|  |
| --- |
|  |
| TEORIJA GRAFOVA: |
| IMPLEMENTACIJA KLASE ZA UPOTREBU GRAFOVA U JAVI |
|  |
|  |
|  |

Sadržaj

[1.UVOD 2](#_Toc30579538)

[1.a O ODABIRU TEME 3](#_Toc30579539)

[1.b UVOD U GRAFOVE 4](#_Toc30579540)

[1.b.1 TERMINOLOGIJA 4](#_Toc30579541)

[1.b.2 PRIMJERI GRAFOVA I NJIHOVE PRIMJENE 6](#_Toc30579542)

[2.OBJAŠNJENJE ALGORITAMA 8](#_Toc30579543)

[2.a BFS 9](#_Toc30579544)

[2.b DFS 11](#_Toc30579545)

[2.c DIJKSTRA 12](#_Toc30579546)

[2.d BELLMAN FORD 15](#_Toc30579547)

[2.e ISPITIVANJE BIPARTITIVNOSTI POMOĆU BFS-A 17](#_Toc30579548)

[3.PRIMJERI GRAFOVA I ALGORITAMA NAD NJIMA 19](#_Toc30579549)

[3.a BFS, DFS I BELLMAN FORD 20](#_Toc30579550)

[3.b BFS, DFS I DIJKSTRA 23](#_Toc30579551)

[4.IMPLEMENTACIJA GLAVNOG PROGRAMA 26](#_Toc30579552)

[5.IMPLEMENTACIJA KLASE „GRAF“ 29](#_Toc30579553)

[6.ZAKLJUČAK 36](#_Toc30579554)

[7.BIBLIOGRAFIJA 37](#_Toc30579555)

# 1.UVOD

-Prije nego što bilo šta napišem o grafovima i njihovoj upotrebi, želio bih da se zna na koji način sam došao do odabira ove teme. Naime, prvobitna tema koju sam trebao istraživati su bile neuralne mreže. Međutim, pošto je vremena malo, a posla puno, odlučio sam se za nešto (barem meni) jednostavnije, ali opet nimalo bezazleno – grafove.

1.a O ODABIRU TEME

-Ovo je jedna od onih tema za koju nije potrebna akademska pozadina da bi se krenula proučavati, jer ne zahtijeva veliko poznavanje matematskih i drugih prirodnih zakona, već iziskuje logički i intuitivan način razmišljanja. Sam projekat će se sastojati iz dva dijela – praktični, koji predstavlja klasu razvijenu u Javi za jednostavniju upotrebu grafova, te teoretski u kojem ću malo analizirati rad algoritama klase. Klasa je napravljena tako da se može koristiti od strane onih koji znaju upotrijebiti grafove, ali nije nužno da poznaju način rada njihovih algoritama (apstrakcija). Ipak, za one koji su znatiželjni, rad će sadržati i objašnjenje načina na koji njihovi algoritmi rade, te matematske dokaze pomoću kojih možemo dokazati korektnost rada istih.

## 1.b UVOD U GRAFOVE

### 1.b.1 TERMINOLOGIJA

-Da biste mogli što bolje i lakše shvatiti teoriju grafova i sam projekat, potrebno je uvesti osnovne pojmove koji bi trebalo da se poznaju. Oni su dati u nastavku:

**Graf –** je struktura podataka koja se sastoji od čvorova i ivica koje povezuju te čvorove. Ivice se u zavisnosti od autora teksta mogu zvati i bridovi, a čvorovi tačke. Mi ćemo ih zvati čvorovi i ivice, a označavati ćemo ih sa V (engl.vertices – čvorovi), i E (engl. edges – ivice).

**Čvor –** apstrahirana struktura koja u sebi sadrži neke informacije, može predstavljati gradove, ljude, ili bilo šta na što bi se teorija grafova mogla korisno primijeniti.

**Ivica –** spona, most između najviše dva čvora. Također apstrahiran pojam, može predstavljati putove između gradova, ili jačinu prijateljstva između ljudi. To opet zavisi od toga gdje se naš graf primjenjuje.

**Težina ivice –** to je broj (ili opet neka druga apstrahirana struktura) koja nam pokazuje koliko je bitna veza između neka dva čvora. Kao primjer možemo navesti graf prijateljstava gdje su ljudi čvorovi, a prijateljstva ivice. Težina ivice bi tu predstavljala koliko je prijateljstvo između dva proizvoljno odabrana čvora (čovjeka) jako. Ukoliko između neka dva čvora ne postoji ivica, kažemo da je težina između ta dva čvora ima neku dogovorenu vrijednost (najčešće nula) koja opet, zavisi od mjesta na kojem primjenjujemo graf.

**Susjedi(Komšije) –** za dva čvora možemo reći da su komšije ukoliko su povezani direktno, samo sa jednom ivicom.

**Jednostavan graf** – za graf kažemo da je jednostavan ukoliko u njemu ne postoje ciklusi, i ukoliko su mu bilo koja dva čvora povezana najviše sa jednom ivicom.

**Ciklus –** zatvoreni put u grafu u kojem su svi čvorovi međusobno različiti.

**Stepen čvora –** predstavlja broj ivica koje ulaze ili izlaze iz promatranog čvora.

**Acikličan graf –** graf koji ne sadrži niti jedan ciklus takav da je moguće doći iz nekog čvora A u taj isti čvor (vidjećemo da je moguće da graf sadrži ciklus koji ne zadovoljava drugi dio definicije, pa da opet bude acikličan).

**Cikličan graf –** graf koji sadrži barem jedan ciklus takav da je moguće doći iz nekog čvora A u taj isti čvor.

**Povezan graf –** za graf kažemo da je povezan ukoliko je moguće iz jednog proizvoljnog čvora doći u svaki drugi čvor, odnosno postoji put do svakog drugog čvora.

**Put u grafu –** put p(A, B) predstavlja skup čvorova i ivica između tih čvorova pomoću kojih možemo doći od **početnog čvora – A** do **odredišta, krajnjeg čvora – B.**

**Nepovezan graf -** graf koji nije povezan, odnosno graf u kojem postoji čvor do kojeg nije moguće doći iz barem jednog od preostalih čvorova.

**Usmjeren graf –** je onaj graf kod kojeg ivica od A do B nije ekvivalentna ivici od B do A, odnosno ukoliko možemo doći od A do B, to ne znači da možemo doći od B do A.

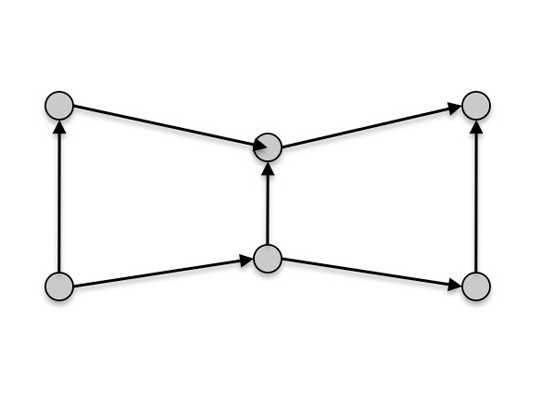
**Neusmjeren graf –** je onaj graf kod kojeg je ivica od A do B ekvivalentna ivici od B do A, odnosno ukoliko je ivica zadana od A do B, to implicira da postoji ivica i od B do A.

**Bipartitan graf –** zagraf kažemo da je bipartitan ako i samo ako je moguće rasporediti njegove čvorove u dva disjunktna skupa tako da svaka ivica povezuje jedan čvor iz prvog, i jedan čvor iz drugog skupa. Također, ako je graf bipartitan, ne smije postojati ivica takva da povezuje dva čvora koja pripadaju istom skupu.

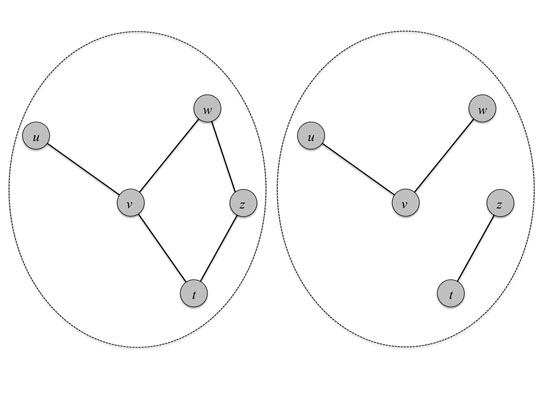
-Bitno je napomenuti da postoji mnogo termina i algoritama korisnih za obradu i proučavanje grafova, ali sam ja odabrao samo neke za koje sam smatrao da su neophodni za daljnje praćenje ovog rada.

### 1.b.2 PRIMJERI GRAFOVA I NJIHOVE PRIMJENE

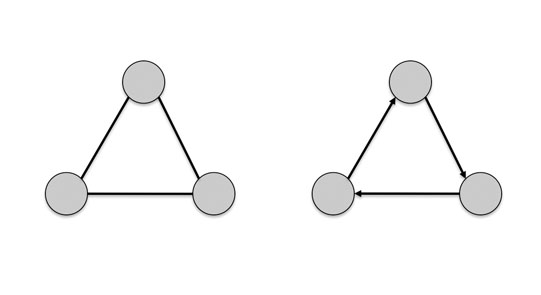
Na slikama ispod su dati primjeri nekih vrsta grafova koje smo definisali u prethodnom dijelu:



Slika 1.1 : usmjeren i acikličan graf



Slika 1.2 : primjer povezanog i nepovezanog neusmjerenog grafa



Slika 1.3 : razlika između usmjerenog i neusmjerenog cikličnog grafa

Grafovi se koriste za modeliranje raznih vrsta informacija, a neke od njih su :

1.Simuliranje rutera : definitivno jedna od korisnijih primjena, većina rutera koristi grafove i neke od njihovih algoritama za optimizaciju povezivanja istih. Dakle, ovdje ruteri predstavljaju čvorove, a ivice su predstavljene kao internetska konekcija između dva rutera.

2.TSP problem – takozvani Travelling Salesman Problem, problem trgovačkog putnika, koji predstavlja problem obilaska svakog od gradova, tako da zbir težina između ivica (putova) koji povezuju te gradove bude minimalan. Ovdje grafovi služe za simulaciju gradova i putova, te pronalazak optimalnih putanja između istih.

3.Predstavljanje molekula pri proučavanju određenih svojstava istih u biologiji – gdje svaki od atoma predstavlja čvor, a veze između njih su naravno ivice.

# 2.OBJAŠNJENJE ALGORITAMA

U nastavku slijedi objašnjenje načina rada (pseudo kod), svrha nekih od algoritama koje sam implementirao u klasi, te dokaz njihove korektnosti, odnosno ispravnosti ukoliko nisu trivijalni. Spomenut ću također i vremensku složenost istih. Također, nakon ovog dijela bit će prikazana implementacija svakog od njih, te implementacija pomoćnog main programa sa kojim sam ispitao napravljenu klasu.

Algoritmi kojima ćemo se baviti u ovom radu su :

1.Bfs (Breadth first search – pretraživanje grafa u širinu)

2.Dfs (Depth first search – pretraživanje grafa u dubinu)

3.Dijkstra (pronalazak najkraćeg puta)

4.Bellman Ford (pronalazak najkraćeg puta)

5.Ispitivanje bipartitivnosti grafa pomoću Bfs-a

## 2.a BFS

-Bfs (engl. Breadth first search – pretraživanje u širinu) je jedan od najpoznatijih algoritama za pretraživanje grafova, ali ujedno i najkorisniji, jer se njegovom laganom modifikacijom mogu dobiti algoritmi korisni za različite primjene.

-Samo ime odaje način njegovog rada – algoritam posjećuje sve čvorove grafa u širinu. To znači da posjećuje prvo sve komšije onog čvora od kojeg kreće, zatim posjećuje sve komšije njegovog prvog komšije, pa sve komšije njegovog drugog komšije…., sve dok svi čvorovi ne budu posjećeni. Očigledno da će nam biti potreban niz u kojem ćemo držati posjećenost svakog od čvorova, te red u kojem ćemo držati čvorove čije komšije treba tek posjetiti.

Pseudo kod kojim bi se BFS mogao predstaviti :

1. BFS (početniČvor):
   1. Inicijaliziraj sve čvorove kao neposjećene;
   2. Označi početniČvor kao posjećen;
   3. Ubaci početniČvor u red (queue);
   4. Sve dok ima neposjećenih čvorova:
      1. Skini trenutni čvor sa reda;
      2. Za svakog komšiju od trenutnog čvora:
         1. Ubaci ga u red za posjećivanje;
         2. Označi ga kao posjećenog;

-Vidimo da smo se koristili strukturom red (engl. queue), koja je FIFO struktura(first in, first out).

-Sličan algoritam će biti i za Dfs, ali će koristiti malo drugačiju strukturu za čuvanje neposjećenih čvorova.

-Što se tiče same vremenske složenosti, ona je O (V + E), gdje je V – broj čvorova, a E broj ivica. Zašto? - Ukoliko pogledamo koliko puta se puta izvrši petlja pod d) – onoliko puta koliko ima čvorova, što je O(V). Sada, ostaje još petlja pod ii). Ona se, kada se sabere broj komšija od svakog od čvorova, svodi na broj ivica O(E). Dakle, konačna složenost je O(V + E).

-Gore navedeno objašnjenje vremenske kompleksnosti možda jeste šturo i sažeto, ali ipak ono nije tema ovog projekta, već samo neka „laička“ procjena načina promatranja vremenske složenosti algoritma.

-Kada je u pitanju povezan graf, ispravnost algoritma je trivijalna. Naime, pretpostavimo suprotno, to jest da poslije izvršenja petlje pod d) ovog algoritma postoji neki čvor A koji nije posjećen. Tada petlja pod d) ne bi mogla nikada završiti, jer je uvjet da bi algoritam završio taj da su svi čvorovi posjećeni – kontradikcija.

-Imamo još jedan problem. Kako znamo da ćemo morati posjetiti sve komponente povezanog grafa ovim algoritmom, odnosno da se algoritam neće odvijati beskonačno vremena? – Recimo da smo od čvorova {1, 2, 3, 4} nekog grafa posjetili čvorove {1, 2, 3}, i da ne možemo doći do čvora 4. Tada bi se algoritam odvijao u beskonačnost – nikada ne bi izašao iz petlje d), jer bi imali neki čvor kojeg je nemoguće posjetiti. Šta onda?

-Pa, kao što ste možda i pretpostavljali, radi se o povezanom grafu. Prema definiciji za povezani graf (vidi 1. b 1 Terminologija), možemo tvrditi da je moguće u povezanom grafu iz bilo kojeg čvora doći u bilo koji drugi, tako da navedeni problem u paragrafu iznad, i nije problem.

-Navedene stvari i potencijalni problemi za nekoga ko se prvi put susreće sa ovim algoritmom su možda nekome iskusnijem trivijalni. Ipak, ovaj algoritam je sam po sebi trivijalan, kao i njegov sljedbenik Dfs, tako da smo sa rečenim objasnili sve moguće probleme u shvatanju istog.

-Kada je u pitanju nepovezan graf, samo je potrebno da se algoritam odvije nad svakom od njegovih nepovezanih komponenti. „Dokaz korektnosti“ tada vršimo za svaki od njegovih komponenata analogno prethodno navedenom.

## 2.b DFS

-Dfs (engl.depth first search – pretraživanje u dubinu) se koristi iz istih razloga kao i Bfs – za pretraživanje svih čvorova grafa. I on se može modifikovati tako da radi korisnije stvari od osnovne pretrage, ali se može koristiti za drugačije stvari od Bfs-a zbog drugačijeg redoslijeda pretrage grafa, što je i osnovna razlika između Bfs i Dfs algoritama.

-On posjećuje čvorove u dubinu, tj. prvog komšiju početnog čvora, nakon toga prvog komšiju komšije početnog čvora…..itd, dok ne posjeti sve čvorove. Sad bi trebalo biti jasno zašto se zove pretraga u dubinu. Detaljniji pregled pretrage može se naći u odjeljku provjere rada napravljene klase.

-Pseudo kod je veoma sličan, samo što se umjesto reda koristi stek, što je i logično jer on oponaša rekurziju – pomoću koje se također može implementirati ovaj algoritam. Pseudo kod je dat u nastavku:

1. DFSPretraga (početniČvor):
   1. Inicijaliziraj sve čvorove kao neposjećene;
   2. Ubaci početniČvor na stek;
   3. Sve dok postoji neposjećen čvor:
      1. Skini čvor sa steka;
      2. Za sve komšije skinutog čvora:
         1. Ako je komšija neposjećen, ubaci ga na stek;
         2. Označi ga kao posjećenog;

-Pošto je algoritam analogan BFS-u, složenost mu je ista : O(V + E). Algoritam koristi stek koji je LIFO struktura (Last In, First Out).

-Također, dokazivanje ispravnosti je analogno BFS-u u slučaju povezanog, ali i u slučaju nepovezanog grafa.

Što se tiče samog DFS-a, on je, može se reći, više intuitivan od BFS-a zbog mogućnosti implementacije pomoću rekurzije, koja je, većini programera koji pokušavaju naći neke razumne i logične razloge zašto algoritmi rade onako kako rade, jasnija i bliža.

## 2.c DIJKSTRA

Algoritam je dobio ime po svome kreatoru, Edsger Wybe Dijkstri, i rješava idući problem:ukoliko nam je dan početni čvor A i graf G (V, E), pronaći najkraće udaljenosti od zadanog čvora A do svih ostalih. Udaljenost između dva čvora može biti definisana na mnoge načine, ali ćemo mi udaljenost između čvorova (A, B) računati kao zbir težina svih ivica koje se nalaze u putu koji počinje čvorom A, i završava čvorom B.

Ovaj algoritam radi samo ukoliko su **težine ivica nenegativni brojevi**.Za one grafove kod kojih se iz bilo kojeg razloga mora sastojati od negativnih težina ivica, morat ćemo koristiti Bellman-Ford algoritam, koji će biti analiziran odmah nakon ovoga.

Princip rada algoritma:

Prvo se mijenjaju udaljenosti komšija čvora od kojeg krećemo, to je trenutni čvor. Nakon toga, trenutni čvor označavamo kao posjećenog, što u ovom slučaju znači da smo našli optimalnu udaljenost od zadanog čvora. Onda odabiremo čvor koji trenutno ima **NAJMANJU** udaljenost od početnog čvora, i postavljamo ga kao trenutni čvor. Proces se ponavlja, sve dok svi čvorovi ne budu posjećeni – odnosno dok ne nađemo optimalne udaljenosti za svaki čvor.

Također je bitno dodati da ovaj algoritam nije trivijalan, te da se za shvatanje načina rada, te dokaza njegove tačnosti treba izdvojiti određena količina vremena i vježbe.…

Pseudo kod je dat u nastavku :

1. Dijkstra (početniČvor):
   1. Postavi sve udaljenosti čvorova od početniČvor na beskonačnost;
   2. Postavi sve posjećenosti čvorova na false;
   3. Postavi udaljenost početniČvor od samog sebe na 0;
   4. Dok postoje neposjećeni čvorova:
      1. Uzmi čvor sa minimalnom udaljenosti od početnogČvora koji nije posjećen(neka je to trenutniČvor);
      2. Označi trenutniČvor kao posjećen(pronađena optimalna udaljenost);
      3. Za svakog komšiju od trenutniČvor:
         1. Ako je udaljenost[komšija] > udaljenost[trenutniČvor] + ivica(trenutniČvor, komšija) onda:
            1. Udaljenost[komšija] = udaljenost[trenutniČvor] + ivica(trenutniČvor, komšija);

Što se tiče vremenske kompleksnosti, ona zavisi od strukture podataka pomoću koje smo implementirali algoritam, pa neću puno govoriti o tome jer to nije tema projekta. Reći ću samo da je ona O((V + E)logV) ukoliko se koristi binarna ili fibonačijeva gomila(binary or fibonacci heap) kao prioritetni red (priority queue) da bi se minimalna vrijednost povlačila što efikasnije.

Dokaz ispravnosti algoritma :

Definišimo prvo neke pojmove korisne za dokaz:

-Neka je A skup koji sadrži sve čvorove za koje smo do sad našli optimalne udaljenosti od početnog čvora, i oni su svi posjećeni. Ovo možemo tvrditi po indukciji za broj posjećenih čvorova, odnosno za veličinu skupa A.

-Neka je B čvor koji od svih neposjećenih (tj. onih za koje nismo našli optimalnu udaljenost) ima minimalnu udaljenost od početnog čvora, ali preko posjećenih čvorova(čvorova iz skupa A). Ta minimalna udaljenost čvora B je predstavljena kao zbir udaljenosti nekog čvora C iz skupa A i ivice od C do B. Neka je put kojim je predstavljena ta udaljenost p(početniČvor, B);

Pošto Dijkstrin algoritam uzima samo čvorove koji se mogu opisati svojstvima prethodno opisanog čvora B i postavlja njihove udaljenosti kao optimalne, dokaz ispravnosti algoritma se svodi na dokaz da je udaljenost čvora B optimalna, tj. da ne postoji bolja udaljenost za čvor B od gore definisane (odnosno one koju koristi sam algoritam).

Pretpostavimo sada da postoji put z (početniČvor, B) koji ima manju udaljenost od početniČvor do B nego put p. Naime, kada bi takav put postojao, on bi imao dvije opcije:

a)Put z sadrži samo već posjećene čvorove(čvorove iz skupa A)

b)Put z sadrži i neposjećene čvorove

a) Ukoliko put z sadrži samo već posjećene čvorove, iz naše definicije puta p(rekli smo da je put p minimalna udaljenost od početnog čvora do B samo kroz posjećene čvorove) slijedi da taj put z mora imati veću ili jednaku udaljenost od puta p, što je kontradikcija sa pretpostavkom da postoji put z koji ima manju udaljenost od puta p.

b) Kada bi put z sadržao i neposjećene čvorove i ujedno bio najkraći, to bi značilo da u tom trenutku postoji čvor X (na prelazu iz posjećenih u neposjećene) takav da je neposjećen, i da ima manju udaljenost od početnog čvora nego čvor B, samo preko čvorova iz skupa A. Naime, to je nemoguće. Zašto? **Zato što smo čvor B definisali kao čvor najkraće udaljen od početnog čvora (preko čvorova iz skupa A) u promatranom trenutku.**

Dakle, uspjeli smo dokazati da ne postoji put z koji ima kraću udaljenost od puta p definisanog na način Dijkstrinog algoritma. Odnosno, uspjeli smo dokazati da je udaljenost čvora definisanog kao čvor B zaista minimalna, a pošto Dijkstrin algoritam uzima čvorove na identičan način na koji smo mi definisali čvor B, to možemo tvrditi da će ovaj algoritam zaista naći najmanju udaljenost za svaki čvor u nekom grafu…

Još ću navesti zašto ovaj algoritam ne može da radi sa negativnim ivicama:

U gore navedenom dokazu koristili smo to da algoritam uvijek uzima čvor koji ima trenutnu minimlanu vrijednost. To se može zato što će svaka iduća ivica taj put samo povećavati (jer je pozitivna) – u najboljem slučaju put će ostati isti (ivica = 0).

Ali, ukoliko graf sadrži i negativne ivice, tada, iako odaberemo čvor sa trenutno minimalnom udaljenosti, niko nam ne može garantovati da će taj put ostati minimalan nakon 2, 3 ili k koraka, upravo zato što imamo negativne ivice, koje će možda udaljenost puta koji u tom trenutku nije bio optimalan smanjiti svojom „negativnošću“.

Tada nam pohlepna (greedy) taktika Dijkstre neće pomoći, već ćemo morati ispitati sve slučajeve pomoću Bellman Forda, koji slijedi u nastavku…

## 2.d BELLMAN FORD

Dobio ime po Richard Ernest Bellman-u, i Lester Randolph Ford-u, jednostavniji je od Dijkstre. Prednost mu je što radi sa negativnim ivicama (jer nije pohlepni algoritam, provjerava sve ivice), ali mana mu je što ima lošiju vremensku kompleksnost, baš zato što nije pohlepan. Ukoliko ne postoje negativni ciklusi, algoritam vraća najkraće udaljenosti svakog čvora od zadanog početnog, u suprotnom javlja da se u grafu nalazi negativni ciklus. Problem sa negativnim ciklusom je taj što bi se mogao ponoviti beskonačno puta u cilju da se minimizira udaljenost, koja bi teoretski tada bila minus beskonačno. Naravno, algoritam se može urediti tako da vraća i takve udaljenosti (ako je u nekom slučaju to potrebno).

Algoritam je veoma jednostavan. On mijenja vrijednosti za svaku od ivica (V – 1) puta, gdje je V broj čvorova u datom grafu. Sada ćemo prikazati pseudo kod za algoritam, a u dijelu gdje ću promatrati ispravnost će biti objašnjeno zašto je ovaj naizgled bezvezni algoritam tačan:

1. BellmanFord (početniČvor):
   1. Postavi udaljenosti svakog od čvorova na beskonačnu vrijednost;
   2. Postavi udaljenost početnog čvora na 0;
   3. Ponavljaj (V – 1) puta :
      1. Za svaku ivicu i(početni, krajnji) u grafu :
         1. Ukoliko je udaljenost[početni ] + i(početni, krajnji) < udaljenost[krajnji],
            1. onda udaljenost[krajnji] = udaljenost[početni] + i(početni, krajnji);
   4. Za svaku ivicu i(početni, krajnji) u grafu: //za provjeru negativnih ciklusa:
      1. Ukoliko je udaljenost[početni] + i(početni, krajnji) < udaljenost[krajnji], onda
         1. ispiši(„Graf sadrži negativni ciklus, nemoguće naći udaljenosti!“);

Što se tiče same složenosti, ona je O(VE), što je puno sporije od Dijkstre, zbog provjeravanja svih ivica u grafu. Naravno, Dijkstra uopće ne može da radi ukoliko graf sadrži negativne ivice, tako da uz pomoć negativnih ivica i jednostavnosti implementacije, Bellman Ford može da parira Dijkstri.

Zašto ovaj algoritam uopće radi?

Počnimo od grafa koji nema negativnih ciklusa. Negativne cikluse ćemo objasniti na kraju. Sada, koliko maksimalno ivica može sadržati optimalan put p od čvora A do čvora B u nekom grafu? – Maksimalno (C – 1) ivica, gdje je C – broj čvorova u datom putu p. Ovo važi ukoliko taj optimalni put p ne sadrži cikluse u sebi. Ali, već smo rekli da sada promatramo grafove bez negativnih ciklusa, pa ukoliko optimalni put u sebi sadrži ciklus, zbroj težina ivica u tom ciklusu mora biti 0, inače put ne bi bio optimalan(uzeli bismo isti taj put bez ciklusa). Odavde vidimo da je moguće izbaciti taj ciklus(jer ne utiče na konačni zbir težina ivica – zbir mu je nula, ili ga eventualno povećava), pa i u grafu u kojem imamo nenegativne cikluse optimalan put p može sadržati najviše (C - 1) ivica.

Sada, koliko maksimalno ivica možemo imati u putu koji povezuje početni čvor sa svim ostalim čvorovima? Pošto smo rekli da je ukupno V čvorova, to je u našem najduljem putu maksimalno (V – 1) ivica.Zbog toga imamo prvu petlju koja se odvija (V – 1) puta.

Sada, dovoljno nam je dokazati da nakon (V – 1) puta ponavljanja petlje pod c) našeg algoritma, imamo optimalne putove sa najviše (V – 1) ivica u sebi.

To ćemo dokazati matematskom indukcijom po broju ponavljanja petlje pod c), taj broj ćemo označiti sa x:

1. za x = 0, minimalnu udaljenost smo našli samo za početniČvor, koji ima 0 ivica koje ga povezuju sa samim sobom, dakle tvrdnja vrijedi.
2. za x = n, pretpostavimo da smo našli minimalne udaljenosti za čvorove koji su od početnog udaljeni najviše za n ivica.
3. Promatrajmo proizvoljan čvor D, koji je udaljen najviše (n + 1) ivicu od početnog, i probajmo dokazati da će optimalni put od početnog čvora do D imati najviše (n + 1) ivicu, odnosno dovoljno dokazati : brojIvica[D] <= (n + 1).
4. Da li čvor D može imati komšije udaljene od početnog za više od n ivica? Ne. Naime, ukoliko bi imao, tada bi čvor D bio udaljen za barem (n+2) ivica od početnog, što se kosi sa uvjetom da je D čvor udaljen najviše za (n+1) ivicu od početnog... Dakle, sve njegove komšije su udaljene **najviše n ivica od početnog čvora.** Očigledno je da minimalna udaljenost od D mora biti : udaljenost[D] = udaljenost[komšija] + ivica(komšija, D), gdje je komšija – takav susjed za kojeg će udaljenost[D] biti minimalna. Ali, znamo da je udaljenost[komšija] udaljenost koja je dobivena iz puta sa najviše n ivica, pa u vidu ivica se ova formula može predstaviti kao: brojIvica[D] = brojIvica[komšija] + 1 <= n + 1, izčega slijedi : brojIvica[d] <= (n + 1), što je trebalo i dokazati…

Pošto smo dokazali da će se optimalne udaljenosti za svaki čvor naći nakon najviše (V – 1) iteracija petlje pod c), to sada možemo promatrati grafove sa negativnim ciklusima. Ukoliko imamo negativni ciklus, optimiziranje (smanjivanje udaljenosti za proizvoljni čvor) nastavit će se i nakon (V – 1) iteracija, pa je dovoljno da napravimo još jednu iteraciju. Ukoliko u toj iteraciji možemo smanjiti udaljenosti za koje smo dokazali da su najmanje, to znači da u našem grafu postoji ciklus kod kojeg je zbir težina njegovih ivica negativan.

Još je bitno napomenuti da se prethodno navedeni algoritam u pseudo kodu može optimizirati. Ako u nekom od ponavljanja petlje pod c) nema promjena, to znači da je algoritam već pronašao optimalne putanje za svaki od čvorova. Tada se može uvesti boolova varijabla koja prati da li ima promjena u trenutnoj iteraciji. Ukoliko nema, to znači da su minimalne udaljenosti pronađene, te da možemo terminirati naš proces(algoritam).

## 2.e ISPITIVANJE BIPARTITIVNOSTI POMOĆU BFS-A

U uvodnom dijelu smo u pojmovima naveli bipartitan graf, i ovaj algoritam nam pomaže da otkrijemo da li je neki graf bipartitan. Da bi se problem bolje shvatio i pojednostavio, potrebno je promijeniti tačku gledišta na isti, odnosno promijeniti perspektivu.

Dakle, koristit ćemo tehniku bojanja. Jasno nam je da ukoliko je graf bipartitan, ne smijemo imati dva čvora iz iste grupe povezana ivicom. Podijelimo ih u disjunktne skupine A i B. Čvorove iz skupine A bojit ćemo plavom, a one iz skupine B crvenom bojom. Sad se problem svodi na to da ne smiju postojati dva čvora iste boje takva da su povezana ivicom.

Ovaj algoritam nije ništa drugo do proširenje BFS algoritma. Krećemo od početnog čvora, i bez gubitka općenitosti ga bojimo u crveno. Sada, pošto ne smiju postojati dva čvora koja su iste boje, a da su povezani ivicom, to svi njegovi susjedi moraju bti druge boje – u našem slučaju plave. Sada za sve njegove susjede radimo isto, dok ne posjetimo sve čvorove u grafu. Za svakog od komšija, ukoliko nije posjećen dodjeljujemo suprotnu boju, i stavljamo ga na red (queue). Ukoliko je posjećen a boja mu je ista kao i kod početnog, algoritam prestaje, jer smo otkrili da graf nije bipartitan. Ukoliko smo posjetili sve čvorove i uspjeli da ih obojimo po zadanom pravilu, tada je graf bipartitan.

Pseudo kod je dat u nastavku:

1. Boolean jeLiBipartitan (početniČvor) :
   1. Inicijalizuj sve čvorove kao neobojene;
   2. Postavi početniČvor kao obojen u crveno;
   3. Dodaj početniČvor na queue;
   4. Dok queue nije prazan:
      1. Uzmi prvi element iz reda(neka je to trenutniČvor);
      2. Obriši prvi element nakon uzimanja;
      3. Za sve komšije od trenutnog čvora:
         1. Ukoliko komšija nije obojen: //(ako nije obojen, nije ni posjećen)
            1. Oboji ga suprotnom bojom;
            2. Umetni ga na queue;
         2. Ukoliko je komšija obojen:
            1. Ako je obojen istom bojom kao i trenutniČvor:

return false;

* 1. //ukoliko sam došao do ovdje, znači da je graf bipartitan:
  2. return true;

Što se tiče složenosti, algoritam je u stvari BFS sa dodanim nizom za čuvanje boja, stoga je njegova kompleksnost ekvivalentna kompleksnosti BFS-a, što je O(V + E).

Objašnjenje složenosti je isto kao i za BFS algoritam.

Još nam je ostao dokaz korektnosti, odnosno ispravnosti:

Dokaz za ispravnost je ovdje dosta trivijalan. Dovoljno je primjetiti da **ukoliko je moguće obojiti čvorove grafa u dvije različite boje tako da ne postoje dva čvora iste boje takva da su povezana ivicom,** graf mora biti bipartitan. Nije nam bitno da li imamo ivice između čvorova iz različitih grupa, ali nam je bitno i neophodno da nemamo veza između čvorova iz istih grupa. Pošto je to jedini uvjet za bipartitnost grafa, dokaz je završen.

# 3.PRIMJERI GRAFOVA I ALGORITAMA NAD NJIMA

U nastavku ćemo prikazati dva primjera grafova na koje sam primjenio algoritme iz klase koju sam implementirao, i uporediti dobivene rezultate sa očekivanim rezultatima. Za svaki od primjera ćemo provjeriti bipartitnost grafa.

Za jedan ćemo koristiti negativne ivice i provjeriti BFS, DFS, te Bellman forda, dok ćemo za drugi sa pozitivnim ivicama provjeriti BFS, DFS, te Dijkstrin algoritam.

## 3.a BFS, DFS I BELLMAN FORD

Na dolje prikazanom grafu ispitat ćemo redoslijed posjećivanja čvorova pomoću DFS, BFS algoritama, te ćemo naći minimalne udaljenosti pomoću Bellman Ford algoritma.

Na ovom grafu nećemo ispitivati Dijkstru zato što su ivice negativne, a već smo rekli da je ovaj algoritam sa svojom greedy metodom nemoćan kada su u pitanju negativne ivice.

B

2

D

4

3 4

A

3 1 -5

2

E

5

C

Slika 3.1 : prvi graf za ispitivanje napravljenih algoritama

Iako gore nacrtani graf ima samo jednu negativnu ivicu, to nas sprečava da na njemu isprobamo dijkstru.

Prvo ćemo pregledati redoslijed posjeta pomoću BFS, nakon toga sa DFS, pa ćemo vidjeti koje su minimalne udaljenosti pomoću Bellman Ford algoritma, pa ćemo ispitati bipartitnost grafa.

Redoslijed posjete čvorova pomoću BFS-a (početni čvor A):

**{A, B, C, D, E}**

Redoslijed posjete čvorova pomoću DFS-a(početni čvor A):

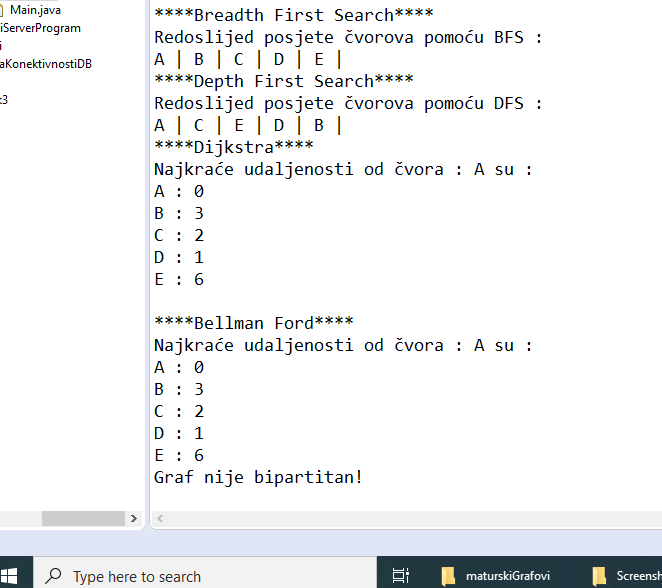
**{A, C, E, D, B}**

Udaljenosti koje vraća Bellman Ford algoritam:

|  |  |
| --- | --- |
| **Ime čvora** | **Konačna udaljenost od čvora A** |
| **A** | **0** |
| **B** | **3** |
| **C** | **2** |
| **D** | **1** |
| **E** | **6** |

Što se tiče bipartitnosti grafa prikazanog na gornjoj slici, očigledno da **ovaj graf nije bipartitan,** što se vidi odmah iz prva tri čvora.

Gore navedene vrijednosti su ispravne, tj. one koje bi moji algoritmi trebali da vrate.Na sljedećoj stranici je data slika stvarnih povratnih vrijednosti mojih algoritama, koje su, kao što se vidi iz priloženog ekvivalentne onima koje su ispravne:



Slika 3.2 : rezultati algoritama za prvi graf

-Vidimo da nam je i Dijkstra dao tačan rezultat, iako imamo negativnu ivicu.Ovdje je bitno napomenuti da **Dijkstra može dati tačan rezultat ukoliko imamo negativne ivice, ali i ne mora (nismo 100 % sigurni da će dati tačan rezultat). To zavisi od strukture grafa koji se analizira.**

## 3.b BFS, DFS I DIJKSTRA

Dolje imamo još jedan primjer, ali sada sa nenegativnim ivicama. Također možemo provjeriti i Bellman Ford algoritam, što nije neophodno zbog toga što bi u principu Dijkstra i Bellman Ford trebali dati identičan izlaz, ako su dobro implementirani.

Na ovom primjeru nisam koristio slova, već brojeve, da bi pokazao varijabilnost u nazivima čvorova u mojoj klasi (poznatiju kao template-i u c++-u). To je pojava kada se implementira klasa na takav način da se neki od parametara apstrahira. Njegova prava vrijednost (String, Integer i sl) se dodjeljuje tek prilikom stvaranja objekta napisane klase.

Stoga je ovaj drugi primjer sa čvorovima čija su imena tipa Integer, dok u prvom primjeru imamo tip String.

3

3

5

6

5

2 4

2

3

2

2

4

3

1

Slika 3.3 : drugi graf za ispitivanje algoritama

Kao početni čvor u ovom grafu uzet ćemo čvor 5.

Redoslijed posjete čvorova pomoću BFS-a:

**{5, 1, 2, 3, 4, 6}**

Redoslijed posjete čvorova pomoću DFS-a:

**{5, 3, 6, 2, 4, 1}**

Udaljenosti od početnog čvora pomoću Dijkstre su :

|  |  |
| --- | --- |
| **Ime čvora** | **Konačne udaljenosti od početnog čvora:** |
| **1** | **2** |
| **2** | **2** |
| **3** | **5** |
| **4** | **4** |
| **5** | **0** |
| **6** | **5** |

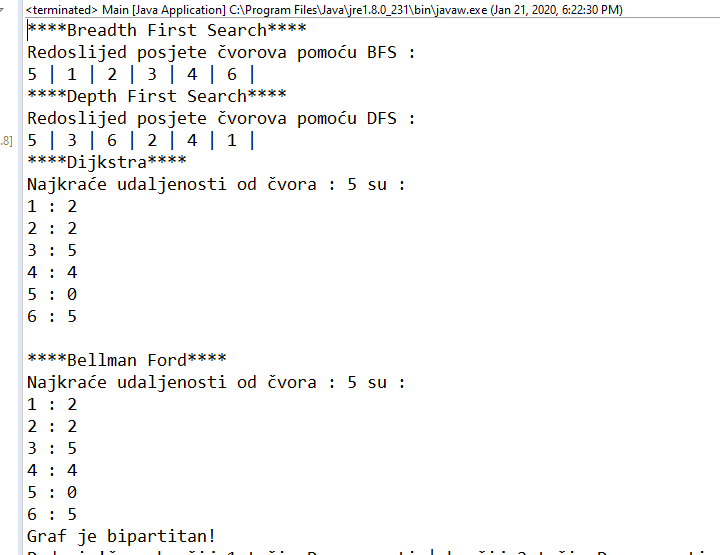
**Ovaj graf je bipartitan.**Naime, moguće ga je rastaviti na dva skupa :

**A {1, 2, 3} i**

**B {4, 5, 6}**

**Tako da zadovoljava uslov bipartitnosti.**

Na idućoj stranici dana je slika koja prikazuje izlaze koje pravi moj program nakon ispitivanja za gore prikazani graf :



Slika 3.4 : rezultati algoritama za drugi graf

-Kao što se vidi iz priloženog, moja klasa vraća tačne vrijednosti za svaki od dva gore analizirana grafa…

# 4.IMPLEMENTACIJA GLAVNOG PROGRAMA

Idući kod prikazuje glavni driver program u kojem se prave objekti napravljene klase, i unose dva grafa sa slike. Ovdje se jasno vidi da su korišteni template-i, jer je prvi graf Integera, dok je drugi graf graf Stringova:

|  |
| --- |
| package mojeKlase;  import java.util.Scanner;  public class Main {  public static void main(String[] args) {  Scanner ulaz = new Scanner(System.in);  Graf<Integer> testniGraf1 = unesiGrafInteger(ulaz);  Graf<String> testniGraf2 = unesiGrafString(ulaz);  Integer početni1 = 0;  String početni2 = "";  System.out.println("Koji je početni čvor grafa sa integerima?");  početni1 = ulaz.nextInt();  ulaz.nextLine();  System.out.println("Koji je početni čvor grafa sa stringovima?");  početni2 = ulaz.nextLine();  testniGraf1.Bfs(početni1);  testniGraf1.Dfs(početni1);  testniGraf1.Dijkstra(početni1);  testniGraf1.BellmanFord(početni1);  if (testniGraf1.jeLiBipartitan(početni1) == true)  System.out.println("Graf je bipartitan!");  else  System.out.println("Graf nije bipartitan!");  testniGraf2.Bfs(početni2);  testniGraf2.Dfs(početni2);  testniGraf2.Dijkstra(početni2);  testniGraf2.BellmanFord(početni2);  if (testniGraf2.jeLiBipartitan(početni2) == true)  System.out.println("Graf je bipartitan!");  else  System.out.println("Graf nije bipartitan!");  ulaz.close();  }  //za čvorove čija su imena stringovi  public static Graf<String> unesiGrafString(Scanner ulaz) {  Graf<String> mojGraf = new Graf<String>();  String početniČvor = "";  String krajnjiČvor = "";  Integer težinaIvice = 0, brojIvica = 0;  Integer usmjerenGraf = 0;  System.out.println("Unesi 1 za usmjereni graf, ili drugi broj za neusmjereni graf:");  usmjerenGraf = ulaz.nextInt();  ulaz.nextLine();  if (usmjerenGraf == 1)  mojGraf.setUsmjereniGraf(true);  else  mojGraf.setUsmjereniGraf(false);  System.out.println("Unesi koliko ivica želiš u tvome grafu?");  brojIvica = ulaz.nextInt();  ulaz.nextLine();  System.out.println("Ivice su oblika (početniČvor, krajnjiČvor, težinaIvice).");  for (int i = 0; i < brojIvica; i++) {  System.out.println("Unesi " + (i + 1) + ". ivicu po redu:");  početniČvor = ulaz.nextLine();  krajnjiČvor = ulaz.nextLine();  težinaIvice = ulaz.nextInt();  ulaz.nextLine();  mojGraf.dodajIvicu(početniČvor, krajnjiČvor, težinaIvice);  }  return mojGraf;  }  //za čvorove čija su imena integeri  public static Graf<Integer> unesiGrafInteger(Scanner ulaz) {  Graf<Integer> mojGraf = new Graf<Integer>();  Integer početniČvor = 0;  Integer krajnjiČvor = 0;  Integer težinaIvice = 0, brojIvica = 0;  Integer usmjerenGraf = 0;  System.out.println("Unesi 1 za usmjereni graf,a drugi broj za neusmjereni graf:");  usmjerenGraf = ulaz.nextInt();  ulaz.nextLine();  if (usmjerenGraf == 1)  mojGraf.setUsmjereniGraf(true);  else  mojGraf.setUsmjereniGraf(false);  System.out.println("Unesi koliko ivica želiš u tvome grafu?");  brojIvica = ulaz.nextInt();  ulaz.nextLine();  System.out.println("Ivice su oblika (početniČvor, krajnjiČvor, težinaIvice).");  for (int i = 0; i < brojIvica; i++) {  System.out.println("Unesi " + (i + 1) + ". ivicu po redu:");  početniČvor = ulaz.nextInt();  krajnjiČvor = ulaz.nextInt();  težinaIvice = ulaz.nextInt();  ulaz.nextLine();  mojGraf.dodajIvicu(početniČvor, krajnjiČvor, težinaIvice);  }  return mojGraf;  }  } |

# 5.IMPLEMENTACIJA KLASE „GRAF“

-Klasa „Graf“ zajedno sa njenim funkcijama, podacima i algoritmima je prikazana ispod. Treba primjetiti da je implementacija grafa izvršena sa strukturom podataka zvanom HashMap, koja ima ključeve i vrijednosti. Ključevi su čvorovi, dok je vrijednost za svaki od čvorova niz njihovih komšija također predstavljen HashMap-om. U ovoj HashMap-i ključevi su komšije, dok su vrijednosti težine ivica između ta dva čvora. Treba dodati i to da algoritmi kao npr Bellman Ford imaju odgovarajuću povratnu vrijednost, ali je ona u demonstraciji ignorirana.

|  |
| --- |
| package mojeKlase;  import java.util.HashMap;  import java.util.List;  import java.util.Queue;  import java.util.LinkedList;  import java.util.Stack;  class Graf<Tip> {  private HashMap<Tip, HashMap<Tip, Integer>> mapa = new HashMap<Tip, HashMap<Tip, Integer>>();  private boolean obaSmjeraDefault = true;  //razne vrste konstruktora//:  //prazan konstruktor.  Graf() {  }  //konstruiranje grafa samo sa čvorovima, bez ivica.  Graf(List<Tip> čvorovi) {  for (int i = 0;i < čvorovi.size(); i++)  dodajČvor(čvorovi.get(i));  }  //konstruktor kopije:  Graf(Graf<Tip> \_graf) {  obaSmjeraDefault = \_graf.jeLiUsmjereniGraf();  for (Tip trenutniČvor : \_graf.mapa.keySet()) {  this.dodajČvor(trenutniČvor);  for (Tip komšija : \_graf.mapa.get(trenutniČvor).keySet())  dodajIvicu(trenutniČvor, komšija, \_graf.mapa.get(trenutniČvor).get(komšija), false);  }  }  //funkcije za rad sa podacima//:  public void setUsmjereniGraf(boolean vrijednost) {  obaSmjeraDefault = !vrijednost;  }  public boolean jeLiUsmjereniGraf() {  return obaSmjeraDefault;  }  public void dodajČvor(Tip \_čvor) {  if (!imaČvor(\_čvor))  mapa.put(\_čvor, new HashMap<Tip, Integer>());  }  public void dodajIvicu(Tip odakle, Tip dokle, Integer težinaIvice, boolean obaSmjera) {  if (!imaČvor(odakle))  dodajČvor(odakle);  if (!imaČvor(dokle))  dodajČvor(dokle);  if (!imaIvicu(odakle, dokle))  mapa.get(odakle).put(dokle, težinaIvice);  if (obaSmjera && !imaIvicu(dokle, odakle))  mapa.get(dokle).put(odakle, težinaIvice);  }  public void dodajIvicu(Tip odakle, Tip dokle) {  dodajIvicu(odakle, dokle, 0, obaSmjeraDefault);  }  public Integer getTežina(Tip početak, Tip kraj) {  return mapa.get(početak).get(kraj);  }  public void promijeniTežinuIvice(Tip početak, Tip kraj, Integer novaVrijednostTežine) {  if (!this.imaIvicu(početak, kraj)) {  System.out.println("Ivica ne postoji, pravim novu...");  dodajIvicu(početak, kraj, novaVrijednostTežine, obaSmjeraDefault);  return;  }  mapa.get(početak).replace(kraj, novaVrijednostTežine);  if (obaSmjeraDefault)  mapa.get(kraj).replace(početak, novaVrijednostTežine);  }  public void dodajIvicu(Tip odakle, Tip dokle, Integer težinaIvice) {  //za defaultnu vrijednost varijable obaSmjera:  dodajIvicu(odakle, dokle, težinaIvice, obaSmjeraDefault);  }  public int getBrojČvorova() {  return mapa.keySet().size();  }  public int getBrojIvica(boolean obaSmjera) {  int broj = 0;  for (Tip mojTip : mapa.keySet())  broj += mapa.get(mojTip).size();  if (obaSmjera)  broj /= 2;  return broj;  }  public int getBrojIvica() {  //sa podrazumijevanom obaSmjera u true:  return getBrojIvica(obaSmjeraDefault);  }  public void SadržajGrafa() { //ispiši sve što graf sadrži  System.out.println("Podaci:'čvor:komšija1,težinaPovezanosti | komšija2,težinaPovezanosti ...");  for (Tip čvor : mapa.keySet()) {  System.out.print(čvor + " :");  for (Tip komšija : mapa.get(čvor).keySet())  System.out.print(komšija + ", " + mapa.get(čvor).get(komšija) + "|");  System.out.println();  }  }  public boolean imaČvor(Tip čvor) {  if (mapa.containsKey(čvor))  return true;  return false;  }  public boolean imaIvicu(Tip početak, Tip kraj) {  if (mapa.get(početak).containsKey(kraj))  return true;  return false;  }  public Queue<Tip> Bfs(Tip početni) { // ispiši redoslijed posjete čvorova pomoću Dfs algoritma  System.out.println("\*\*\*\*Breadth First Search\*\*\*\*");  Tip trenutniČvor = početni; //neka vrijednost za početni.  Queue<Tip> queue = new LinkedList<Tip>();  HashMap<Tip, Boolean> posjećen = new HashMap<Tip, Boolean>();  Queue<Tip> rezultat = new LinkedList<Tip>();  for (Tip mojTip : mapa.keySet())  posjećen.put(mojTip, false);  queue.add(početni);  posjećen.replace(početni, true);  System.out.println("Redoslijed posjete čvorova pomoću BFS :");  while (!queue.isEmpty() ) {  trenutniČvor = queue.remove();  rezultat.add(trenutniČvor);  System.out.print(trenutniČvor + " | ");  for (Tip komšija : mapa.get(trenutniČvor).keySet()) {  if (!posjećen.get(komšija)) {  queue.add(komšija);  posjećen.replace(komšija, true);  }  }  }  System.out.println();  return rezultat;  }  public Queue<Tip> Dfs(Tip početni) { // ispiši redoslijed posjete čvorova pomoću Bfs algoritma  System.out.println("\*\*\*\*Depth First Search\*\*\*\*");  Tip trenutniČvor = početni;  Stack<Tip> stack = new Stack<Tip>();  Queue<Tip> rezultat = new LinkedList<Tip>();  HashMap<Tip, Boolean> posjećen = new HashMap<Tip, Boolean>();  for (Tip mojTip : mapa.keySet())  posjećen.put(mojTip, false);  stack.push(početni);  posjećen.replace(početni, true);  System.out.println("Redoslijed posjete čvorova pomoću DFS :");  while (!stack.isEmpty()) {  trenutniČvor = stack.pop();  rezultat.add(trenutniČvor);  System.out.print(trenutniČvor + " | ");  for (Tip komšija : mapa.get(trenutniČvor).keySet()) {  if (!posjećen.get(komšija)) {  stack.push(komšija);  posjećen.replace(komšija, true);  }  }  }  System.out.println();  return rezultat;  }  private Tip minimalnaUdaljenost(HashMap<Tip, Integer> r, HashMap<Tip, Boolean> a) {  Tip najmanji = null;  Integer najmanjaVrijednost = Integer.MAX\_VALUE;  for (Tip i : r.keySet()) {  if (r.get(i) <= najmanjaVrijednost && !a.get(i)) {  najmanjaVrijednost = r.get(i);  najmanji = i;  }  }  return najmanji;  }  public HashMap<Tip, Integer> Dijkstra(Tip početak) {  System.out.println("\*\*\*\*Dijkstra\*\*\*\*");  //varijable potrebne za rad algoritma:  Tip početni = početak;  HashMap<Tip, Boolean> posjećen = new HashMap<Tip, Boolean>();  HashMap<Tip, Integer> rezultat = new HashMap<Tip, Integer>();  //inicijalizacija varijabli potrebnih za rad.  for (Tip trenutniČvor : mapa.keySet()) {  posjećen.put(trenutniČvor, false);  rezultat.put(trenutniČvor, Integer.MAX\_VALUE);  }  rezultat.replace(početak, 0);  while (posjećen.containsValue(false)) {  početak = minimalnaUdaljenost(rezultat, posjećen);  posjećen.replace(početak, true);  for (Tip komšija : mapa.get(početak).keySet())  if (getTežina(početak, komšija) + rezultat.get(početak) <= rezultat.get(komšija))  rezultat.replace(komšija, getTežina(početak, komšija) + rezultat.get(početak));  }  System.out.println("Najkraće udaljenosti od čvora : " + početni + " su :");  for (Tip mojTip : mapa.keySet())  System.out.println(mojTip + " : " + rezultat.get(mojTip));  System.out.println();  return rezultat;  }  public HashMap<Tip, Integer> BellmanFord(Tip početni) {  System.out.println("\*\*\*\*Bellman Ford\*\*\*\*");  //ako želimo da algoritam radi i sa negativnim ivicama, onda graf mora biti usmjeren  //jer bi inace negativna dvostruka ivica predstavljala ciklus...  HashMap<Tip, Integer> rezultat = new HashMap<Tip, Integer>();  for (Tip trenutniČvor : mapa.keySet())  rezultat.put(trenutniČvor, Integer.MAX\_VALUE);  rezultat.replace(početni, 0);  for (int i = 0; i < this.getBrojČvorova() - 1; i++) {  for (Tip trenutniČvor : mapa.keySet()) {  if (rezultat.get(trenutniČvor) == Integer.MAX\_VALUE)  continue;  for (Tip komšija : mapa.get(trenutniČvor).keySet())  if (rezultat.get(komšija) > rezultat.get(trenutniČvor)  + this.getTežina(trenutniČvor, komšija))  rezultat.replace(komšija, rezultat.get(trenutniČvor)  + this.getTežina(trenutniČvor, komšija));  }  }  //provjera da li postoji negativni ciklus:  for (Tip trenutniČvor : mapa.keySet()) {  for (Tip komšija : mapa.get(trenutniČvor).keySet())  if (rezultat.get(komšija) >  rezultat.get(trenutniČvor) + this.getTežina(trenutniČvor, komšija)) {  //javljam negativni ciklus u grafu.  System.out.println("Vaš graf sadrži negativni ciklus.");  System.out.println("Prema tome, nemoguće naći udaljenosti...");  return null;  }  }  System.out.println("Najkraće udaljenosti od čvora : " + početni + " su :");  for (Tip mojTip : mapa.keySet())  System.out.println(mojTip + " : " + rezultat.get(mojTip));  return rezultat;  }  public boolean jeLiBipartitan(Tip početni) {  //ovaj algoritam radi samo za povezane grafove  //ali ipak ga je lagano modifikovati da proradi i sa  //nepovezanim grafovima...  Tip trenutniČvor = početni;  HashMap<Tip, Integer> boja = new HashMap<Tip, Integer>();  Queue<Tip> queue= new LinkedList<Tip>();  //postavi sve boje na neobojeno -1:  for (Tip a : mapa.keySet())  boja.put(a, -1);  queue.add(početni);  boja.replace(početni, 0);  while (!queue.isEmpty()) {  trenutniČvor = queue.remove();  for (Tip komšija : mapa.get(trenutniČvor).keySet()) {  if (boja.get(komšija) == -1) { //ako nije posjećen  boja.replace(komšija, 1 - boja.get(trenutniČvor));  queue.add(komšija);  } else if (boja.get(komšija) == boja.get(trenutniČvor)) {  return false;  }  }  }  return true;  }  } |

# 6.ZAKLJUČAK

Cilj ovog rada je bio implementacija klase i glavnog main programa za demonstraciju rada grafova, a usput smo izvršili i analizu implementiranih algoritama.

Ovo može biti jedan solidan uvod u grafove svakome ko se ozbiljno planira baviti istima, a može dobro doći kao podsjetnik i ispomoć na bilo kojoj vrsti tehničkog fakulteta, jer se na većini njih proučava teorija grafova. U tim situacijama nije od viška imati ovakvu klasu da bi se probali određeni primjeri bez puno truda. Veoma je jednostavna za upotrebu, i može se koristiti bez poznavanja oblasti grafova, bilo da ste početnik koji se tek njima počinje baviti, ili vam oni trebaju usput, da biste odradili neke druge stvari.

Ciljao sam da napravim nešto korisno, odnosno što bi mi nekada moglo zatrebati, što sam, nadam se i uspio. Ne samo da ovo može biti korisno meni, nego bilo kome ko bude htio proučavati ovu oblast, ili bude htio proširiti i obogatiti način razmišljanja kroz nekoliko (relativno) jednostavnih primjera matematskih dokaza implementiranih algoritama……

# 7.BIBLIOGRAFIJA

1)Bellman Ford algoritam, GeeksForGeeks,

<https://www.geeksforgeeks.org/bellman-ford-algorithm-dp-23/>

2)Dijkstra algoritam, GeeksForGeeks,

<https://www.geeksforgeeks.org/dijkstras-shortest-path-algorithm-greedy-algo-7/>

3)ETF Courseware,Algoritmi i Strukture podataka,predavanje 13 i 14,

<https://c2.etf.unsa.ba/course/view.php?id=50>

4)Primjer grafa za provjeru Bellman Ford-a,

<https://www.programiz.com/dsa/bellman-ford-algorithm>