Odsjek za matematiku
Prirodno-matematički fakultet
Univerzitet u Sarajevu

NAJKRAĆI PUT

Seminarski rad iz predmeta „Teorija grafova“

Student:
Nadin Zajimović (4818/M)

Mentor:
Damir Hasić

Sarajevo, juni 2013.

1. Abstrakt

Nalaženje najkraćeg puta je problem koji se u praksi javlja često. To je jedan od familije srodnih problema koji se rješavaju algoritmima na grafu. Veliki broj stvarnih problema sa kojim se surećemo u praksi se može modelirati pomoću grafa. Teorija grafova je oblast koja proučava svojstva matematičkih struktura nazvanih „grafovi“ i koja, između ostalog, nudi mnoge algoritme i načine rješavanja grafovskih problema. Zbog velika primjene, to je jedna od disciplina kojom se matematičari ponajviše bave. Mnogi od grafovskih problema su „NP-teški“, tj. ne postoji efikasan (polinomijalan) algoritam koji daje tačno rješenje tih problema, te se zbog toga razvijaju razne heuristike koje daju približno optimalna rješenja u što kraćem vremenu. Postoje problemi koji su naizgled jako srodni NP-teškim problemima, a za koje postoje efikasni algoritmi. Tako za nalaženje najkraćeg puta između proizvoljna dva čvora u grafu postoji nekoliko algoritama, dok ukoliko se na taj problem dodaju neki, naizgled ne tako jaki, uslovi dobijamo NP-težak problem. U ovom seminarskom radu ćemo se osvrnuti na rješavanje problema najkraćeg puta uz neke dodatne uslove.

2. Sadržaj

3. Uvod

Kao što je već rečeno, u ovom seminarskom radu ćemo dati algoritam za rješavanje specijalizovanog problema najkraćeg puta u grafu. Neformalno gledano, graf je objekat koji se sastoji određenog broja manjih objekata – čvorova, koji mogu biti povezani međusobnim vezama. Formalno, graf G je uređeni par (V,E), pri čemu se skup V naziva „skup čvorova“, a njegovi elementi „čvorovi“. Skup E je „skup grana“ i to je skup dvoelementnih skupova {u,v} takvih da su u i v elementi skupa V. Opisani graf se naziva neusmjereni, jer poredak čvorova za granu nije bitan, tj. grana nema smjer, grane {u,v} i {v,u} su ekvivalentne. Ukoliko su elementi skupa E uređeni parovi (u,v) takvi da su u i v elementi skupa V, onda pričamo o usmjerenom grafu i usmjerenim granama, pri čemu grana počinje u u, a završava u v. Iako se ovakva formalna definicija, na prvi pogled čini teško primjenjiva u praksi, to nije slučaj. Mnoge stvarne probleme i strukture možemo modelirati grafovima. Primjerice radi, čvorovi mogu predstavljati gradove, dok grane mogu predstavljati ceste između njih. Ukoliko svakoj grani dodijelimo neki broj, onda dobijamo težinski graf. U našem primjeru sa gradovima i cestama, taj broj može predstavljati dužinu ceste između dva grada. U teoriji grafova, taj broj se naziva „težina grane“. U našem primjeru možemo pretpostaviti da je težina grane novčana vrijednost koju ćemo potrošiti prelazeći cestu (recimo potrošnja goriva, cestarine i sl.). Tada bi nalaženje najkraćeg puta između dva čvora u grafu predstavljalo najoptimalniji put između dva grada, tj. onaj put na kojem ćemo potrošiti najmanje novca. Formalno gledano, put u grafu je niz čvorova među kojim se nijedan ne ponavlja, a između svaka dva susjedna postoji grana. Naš konkretan problem je naći najkraći put između dva čvora, ali tako da on prolazi kroz sve čvorove iz nekog zadanog podskupa A, a ne prolazi nijednim čvorom iz podskupa B. Ukoliko to nije moguće potrebno je naći najkraći put koji maksimizira broj „obaveznih“ čvorova koje smo posjetili i minimizira broj „zabranjenih“ čvorova koje smo posjetili.

4. Opis problema i model

4.1. Kratki opis

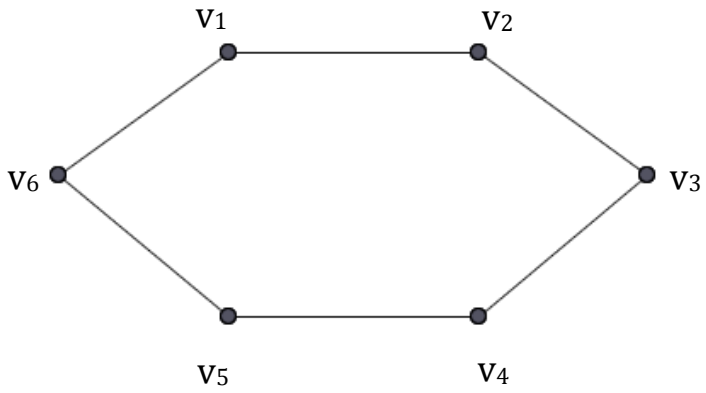
Dakle, problem najkraćeg puta sa restrikcijama pronalazi najkraći put između dva čvora u grafu takav da posjećuje sve čvorove iz zadanog podskupa čvorova i da ne posjećuje nijedan čvor iz drugog zadanog podskupa. Ukoliko takav put ne postoji, a ne mora postojati, odgovor na ovaj problem je put najkraći put takav da posjećuje što više čvorova iz zadanog podskupa „obaveznih“ čvorova, a što manje iz zadanog podskupa „zabranjenih“ čvorova.

4.2. Formalni opis

Naravno, kao i svaki problem/zadatak, tako i ovaj podrazumijeva da tačno definišemo šta su ulazni podaci, a šta izlazni podaci, te kako su oni povezani. Ulazni podaci za ovaj problem, u svojoj najopštijoj formi su: graf G=(E,V), skupovi A⊂V, B⊂V, te dva čvora u∈V, v∈V. Graf G je isti graf iz opisa problema, skup A je skup obaveznih čvorova, skup B je skup zabranjenih čvorova, dok put treba da ide iz čvora u u čvor v. Rezultat/odgovor na ovaj problem je put: $p=(v_1,v_2,...,v_k)$ takav da $v_i \in V$ (i=1,..,k) i da taj put optimizira funkciju:

$$\sum_{i=1}^{k-1} w(v_i, v_{i+1})$$

po svim putevima p, takvim da je $A \setminus V(p) = \varnothing$ i $B \cap V(p) = \varnothing$, pri čemu nam je $w(v_1,v_2)$ težina grane koja spaja čvorove v_1 i v_2, V(p) je skup čvorova na putu p. Naravno, čvorovi v_i i v_{i+1} moraju biti povezani za sve i=1,...,k-1, te mora vrijediti $v_1=u$, $v_k=v$. Naravno, ovo je postavka problema za slučaj da želimo najoptimalnije moguće rješenje, tj. da posjetimo sve obavezne i nijedan zabranjeni čvor. Međutim, može se pokazati (kontraprimjerom) da postoji graf G i dva čvora u,v, te dva podskupa čvorova, takvi da ne postoji put koji obilazi sve obavezne i zaobilazi sve zabranjene čvorove. Neka je graf G:



i neka je u=v_1, v=v_3, A={v_2,v_4,v_5}, B={v_6}. Pošto je u pitanju ciklus, postoje samo dva puta između u i v, to su p_1 = (u,v_2,w) i p_2 = (u,v_6,v_5,v_4,w). Vidimo da put p_1 minimizira broj posjećenih zabranjenih čvorova (nema ih, dok ih na putu p_2 ima), dok put p_2 maksimizira broj posjećenih obaveznih čvorova (na putu p_2 su dva obavezna čvora, na putu p_1 je jedan). U ovom slučaju težine grana su nebitne za kontraprimjer, pa su zato i izostavljene sa slike. Dakle, nakon ove primjedbe, dolazimo u situaciju da se moramo odlučiti za jednu od tri moguće verzije ovog problema. Prva verzija podrazumijeva da se kao rješenje problema javi put koji pod svaku cijenu maksimizira broj posjećenih obaveznih čvorova, a da se među takvim bira onaj sa najmanje posjećenih zabranjenih čvorova. Druga verzija je analogna, rezultat bi bio put sa najmanjim brojem posjećenih zabranjenih čvorova, a među takvim se bira onaj koji maksimizira broj posjećenih obaveznih čvorova. Treća verzija, ona za koju smo se odlučili posmatrati u ovom seminarskom radu, je ta da se kao rezultat vrati put koji maksimizira razliku broja posjećenih obaveznih čvorova i broja posjećenih zabranjenih čvorova. Principijelno, ova verzija minimizira napravljenu „štetu“, jer ona minimizira broj čvorova u grafu koji narušavaju postavljene uslove.

4.3. Model

Ovaj problem je u principu srodan nalaženju najkraćeg puta u grafu. Za nalaženje najkraćeg puta bez dodatnih uslova postoje razni algoritmi, od kojih su najpoznatiji Dijsktrin i Bellman-fordov algoritam.

Dijkstrin algoritam je poseban slučaj „generalnog“ algoritma za konstruisanje pokrivajućeg stabla. Dijsktrinim algoritmom nalazimo pokrivajuće stablo najkraćih

puteva u odnosu na početni čvor. Naime, poznato je da svaki povezan graf (sve komponente nepovezanog grafa) imaju pokrivajuće stablo, stablo koje sadrži sve čvorove grafa. Također, poznato je da je u stablu put između dva čvora jedinstven. U Dijkstrinom stablu, put od početnog čvora do svih ostalih čvorova je jedinstven i ujedno najkraći u grafu. Kao što smo rekli, ovo je poseban slučaj algoritma za konstruisanje pokrivajućeg stabla. Taj algoritam kreće od početnog čvora koji je ujedno jedini čvor stabla na početku algoritma. U svakoj iteraciji se dodaje proizvoljna grana koja povezuje neki čvor koji je u stabli i neki koji nije. Algoritam staje kad stablo koje dobijemo postane pokrivajuće. Dijsktrin algoritam u svakom koraku ne bira proizvolju granu nego onu koja zadovoljava određen uslov. U tu svrhu se uvodi funkcija potencijala, svakom čvoru se dodijeli jedan broj, nazvan potencijal – d(v). Sada, u svakom koraku se bira grana koja minimizira potencijal čvorova koji nisu u stablu. Potencijal čvora koji nije u stablu se računa samo za čvorove koji su incidentni sa nekom granom e, čiji je drugi čvor, v, u stablu. Taj potencijal je jednak zbru potencijala čvora v i težini grane e. Potencijal početnog čvora se po definiciji uzima da je jednak nuli. Ovo je algoritam iz porodice pohlepnih algoritama. To su algoritmi koji u svakoj iteraciji iz određenog skupa objekata biraju onaj koji ima najveću vrijednost neke funkcije (ili najmanju, kao u ovom slučaju). Dodatna pogodnost ovog algoritma je, da se uz malu modifikaciju može i rekonstruisati najkraći put. Naime, svaki put kada dodamo neku granu e={u,v} u stablo, pri čemu se u nalazio u stablu, v nije, dovoljno je za čvor v zapamtiti da je njegov potencijal izračunat u odnosu na čvor u. Dakle, definišemo funkciju prethodnik(v), koja svakom čvoru dodijelu čvor u odnosu na koji je izračunat njegov potencijal. Sada, krenemo od čvoro do kojeg želimo najkraći put, i „odmotajemo“ put, koristeći vrijednosti funkcije „prethodnik“ i tako dobijamo, osim dužine najkraćeg puta, i niz čvorova koji ga grade. Dijsktrin algoritam, modifikovan za implementaciju u programskim jezicima, ima sljedeći pseudokod:

```
Dijkstra (graf G, cvor pocetni_cvor)
{
      za svaki čvor v u G definiši:
            d(v) := ∞
            prethodnik(v) := nedefinisan
      d(pocetni_cvor) := 0
      Q := skup svih čvorova u G
      sve dok Q nije prazan:
            u := čvor u Q sa najmanjom vrijednosti d
            izbaci u iz Q
            ako je d(u) = ∞
                  završi algoritam
            za svaki čvor v koji je susjed od u i nije u Q:
                  privremeni_potencijal := d(u) + w(u,v)
                  ako je privremeni_potencijal < d(v)
                        d(v) = privremeni_potencijal
                        prethodnik(v) = u
}
```

Po završetku ovog algoritma, vrijednost funkcije d za svaki čvor će biti ujedno i dužina najkraćeg puta koji ćemo moći rekonstruisati pomoću funkcije „prethodnik“.

Naš problem je vrlo srodan problemu najkraćeg puta, pa će nam Dijsktrin algoritam poslužiti kao osnova za konstruisanje algoritma za rješavanje našeg problema.

5. Algoritam

5.1. NP-kompletnost problema

Prije svega, primjetimo jednu stvar. Specijalan slučaj za naš problem je problem nalaženja hamiltonovog puta u grafu. Naime, za proizvoljan graf G se može konstruisati instanca našeg problema čijim rješenjem dobijamo rješenje problema hamiltonovog puta za graf G. Hamiltonov put u grafu G je put koji posjećuje svaki čvor tačno jednom. Podsjetimo se još jednom definicije našeg problema: za graf G=(V,E), čvorove u,v, i skupove A⊂V, B⊂V potrebno je naći put nakraće dužine između u i v, takav da sadrži sve čvorove iz A, a nijedan iz B, ili najbliže optimalno rješenje, ukoliko to nije moguće. Ako za prozivoljan graf G posmatramo naš problem za sve moguće parove čvorova u,v iz G, te ako za svaki taj par definišemo A=V(G)\{u,v}, B=∅, onda bi rješenje za jedan od tih problema ujedno bio hamiltonov put. Naime, hamiltonov put obilazi sve čvorove, a mi postavljamo uslov da mora postojati put između dva čvora u,v koji sadrži sve ostale, što je ekvivalentno. S druge strane, može se pokazati da bi iz polinomijalnog algoritma za rješenje našeg problema slijedio i polinomijalni algoritam za rješenje problema najdužeg puta u grafu G. Naime, za proizvoljan graf G, konstruišimo graf G', takav da vrijedi V(G) ⊂ V(G'), za čvorove u,v koji se nalaze i u G i u G' i spojeni su granom u G vrijedi da je težina grane u G najkraći put od u do v u G'. Definišimo sada A=V(G), B=∅. Pozivajući naš algoritam za svaki par čvorova (u,v) iz G' i birajući najkraći put, dobili bi ujedno i najduži put u grafu G. A poznato je da je problem najdužeg puta NP-težak.

5.2 Opis algoritma

Dakle, najbolje što možemo je razviti heuristički algoritam koji daje približno dobra rješenja. U tu svrhu ćemo modifikovati Dijsktrin algoritam. U svakoj iteraciji Dijsktrinog algoritma birali smo čvor iz skupa neposjećenih čvorova koji ima najmanju udaljenost. Dakle, poredak čvorova smo definisali po kriteriju udaljenost, tj. čvorovima smo pridužili jedan broj, skalar, koji predstavlja udaljenost. To je rezultovalo da na kraju put bude najkraći. U našem slučaju se traži da put ima što više obaveznih čvorova i što manje zabranjenih, te da od takvih bude najkraći. U tu svrhu ćemo svakom čvoru pridružiti

uređenu četvorku, za razliku od Dijsktring algoritma u kojem smo čvoru pridružili samo jedan broj – udaljenost. Ta četvorka će se sastojati od sljedećih informacija: broj obaveznih čvorova koje smo do određenog trenutka posjetili na putu do čvora, broj zabranjenih čvorova koje smo posjetili na putu do datog čvora, dužina do sada najkraćeg puta koji zadovoljava uslove, te prioritet čvora. Prve tri informacije su jasne, ostaje još da se definiše prioritet čvora. Prioritet nam po definiciji predstavlja jedan broj koji određuje da li je taj čvor obavezan, zabranjen ili nijedno od toga. Obaveznim čvorovima pridružujemo prioritet 0, zabranjenim 2, ostalim 1. Dakle, što je prioritet manji, to nam je čvor „važniji“. Sada možemo definisati kriterij po kojem biramo čvorove, tj. funkciju poređenja dva čvora. Za jedan čvor, *u*, kažemo da je „*bolji*“ od *v*, ako je njegova razlika broja posjećenih obaveznih čvorova i broja posjećenih zabranjenih čvorova veća od iste razlike za čvor *v*. Dakle, biramo čvor koji će maksimizirati navedenu razliku. Takav kriterij će nam osigurati da na putu bude što manje čvorova koji narušavaju postavljene uvjete, tj. da ako je moguće povećati broj posjećenih obaveznih i ne povećati broj posjećenih zabranjenih čvorova, to ćemo i učiniti. Ukoliko je ta razlika ista za dva čvora, onda ih poredimo po dužini puta koji vodi do njih, što će garantovati da će put koji zadovoljava uslove biti što kraći među takvim. Dakle, biramo onaj čvor do kojeg je put kraći. Sada već možemo rezimirati naš algoritam i pretvoriti ga u pseudokod, što će biti učinjeno u narednom pargrafu.

5.3. Pseudokod

Modifikujući Dijsktrin algoritam, mijenjanjem funkcije d, sa funkcijom koja svakom čvoru pridružuje uređenu četvorku opisanu u prethodnom paragrafu, možemo napisati pseudokod. Uvedimo oznaku za tu funkciju:

$f(v) = (n_v, m_v, pr_v, d_v)$;

n – broj posjećenih obaveznih čvorova na putu do čvora v

m – broj posjećenih zabranjenih čvorova na putu do čvora v

pr – prioritet čvora

d – dužina puta do čvora v

Definišimo sada kriterij poređenja za čvorove, koristeći funkciju f. Dakle, vrijedi:

$f(v) < f(w) \Leftrightarrow (n_v - m_v < n_w - m_w) \ \vee \ (n_v - m_v = n_w - m_w \ \wedge d_v < d_w)$

Sada kad smo definisali funkciju, napišimo pseudokod algoritma koji za graf G=(V,E), čvorove *start* i *end*, te skupove A⊂V, B⊂V nalazi traženi najkraći put pod pomenutim uslovima:

```
Najkraci_put (graf G, cvor start, cvor end, skup A, skup B)
{
      za svaki čvor v u G definiši:
            f(v) := (0,0,1,∞)
            prethodnik(v) := nedefinisan
      za svaki čvor v u A definiši:
            f(v) := (0,0,0,∞)
      za svaki čvor v u B definiši:
            f(v) := (0,0,2,∞)
      f(start) := (0,0,0,0)
      Q := skup svih čvorova u G
      sve dok Q nije prazan:
            u := čvor u Q sa najmanjom vrijednosti f
            izbaci u iz Q
            ako je d_u = ∞
                  ne postoji put između start i end, kraj algoritma
            ako je u = end
                  Ispisi_rezultat(), kraj algoritma
            za svaki čvor v koji je susjed od u i nije u Q:
                  indikator_obavezan := 1 ako v ∈ A, 0 inače
                  indikator_zabranjen := 1 ako v ∈ B, 0 inače
                  f_priv :=(n_v + indikator_obavezan,
                            m_v + indikator_obavezan,
                            pr_v, d_v + w(u,v))
                  ako je f_priv < f(v)
                        f(v) = f_priv
                        prethodnik(v) = u
}
```

Dakle, algoritam počinje dodjeljujući svim čvorovima početne vrijednosti za funkciju f, u početku svim čvorovima se dodijeli broj 0 za prve dvije komponente funkcije f, jer na početku ni do jednog čvora nismo konstruisali put, pa na tom putu ne mogu biti ni zabranjeni ni dozvoljeni čvorovi. Vrijednost udaljenosti je beskonačna za sve čvorove, osim za početni, kojem je ta vrijednost 0, dok se prioritet određuje na osnovu pripadnosti skupovima A ili B. Sa tako definisanim vrijednostima funkcije f, formira se skup Q, skup svih čvorova i u svakoj iteraciji se uzima čvor sa najmanjom vrijednosti funkcije f. Ukoliko je udaljenost do tog čvora beskonačna, to znači da ne možemo doći do tog čvora, pa algoritam staje sa negativnim odgovorm. U slučaju da naš krajnji čvor ima najmanju vrijednost funkcije f, algoritam staje, jer je u toj vrijednosti sadržan najkraći put. Ukoliko nijedno od tog dvoje nije slučaj, posmatraju se svi susjedi izabranog čvora koji nisu u Q i računa se nova vrijednost funkcije f, tj. računa se vrijednost funkcije f koja bi bila u slučaju da put produžimo baš preko tog susjeda. Ukoliko je novoizračunata vrijednost manja od trenutne, onda se dosadašnji optimum ažurira. Primjetimo da se nova vrijednost funkcije f računa tako što se posmatra pripadnost susjeda čvora v skupovima A i B, te zbir udaljenosti izabranog čvora i težine granje od njega do susjeda.

Osvrnut ćemo se još na funkciju „*Ispisi_rezultat()*“ koja je zadužena za ispis rezultata u slučaju da je identifikovan najkraći put iz opisa problema. Dužinu puta je trivijalno odrediti, to je vrijednost d_{end}. Sada je potrebno odrediti stvarni put. Kao što smo to spomenuli ranije, dovoljno je krenuti od zadnjeg cvora i pratiti vrijednosti funkcije „prethodnik“ sve dok ne dođemo do početnog čvora. Međutim to je put ispisan unazad. Iz tog razloga ćemo te čvorove, počevši od zadnjeg stavljati na stek, pa ćemo na kraju ispisati elemente redom sa steka (stek je LIFO struktura i zato je pogodan za ovu primjenu). Stoga funkcija „*Ispisi_rezultat*“ treba da izgleda ovako:

```
Ispisi_rezultat()
{
      ispiši vrijednost d_end
      S := prazan stek, u := end
      sve dok je definisan prethodnik(u)
            ubaci u na S; u = prethodnik(u)
      sve dok S nije prazan
            ispiši vrh steka S i izbaci ga sa steka
}
```

5.4. Implementacija

Postavlja se pitanje koje strukture podataka koristiti za efikasnu implementaciju algoritma. Naime, inicijalizacija svakom čvoru dodjeljuje vrijednost funkcije f, i taj dio algoritma mora biti linearan po broju čvorova. Međutim, efikasnost implementacije dosta zavisi od petlje algoritma, tj. od strukture podataka kojom modeliramo skup Q. Ta struktura treba da efikasno podrži ubacivanje elemenata, izbacivanje elemenata, traženje najmanjeg elementa, te ažuriranje vrijednosti, jer to su operacije koje radimo sa Q u toku algoritma. Ukoliko malo bolje razmotrimo strukture podataka i efikasnost operacija sa njima, doći ćemo do zaključka da je gomila (heap) najbolje rješenje. U zavisnosti od vrste gomile koja se koristi, vremenska složenost navedenih operacija može varirati, ali mi smo se odlučili za korištenje binarne gomile. Naime, binarna gomila je posebna vrsta binarnog stabla sa osobinom da su svi potomci nekog čvora manji/veći od od njega. U našem slučaju je pogodno koristiti tzv. „min-heap“, gomilu kod kojeg je u korijenu najmanji čvor i kod koje vrijedi da su svi potomci nekog čvora veći od njega. Konkretno, mi ćemo u gomili čuvati uređeni par (ključ, vrijednost), gdje će nam ključ biti čvor, a vrijednost će predstavljati vrijednost funkcije f za taj čvor. Gomila će biti uređena po vrijednostima funkcije f, tj. u korijenu će biti čvor sa najmanjom vrijednosti funkcije f. Tako organizovana gomila nam garantuje da se će operacije ubacivanja elementa, izbacivanje najmanjeg elementa (što je nama i dovoljno, jer uvijek izbacujemo iz Q čvor sa najmanjom vrijednosti funkcije f), te ažuriranje vrijednosti elementa obavljati u vremenu O(log n), dok se operacija dobavljanja najmanjeg elementa obavlja u vremenu O(1). Primjetimo da ovom metodom inicijalizacija gomile Q bi trajala O(nlog n) vremena (pri čemu je n broj čvorova u grafu). Međutim, prednost gomile je što može u linearnom vremenu rasporediti sve elemente nekog niza u gomila, što je kod nas slučaj, mi na početku imamo dostupne sve čvorove i samo ih je potrebno reorganizovati u gomilu, bez potrebe za sekvencijalnim ubacivanjem.

Kada pričamo o implementaciju u C++-u, prvo treba implementirati strukturu „Graf“, jer ona nije dio standarda jezika C++. U našoj implementaciji svaki čvor grafa je interno označen rednim brojem iz raspona [0,n), pri čemu je n broj čvorova. Također, dopustili smo da se čvorovi mogu označiti i drugom oznakom koja je generičkog tipa, korisnik bira kojeg tipa će biti oznaka. Recimo, praktično je uzeti za tip oznake niz znakova, tako svakom čvoru možemo dodijeliti ime, iako oni već imaju jedinstven redni broj. Unutar

same strukture (bolje rečeno klase) Graf se čuva mapiranje oznaka, tj. za svaku *vanjsku* oznaku se čuva redni broj odgovarajućeg čvora i obratno. Napomenimo da se za mapiranje vanjskih oznaka u redni broj koriste hash-tabele (implementirane strukturom podataka „unordered_map“ u standardnom C++ jeziku), čije je vrijeme pristupa elementima konstantno (u prosjeku). Dalje, grane i njihove težine se čuvaju u kvadratnoj nxn matrici, i element matrice (i,j) govori da li ima grana između čvorova i i j. Vrijednost 0 predstavlja nepostojanje grane, dok vrijednost različita od nule predstavlja težinu grane. Toliko o implementaciji klase „Graf“. Vrijednosti funkcije f za svaki čvor implementirane su klasom „vertex_info“ koja u sebi čuva informacije o četiri komponente funkcije f, kao i za koju su definisani odgovarajući operatori poređenja. Još nam ostaje da razjasnimo detalje oko implementacije gomile. Standard C++ jezika podržava rad sa gomilama, bilo da koristimo strukturu podataka nazvanu „priority_queue“, bilo da koristimo funkcije iz STL-a (standard template library) koje podržavaju rad sa gomilama, kao npr. „make_heap“, „push_heap“, „pop_heap“ i sl. Međutim, nijedan od dva navedena načina ne podržava operaciju ažuriranja vrijednosti unutar gomile. Zbog toga smo razvili našu, korisnički definisanu klasu „gomila“ koja modelira binarnu gomilu, a koja podržava sve standardne operacije sa gomilama u očekivanom vremenu. Podaci koji se čuvaju u gomili su uređeni parovi (instance strukture „pair“ iz standarda C++ koja modelira uređeni par) od kojih prva komponenta predstavlja redni broj čvora u grafu, dok druga komponenta predstavlja vrijednost funkcije f za dati čvor, tačnije rečeno instancu klase „vertex_info“ koja modelira vrijednost funkcije f. Sada smo razjasnili sve detalje implementacije, te kod algoritma možete naći u prilogu koji dolazi uz ovaj seminarski rad. Sada se možemo posvetiti testiranju našeg programa koji implementira opisani algoritam.

5.5. Analiza složenosti

Uvedimo oznake, neka nam n označava broj čvorova u grafu, m broj grana. Algoritam počinje inicijalizacijom svih čvorova te kreiranjem gomile od njih, što smo već konstatovali da se može izvršiti u vremenu O(n). Dalje, u najgorem slučaju algoritam će ispitati sve moguće čvorove, pa samim tim i njihove susjede. Dakle, za svaku granu će se izvršiti operacija ažuriranja (u najgorem slučaju), te taj dio se vrši u vremenu O(mlog n), uzimajući našu implementaciju Q preko gomile. I još ostaje da svaki čvor uklonimo sa

gomile (svaki u najgorem slučaju), što se izvršava u vremenu O (nlog n). Ukupno vrijeme izvršavanja je:

$$O(n) + O(n\ \log n) + O(m\ \log n) = O(n\ \log n + m\ \log n) = O((n+m)\log n)$$

Dakle, vremenska složenost našeg algoritma je:

$$O\big((n+m)\log n\big),$$

pri čemu je:

n – broj čvorova u grafu

m – broj grana u grafu.

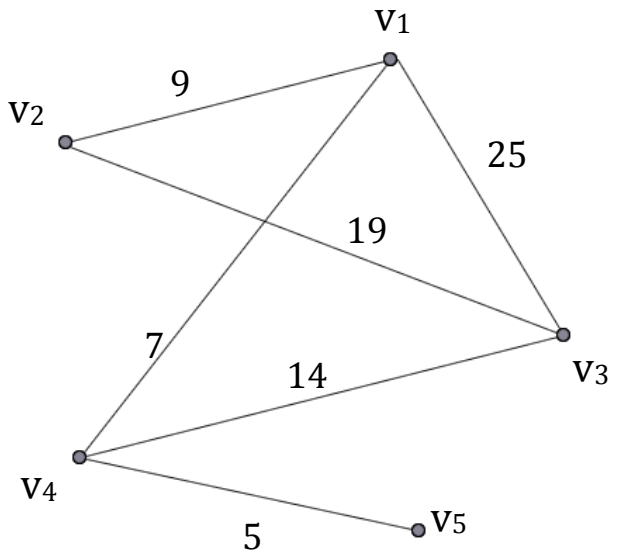
6. Testiranje

Kada pokrenemo program, on korisniku nudi jedan od tri načina korištenja, unos podataka o grafu i dodatnim uslovima sa tastature, unos istih podataka iz vanjske datoteke, te generisanje slučajnih podataka i testiranje algoritma na istim. Prva dva načina su u principu jako slična i svode se na ručno upisivanje.

6.1. Testiranje unosom podataka o grafu

Bilo da se radi o unošenju podataka sa tastature ili iz vanjske datoteke, pred korisnikom je isti posao, definisati broj čvorova, imena čvorova unutar grafa (u ova dva načina se koristi označavanje čvorova imenima, mada principijelno u samoj klasi „Graf“ oni su i dalje označeni rednim brojevima i postoji bijekcija između rednih brojeva i oznaka), broj grana te za svaku granu dva čvora koja su spojeni s njom i njenu težinu. I, naravno, korisnik još treba da definiše čvorove koje obavezno želi posjetiti, te one koje želi zaobići. Program je testiran na manjim grafovima. Svi ulazni i izlazni fajlovi za ovaj vid testiranja se nalaze u folderu „rezultati_testiranja/podaci iz datoteke“. Na slici se nalazi graf G, čiji se ulazni podaci nalaze u datoteci „input1.txt“, dok se izlazni podaci nalaze u dvije datoteke, „optimalan_put1.txt“, koja daje dužinu najkraćeg traženog puta i sam put, kao i „graf1.txt“ koja predstavlja internu strukturu klase „Graf“ za uneseni graf.

Primjer 1



Čvorovi koje želimo posjetiti:

v_3

Čvorovi koje ne želimo posjetiti:

v_4

Optimalan put:

$v_1 - v_3 - v_4 - v_5$

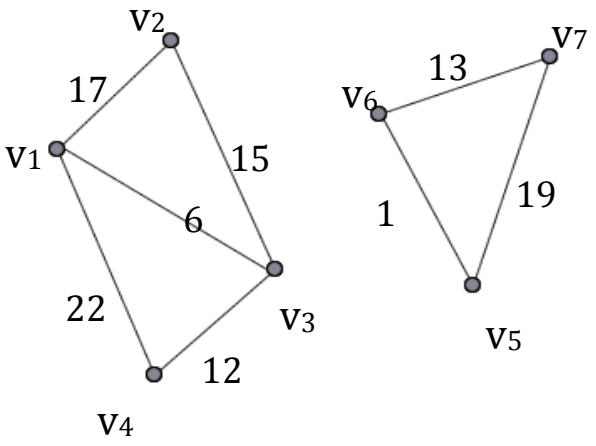
Dužina optimalnog puta:

44

Početni i završni čvor:

v_1, v_5

Primjer 2 (input2.txt, graf2.txt, optimalan_put2.txt)



Čvorovi koje želimo posjetiti:

v_1, v_3, v_7

Čvorovi koje ne želimo posjetiti:

v_4, v_5

Optimalan put:

Ne postoji

Dužina optimalnog puta:

-1 (indikator da ne postoji put)

Početni i završni čvor:

v_2, v_6

6.2. Testiranje slučajno generisanim podacima

Svi primjeri koje čovjek može ručno unijeti su relativno mali, prema broju čvorova i grana, te ne mogu testirati stvarnu efikasnost algoritma. Algoritam stavljamo na kušnju tek za mnogo veće vrijednosti broja čvorova i broja grana, vrijednosti koje se broje u hiljadama. Međutim kako je čovjeku (gotovo) nemoguće ručno generisati takav primjer, u tu svrhu smo razvili generator slučajnih primjera. Generator prihvaća kao parametre osvnovne informacije o veličini i struktruri grafa, te generiše slučajan graf po uzoru na proslijeđene podatke. Također, generator je zadužen i za generisanje slučajnog niza čvorova koje želim, odnosno ne želimo posjetiti. Generator slučajnih grafova koji smo razvili prihvata četiri ulazna parametra: broj čvorova, broj grana, najveći stepen čvora u grafu te informaciju o tome da li graf mora ili ne mora biti povezan. Kada korisnik odabere u programu opciju da se program testira na slučajno generisanom grafu, pristupa unošenju navedenih parametara. Prvo se unosi broj čvorova, pa zatim broj grana, najveći stepen te „povezanost“ – informacija o tome da li se mora forsirati povezanost grafa. Primjetimo da je unos broja grana ograničen brojem čvorova (uz pretpostavku da se radi o jednostvnom grafu, što, za praktične primjene, uglavnom i nije previše jako ograničenje). Naime, gornja granica za broj grana u grafu sa n čvorova je $n(n-1)/2$ (K_n kompletan graf). Unošenje tih podataka ide uz obaveznu provjeru ispravnosti, te program neće nastaviti sa radom dok korisnik ne unese ispravan podatak. Slično, i najveći stepen čvora u grafu je ograničen brojem grana i čvorova. Primjerice radi, najveći stepen sigurno mora biti manji i od broja

grana i od broja čvorova, te se mora voditi računa o tome da je zbir svih stepena u grafu jednak dvostrukom broju grana. Isto tako, graf sa do sada unešenim parametrima ne mora nužno biti „povezljiv“. Naime u koliko je broj grana manji od n-1 (n - broj čvorova), graf ne može biti povezan, obzirom da je graf povezan akko ima pokrivajuće stablo, a stablo ima n-1 granu. Kod unosa ovih parametara vodi se računa o zadovoljenju navedenih uslova. Nakon što korisnik unese parametre, oni se šalju generatoru (C++ funkciji) koja generiše slučajan graf. Postupak generisanja grafa traje u dva dijela. Prvi dio se izvršava samo ukoliko se forsira povezanost. Napomenimo da generator obavlja svoju funkciju uz pretpostavku da su uneseni ispravni parametri. Prvi dio generisanja grafa, ukoliko se forsira povezanost, sastoji se u konstruisanju povezujućeg stabla koje će garantovati povezanost. Dakle, ubacuje se n-1 grana (što će biti moguće jer smo se pri unosu ograničili samo na broj grana veći li jednak n-1, ukoliko forsiramo povezanost). U ranijem dijelu seminarskog rada izložen je algoritam za generisanje pokrivajućeg stabla. Naravno, treba se voditi računa o randomiziranju tog stabla. U tu svrhu, koristi se niz u kojem se čuvaju svi čvorovi, te za taj niz postoji „graničnik“, tj. indeks koji dijeli taj niz na lijevi i desni dio (obična promjenjiva u kojoj se čuva indeks graničnika). U svakom trenutuku čvorovi sa lijeve strane graničnika se nalaze u stablu, dok su sa desne strane čvorovi koji nisu u stablu. Algoritam kreće stavljanjem graničnika na početak, te stalnim pomjeranjem udesno, pri tome birajući granu koja će se dodati u stablo tako što se slučajno izabere jedan čvor iz lijevog dijela niza, jedan iz desnog. Randomiziranost je postignuta slučajnim ispremetanjem čvorova u nizu prije pokretanja algoritma. Slučajno ispremetanje elemenata niza je linearna operacija po broju elemenata i sastoji se u tome da petljom prođemo kroz sve elemente niza i zamijenimo ih sa nekim slučajnim, drugim elementom niza. Dakle, nakon prve faze imamo osiguranu povezanost grafa, ukoliko smo je morali forsirati. Druga faza se sastoji u dodavanju grana da bi se zadovoljio traženi broj grana (m). Prvo primjetimo da za graf sa n čvorova označenih sa {0,1,...,n-1} imamo n^2 mogućih grana. Zgodna stvar je što postoji bijekcija između grane (uređenog para (i,j), i,j=0,..,n-1) i skupa {0,1,2,...,n^2-1}. Naime, granu (i,j) možemo kodirati sa brojem in+j, i to kodiranje je jedinstveno. Primjetimo da na jedinstven način možemo i dekodirati granu u broj. Ukoliko je grana (i,j) kodirana sa brojem k, tada je:

$$i = \left\lfloor \frac{k}{n} \right\rfloor; \;\; j = k \; mod \; n$$

Dakle, sada kad kažemo „grana“ možemo pretpostaviti da se radi o cijelombroju iz opsega $[0,n^2)$. Sada možemo uzeti sve grane i staviti ih u niz od n^2 elemenata, te slučajno ih izmiješati. Onda kreće postupak dodavanja jedne po jedne grane u graf. Naravno, grana se dodaje samo ako zadovoljava uslove, tj. ako se ne narušava najveći stepen u grafu, te ako nije petlja i ako je već nema u grafu. Postupak se nastavlja sve dok ne dodamo broj grana koji nam je potreban (m). Primjetimo da će taj postupak uvijek stati, jer smo se kod unosa podataka ograničili na broj grana. Kada se postupak dodavanja grana završi imamo slučajan graf koji zadovoljava tražene uslove. Primjetimo još i da je vremenska složenost slučajnog generatora $O(n^2)$, pri čemu je n broj čvorova, jer od svih navedenih operacija, najviše vremena troši kreiranje niza od n^2 elemenata i njegovo slučajno ispremetanje, te dodavanje grane jednu po jednu.

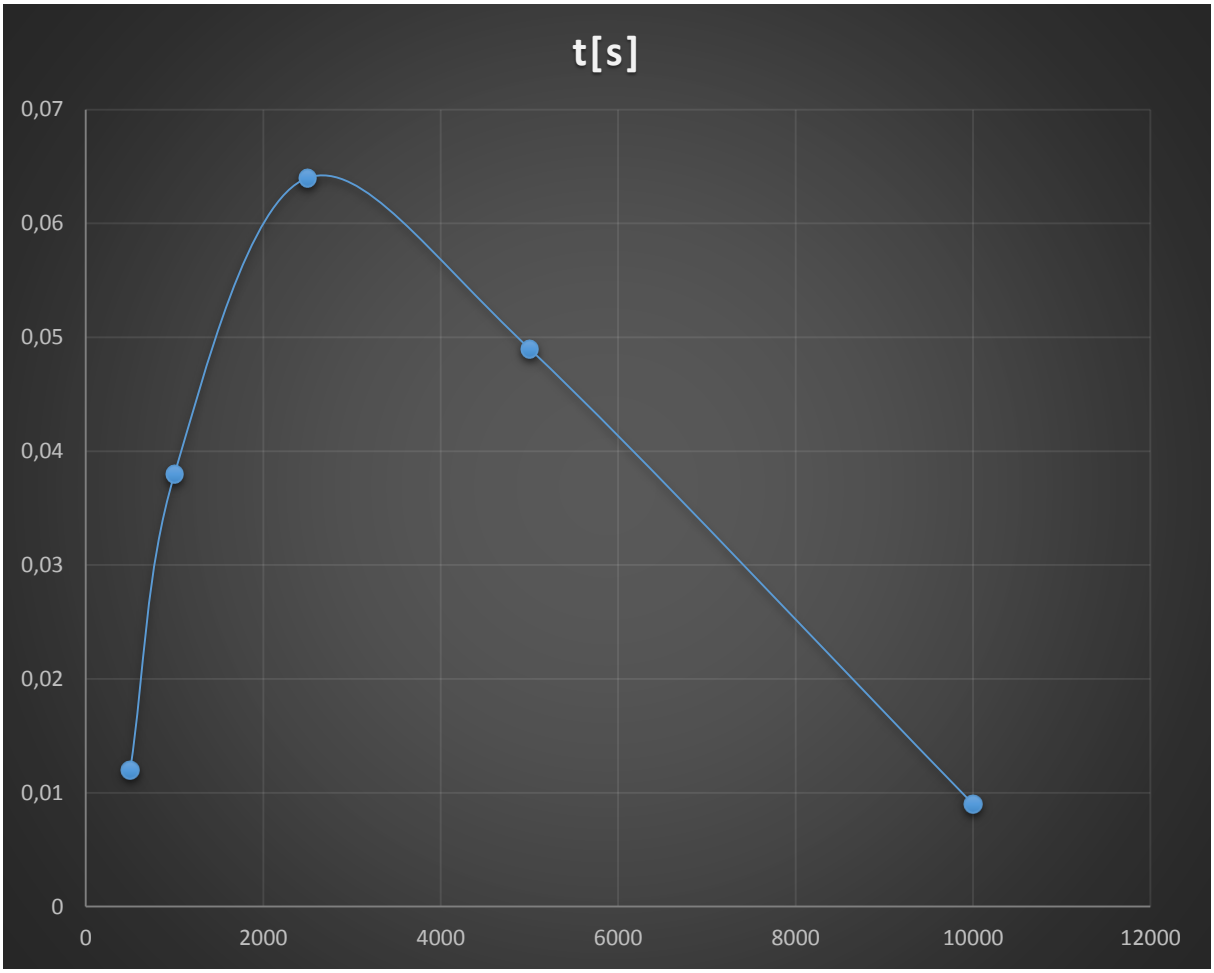
Nakon što je generator generisao graf, on generiše i slučajne nizove obaveznih i zabranjenih čvorova (čije veličine korisnik također unosi, uz ograničenja na veličine). Slučajni generator podrazumijeva da se traži nakraći put od čvora 0 do čvora n-1, što se može pretpostaviti bez umanjenja opštosti, obzirom da su svi procesi tokom kreiranja grafa slučajni, te nijedan čvor nije „dominantan“ u nekim osobinama. Nakon što su generisani čvorovi za posjetiti i zaobići, oni se šalju grafu kao parametar za rješavanje našeg problema. Sve rezultate testiranja program pohrani u četiri odvojene datoteke: „graf.txt“, „optimalan_put.txt“, „cvorovi_za_posjetitu.txt“, „vrijeme.txt“. U njih su redom pohranjene informacije o internoj strukturi klase Graf za slučajno generisani graf, optimalan put i njegova dužina, informacije koje čvorove je trebalo zaobići, a koje posjetiti, te vrijeme izvršavanja algoritma, generatora i podaci o veličini testnog primjera.

6.2.1. Testiranje prema fiksnom broju čvorova

U ovoj metodi testiranja odabrat ćemo neku fiksnu vrijednost za broj čvorova, te mijenjati broj grana i vidjeti kako se prema tome ponaša vrijeme izvršavanja algoritma. To ćemo uraditi za nekoliko vrijednosti broja čvorova. Rezultate testiranja možete pogledati u tabelama ispod, kao i grafikonima, dok su svi izlazni i ulazni fajlovi pohranjeni u folder „rezultati_testiranja/slucajno generisani podaci“.

- **n=1000**

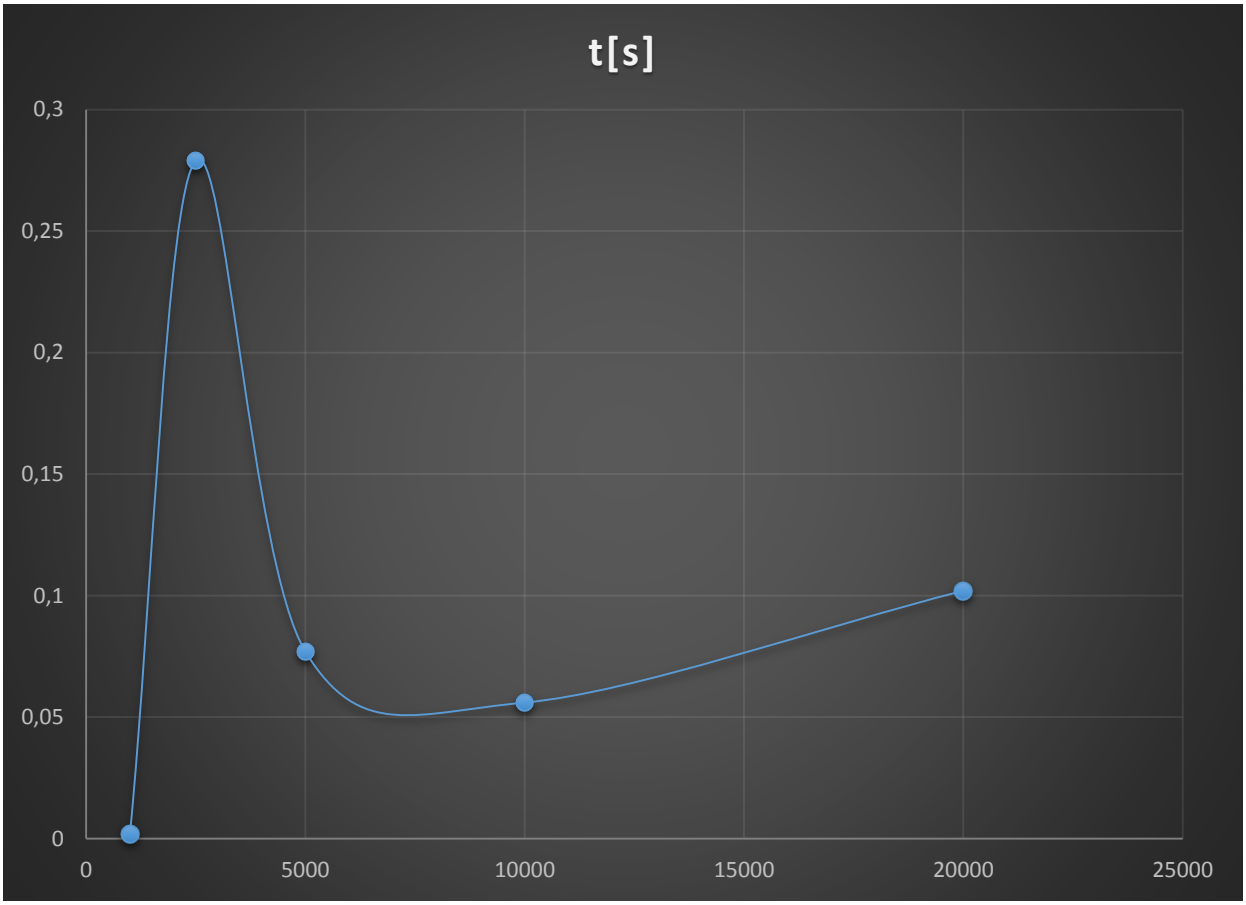
	Broj grana (m)	Vrijeme izvršavanja [s]
1	500	0.012
2	1000	0.038
3	2500	0.064
4	5000	0.049
5	10000	0.009



Vidimo da je za n=1000 čvorova vrijeme izvršavanja jako malo, te pri ovako malim vrijednostima, ono i ne zavisi samo od veličine inputa, nego i od negih drugih parametara računara, npr. korištenje RAM memorije u trenutku izvršavanja i sl., pa zato i ne treba čuditi da se za 10000 grana program „mnogo brže“ završi nego za recimo 2500 grana.

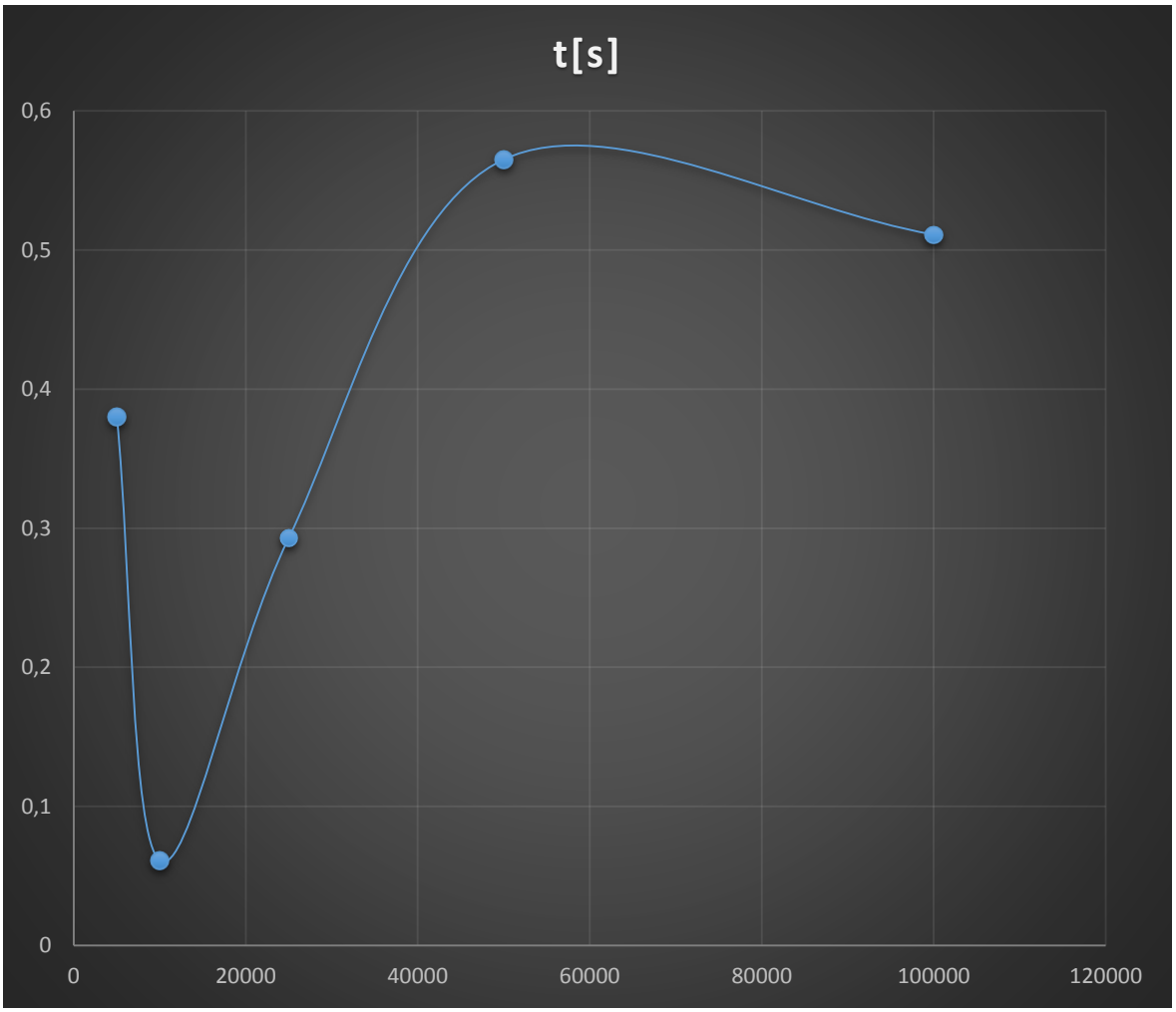
- **n=2500**

Redni broj testa	Broj grana (m)	Vrijeme izvršavanja [s]
6	1000	0.002
7	2500	0.279
8	5000	0.077
9	10000	0.056
10	20000	0.102



- **n=5000**

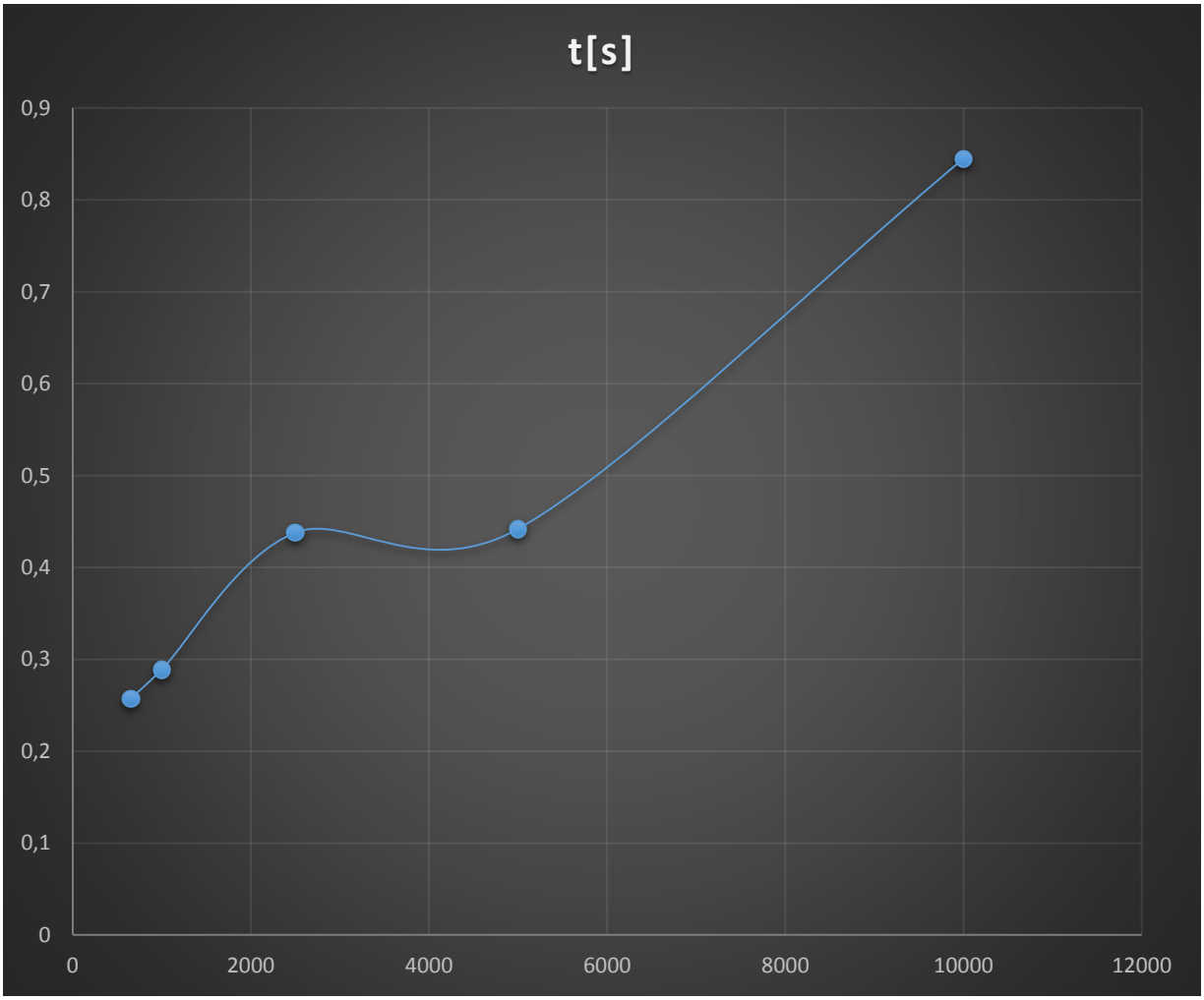
Redni broj testa	Broj grana (m)	Vrijeme izvršavanja [s]
11	5000	0.380
12	10000	0.061
13	25000	0.293
14	50000	0.565
15	100000	0.511



6.2.2. Testiranje prema fiksnom broju grana

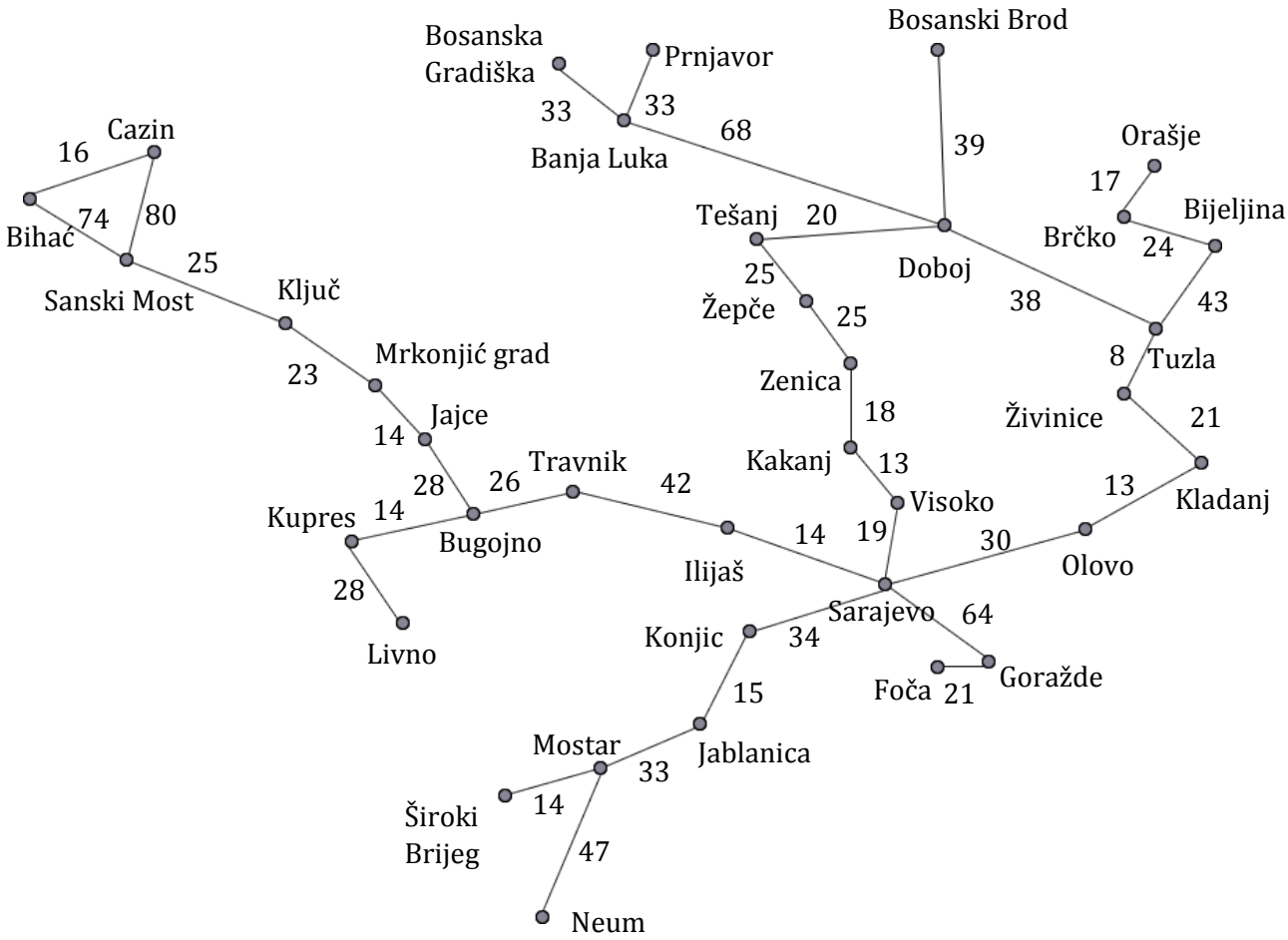
- **m=200000**

Redni broj testa	Broj čvorova (n)	Vrijeme izvršavanja [s]
16	650	0.257
17	1000	0.289
18	2500	0.438
19	5000	0.442
20	10000	0.845



6.3. Testiranje na primjeru mape BiH

Osim standardnih testiranja, ovaj program smo testirali i na jednom stvarnom primjeru koji se može pojaviti u praksi. Unijeli smo podatke o 30-ak najvećih mjesta u BiH i njihovoj povezanosti. Graf za mapu BiH izgleda ovako:



Ulazni podaci za ovaj testni primjer se mogu pronaći u datoteci „input.txt“ u folderu „rezultati_testiranja/mapa bih“, dok se izlazni rezultati mogu naći u istom folderu. Naš primjer se sastojao u pronalasku najkraćeg puta od Tešnja do Bihaća, ali takvog da se posjete Cazin i Neum, a ne posjeti Bugojno. Sa slike se očigledno vidi da od Tešnja do Sanskog Mosta postoji jedinstven put, a od Sanskom Mosta do Bihaća možemo doći ili direktno ili preko Cazina. Pošto se upravo Cazin nalazi na spisku gradova koje bi trebali obići, to je naše optimalno rješenje, iako bi bez tog ograničenja najkraći put vodio direktno iz Sanskog Mosta u Bihać.

7. Literatura

[1] - S. Dasgupta, C. H. Papadimitriou, U. V. Vazirani (2006) „Algorithms“

[2] - Steven S. Skiena (2008) „The Algorithm Design Manual“

[3] - Željko Jurić (2011) „Radna skripta za kurs 'Diskretna matematika' na ETF u Sarajevu“

[4] - Radna skripta za kurs „Algoritmi i strukture podataka“ na ETF u Sarajevu

[5] - S. Sahni (2005) „Handbook of Data Structures and Applications“