preciznije, to je omjer broja ciklusa duljine 3 koji

prolaze kroz vrh v i broj svih mogućih ciklusa duljine 3 koji bi mogli prolaziti kroz taj čvor. Neka je za čvor v broj susjednih čvorova dana sa k_v, a e_v broj povezanih susjeda od v. Tada je koeficijent grupiranja

$$c_n = \frac{2e_v}{k_v(k_v - 1)}$$

U usmjerenom grafu broj mogućih ciklusa je dvostuko veći, pa je koeficijent grupiranja

$$c_n = \frac{e_v}{k_v(k_v - 1)}$$

Koeficijent grupiranje je u uskoj vezi sa gustoćom mreže. Gustoća predstavlja omjer broja bridova u odnosu na mogući broj bridova u mreži. Vrijednost gustoće je između 0 i 1. Graf bez bridova ima gustoću 0, a potpun povezan graf ima gustoću 1.

3.1.4 Broj povezanih komponenti i jako povezanih komponenti

Komponenta u grafu G je maksimalni mogući podgraf grafa u kojem su svaka dva čvora povezana. Broj povezanih komponenti je broj takvih podgrafova u grafu. U usmjerenom grafu komponenta je jako povezana (engl. *strongly conncected*) ako postoji usmjereni put između svaka dva čvora u podgrafu.

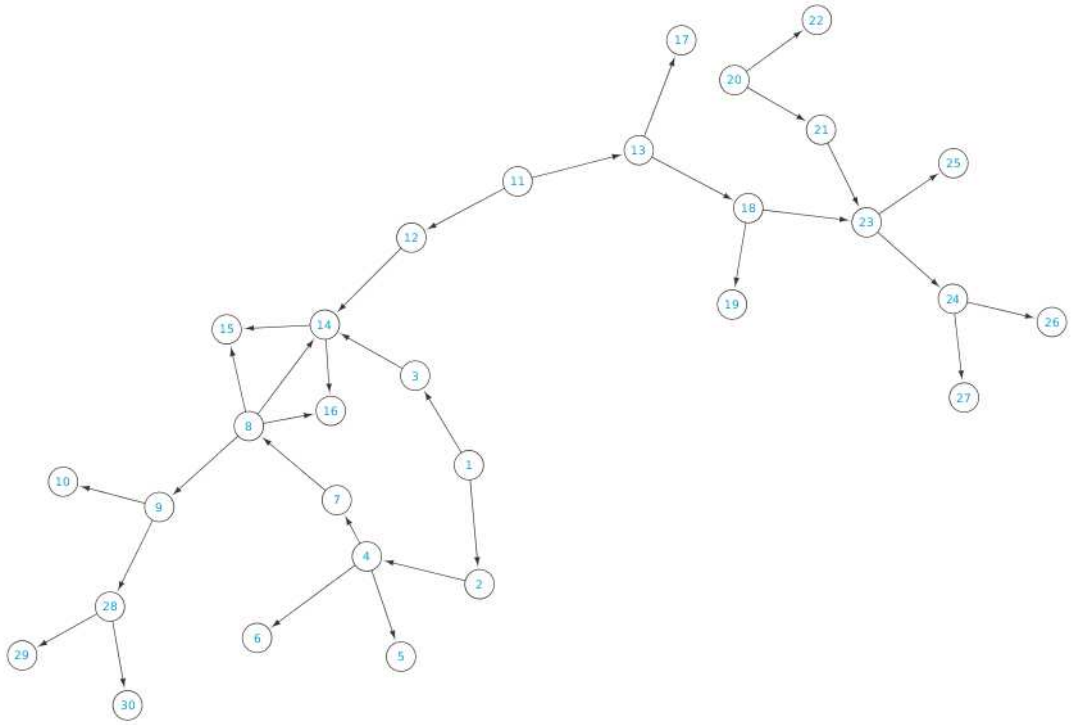
3.2 Metode za uzorkovanje

Već smo spomenili u prethodnom poglavlju uzrokovanje. Uzorkovanje je proces odabira čvorova iz mreže tako da uzorak bude manja mreža koja zadržava i svojstva i sličnu strukturu koju je imala veća mreža. Preciznije, za graf $G = (V, E)$ uzorak je graf $G_1 \subseteq G$ takav da je $G_1 = (V_1, E_1)$, $V_1 \subseteq V$ i $E_1 \subseteq E$. Dva su cilja uzorkovanja. Prvi je naći uzorak tako da se sačuvaju svojstva mreže. Pritom se misli na svojstva mreže kao što su prosječan stupanj čvora ili prosječni koeficijent grupiranja u mreži. Drugi je naći uzorak tako da se sačuva struktura mreže, primjerice distribucija najkraćih puteva u mreži ili distribucija koeficijenata grupiranja.

Algoritmi za uzorkovanje obrađeni u ovom radu uglavnom su nastali iz potrebe za analiziranjem društvenih mreža. Izuzetak od toga je Represent algoritam koji je napravljen za potrebe uzorkovanja područnog znanja u inteligentnim tutorskim sustavima. Društvene mreže se mogu reprezentirati neusmjerenim grafom, kao u slučaju Facebook mreže, ili usmjerenim grafom, kao u slučaju Twitter mreže. Nadalje, mreže citata su mreže koje se mogu reprezentirati usmjerenim acikličkim grafom. Većina predstavljenih algoritama namijenjena je za neusmjerene grafove te smo ih stoga modificirali za potrebe usmjerene acikličke mreže koja reprezentira područno znanje.

Osim Represent algoritma, svi navedeni algoritmi su algoritmi u kojima krećemo od inicijalnog čvora, a zatim, u svakoj iteraciji, ispitujemo susjede čvora i neka od njih dodajemo u uzorak. Pod susjedom čvora *u* misli se na čvor *v* takav da postoji brid koji ide od *u* prema *v*. Problem koji može nastati prilikom generiranja uzorka kod acikličkog usmjerenog grafa je nepostojanje susjeda. U slučaju da dođemo do čvora koji je *list*, tj. iz kojeg ne izlazi niti jedan brid, uzorkovanje nastavljamo od nekog drugog slučajno odabranog čvora koji već nije u uzorku.

Djelovanje pojedinih metoda odnosno algoritama prikazat ćemo u nastavku na primjeru grafa od 30 čvorova.



Slika 3.3. Primjer početnog grafa koji se sastoji 30 čvorova

3.2.1 Slučajna šetnja (engl. *Random Walk*)

Problem slučajne šetnje (engl. *Random Walk*) prvi put je postavio matematičar Karl Pearson u časopisu Nature 1905.g. (PEARSON, 1905). U njemu poziva čitatelje da mu pomognu kod sljedećeg problema: čovjek kreće iz točke *O* i hoda *k* metara naprijed. Nakon toga okreće se za neki, bilo koji kut i opet hoda ravno *k* metara. Ponavlja taj proces *n* puta. Pitanje je kolika je vjerojatnost da je on nakon *n* takvih okretanja između *r* i *r+dr* od početne točke. U Natureu taj je problem adresiran kao slučajna šetnja ili šetnja pijanca (engl. *Drunkard's walk*).

Jedan od čitatelja koji se odaziva na poziv i daje rješenje je Lord Rayleigh, dobitnik Nobelove nagrade za fiziku. Njegovo rješenje izaziva Pearsona da izjavi da je "najvjerojatnije

mjesto gdje možeš pronaći pijanca koji stoji na nogama negdje u blizini njegove početne točke" (NATURE, 2014).

Slučajna šetnja je postala tema u raznim disciplinama, posebno teoriji vjerojatnosti, ali i biologiji, psihologiji, fizici, kemiji i ekonomiji. Međutim, jedino područje u kojem nije vezana uz problem, već uz model je teorija grafova.

U teoriji grafova slučajna šetnja obilazi vrhove u grafu tako da, krenuvši od nekog početnog vrha odaberemo njegov susjedni vrh slučajnim odabirom, pomaknemo se na taj vrh i zatim ponovo biramo susjedni vrh (LOVÁSZ, 1993). Postupak se nastavlja dok slučajnom šetnjom ne dođemo do željene duljine šetnje ili do određenog vrha. Prilikom uzorkovanja cilj nam je dostići određenu veličinu uzorka. Međutim, algoritam slučajne šetnje dozvoljava ponovno posjećivanje vrha (GJOKA, KURANT, BUTTS, MARKOPOULOU, 2010), pa je moguće da se u uzorku vrhovi ponavljaju i da je ukupan broj različitih vrhova u uzorku manji od ciljanog. Tome možemo doskočiti tako da algoritam ponavljamo dok ne dođemo do ciljanog broja različitih vrhova u uzorku.

Algoritam slučajne šetnje koristi vjerojatnost povratka na početak (engl. *fly back probability*) kako bi u nekim koracima algoritam vratio u ishodišni vrh. Na taj način uzorak postaje kombinacija više šetnji sa istim ishodišnim vrhom. Obično je ta vrijednost 0.15 (LESKOVEC, FALOUTSOS, 2006).

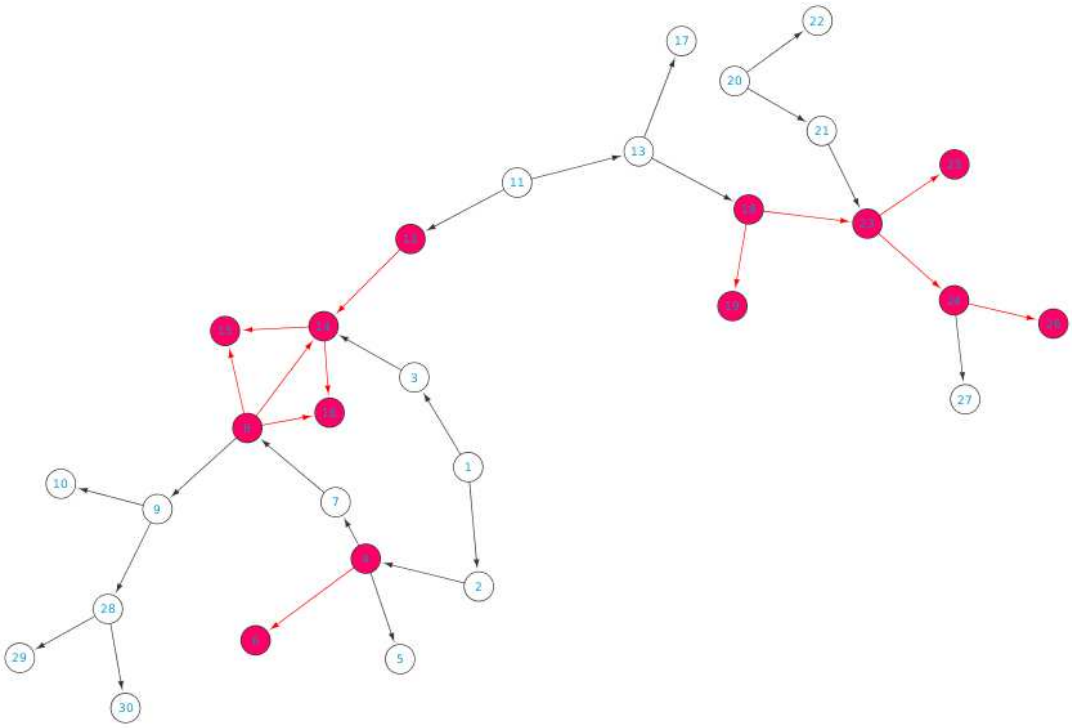
Osim toga, postoji i mogućnost da uzorak ne raste dovoljno brzo, recimo ako je ishodišni vrh izlaznog stupnja nula ili ako je dio male komponente. U tom slučaju bira se novi ishodišni vrh i šetnja se nastavlja od njega. Definira se broj iteracija T i očekivani rast M tokom T iteracija, te ako rast nije dovoljan bira se novi ishodišni vrh.

Algoritam slučajne šetnje

1. Dan je usmjereni aciklički graf, broj iteracija *T*, veličina rasta uzorka *M* u periodu *T* i povratna vjerojatnost *p*
2. Odabere se čvor slučajnim uniformnim odabirom
3. Dok se ne dosegne tražena veličina uzorka

    a. Sa vjerojatnošću *1-p* odaberi susjedni čvor i ako nije u uzorku dodaj ga
    b. Sa vjerojatnošću *p* (*fly-back probability*) vrati se na početni vrh
    c. Ako je broj dodanih čvorova sa brojem iteracija *T* manji od *M* vrati se na korak 2

Slika 3.4. Algoritam slučajne šetnje



Slika 3.5. Početni graf nakon uzorkovanja algoritmom slučajne šetnje

3.2.2 Metropolis-Hastings slučajna šetnja (engl. *Metropolis-Hastings Random Walk, MHRW*)

Originalno, Metropolis algoritam razvio je Nicholas Metropolis sa suradnicima 1953. u Los Alamos Scientific Laboratory za potrebe fizikalne kemije (engl. *chemical physics)* (METROPOLIS et al., 1953). Algoritam je baziran na Monte Carlo algoritmima razvijenim za vrijeme drugog svjetskog rata u istom laboratoriju, poznatom još i kao jedno od mjesta gdje se razvijao projekt Manhattan. Naziv Monte Carlo odnosi se na familiju algoritama u čijoj je osnovi ponovljeno slučajno uzorkovanje. Kasnije je Hastings generalizirao algoritam, te je algoritam dobio ime Metropolis-Hastings.

Za uzorkovanje grafova Metropolis-Hastings algoritam koristi distribuciju vjerojatnosti stupnja čvora. Naime, kao i u algoritmu slučajne šetnje za neki vrh *u* odabire se susjedni vrh *v* slučajnim uniformnim odabirom. Međutim, odabrani susjed ne mora nužno ući u uzorak. Odluka o tome ovisi o stupnju vrha i slučajnom broju *p* između 0 i 1. Ukoliko je odabrani susjed stupnja manjeg od samog vrha, dodati će se u uzorak. Ukoliko ima veći stupanj dodat će se ukoliko je $\mathrm{d}(u)/\mathrm{d}(v) >= p$ (LEE, XU, EUN, 2012).

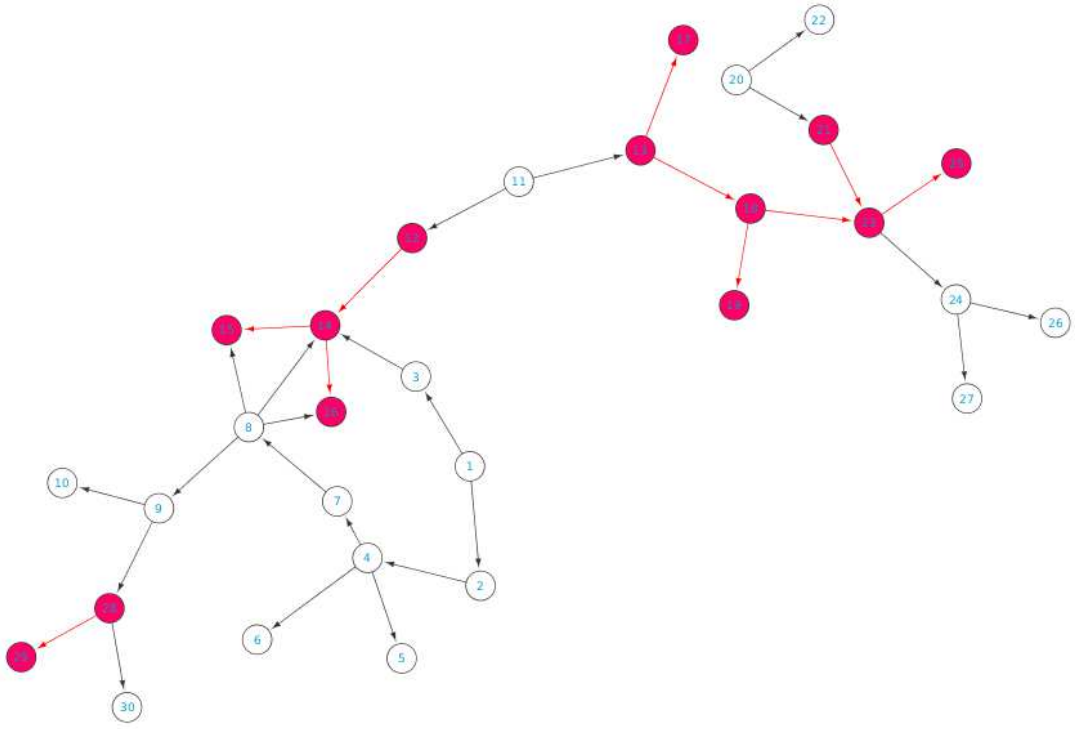
Algoritam Metropolis-Hastings slučajna šetnja

1. *v* je početni čvor
2. Dok se ne ispuni uvjet za zaustavljanje

    a. Odaberi čvor *w* uniformno među susjedima od *v*
    b. Generiraj broj *p* između 0 i 1
    c. Ako je $p \leq min\left(\frac{Q(v)}{Q(w)}\right)$, *w* je sljedeći čvor, inače ostani na čvoru *v*

Slika 3.6.Algoritam Metropolis-Hastings slučajna šetnja

Problem zaustavljanja algoritma u slučaju da u uzorak uđe vrh koji ima izlazni stupanj nula rješava se slučajnim ponovnim odabirom početnog vrha.



Slika 3.7. Početni graf nakon uzorkovanja Metropolis-Hastings algoritmom

3.2.3 Snježna gruda (engl. *Snowball*)

Još jedan algoritam s korijenima u sredini 20. stoljeća je algoritam snježne grude. Ovaj put algoritam generalizira problem iz društveno humanističkih znanosti, a razvio ga je Leo A. Goodman 1961. godine (GOODMAN, 1961). Algoritam je nastao kao strategija prilikom

proučavanja "skrivene populacije". To mogu biti manjine, homoseksualci, narkomani ili naprosto grupe ljudi do kojih je teško doći: djeca, adolescenti ili starci.

Početni uzorak odabire se slučajnim odabirom iz dane populacije. Svaka individua u uzorku imenuje k ljudi. Ovisno o cilju uzorkovanja, to mogu biti prijatelji, ljudi koji im naprosto padnu na pamet ili uzori. Ljudi koji nisu bili u slučajnom uzorku na početku, već su odabrani od strane ljudi iz slučajnog uzorka pripadaju prvoj razini (engl. *stage*). Svaki čovjek iz prve razine opet imenuje k ljudi. Oni među imenovanima koji nisu u početnom uzorku niti u prvoj razini dodaju se u uzorak i čine drugu razinu. Postupak se nastavlja sve dok uzorak ne dosegne ciljanu veličinu ili se ne dosegne ciljana razina.

S obzirom na način odabira čini sa da će uzorak eksponencijalno rasti, međutim u uzorak na svakoj razini ulaze samo oni ljudi koji nisu na prethodnim razinama, pa uzorak ne raste tom brzinom. Kada bi navedenu situaciju modelirali kao graf, to bi bio usmjereni graf sa ciklusima. Smjer brida ide od osobe koja imenuje prema osobi koja je imenovana, a ciklusi su mogući jer ne postoji zapreka da čovjek imenuje nekoga tko je na razinama nižim od njega.

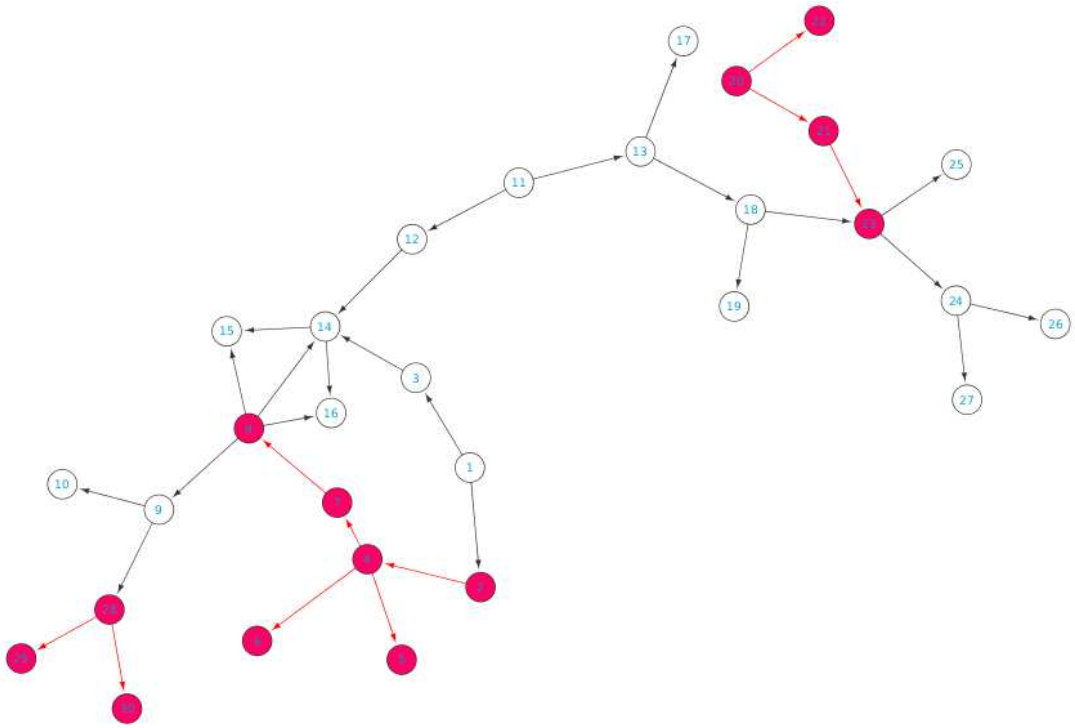
Goodman analizira različite varijante algoritma ovisno o razinama *s* i broju susjednih (imenovanih) vrhova *k*. Slučajevi $s = k = 1$ ili *s* je pozitivan cijeli broj, a $k = 1$ odražavaju situacije kada se imenuje samo jedan čovjek. S obzirom na problem kojim se bavimo u ovom radu, to nam nije od interesa. Od interesa nam je varijanta kada su *s* i *k* dva pozitivna cijela broja. Tada se na svakoj razini dodaje *k* vrhova koji već nisu u uzorku.

Razlika u primjeni ovog algoritma na društvene veze i kontakte među ljudima i mreže područnog znanja ili mreže citata su u tome što kod područnog znanja i citata nema onog tko bi imenovao k susjednih vrhova. Osim toga, mreže koje su nama od interesa su acikličke, pa se biranjem susjeda ipak ne možemo vratiti na prethodnu razinu. Stoga u našem slučaju mreža brzo raste. Susjedi koji će ući u uzorak se delegiraju na način da ako vrh ima stupanj manji ili jednak ciljanom broju k, dodaju se svi susjedi u uzorak, a ukoliko ima više od k susjeda, odabire se k susjeda i dodaje u uzorak.

Algoritam snježne grude

1. *S* je red (*queue*) *k* slučajno odabranih čvorova
2. Uzmi iz reda (*dequeue*) čvor *v*
    a. Ako je veličina uzorka dostignuta, zaustavi se
    b. Inače, uzmi *k* slučajno odabranih susjeda čvora *v* i dodaj ih u red *S*
3. Ako red *S* nije prazan, nastavi od koraka 2

Slika 3.8. Algoritam snježne grude



Slika 3.9. Početni graf nakon uzorkovanja algoritmom snježne grude

3.2.4 Šumski požar (engl. Forest Fire)

Model šumskog požara (BAK, CHEN, TANG, 1990) je model u kojem se šuma rasprostire na mreži od *nxn* lokacija tj. čvorova, a svaka od lokacija može biti prazna, na njoj može biti stablo koje gori ili na njoj može biti stablo koje ne gori. Dinamika požara opisana je sa tri pravila:

1. stablo izraste sa vjerojatnošću p u svakom koraku na nekoj praznoj lokaciji,
2. stabla koja gore, izgoriti će u sljedećem koraku (lokacija postaje prazna),
3. požar će se proširiti na sva susjedna stabla u sljedećem koraku.

Model su modificirali Drossel i Schwabl (DROSSEL, SCHWABL, 1992) mijenjajući drugo pravilo sa sljedeća dva pravila:

2a. stablo će se zapaliti s vjerojatnošću 1-*g* ako barem jedno susjedno stablo gori,

2b. stablo će se zapaliti sa vjerojatnošću *f*<<1 ako nijedno susjedno stablo ne gori.

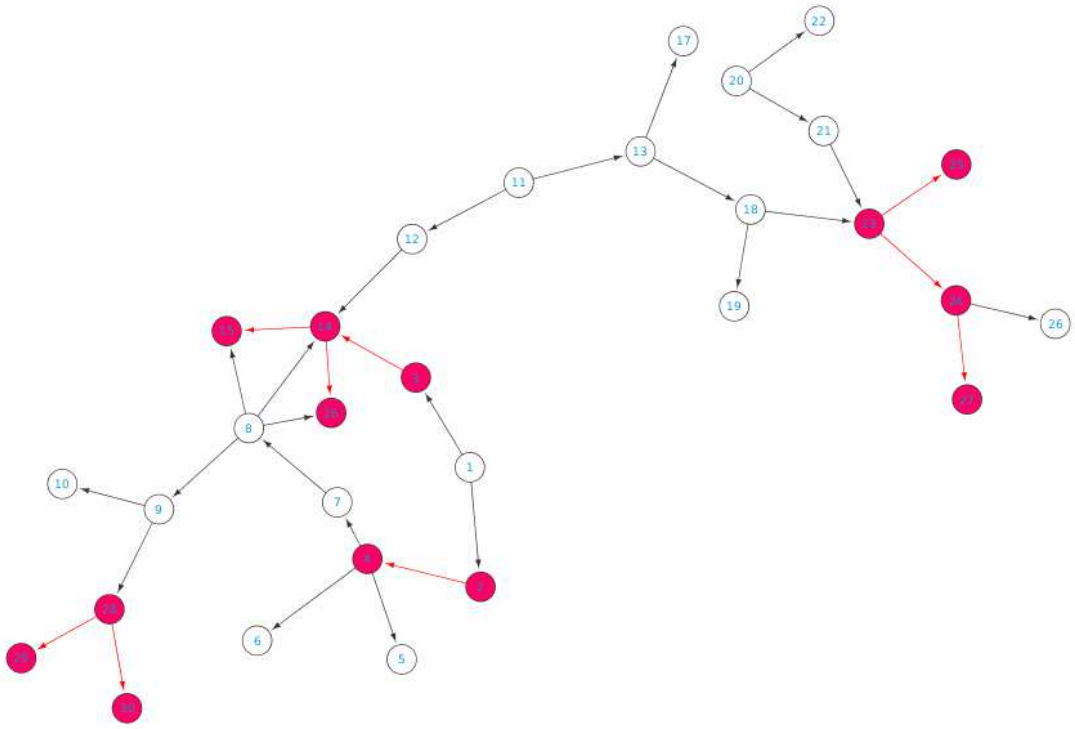
Algoritam šumskog požara za potrebe uzorkovanja u grafu je prilagodio Jure Leskovec (LESKOVEC, FALOUTSOS, 2006). Uniformnim slučajnim odabirom odabire se početni čvor *v*. Nakon toga se generira slučajni broj *x*, te se odabire *x* izlaznih bridova iz čvora *v* incidentnih s čvorovima koji još nisu posjećeni. Neka su to čvorovi *w1,...,wx*. Primjenjujemo prethodni korak rekurzivno na svaki *w1,...,wx* dok uzorak ne dostigne željenu veličinu. Ako se "vatra ugasi" uzorak se nastavlja puniti počevši od nekog novoizabranog čvora *v* odabranog uniformnim slučajnim odabirom.

Algoritam šumskog požara

Dok se ne dosegne ciljana veličina uzorka:

1. *v* je slučajno odabrani čvor, dodaj ga u uzorak
2. Generiraj *x* i odaberi *x* susjeda čvora koji još nisu bili posjećeni i dodaj ih u uzorak
3. Ponovi prethodni postupak na odabrane susjede
4. Ako se uzorak nije povećao, vrati se na korak 1

Slika 3.10. Algoritam šumskog požara



Slika 3.11. Početni graf nakon uzorkovanja algoritmom šumskog požara

3.2.5 Represent algoritam

Represent algoritam je jedini algoritam napravljen specifično za potrebe reprezentacije područnog znanja u inteligentnim tutorskim sustavima, te jedini od proučavanih algoritama koji ne koristi slučajne veličine za odabir vrhova koji će ući u uzorak (GRUBIŠIĆ, 2012.). Graf područnog znanja sastoji se od više cjelina, a svaka cjelina ima centralni vrh (korijen) koji nema prethodnika i od njega vodi put do svih ostalih vrhova u cjelini. Drugim riječima, cjelina je povezani podgraf u kojem je svaki vrh povezan s centralnim vrhom.

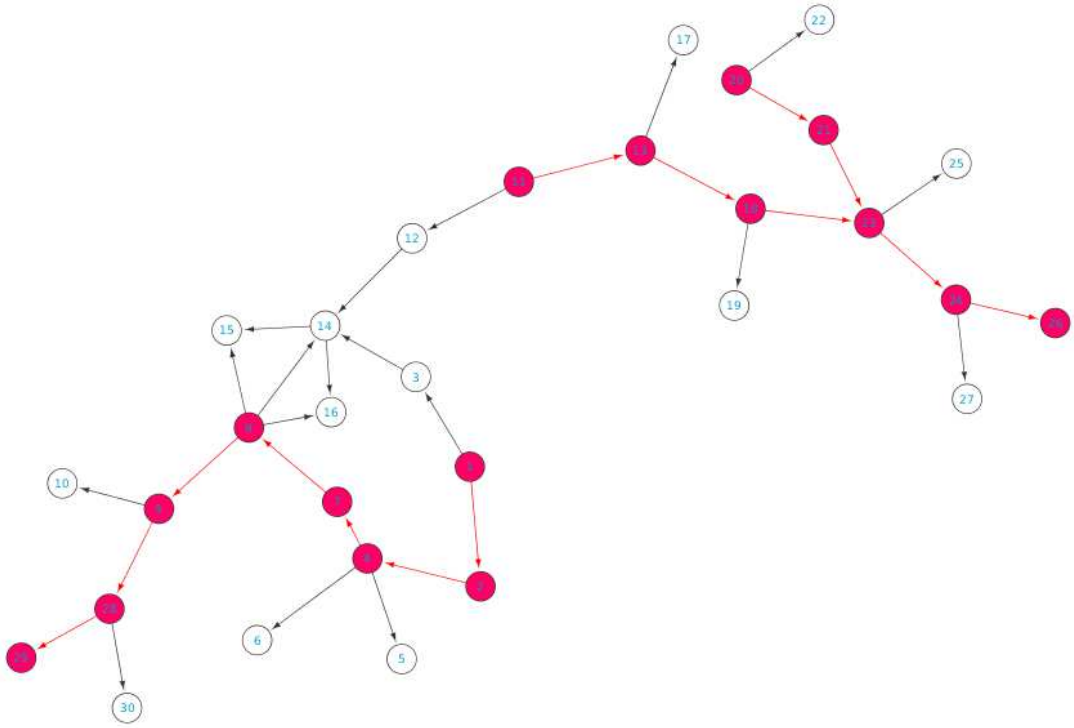
Ideja Represent algoritma je da se u uzorak stavi unija vrhova koji predstavljaju najduže puteve u svakoj cjelini. Pojam najdužeg puta ustvari označava put koji je najduži među svim najkraćim putevima u grafu ili podgrafu koji promatramo.

Algoritam radi tako da za svaki korijen iz skupa svih korijena grafa, pro nalazi najkraći put najveće duljine do nekog lista. Kako se radi o usmjerenim acikličkim grafovima, a algoritam ne ovisi o slučajnim veličinama, dobiveni uzorak je jedinstven.

Represent algoritam

1. Neka je *roots* lista svih korijena, a *leafs* lista svih listova u usmjerenom acikličkom grafu
2. Za svaki korijen:
    a. Nađi najkraće puteve od korijena do svih listova i među njima odaberi onaj najveće duljine i dodaj ga u uzorak

Slika 3.12. Represent algoritam



Slika 3.13. Početni graf nakon uzorkovanja Represent algoritmom

4. Implementacija metoda uzorkovanja

U ovom poglavlju najprije ćemo opisati tehnologije koje smo koristili za implementaciju odabranih metoda uzorkovanja, a zatim ćemo opisati samu implementaciju.

4.1 Python

Za implementaciju algoritama za uzorkovanje odabrali smo programski jezik Python. Razlog za to se krije u jednostavnosti jezika i jednostavnom korištenju struktura podataka koje su potrebne za implementaciju algoritama, ali i dobroj podršci za teoriju grafova u vidu Python biblioteka.

Tijekom godina razvoja Pythona, razvijeno je mnogo modula i paketa za Python, a navest ćemo samo korištene.

4.1.1 Networkx

Networkx je multiplatformska Python biblioteka za istraživanje, analizu mreža i mrežnih algoritama. Izdana je pod licencom *BSD-new license*, što je svrstava u slobodni softver otvorenog koda. Omogućava rad sa jako velikim mrežama, te implementira brojne algoritme teorije grafova.

U biblioteci postoje klase za objekte grafa (vrhove i bridove, grafove, digrafove...), metode za generiranje grafova, čitanje i pisanje iz raznih skupova podataka (engl. *datasets*), metode za analizu mreža i metode za prikazivanje grafa.

Uključivanje biblioteke u projekt postiže se sljedećom naredbom

```
import networkx
```

a zbog lakšeg korištenja preporuča se sljedeći oblik

```
import networkx as nx
```

Sam objekt graf može biti tipa Graph, DiGraph, MultiGraph i MultiDiGraph, dakle podržani su neusmjereni, usmjereni i multigrafovi. Implementacija grafa bazirana je na listi susjeda konkretno realizirana kao rječnik rječnika (engl. *dictionary of dictionary*). Ključevi (vanjskog) rječnika su čvorovi kojima su pridružene vrijednosti predstavljene rječnikom u kojem je ključ susjedni vrh, a vrijednost eventualni atribut brida (npr. težina) (NETWORKX, 2014).

4.1.2 Pandas

Biblioteka pandas je multiplatformska Python biblioteka otvorenog koda. Biblioteka obuhvaća strukture podataka i statističke alate za analizu podataka. U ovom radu koristi se za manipulaciju csv (*comma-separated values*) datoteka. Naime, distribucije stupnjeva, koeficijente grupiranja i najkreće puteve za svaki uzorak se zapisuje u cvs datoteku. Metode iz biblioteke pandas omogućuju spajanje csv datoteka, kako bi nastao skup podataka za crtanje grafova kumulativne distribucije.

4.1.3 Matplotlib

Biblioteka matplotlib je multiplatformska Python biblioteka otvorenog kôda za crtanje 2D grafova (MATPLOTLIB, 2014). U ovom diplomskom radu koristi se klasa pyplot za crtanje grafova kumulativnih distribucija iz csv (*comma-separated values*) datoteka.

4.2 Gephi

Vizualno prikazivanje mreža važno je za razumijevanje prikupljenih podataka i za provođenje rezultata analize. Mnogo je primjera softvera za analizu mreža, te je velik broj njih specijaliziran za određeno područje istraživanja. U ovom radu korišten je Gephi.

Gephi je multiplatformski alat otvorenog kôda namijenjen vizualizaciji i analizi mreža. Može obrađivati mreže do 50000 čvorova i 1000000 veza (GEPHI, 2014). Koristi nekoliko algoritama za razmještaj čvorova i veza prilikom vizualizacije, te cijeli spektar funkcija za računanje metrika na grafu. Metrike računate u ovom radu su prosječni stupanj u grafu, prosječna dužina najkraćeg puta, gustoća, prosječni koeficijent grupiranosti, međutim u Gephiju se mogu računati i centralnost, dijametar, međupoloženost (engl. betweenness) i drugi. Nekoliko je formata za ulazno izlazne podatke koje podržava Gephi, a u ovom radu koristili smo eksportiranje u pajek format[1].

Ipak, koristeći Gephi došli smo do čudnih rezultata. Naime, za graf koji se sastoji samo od usmjerenog ciklusa duljine recimo 4, Gephi za prosječan stupanje daje 1, iako je očito da je prosječan stupanj 2. Provjerom na različitim grafovima ustanovili smo da daje dvostruko manji rezultat od očekivanog. Stoga smo Gephi koristili za provjeru ostalih rezultata.

[1] Podržani formati u Gephi okruženju su: GEXF, GML, GraphML, Tulip, Pajek NET, GDF, CSV i Compressed ZIP (GEPHI, 2014).

4.3 Opis implementacije

Praktični dio ovog diplomskog rada sastoji se od programa koji za neki graf kreira podgrafove pomoću algoritama opisanim u prethodnom poglavlju, te radi statističku analizu novonastalih grafova. Program je podijeljen na nekoliko dijelova:

- ulaz i izlaz,
- uzorkovanje,
- statistika.

4.3.1 Ulaz i izlaz

Predviđeni format ulaza u ovom programu je format u kojem se podaci u formi liste bridova. Veliki skupovi podataka objavljeni na stranicama http://snap.stanford.edu/data/index.html zapisani su u tom formatu. U tom formatu linije koje počinu znakom '#' su komentari i obično se pišu na početku gdje se opisuje kakvu mrežu predstavlja datoteka, koliko ima čvorova i bridova i koja je vrsta grafa zapisana. Nakon toga slijede reci u kojima su opisani bridovi u grafu. Svaki redak sastoji se od dva identifikatora koji predstavljaju dva čvora koje brid spaja odvojenih znakom '\t'.

Implementacija funkcije za čitanje je jednostavna. Pročitani redci spremaju se u networkx DiGraph objekt.

```
def read_snap(filename):
f = open(filename,  'r')
dg=nx.DiGraph()
for line in skip_comments(f):
edge = [x for x in line.strip().split('\t')]
dg.add_edge(edge[0], edge[1])
f.close()
return d

def skip_comments(file):
""" vraca generator objekt -
linije koje nisu komentari """
for line in file:
if not line.strip().startswith('#'):
yield line
```

Zapisivanje grafa omogućuju dvije networkx metode: `write_edgelist` i `write_pajek`. Metoda `write_edgelist` će zapisati graf u gore spomenutom formatu, dok će `write_pajek` zapisati graf u pajek formatu koji je zgodan jer ga čitaju svi alati za vizualizaciju uključujući Gephi.

4.3.2 Uzorkovanje mreže

Uzorkovanje se obavlja u metodama klase GraphSample koja za svoje podatkovne članove ima sam graf. Metode, izuzevši Represent metodu, imaju za parametar veličinu koja govori koliku ciljanu veličinu grafa želimo postići. Predefinirana vrijednost je 0.3 odnosno želimo uzorak veličine 30% originalnog grafa. Sve metode kao povratnu vrijednost vraćaju podgraf grafa kreiranog u skladu s algoritmom koji implementira.

4.3.2.1 Slučajna šetnja

Metoda za kreiranje podgrafa baziranog na algoritmu slučajne šetnje osim veličine grafa ima još i parametre koji određuju vjerojatnost povratka na početak (engl. f*lyback probability*), te broj koraka (engl. *flyback*) i veličinu za koju u tim koracima treba narasti uzorak (engl. *flyback size*).

```
def randomwalk(self, ratio = 0.3, flyback = 0.15, T = 10, M = 6)
```

Prvi dio metode odnosi se na inicijaliziranje podataka:

```
size = ratio * G.number_of_nodes()
start = random.choice(G.nodes())
sample = [start]
v = sample[-1] # zadnji element liste
period_cnt = 1
size_cnt = 1
```

Inicijalno u uzorku se nalazi samo slučajno odabrani čvor. Varijable `period_cnt` i `size_cnt` omogućuju kontrolu nad rastom uzorka. Sam rad metode događa se dok je veličina uzorka, odnosno broj različitih čvorova u uzorku, manja od ciljane veličine:

```
while len(set(sample)) < size:
```

U svakom koraku dohvaća se zadnji element list `sample`, te svi njegovi susjedi. Ukoliko susjeda nema, početni čvor se bira iz početka, te se dohvaćaju svi njegovi susjedi.

```
while len(candidates) == 0:
      v = random.choice((G.nodes()))
      candidates = nx.neighbors(G, v)
```

Nadalje, računa se vjerojatnost povratka na početni čvor. Ako je slučajno uniformno odabrani broj vraćamo se na početni čvor, inače se u uzorak dodaje slučajno odabrani susjed.

```
if random.random() > flyback:
      v = random.choice(candidates)
      size_cnt += 1
      sample.append(v)
else:
      v = start
```

Prilikom dodavanja čvora u uzorak, brojač dodanih čvorova se uvećava (`size_cnt`), a usvakom prolazu kroz iteraciju uvećava se i brojač koraka (`period_cnt`). Sada provjeravamo da li uzorak raste traženim tempom. Ako ne raste, biramo novi početni čvor i vraćamo vrijednosti brojača na početne.

```
period_cnt += 1
if period_cnt == T:
    if size_cnt < M:
        while 1:
         v = random.choice(G.nodes())
         sample.append(v)
      break
     period_cnt = 1
     size_cnt = 1.
```

Na kraju se iz uzorka kreira podgraf:

```
SG=G.subgraph(sample)
```

4.3.2.2 Metropolis-hastings slučajna šetnja

Metoda za generiranje uzorka bazirana na Metropolis algoritmu ima kao parametar samo veličinu uzorka. Ponovo se uzorak gradi počevši od slučajno uniformno generiranog čvora koji se dodaje u listu čvorova, a algoritam se ponavlja dok je broj različitih elemenata u uzorku manji od ciljane veličine. Nadalje, kao i u metodi slučajne šetnje, uzima se zadnji element u listi čvorova, te se dohvaćaju svi njegovi susjedi i smještaju u listu `candidates`. Ako čvor nema susjeda, bira se slučajno odabrani čvor za nastavak procesa i njegovi susjedi se dodaju u listu `candidates`.

Iz liste kandidata `candidates` bira se čvor za koji se onda računa vjerojatnost prihvaćanja (engl. *acceptance probability*). Ukoliko je stupanj kandidata manji od stupnja čvora iz kojeg smo stigli do kandidata, kandidat će se dodati u listu. Ako je veći dodat će se u listu ako je omjer stupnjeva veći od vjerojatnosti prihvaćanja. Postupak se ponavlja dok ima kandidata.

```
found = 0
while len(candidates) != 0 and found == 0:
    c = random.choice(candidates)  # odaberi bilo kojeg
susjeda
    candidates.remove(c)
    if random.random() < min(1,
float(G.degree(v))/G.degree(c)):
        if c not in sample:
             sample.append(c)   # cvor je prihvacen
             found = 1
```

Ako na ovaj način nije dodan kandidat u listu čvorova bira se novi čvor koji već nije u uzorku kako bi se punjenje uzorka nastavilo dalje.

```
if found == 0:  # nitko od susjeda nema tko nije već u uzorku
    while 1:   # trazi novi cvor koji vec nije u uzorku
        v = random.choice(G.nodes()) # mijenjamo početni cvor
        if v not in sample:
            sample.append(v)
            break
```

Na kraju se iz liste čvorova kreira podgraf koristeći networkx funkciju `subgraph`.

4.3.2.3 Snježna gruda

Implementacija algoritma snježne grude nalazi se u dvije metode: `snowball_direct` i `snowball`. U metodi `snowball_direct` određuje se ciljana veličina uzorka, te se puni inicijalna lista čvorova S (lista kandidata) tako da se slučajnim uniformnim odabirom odabere k čvorova.

```
S = []
for i in range(k):
    v = random.choice(G.nodes())
    S.append(v)
```

Algoritam snježne grude traži vrijednosti za k, te vrijednosti za maksimalni nivo `max_depth` do kojeg će se mreža istraživati.

U metodi `snowball` uvodi se varijabla `current_depth` koja će brojati na kojoj dubini od početnog čvora se nalazimo. Iz liste čvorova S uzima se čvor s početka, te se dodaje u uzorak (listu `sample`) ako već nije u njemu. Ako je s tim dosegnuta željena veličina uzorka ili maksimalna dubina algoritam staje, inače se uzimaju susjedi dodanog čvora, te se prvih k među njima dodaje u listu kandidata S na kraj. Ako čvor nema susjeda, prelazi se na sljedeći čvor u listi kandidata.

```
while S:
    current_depth += 1
    center = S.pop(0)
    if center in sample:
         # Visited this person -- continue
         continue
    else:
        # New person! Don't visit again
        sample.append(center)

        if len(set(sample))>size:
           return sample
        if current_depth == max_depth:
           return sample
        neighbors = G.neighbors(center)
        if len(neighbors) == 0:
            continue
        # extend queue with k neighbors
```

```
if len(neighbors) < k:
     S.extend(neighbors)
 else:
     S.extend(neighbors[0:k])
```

4.3.2.4 Šumski požar

Implementacija algoritma šumskog požara realizirana je u metodi `forestfire`. Inicijalno se bira jedan čvor u mreži i dodaje u listu burning_nodes koja predstavlja listu kandidata iz koje čvorove dodajemo u uzorak. Iz te liste kreira se lista `new_burning_nodes` na sljedeći način: za svaki čvor iz liste `burning_nodes` uzimaju se svi susjedi koji već nisu u uzorku te se među njima bira slučajan broj čvorova koji će se dodati u uzorak, ako već nisu u njemu i u listu `new_burning_nodes`.

```
for burning_node in burning_nodes:
tree_neighbors =
list(set(G.neighbors(burning_node)).difference(sample))
if len(tree_neighbors) == 0:
    continue
else:
    z = random.randint(1,len(tree_neighbors))
    for i in range(z):
    new_burning_node =
tree_neighbors.pop(random.randint(0,len(tree_neighbors)-1))
    sample.add(new_burning_node)
    if len(sample) >= size:
        SG = G.subgraph(sample)
        return SG
    new_burning_nodes.append(new_burning_node)
```

Ako se nakon ovog postupka uzorak nije povećao, vatra se ugasila, nastavljamo potragu od nekog slučajno odabranog čvora koji već nije u uzorku.

```
if visited_size_after == visited_size_before:
    #print "nista novo, palimo novu vatru"
    while 1:
        burning_node =
G.nodes()[random.randint(0,len(G.nodes())-1)]
        if burning_node not in sample:
            new_burning_nodes.append(burning_node)
            break

burning_nodes = new_burning_nodes
```

4.3.2.5 Represent metoda

Represent algoritam implementiran je u metodi `represent`. Ova metoda nema parametara, jer se veličina uzorka ne određuje unaprijed. Na početku u listu `roots` stavljaju se svi korijeni mreže tj. čvorovi kojima je ulazni stupanj nula, a u listu `leafs` stavljaju se svi listovi, tj. čvorovi kojima je izlazni stupanj nula.

```
roots=[n for n in G.nodes() if G.in_degree(n)==0]
leafs=[n for n in G.nodes() if G.out_degree(n)==0]
```

Uzorak je skup (set) `candidates` koji je inicijalno prazan. Punit će se čvorovima koje nađemo na najdužim najkraćim putevima između korijena i listova. Za svaki korijen u listi korijena `roots` pronalazi se najkraći put do svih listova u listi listova `leafs`. Među tim putevima traži se onaj najveće duljine i njegovi čvorovi se dodaju u uzorak.

```
for root in roots:
    paths = []
    for leaf in leafs:
        try:
            paths.append(nx.shortest_path(G, root, leaf))
        except:
            pass
        if paths:
            maxpath = max(paths, key = len)
            candidates.update(maxpath)
```

4.3.3 Statistika

Statistička analiza obavlja se unutar klase `GraphStatistic` čiji su podatkovni članovi sam graf, ime datoteke i metoda za koju se radi statistika. Za svaku metodu posebno radi se statistika koja se sprema u privremene csv datoteke, a koje će se na kraju spojiti u jednu sljedećeg oblika.

method	num nodes	num edges	num con	num strongly	density	avg.degree	avg. clustering coef.	avg. shortest path
graph	115	613	1	115	0.046758199847400005	10.6608695652	0.201608005521	2.7987113402099997
randomwalk	35	76	1	35	0.06386554621849999	4.34285714286	0.180136054422	2.0917431192700002
metropolis	35	69	1	35	0.0579831932773	3.9428571428600003	0.176530612245	2.71244635193
snowball	35	105	1	35	0.0882352941176	6.0	0.248299319728	2.04166666667
forestfire	35	68	1	35	0.0571428571429	3.8857142857099998	0.14952380952400002	2.6049382716
represent	32	60	1	32	0.0604838709677	3.75	0.153459821429	1.8120300751900003

Slika 4.1. Krajnji ispis csv datoteke sa svim metodama

Za svaku metodu računa se broj čvorova, broj bridova, broj povezanih komponenti, broj jako povezanih komponenti i gustoća. Te veličine računaju se koristeći biblioteku networkx. Nadalje, računa se prosječna vrijednost stupnja čvora, prosječna vrijednost koeficijenata grupiranja te prosječna vrijednost duljine najkraćeg puta. Te prosječne vrijednosti računaju se tako da se prvo kreira rječnik frekvencija za svaku od metrika, a onda se u funkciji `calculate_average` računa prosječna vrijednost.

```
def calculate_average(self, freq):
        """ prosječna vrijednost iz dictionary-ja frekvencija """
        naz = sum([freq[key] for key in freq])
        if naz:
            return str(float(sum([key*freq[key] for key in freq]))/naz)
        return 0
```

Frekvencije stupnja računaju se u metodi `degree_frequency`. Uzimaju se svi stupnjevi čvorova u grafu koristeći networkx metodu `degree`, te se za svaku vrijednost stupnja broji broj pojavljivanja čvora tog stupnja.

```
def degree_frequency(self):
        """ računa se frekvencija stupnjeva """
        fq = defaultdict( int )
        for w in nx.degree(self.graph).values():
            fq[w] += 1
        return fq
```

Frekvencije koeficijenata grupiranja računaju se u metodi `clustering_coef_frequency` koristeći networkx metodu `clustering`. Međutim, kako networkx metoda računa koeficijent grupiranja za neusmjerene grafove, prilagodili smo izračun usmjerenim grafovima u skladu sa formulom danom u poglavlju 4.1.3.

```
def clustering_coef_frequency(self):
    cc = nx.clustering(self.ugraph)
    cc.update({k : cc[k]/2. for k in cc.iterkeys()})
    fq = defaultdict( int )
    for w in cc.values():
        fq[w] += 1
    return fq
```

Na kraju, frekvencije duljina najkraćeg puta dobiju se iz networkx metode `shortest_path_length`.

```
def shortest_path_frequency(self):
        G = self.graph
        """ dictionary (duljina najkraćeg puta,
frekvencija)"""
        freq_len = {}
for p in G.nodes():
            """ cl je dictionary, key je vrh do kojeg ide
najkraći put,
                a value duljina najkraćeg puta """
            cl = nx.shortest_path_length(G,p)
            for c, l in cl.items():
                if p != c:
                    if l in freq_len:
                        freq_len[l] += 1
                    else:
                        freq_len[l] = 1
        return freq_len
```

Nakon što su se izračunale vrijednosti za sve metode i zapisale u privremene csv datoteke, datoteke se spajaju u jednu koristeći biblioteku pandas. Kako su podaci u svakoj datoteci bili zapisani u jednom retku, to će se finalna datoteka dobiti spajanjem redaka.

```
df=pd.read_csv(oldbase[0]+'_graph.csv')

for method in l:
    if method == "graph":
        continue
    name = oldbase[0]+'_'+method+'.csv'
    df2=pd.read_csv(name)
    df=pd.concat([df,df2])
df.to_csv(oldbase[0]+'_average.csv', index=None)
```

Same frekvencije stupnja, koeficijenata grupiranja i duljina najkraćeg puta spremaju se u privremene csv datoteke čija su imena kombinacija baze imena (baza imena datoteke u kojoj se nalazi originalni graf), broja koji nam služi tome da se očuva redoslijed metoda u ispisu, imena metode i imena metrike. Privremene datoteke imat će stupčane zapise u kojem će prvi stupac biti vrijednost, drugi frekvencija, treći frekvencija vrijednosti manje od tekuće i četvrti udio frekvencija manjih od tekuće u ukupnoj sumi vrijednosti. Ovo se sve obavlja u metodi `print_freq`.

```
def print_freq(self, fq,  name):
        oldbase = os.path.splitext(self.filename) # rezultat je tuple
        # zelimo sortirani ispis u csv file (sortiran po metodama)
        newname = oldbase[0] + '_'+ str(GraphStatistic.ccnt).zfill(3)
+ '_' + self.method + '_' + name +'.csv'
GraphStatistic.ccnt += 1
        """ pisemo pojedinacne vrijednosti frekvencije u datoteke
            - svaki u svoju"""
        f = open(newname, 'w')
        f.write("%s, freq, Actual freq, %s\n" % (name, self.method))

        num = sum(fq.values())
        rmb = 0
        afq = defaultdict( int )
        ratio = defaultdict( int )
        afq[0]=0
        for w in sorted(fq):
            afq[w] = rmb + fq[w]
            rmb = afq[w]
            ratio[w] = float(afq[w])/num
  f.write( "%s, %s, %s, %s\n" % (w, fq[w], afq[w], ratio[w]))
        f.close()
        return ratio
```

Ovako kreirane datoteke spajaju se za sve metode po tipu metrike. To se izvodi u metodi `write_result_to_csv` koristeći pandas biblioteku. U ovom slučaju ne spajaju se retci, već stupci i to stupci nejednake duljine. Postupak za frekvencije stupnjeva za sve uzorke ide tako da se uzmu sve privremene csv dateteke čije ime završava na "degree_distribution", te se kreira lista podataka (DataFrame) iz svake od njih. Biblioteka pandas omogućava spajanje (concat) listi podataka u jednu, te zapisivanje rezultata u novu csv datoteku. Spajanje za koeficijent grupiranja i duljine najkraćih puteva radi se analogno tome.

```
        text_files = [f for f in os.listdir(path) if
        f.endswith('degree_distribution.csv')]
        a = []
        for f in sorted(text_files):
        """ kreira se lista podataka (DataFrame) iz csv datoteke """
        a.append(pd.read_csv(path+'/'+f))
        """ spajanje u jedan DataFrame """
        m1=pd.concat(a, axis=1)
        """ zapisivanje u csv """
        m1.to_csv(oldbase[0]+'_degree.csv', index=None)
        """ uklonit cemo pojedinacne csv datoteke """
```

```
            for f in text_files:
            os.remove(path+'/'+f)
```

I konačno, statističke podatke želimo nacrtati u obliku grafa kumulativnih frekvencija. U tu svrhu koristimo biblioteku matplotlib i njenu klasu pyplot. Grafovi se zapisuju u pdf datoteke čije je ime kombinacija baznog imena grafa i imena metrike čije se frekvencije iscrtavaju.

```
def draw_ratios(self, ratio, name, l, methods):
        oldbase = os.path.splitext(self.filename) # rezultat je tuple
        newname = oldbase[0] + '_'+ name +'.pdf'
        f = open(newname, 'w')
        plt.figure()
        colors = itertools.cycle(["r", "b", "g", "y", "cyan", "magenta"])
        lines = ["-","--","-.",":","-..","--."]
        linecycler = itertools.cycle(lines)
        for key in l:
            x = sorted(ratio[key].keys())
            y = [ratio[key][xx] for xx in x]
            plt.plot(x,y, next(linecycler), color=next(colors))
        if name == "degree":
            plt.xscale('log')
        plt.legend(l, loc = 4)
        plt.xlabel(name)
        plt.ylabel('Ratio')
        plt.title(name)
        plt.savefig(newname)
        plt.close()
        f.close()
```

5. Usporedna analiza implementiranih metoda uzorkovanja

Sljedeće tablice (tablica 5.1. i 5.2.) odgovaraju rezultatima dobivenim alatom Gephi, a prikazuju osnovne podatke o mreži: broj čvorova, veza, te broj povezanih komponenti i jako povezanih komponenti. S obzirom da Represent metoda nema unaprijed zadanu veličinu uzorka, već broj čvorova u uzorku ovisi o broju čvorova u najkraćim putevima između svih korijena i svih listova, ta metoda nam je bila okvir za ciljanu veličinu uzorka za ostale metode. Sve ostale metode imaju kao parametar ciljanu veličinu. Na našim testnim primjerima ciljana veličina je 30% veličine originalnog grafa. Broj veza između čvorova, odnosno gustoća mreže varira od metode do metode.

method	num nodes	num edges	num conn comp.	num strongly conn. comp
graph	34546	421578	61	21608
randomwalk	10364	95163	158	8281
metropolis	10364	77921	208	10187
snowball	7482	58442	1	5240
forestfire	10364	94088	1	5918
represent	12861	75215	96	10228

Tablica 5.1. Osnovni podaci o uzorcima za mrežu cit-HepPh

method	num nodes	num edges	num conn comp	num strongly conn. comp
graph	27770	352807	143	20086
randomwalk	8331	86701	283	7194
metropolis	8331	90995	267	6909
snowball	8332	102715	1	5468
forestfire	8331	109496	1	5519
represent	10229	54040	218	8946

Tablica 5.2. Osnovni podaci o uzorcima za mrežu cit-HepTh

U **Error! Reference source not found.**5.3. i 5.4. vidimo da se najgušća mreža dobije algoritmom šumskog požara, dok je najmanje gusta i najbliža originalnoj mreži, mreža dobivena Represent metodom.

method	density	avg.degree	avg. clustering coef.	avg. shortest path
graph	0.000353260413931	24.4067619985	0.142398066045	11.6929683986
randomwalk	0.000886043863636	18.3641451177	0.142349507513	10.96629365210000
metropolis	0.000725507013213	15.0368583558	0.138338213494	6.1395873218
snowball	0.00104411350062	15.622026196199998	0.14142749889599998	11.7135521588
forestfire	0.0008760347513400001	18.156696256300002	0.143500729373	11.37414431940000
represent	0.00045476680133999995	11.696602130499999	0.12131228052100002	11.4584582118

Tablica 5.3. Prosječne vrijednosti u uzorcima mreže cit-HepPh

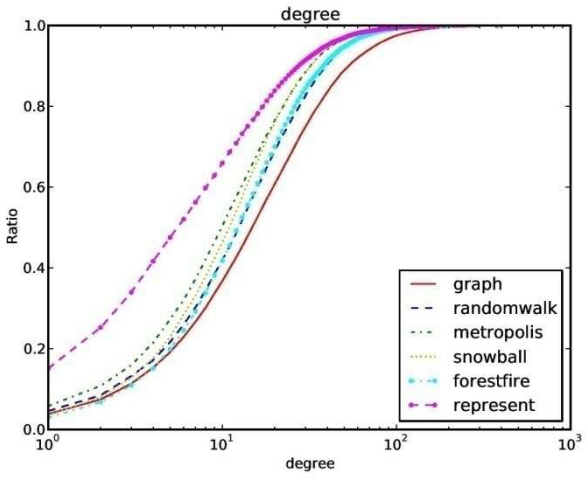
method	density	avg.degree	avg. clustering coef.	avg. shortest path
graph	0.00045751050778199994	25.409218581199998	0.156009747903	8.46013729391
randomwalk	0.0012493438138699998	20.814067939	0.151964304776	5.421517193350001
metropolis	0.00131121948239	21.8449165766	0.150541127893	5.517824442359999
snowball	0.0014797470224	24.655544887199998	0.160589864595	7.258483064810001
forestfire	0.00157781513758	26.2864001921	0.15967290235200002	6.48064954113
represent	0.000516525114189	10.5660377358	0.120650411243	8.73499406118

Tablica 5.4. Prosječne vrijednosti u uzorcima mreže cit-HepTh

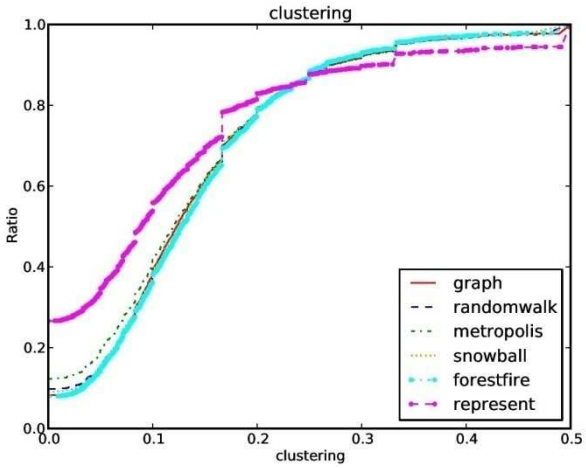
Što se tiče broja povezanih komponenti, algoritmi snježne grude i šumskog požara za rezultat imaju mrežu koja ima jednu komponentu, dok ostale metode za rezultat daju više povezanih komponenti nego što ih ima u originalnom grafu. Razlog za to je što se kod te dvije metode dodaje odjednom veći broj susjeda nekog čvora, dok kod slučajne šetnje i Metropolis metode dodaje se u svakom koraku najviše jedan susjed ovisno o random funkciji. Represent algoritam s druge strane, koristi potpuno drugačiju metodu građenja mreže, te povezujući korijene sa listovima i tražeći najdulji najkraći put među njima, očekivano najdulje puteve nalazi u različitim komponentama.

Prosječni stupanj čvora najbliži je prosječnom stupnju čvora u originalnoj mreži u mreži dobivenoj primjenom algoritma šumskog požara. Razlog za to je što se broj susjeda nekog čvora dodanog u uzorak mijenja ovisno o random funkciji, te se u svako koraku dodaje veći broj susjeda. U ostalim metodama je ili broj susjeda ograničen (algoritam snježne grude) ili se dodaje po jedan odabrani susjed. U Represent metodi dodaju se oni susjedi nekog čvora koji se nalaze na najkraćem putu između nekog korijena i lista, a njih za jedan najkraći put može biti najviše dva.

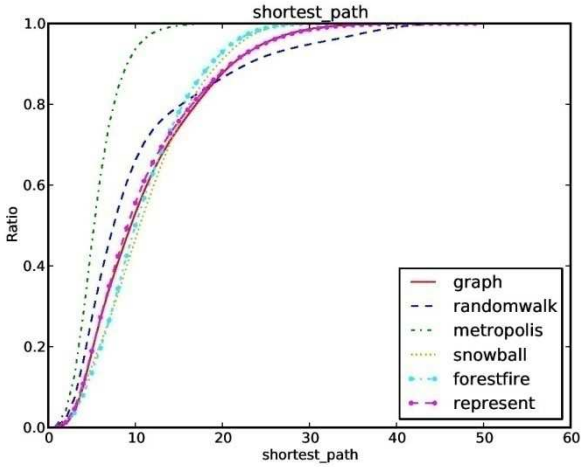
Prosječni stupanj grupiranja u svim uzorcima oslikava originalnu mrežu (od 0.3% odstupanja u algoritmu snježne grude do 15% razlike u Represent algoritmu u odnosu na originalnu mrežu), dok je prosječna duljina najkraćeg puta u uzorku gotovo jednaka prosječnoj duljini najkraćeg puta u originalnoj mreži u uzorcima dobivenim algoritmom snježne grude, šumskog požara i Represent algoritmom.



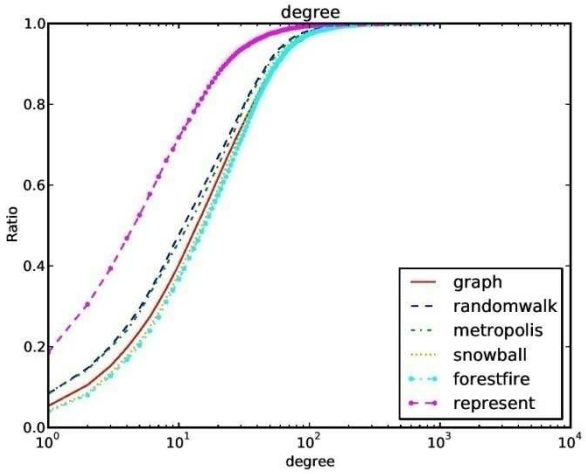
Slika 5.1. Kumulativna distribucija stupnjeva za cit-HepPh



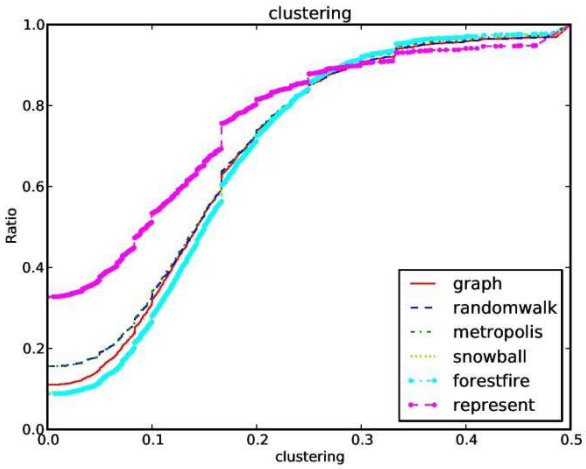
Slika 5.2. Kumulativna distribucija koeficijenata grupiranja za cit-HepPh



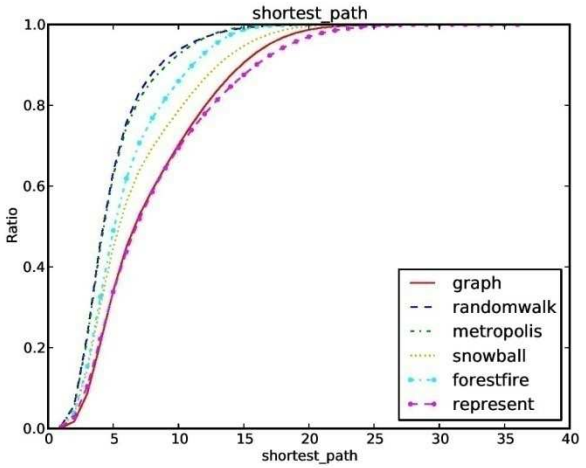
Slika 5.3. Kumulativna distribucija duljina najkraćih puteva za cit-HepPh



Slika 5.4. Kumulativna distribucija stupnjeva za cit-HepTh



Slika 5.5. Kumulativna distribucija koeficijenata grupiranja za cit-HepTh



Slika 5.6. Kumulativna distribucija duljina najkraćih puteva za cit-HepTh

Slika 5.1. i 5.4. prikazuju kumulativnu distribuciju stupnjeva, a slika 5.2. i 5.5. kumulativnu distribuciju koeficijenata grupiranja, a slika 5.3, i 5.6. kumulativnu distribuciju najkraćih puteva u mreži.

Iz slika možemo zaključiti da uzorci dobiveni svim algoritmima dobro čuvaju stupanj vrha, među njima najbliže originalnoj distribuciji stupnjeva je uzorak dobiven algoritmom šumskog požara. Graf kumulativne distribucije koeficijenata grupiranja pokazuje nam također da svi algoritmi dobro oslikavaju grupiranje u originalnoj mreži. Najveće odstupanje imaju uzorci dobiveni Represent algoritmom i algoritmom snježne grude. Naime, za neki čvor u uzorku dobivenom Represent algoritmom sigurno su u uzorku dva susjeda s kojim je čvor zajedno na najkraćem putu najveće duljine između nekog korijena i lista, dok ostali susjedi ne moraju biti u uzorku. Što se tiče uzorka dobivenim algoritmom snježne grude, on za primjer cit-HepTh pokazuje dosta veliko odstupanje. To je posljedica toga što taj algoritam staje ukoliko se iscrpe susjedi ili se dođe do max_depth dubine pretrage, a potraga se ne nastavlja dalje od nekog drugog čvora. Konačno, frekvencija duljina najkraćih puteva u grafu najtočnija je u uzorku dobivenim Represent algoritmom, što je posljedica načina na koji se uzorak stvarao - kao unija najkraćih puteva.

6. Zaključak

Cilj inteligentnih tutorskih sustava je da se nastavni sadržaj ponuđen učeniku računalno generira ovisno o napredovanju učenika. Da bi to bilo moguće potrebno je iz skupa područnog znanja odabrati podskup koji bi reprezentirao cijeli skup. To se može postići algoritmima za uzorkovanje preuzetim iz područja analize kompleksnih mreža ili algoritmima kreiranim baš za reprezentaciju područnog znanja.

U ovom radu opisani su rezultati izvođenja Python implementacije pet algoritama za uzorkovanje: algoritam slučajne šetnje, Metropolis-Hastings algoritam slučajne šetnje, algoritam snježne grude, algoritam šumskog požara i Represent algoritam. Represent algoritam je napravljen baš za generiranje reprezentativnog podskupa područnog znanja.

Implementacija je testirana na dvije mreže citata, jedne sa 34546 čvorova i 421578 bridova, a druge sa 27770 čvorova i 352807 bridova. Uzorci generirani algoritmom šumskog požara i Represent algoritmom točnije predstavljaju originalnu mrežu, s tim da među tim metodama Represent metoda ima nešto dulje vrijeme izvođenja. Zbog toga se ta metoda preporuča na ne prevelikim skupovima podataka. Ipak, za potrebe reprezentacije područnog znanja taj je algoritam najbolji jer najbolje čuva svojstvo duljine najkraćeg puta koji u ovom slučaju predstavljaju putanje od nadkoncepata prema podkonceptima.

7. Literatura

Urediti literaturu prema pravilima za seminar

(ANDERSON, 1988) J. R. Anderson: *The Expert Module*. Foundations of Intelligent Tutoring System. Hillsdale, New Jersey: Lawrence Erlbaum Associates; 1988. pp. 21-53.

(BAK, CHEN, TANG, 1990) Per Bak, Kan Chen (Department of Physics, Brookhaven National Laboratory, Upton, NY, USA), Chao Tang (Institute for Theoretical Physics, University of California, Santa Barbara, CA, USA): *A forest-fire model and some thoughts on turbulence*, 1990.

[BILGIN, YENER, 2008] Bilgin., C.C.; Yener, B.: „Dynamic Network Evolution: Models, Clustering, Anomaly Detection“, 2008.

[BOCCALETTI, LATORA, et al., 2006] Boccaletti, S.; Latora V.; Moreno, V.; Chavez M.; Hwang, D.-U.: „Complex networks: Structure and dynamics“, 2006.

(BURNS, 1998) H. L. Burns, C. G. Capps: *Foundations of Intelligent Tutoring Systems: AnIntroduction.* In Polson MC, Richardson JJ, editors. Foundations of IntelligentTutoring Systems. Hillsdale, New Jersey: Lawrence Erlbaum Associates;1988. pp. 1-19.

(DROSSEL, SCHWABL, 1992) B. Drossel, F. Schwabl: *Self-Organized Critical Forest-Fire Model.* Physik-Department der Technichen Universität München, Germany, 1992.

(GEPHI, 2014) dostupno 25.08.2014. na: https://gephi.github.io/features/

(GJOKA, KURANT, BUTTS, MARKOPOULOU, 2010) Minas Gjoka, Maciej Kurant, Carter T. Butts, Athina Markopoulou: *Walking in Facebook: A Case Study of Unbiased Sampling of OSNs*, 2010.

(GOODMAN, 1961) Leo A. Goodman: *Snowball Sampling*. University of Chicago, USA, 1961.

(GRUBER, 1993) Gruber, T. R. (1993). A translation approach to portable ontology specifications. Knowledge acquisition, 5(2), pp. 199-220.

(GRUBIŠIĆ, 2012) Grubišić, A.: Model prilagodljivoga stjecanja znanja učenika u sustavima e-učenja. Doktorski rad. Zagreb, 2012.

(HONKELA, 2007) T. Honkela: *Philosophical Aspects of Neural, Probabilistic and Fuzzy Modeling of Language Use and Translation*. In IJCNN 2007. International Joint Conference on Neural Networks; 2007; Orlando, Florida. pp. 2881-2886.

(KLEINBERG, 1999) J. Kleinberg: The *Small-World Phenomenon: An Algorithmic Perspective*. Cornell Computer Science Technical Report 99-1776, Department of Computer Science, Cornell University, October 1999.

(LEE, XU, EUN, 2012) Chul-Ho Lee, Xin Xu, Do Young Eun: *Beyond Random Walk and Metropolis-Hastings Samplers: Why You Should Not Backtrack for Unbiased Graph Sampling*. Department of Electrical and Computer Engineering, North Carolina State University Raleigh, NC, USA, 2012.

(LESKOVEC, FALOUTSOS, 2006) J. Leskovec, C. Faloutsos: *Sampling from Large Graphs*. School of Computer Science, CAMEGIE Mellon University, Pittsburgh, PA, USA, 2006.

(LOVÁSZ, 1993) L. Lovász: *Walks on Graphs: A Survey*, 1993.

(MATPLOTLIB, 2014) *Plotting commands summary*, dostupno 25.08.2014. na: http://matplotlib.org/api/pyplot_summary.html

(METROPOLIS et al., 1953) Nicholas Metropolis et al.: *Equation of State Calculations by Fast Computing Machines*. Los Alames Scientific Laboratory, Los Alamos, New Mexico

(NATURE, 2014) Physics portal, dostupno 20.08.2014. na: http://www.nature.com/physics/looking-back/pearson/index.html

(NETWORKX, 2014) dostupno 25.08. 2014. na: http://networkx.github.io/

(NEWMAN, 2003) Newman M. E. J.: „The structure and function of complex networks“, SIAM review, 2003.

(PEARSON, 1905) Karl Pearson, časopis Nature, 1905.

(SELF, 1974) J. A. Self: *Student models in computer-aided instruction*. International Journal of Man-Machine Studies. 1974. 6 pp. 261-276.

(SELF, 1990) J. Self: *Theoretical Foundations for Intelligent Tutoring Systems*. Journal of Artificial Intelligence in Education. 1990. 1(4). pp. 3-14.

(VELJAN, 1989) D. Veljan (1989). Kombinatorika s teorijom grafova. Zagreb: Školska knjiga.

(WAY, 1991) E. C. Way: "Knowledge Representation and Metaphor" Norwell, MA, USA: Kluwer Academic Publishers; 1991.

[WATTS, STROGATZ, 1998] Watts, D. J. ; Strogatz, S. H.: „Collective dynamics of small world networks“, Nature, 1998.

(WENGER, 1987) E. Wenger: *Artificial intelligence and tutoring systems: Computational approaches to the communication of knowledge*. Los Altos, CA: Morgan Kaufman; 1987.

(WOOLF, 1992) B. P. Woolf: AI in Education. In Shapiro SC, editor. Encyclopedia of Artificial Intelligence. New York: John Wiley & Sons, Inc.; 1992. pp. 434-444.

8. Prilozi

8.1 Dodatak I – dijelovi programskog koda

8.1.1 Start.py

```
1.  from IO.io import *
2.  from Stat.stat import *
3.  from Sample.sample import *
4.  import os
5.  import networkx as nx
6.
7.
8.  def main():
9.      filename = raw_input("enter filename: ")
10.     if not os.path.exists(filename):
11.         print "not exists"
12.         return
13.     try:
14.         digraph, graph = read_snap(filename)
15.
16.     except TypeError:
17.         print "read graph error"
18.         return
19.
20.     oldbase = os.path.splitext(filename)
21.
22.     stats={}
23.
24.     sample = GraphSample(filename, digraph)
25.
26.     l=["graph","randomwalk", "metropolis", "snowball", "forestfire", "represent"]
27.     methods = {l[0]:sample.graph, l[1]:sample.randomwalk, l[2]:sample.metropolis_direct,
     l[3]:sample.snowball_direct, l[4]:sample.forestfire, l[5]:sample.represent}
28.
29.     for method in l:
30.         print "method: ", method
31.         if method == "graph":
32.             smp = digraph
33.         else:
34.             smp = methods[method]()
35.
36.         fn_out = oldbase[0]+"_"+method+"_pajek.net"
37.         write_pajek(smp, fn_out)
38.         usmp = smp.to_undirected()
39.         stat = GraphStatistic(oldbase[0], smp, usmp, method)
40.         stats[method] = stat.do_statistic()
41.
42.         df=pd.read_csv(oldbase[0]+'_graph.csv')
43.
44.     for method in l:
45.         if method == "graph":
```

```
46.             continue
47.         name = oldbase[0]+'_'+method+'.csv'
48.         df2=pd.read_csv(name)
49.         df=pd.concat([df,df2])
50.     df.to_csv(oldbase[0]+'_average.csv', index=None)
51.
52.     draw = DrawStatistic(filename)
53.     draw.draw_all(stats, l, methods)
54.
55.     write_results_to_csv(filename)
56.
57.
58. if __name__ == "__main__":
59.     main()
```

8.1.2 IO.io

```
1.  import networkx as nx
2.  import pandas as pd
3.  import os
4.
5.  def skip_comments(file):
6.      for line in file:
7.          if not line.strip().startswith('#'):
8.              yield line
9.
10. def read_snap(filename):
11.     f = open(filename,  'r')
12.     dg=nx.DiGraph()
13.     g=nx.Graph()
14.     for line in skip_comments(f):
15.         edge = [x for x in line.strip().split('\t')]
16.         dg.add_edge(edge[0], edge[1])
17.         g.add_edge(edge[0], edge[1])
18.     f.close()
19.     return dg, g
20.
21. def write_graph(graph, filename):
22.     nx.write_edgelist(graph, filename)
23.
24. def read_graph(filename):
25.     return nx.read_edgelist(filename,create_using=nx.DiGraph())
26.
27. def write_pajek(graph, filename):
28.     nx.write_pajek(graph, filename)
29.
30. def write_results_to_csv(filename):
31.     path = os.path.dirname(filename)
32.     oldbase = os.path.splitext(filename)
33.     print "write_results_to_csv oldbase ", oldbase[0]
34.     text_files = [f for f in os.listdir(path) if f.endswith('degree_distribution.csv')]
35.
36.     a = []
37.     for f in sorted(text_files):
38.         a.append(pd.read_csv(path+'/'+f))
39.
```

```
40.     m1=pd.concat(a, axis=1)
41.
42.     m1.to_csv(oldbase[0]+'_degree.csv', index=None)
43.
44.     for f in text_files:
45.         os.remove(path+'/'+f)
46.
47.     text_files = [f for f in os.listdir(path) if f.endswith('cluster_distribution.csv')]
48.     a = []
49.     for f in sorted(text_files):
50.         a.append(pd.read_csv(path+'/'+f))
51.     m2=pd.concat(a, axis=1)
52.     m2.to_csv(oldbase[0]+'_clustering.csv', index=None)
53.     for f in text_files:
54.         os.remove(path+'/'+f)
55.
56.     text_files = [f for f in os.listdir(path) if f.endswith('path_length_distribution.cs
    v')]
57.     a = []
58.     for f in sorted(text_files):
59.         a.append(pd.read_csv(path+'/'+f))
60.     m3=pd.concat(a, axis=1)
61.     m3.to_csv(oldbase[0]+'_shortest_path.csv', index=None)
62.     for f in text_files:
63.         os.remove(path+'/'+f)
```

8.1.3 Sample.sample

```
1.  import networkx as nx
2.  import os
3.  import random
4.  import time
5.  from collections import defaultdict
6.  import sys
7.  sys.setrecursionlimit(10000)
8.
9.  class GraphSample():
10.     def __init__(self, filename, graph):
11.         self.graph = graph
12.         self.filename = filename
13.
14.     def graph(self):
15.         print "graph ", self.graph.number_of_nodes()
16.         return self.graph
17.
18.     def randomwalk(self, ratio = 0.3, flyback = 0.15, T = 10, M = 6):
19.         print "random walk"
20.         G = self.graph
21.
22.         size = ratio * G.number_of_nodes()
23.         start = random.choice(G.nodes())
24.         sample = [start]
25.         v = sample[-1]
26.         period_cnt = 1
27.         size_cnt = 1
28.
```

```
29.         while len(set(sample)) < size:
30.             v = sample[-1]
31.             candidates = nx.neighbors(G, v)
32.
33.             while len(candidates) == 0:
34.                 v = random.choice((G.nodes()))
35.                 candidates = nx.neighbors(G, v)
36.
37.             if random.random() > flyback:
38.                 v = random.choice(candidates)
39.                 size_cnt += 1
40.                 sample.append(v)
41.             else:
42.                 v = start
43.
44.             period_cnt += 1
45.             if period_cnt == T:
46.                 if size_cnt < M:
47.                     while 1:
48.                         v = random.choice(G.nodes())
49.                         sample.append(v)
50.                         break
51.                 period_cnt = 1
52.                 size_cnt = 1
53.
54.         SG=G.subgraph(sample)
55.         print "random walk ", SG.number_of_nodes()
56.         return SG
57.
58.     def metropolis_direct(self, ratio = 0.3):
59.         print('sample metropolis direct')
60.         G = self.graph
61.         size = ratio * G.number_of_nodes()
62.         sample = [random.choice(G.nodes())]
63.
64.         while len(set(sample)) < size:
65.             v = sample[-1]
66.             candidates = nx.neighbors(G, v)
67.                         while len(candidates) == 0:
68.                 candidates = nx.neighbors(G, v)
69.
70.             found = 0
71.             while len(candidates) != 0 and found == 0:
72.                 c = random.choice(candidates)
73.                 candidates.remove(c)
74.
75.                 if random.random() < min(1, float(G.degree(v))/G.degree(c)):
76.                     if c not in sample:
77.
78.                         sample.append(c)
79.                         found = 1
80.             if found == 0:
81.                 while 1:
82.                     v = random.choice(G.nodes())
83.                     if v not in sample:
84.                         sample.append(v)
85.
86.                         break
87.
```

```
88.
89.         SG=G.subgraph(sample)
90.         print "metropolis ", SG.number_of_nodes()
91.
92.         return SG
93.
94.     def snowball(self, size, S, k, max_depth, sample):
95.         G = self.graph
96.         current_depth = 0
97.
98.         while S:
99.             current_depth += 1
100.                center = S.pop(0)
101.                if center in sample:
102.                    continue
103.                else:
104.                    sample.append(center)
105.
106.                if len(set(sample))>size:
107.                    return sample
108.
109.                if current_depth == max_depth:
110.                    return sample
111.
112.                neighbors = G.neighbors(center)
113.
114.                if len(neighbors) == 0:
115.                    continue
116.
117.                if len(neighbors) < k:
118.                    S.extend(neighbors)
119.                else:
120.                    S.extend(neighbors[0:k])
121.
122.            print "sample ", sample
123.            return sample
124.
125.        def snowball_direct(self, ratio = 0.3, center = None, k = 5, max_depth = 100):
126.            print('sample snowball direct')
127.            G = self.graph
128.            size = ratio * G.number_of_nodes()
129.            S = []
130.            for i in range(k):
131.                v = random.choice(G.nodes())
132.                S.append(v)
133.            sample = self.snowball(size, S, k = 10, max_depth = 100, sample = [])
134.            SG=G.subgraph(sample)
135.            print "snowball ", SG.number_of_nodes()
136.
137.            return SG
138.
139.        def forestfire(self, ratio = 0.3):
140.            print "forest fire"
141.            G = self.graph
142.            size = ratio * G.number_of_nodes()
143.
144.            burning_node = G.nodes()[random.randint(0,len(G.nodes())-1)]
145.            burning_nodes = [burning_node]
146.            sample = set([burning_node])
```

```
147.
148.            new_burning_nodes = []
149.            while len(sample) <= size:
150.                visited_size_before = len(sample)
151.                for burning_node in burning_nodes:
152.                    tree_neighbors = list(set(G.neighbors(burning_node)).difference(samp
    le))
153.                    if len(tree_neighbors) == 0:
154.                        continue
155.                    else:
156.                        z = random.randint(1,len(tree_neighbors))
157.                        for i in range(z):
158.                            new_burning_node = tree_neighbors.pop(random.randint(0,len(t
    ree_neighbors)-1))
159.                            sample.add(new_burning_node)
160.                            if len(sample) >= size:
161.                                SG = G.subgraph(sample)
162.                                return SG
163.                            new_burning_nodes.append(new_burning_node)
164.                visited_size_after = len(sample)
165.                if visited_size_after == visited_size_before:
166.                    while 1:
167.                        burning_node = G.nodes()[random.randint(0,len(G.nodes())-1)] #
168.
169.                        if burning_node not in sample:
170.                            new_burning_nodes.append(burning_node)
171.                            break
172.                burning_nodes = new_burning_nodes
173.
174.            SG = G.subgraph(sample)
175.
176.            print "forest fire ", .number_of_nodes()
177.            return SG
178.
179.        def represent(self):
180.
181.            G = self.graph
182.
183.            roots=[n for n in G.nodes() if G.in_degree(n)==0]
184.            leafs=[n for n in G.nodes() if G.out_degree(n)==0]
185.
186.            i = len(roots)
187.            candidates = set()
188.            t = 0
189.            for root in roots:
190.                print(i, len(candidates), round(t,2))
191.                i -= 1
192.                t_start = time.time()
193.                paths = []
194.                for leaf in leafs:
195.                    try:
196.                        paths.append(nx.shortest_path(G, root, leaf))
197.                    except:
198.                        pass
199.                if paths:
200.                    maxpath = max(paths, key = len)
201.                    candidates.update(maxpath)
202.
203.                t = time.time() - t_start
```

```
204.
205.            SG = nx.subgraph(G, candidates)
206.            print "represent ", SG.number_of_nodes()
207.            return SG
```

8.1.4 Stat.stat

```
1.  import networkx as nx
2.
3.  from collections import defaultdict
4.  from draw import *
5.  import os
6.
7.  class GraphStatistic():
8.      ccnt = 0
9.      def __init__(self, filename, graph, ugraph, method):
10.         self.graph = graph
11.         self.ugraph = ugraph
12.         self.filename = filename
13.         self.method = method
14.
15.     def num_nodes(self):
16.         return self.graph.number_of_nodes()
17.
18.     def num_edges(self):
19.         return self.graph.number_of_edges()
20.
21.     def num_conn_comp(self):
22.         return nx.number_connected_components(self.ugraph)
23.
24.     def num_strongly_conn_comp(self):
25.         return nx.number_strongly_connected_components(self.graph)
26.
27.     def degree_frequency(self):
28.         """ racuna se frekvencija stupnjeva """
29.         fq = defaultdict( int )
30.         for w in nx.degree(self.graph).values():
31.             fq[w] += 1
32.         return fq
33.
34.     def shortest_path_frequency(self):
35.         G = self.graph
36.
37.         cnt, maxcnt = 0,len(G.nodes())
38.
39.         freq_len = {}
40.
41.         for p in G.nodes():
42.             if cnt%500==0:
43.                 print cnt, 'of', maxcnt
44.             cnt += 1
45.
46.             cl = nx.shortest_path_length(G,p)
47.
48.             for c, l in cl.items():
```

```
49.                 if p != c:
50.                     l = int(l)
51.                     if l in freq_len:
52.                         freq_len[l] += 1
53.                     else:
54.                         freq_len[l] = 1
55.         return freq_len
56.
57.     def density(self):
58.         return nx.density(self.graph)
59.
60.     def clustering_coef_frequency(self):
61.         cc = nx.clustering(self.ugraph)
62.         cc.update({k : cc[k]/2. for k in cc.iterkeys()})
63.
64.         fq = defaultdict( int )
65.         for w in cc.values():
66.             fq[w] += 1
67.         return fq
68.
69.     def calculate_average(self, freq):
70.         naz = sum([freq[key] for key in freq])
71.         if naz:
72.             return str(float(sum([key*freq[key] for key in freq]))/naz)
73.         return 0
74.
75.     def print_freq(self, fq,  name):
76.         oldbase = os.path.splitext(self.filename)
77.
78.         newname = oldbase[0] + '_'+ str(GraphStatistic.ccnt).zfill(3) + '_' + self.metho
    d + '_' + name +'.csv'
79.         GraphStatistic.ccnt += 1
80.
81.         f = open(newname, 'w')
82.         f.write("%s, freq, Actual freq, %s\n" % (name, self.method))
83.
84.         num = sum(fq.values())
85.         rmb = 0
86.         afq = defaultdict( int )
87.         ratio = defaultdict( int )
88.         afq[0]=0
89.         for w in sorted(fq):
90.             afq[w] = rmb + fq[w]
91.             rmb = afq[w]
92.             ratio[w] = float(afq[w])/num
93.             f.write( "%s, %s, %s, %s\n" % (w, fq[w], afq[w], ratio[w]))
94.
95.         f.close()
96.
97.         return ratio
98.
99.     def write_metrics(self):
100.            oldbase = os.path.splitext(self.filename)
101.            newname = oldbase[0] +'_'+self.method+'.csv'
102.            f = open(newname, 'w')
103.
104.            f.write("method, num nodes, num edges, num conn comp., num strongly conn. co
    mp., density, avg.degree, avg. clustering coef., avg. shortest path\n")
105.            f.write( "%s," % self.method)
```

```
106.            f.write( "%s," % self.num_nodes())
107.            f.write( "%s," % self.num_edges())
108.            f.write( "%s," % self.num_conn_comp())
109.            f.write( "%s," % self.num_strongly_conn_comp())
110.            f.write( "%s," % self.density())
111.
112.            fq_degree = self.degree_frequency()
113.            f.write("%s," % self.calculate_average(fq_degree))
114.
115.            fq_clustering = self.clustering_coef_frequency()
116.            f.write("%s," % self.calculate_average(fq_clustering))
117.
118.            fq_spath = self.shortest_path_frequency()
119.            f.write("%s" % self.calculate_average(fq_spath))
120.
121.            return fq_degree, fq_clustering, fq_spath
122.
123.        def do_statistic(self):
124.            fq_degree, fq_clustering, fq_spath = self.write_metrics()
125.
126.            ratio_degree = self.print_freq(fq_degree, 'degree_distribution')
127.            ratio_clustering = self.print_freq(fq_clustering,  'cluster_distribution' )

128.            ratio_spath = self.print_freq(fq_spath,  'path_length_distribution')
129.            return [ratio_degree, ratio_clustering, ratio_spath]
```

8.1.5 Stat.draw

```
1.  import os
2.  import matplotlib.pyplot as plt
3.  import itertools
4.  class DrawStatistic():
5.      def __init__(self, filename):
6.          self.filename = filename
7.
8.      def draw_ratios(self, ratio, name, l, methods):
9.          oldbase = os.path.splitext(self.filename)
10.         newname = oldbase[0] + '_'+ name +'.pdf'
11.
12.         f = open(newname, 'w')
13.         plt.figure()
14.         colors = itertools.cycle(["r", "b", "g", "y", "cyan", "magenta"])
15.
16.         lines = ["-","--","-.",":","-..","--."]
17.         linecycler = itertools.cycle(lines)
18.         for key in l:
19.             x = sorted(ratio[key].keys())
20.             y = [ratio[key][xx] for xx in x]
21.             plt.plot(x,y, next(linecycler), color=next(colors))
22.
23.         if name == "degree":
24.             plt.xscale('log')
25.         plt.legend(l, loc = 4)
26.         plt.xlabel(name)
27.         plt.ylabel('Ratio')
28.
29.         plt.title(name)
```

```
30.
31.         plt.savefig(newname)
32.         plt.close()
33.         f.close()
34.
35.     def draw_all(self, stats, l, methods):
36.         ratio_degree={}
37.         ratio_cluster={}
38.         ratio_spath={}
39.         ratio_degree["graph"]=stats["graph"][0]
40.         ratio_cluster["graph"]=stats["graph"][1]
41.         ratio_spath["graph"]=stats["graph"][2]
42.         for key in l:
43.             ratio_degree[key]=stats[key][0]
44.             ratio_cluster[key]=stats[key][1]
45.             ratio_spath[key]=stats[key][2]
46.
47.
48.         self.draw_ratios(ratio_spath, "shortest_path", l, methods)
49.         self.draw_ratios(ratio_degree, "degree", l, methods)
50.         self.draw_ratios(ratio_cluster, "clustering", l, methods)
```