Apstraktni tip podataka graf

Đaić, Katarina

Sveučilište Jurja Dobrile u Puli

Fakultet informatike

KATARINA ĐAIĆ

APSTRAKTNI TIP PODATAKA GRAF

Diplomski rad

Pula, rujan, 2019. godine.

Sveučilište Jurja Dobrile u Puli

Fakultet informatike

KATARINA ĐAIĆ

APSTRAKTNI TIP PODATAKA GRAF

Diplomski rad

JMBAG: 0303054563, redoviti student

Studijski smjer: Sveučilišni diplomski studij informatika

Predmet: Napredni algoritmi i strukture podataka

Znanstveno područje: Društvene znanosti

Znanstveno polje: Informacijske i komunikacijske znanosti

Znanstvena grana: Informacijski sustavi i informatologija

Mentor: doc.dr.sc. Tihomir Orehovački

Pula, rujan, 2019. godine

IZJAVA O AKADEMSKOJ ČESTITOSTI

Ja, dolje potpisana, Katarina Đaić, kandidat za magistra informatike, ovime izjavljujem da je ovaj Diplomski rad rezultat isključivo mojega vlastitog rada, da se temelji na mojim istraživanjima te da se oslanja na objavljenu literaturu kao što to pokazuju korištene bilješke i bibliografija. Izjavljujem da niti jedan dio Diplomskog rada nije napisan na nedozvoljen način, odnosno da je prepisan iz kojega necitiranog rada, te da ikoji dio rada krši bilo čija autorska prava. Izjavljujem, također, da nijedan dio rada nije iskorišten za koji drugi rad pri bilo kojoj drugoj visokoškolskoj, znanstvenoj ili radnoj ustanovi.

Student

_____

U Puli, _____, 2019. godine

IZJAVA

o korištenju autorskog djela

Ja, Katarina Đaić, dajem odobrenje Sveučilištu Jurja Dobrile u Puli, kao nositelju prava iskorištavanja, da moj diplomski rad pod nazivom „Apstraktni tip podataka graf“ koristi na način da gore navedeno autorsko djelo, kao cjeloviti tekst trajno objavi u javnoj internetskoj bazi Sveučilišne knjižnice Sveučilišta Jurja Dobrile u Puli te kopira u javnu internetsku bazu završnih radova Nacionalne i sveučilišne knjižnice (stavljanje na raspolaganje javnosti), sve u skladu s Zakonom o autorskom pravu i drugim srodnim pravima i dobrom akademskom praksom, a radi promicanja otvorenoga, slobodnoga pristupa znanstvenim informacijama.

Za korištenje autorskog djela na gore navedeni način ne potražujem naknadu.

U Puli, _____ 2019. godine

Potpis

_____

Sažetak

U ovom diplomskom radu, pod naslovom „Apstraktni tip podataka graf“ je definiran sami graf te su napravljene implementacije u programskim jezicima Python te C++. Preciznije, graf predstavlja vrstu podatkovne strukture putem koje se implementira matematički koncept grafa. Isti se uglavnom sastoji od konačnog (promjenjivog) skupa uređenih parova koji se zovu lukovi (rubovi) i od entiteta koji se zovu čvorovi. Naime, postoje dvije glavne vrste grafa, a to su usmjereni te neusmjereni. Oni se razlikuju po načinu na koji se crtaju, ali i po implementaciji. Osim toga, postoje i dvije vrste pretraživanja grafa, a to su pretraživanje u dubinu i u širinu. Također, opisani su još razni algoritmi te implementirani, a to su Primov, Kruskalov, Dijkstrin, Floyd- Warshall i Bellman- Ford algoritam.

Ključne riječi: graf, rub, vrh, usmjereni graf, neusmjereni graf, pretraživanje u dubinu, pretraživanje u širinu, Primov algoritam, Kruskalov algoritam, Dijkstrin algoritam, Floyd- Warshall algoritam, Bellman- Ford algoritam

Abstract

In this graduation thesis, the "Abstract data type graph" is defined graph itself and implemented in the programming languages of Python and C++. More precisely, the graph represents the type of data structure by which the mathematical concept of the graph is implemented. The same is mainly composed of the final (variable) set of arranged pairs called the arches (edges) and of the entities called the nodes. Namely, there are two main types of graphs that are directed and undirected. They differ in the way they are drawn but also by their implementation. In addition, there are also two types of graph lookups, which are search depth and breadth. Also various algorithms have been introduced, such as Prim's, Kruskal's, Dijkstra's, Floyd-Warshall and Bellman-Ford algorithm.

Keywords: graph, edge, node, directed graph, undirected graph, depth- first search, breadth- first search, Prim's algorithm, Kruskal's algorithm, Dijkstra's algorithm, Floyd-Warshall algorithm, Bellman-Ford algorithm

Sadržaj

Uvod

Kako bi se upravljalo složenošću problema i procesom rješavanja problema, računalni znanstvenici koriste apstrakcije kako bi im se omogućilo da se usredotoče na "veliku sliku", a da se pritom ne izgube u detaljima. Preciznije, stvaranjem modela problemske domene u mogućnosti je za iskoristiti bolji i učinkovitiji proces rješavanja problema. Ti modeli omogućuju da se opišu podatci gdje će sami algoritmi manipulirati na više konzistentniji način s obzirom na sam problem.

U računalnoj znanosti, apstraktni tip podataka je teorijski tip podataka koji je u velikoj mjeri određen operacijama i radom na njemu i ograničenjima koja se primjenjuju. Profesionalci opisuju apstraktnu vrstu podataka kao “matematički model” za skupine tipova podataka ili kao “vrijednost s povezanim operacijama” koja je neovisna o određenoj implementaciji. Tako kada se govori o apstraktnim tipovima podataka, korisno je imati poznavanje konkretnijih tipova podataka koji su jasnije definirani. Nasuprot tome, apstraktni tipovi podataka trebali bi biti šire definicije. Stoga, IT stručnjaci također mogu koristiti ideju „stvarnog života“ gdje apstraktniji identifikator poput „automobila“ predstavlja apstraktni tip podataka, a uži, precizniji identifikator poput „Toyote Yaris“, predstavlja definirani ili konkretni tip podataka.

Apstraktni tip podatka predstavlja tip ili klasu za objekte čije je ponašanje definirano skupom vrijednosti, odnosno skupom operacija. To znači da se valja brinuti samo o tome što podaci predstavljaju, a ne o načinu na koji će se konačno isti izgraditi. Osiguravanjem ove razine apstrakcije stvara se enkapsulacija oko podataka. Ideja toga je uklapanje pojedinosti implementacije koja se skriva iz korisničkog prikaza. To se naziva skrivanje informacija.

Sama definicija apstraktnog tipa podatka označava koje se operacije trebaju obaviti, ali ne i kako će se te operacije provoditi. Isto tako, ne navodi se kako će se podaci organizirati u memoriji te koji će se algoritmi koristiti za provedbu operacija. Prethodno definirano se naziva „apstraktnim“ jer daje pogled neovisan o samoj provedbi. Konkretno, proces pružanja samo bitnih stvari i skrivanje detalja je poznat kao apstrakcija.

Korisnik tipa podatka ne mora znati kako je taj određeni tip podatka implementiran (primjerice, korištenje primitivnih vrijednosti kao što su int, float, char, i sl.). Naime,

apstraktni tip podatka tako predstavlja crnu kutiju koja skriva unutarnju strukturu i dizajn tipa podatka.

Implementacija apstraktnog tipa podataka, često nazvana strukturom podataka, zahtijeva da se pruži fizički prikaz podataka koristeći neku zbirku programskih konstrukata i primitivnih tipova podataka. Točnije, odvajanje ove dvije perspektive omogućit će da se definiraju složeni modeli podataka za same probleme bez davanja bilo kakvih naznaka o tome kako će se model zapravo izgraditi. To osigurava, neovisno o provedbi, prikaz podataka. Budući da obično postoji mnogo različitih načina za implementaciju apstraktnog tipa podataka, ova neovisnost implementacije omogućuje programeru da prebaci detalje implementacije bez promjene načina na koji korisnik podataka komunicira s njim. Korisnik stoga može ostati usredotočen na proces rješavanja problema.

Postoje sljedeće vrste apstraktnih tipova podataka, a to su: lista, stog, red, stablo, binarno stablo, skup, rječnik, prioritetni red, preslikavanje, relacija i graf. U ovom diplomskom radu, pisat će se o apstraktnom tipu podatka graf.

U praksi se koriste različite strukture podataka za prikaz grafova, a to su: lista susjedstva, matrica susjedstva i matrica incidencije. Kod liste susjedstva, isti se pohranjuju kao zapisi ili objekti, dok svaki vrh pohranjuje popis susjednih vrhova. Ova struktura podataka omogućuje pohranjivanje dodatnih podataka na vrhovima. Ti se dodatni podaci mogu pohraniti ako su i rubovi pohranjeni kao objekti, u kojem slučaju svaki vrh pohranjuje svoje incidentne rubove i svaki rub pohranjuje svoje incidentne vrhove. Za razliku od liste susjedstva, matrica susjedstva (dvodimenzionalna matrica) sadrži redove koji predstavljaju izvorne točke i stupce koji stoga predstavljaju odredišne vrhove. Naime, podaci o rubovima i vrhovima moraju biti pohranjeni izvana. Preciznije, između svakog para vrhova može se pohraniti samo trošak (težina) za jedan rub. Posljednja struktura podataka koja se koristi, matrica incidencije (dvodimenzionalna Booleova matrica), sadrži redove koji predstavljaju vrhove i stupce koji predstavljaju rubove. Unosi ovdje označavaju je li vrh u nizu upadljiv na rub stupca.

Apstraktni tip podatka graf obuhvaća sljedeće operacije: *Init, InsertVertex (AddVertex), DeleteVertex (RemoveVertex), InsertEdge (AddEdge), DeleteEdge (RemoveEdge), IsEmpty i Adjacent*. Operacija *Init* služi za iniciranje praznog grafa. *InsertVertex* dodaje novi vrh u graf, dok *DeleteVertex* briše vrh iz grafa te sve njegove bridove koji su s njim

incidentni. Za razliku od *InsertVertexa* i *DeleteVertexa*, *InsertEdge* dodaje brid u graf koji je incidentan s vrhovima, dok *DeleteEdge* briše brid iz grafa. Sljedeća logička funkcija, *IsEmpty,* provjerava da li je graf prazan, odnosno provjerava da li je broj vrhova grafa jednak nuli. Posljednja funkcija, *Adjacent*, provjerava da li su vrhovi susjedni u grafu.

U prvom poglavlju će biti opisana definicija grafa, dok će u njegovim podpoglavljima biti opisan prikaz i terminologija grafa, logička, aplikacijska i implementacijska razina.

Drugo poglavlje će opisivati grafičke prikaze.

U trećem poglavlju će biti objašnjeni algoritmi elementarnih grafova, odnosno pretraživanje u širinu te u dubinu. Podpoglavlje pretraživanje u širinu će imati primjer grafa, njegovu implementaciju u programskim jezicima Python te u C++- u. Pretraživanje u dubinu će sadržavati primjer grafa, implementaciju u programskim jezicima Python te u C++- u. Nadalje, ovo podpoglavlje još će opisivati neusmjerene grafove, pojam bikonektivnosti, Eulerove te Hamiltonove grafove. Sljedeće podpoglavlje trećeg poglavlja, objašnjava usmjerene grafove, odnosno pronalaženje jakih komponenti. Isto tako, treće poglavlje još sadrži i topološki poredak.

Kod četvrtog poglavlja će biti definiran pojam povezivanja. Točnije, povezivanje kod usmjerenih, odnosno kod neusmjerenih grafova.

U petom poglavlju će biti opisan uvod u NP- potpunost, dok će u njegovim podpoglavljima biti definirani lagani i teški problemi (njihove razlike), klasa NP te NP kompletni problem.

Šesto poglavlje će sadržavati minimalno razgranato (razapinjuće) stablo, tj. rast minimalnog razgranatog stabla i Primov te Kruskalov algoritam. Ti će algoritmi još definirati primjere, implementacije u programskim jezicima Python te C++- u.

Kod sedmog poglavlja će biti objašnjeni algoritmi najkraćeg puta, odnosno sama definicija najkraćeg puta, problem algoritma najkraćeg puta, najkraće puteve s jednim izvorom u usmjerenim acikličkim grafovima, Dijkstrin algoritam, razliku između ograničenja i najkraćeg puta te dokaze u svojstvima najkraćih puteva. Problem algoritma najkraćeg puta će opisivati najkraće puteve s jednim izvorom te Bellman-Ford algoritam koji će ujedno imati primjer grafa, implementacije u programskim

jezicima Python te C++- u. Dijkstrin algoritam će također sadržavati primjer grafa, implementacije u programskim jezicima Python te C++- u.

Osmo poglavlje će definirati najkraće puteve svih parova, dok će njegova podpoglavlja opisivati najkraće puteve i množenje matrica, Floyd- Warshall algoritam te Johnsonov algoritam za rijetke grafove. Podpoglavlje Floyd- Warshall algoritam još će imati primjer grafa, implementacije u programskim jezicima Python te C++- u.

Kod devetog poglavlja su objašnjeni grafovi s negativnim rubnim troškovima, posebice otkrivanje ciklusa (problem s pronalaženjem unije) te aciklički grafovi.

Deseto poglavlje će sadržavati maksimalni protok, tj. maksimalni protok minimalne cijene, mreže protoka, Ford- Fulkersenovu metodu, maksimalno dvostrano podudaranje, Push- relabel algoritme i algoritam Relabel- to- front. Mreže protoka će još dodatno opisivati probleme s mrežnim protokom.

Posljednje, jedanaesto poglavlje, će definirati probleme stabilnog podudaranja, probleme zadatka te usklađivanje kod nebibarnih grafova.

1. Definicija grafa

Grafovi pružaju krajnju fleksibilnost kod struktura podataka. Samim time, oni mogu modelirati i sustave iz stvarnog svijeta te apstraktne probleme, pa se koriste u stotinama aplikacija.

Grafovi su raširena podatkovna struktura u računalnoj znanosti i različitim računalnim aplikacijama, a algoritmi temeljeni za rad s njima su temeljeni za polje. Postoje stotine zanimljivih računskih problema koji su definirani u grafovima. Ovdje se ne radi o strukturi podataka, već o tome što grafovi znače pohranjivanje i analizu metapodataka te veza koje se prikazuju u podacima (primjerice, gradovi u nekoj zemlji). Tako je cestovna mreža, koja ih povezuje, prikazana kao graf (ista se mora analizirati). Osim toga, može se ispitati da li je iz jednog grada moguće doći u drugi grad ili pronaći najkraći put između ta dva grada.

Graf je nelinearna podatkovna struktura koja se sastoji od čvorova i rubova. Čvorovi se ponekad nazivaju i vrhovima, a rubovi linije ili lukovi koji povezuju bilo koja dva čvora u grafu. Formalno, graf se može definirati na sljedeći način: “Graf se sastoji od konačnog skupa vrhova (ili čvorova) i skupa rubova koji povezuju par čvorova.“ (Geeksforgeeks_1, n_d).

Grafovi se koriste za rješavanje mnogih stvarnih problema (primjerice, za predstavljanje mreža). Mreže tako mogu uključivati staze kod gradske ili telefonske mreže (mreže strujnih krugova). Isti se također koriste u društvenim mrežama (primjerice, Facebook). Na Facebooku je svaka osoba predstavljena vrhom (ili čvorom). Svaki čvor je struktura i sadrži informacije (primjerice, ID osobe, ime, spol, mjesto, itd.).

Postoje dva važna skupa objekata koji određuju graf i njegovu strukturu: (Algolist, n_d)

1. Prvi skup je *V*, koji se naziva skup točaka. U primjeru s cestovnom mrežom gradovi su vrhovi. Stoga, svaki se vrh može nacrtati kao krug s brojem vrha.
2. Drugi skup je *E*, koji se naziva skup rubova (bridova). *E* je podskup od *V**V*. Jednostavnije rečeno, svaki rub povezuje sve vrhove, uključujući i slučaj, kada je vrh povezan sa samim sobom (takav se rub naziva petlja).

Ako dva ili više bridova povezuju isti par vrhova, isti se zove višestruki brid. U slučaju da graf sadrži višestruke bridove, naziva se multigraf, a ako nema ni petlje ni

višestrukih bridova, isti se zove jednostavan graf. Broj vrhova u grafu se označava s *v(G)*, a broj bridova s *e(G)*. (Nikšić, 2004., str.7).

Mali uzorak raspona problema gdje se rutinski upotrebljavaju, odnosno primjenjuju grafovi: (Clifford, 2013., str. 381)

1. Modeliranje povezanosti u računalnim i komunikacijskim mrežama.
2. Predstavljanje karte kao skupa mjesta s razmacima između lokacija (koristi se za izračun najkraćih ruta između lokacija).
3. Modeliranje pogonskih kapaciteta u transportnim mrežama.
4. Pronalaženje puta od početnog stanja do ciljnog stanja (primjerice, u rješavanju problema umjetne inteligencije).
5. Modeliranje računalnih algoritama koji pokazuju prijelaze iz jednog programa u drugi.
6. Pronalaženje prihvatljive mjere za završavanje zadaća u složenoj aktivnosti (primjerice, izgradnja velikih zgrada).
7. Modeliranje odnosa (primjerice, obiteljska stabla, poslovne ili vojne organizacije i znanstvene taksonomije).

1.1 Prikaz i terminologija grafa

Graf *G=(V, E)* sastoji se od skupa vrhova *V* te skupa rubova *E*, tako da je svaki rub u *E* veza između para vrhova *V*. Broj vrhova je napisan kao |*V*|, a broj rubova je napisan u obliku |*E*|. Konkretnije, |*E*| se može kretati između nule te $|V|^2$- |*V*|. Graf s relativno malim rubovima se naziva rijedak graf, dok se graf koji ima puno rubova naziva gusti graf. Također, smatra se da graf koji sadrži sve moguće rubove, se naziva potpunim grafom.

Graf s rubovima usmjerenim od jednog vrha ka drugome naziva se usmjereni graf (ili digraf). Za razliku od prethodno navedenog grafa, graf čiji rubovi nisu usmjereni naziva se neusmjereni graf. Postoji i graf s oznakama koji su povezani s njegovim vrhovima, i ono se naziva označeni graf. Dvije točke kod grafa su susjedne ako su povezane rubom. Isti se vrhovi nazivaju susjedi. Zapisani rub je rub koji povezuje vrhove *U* i *V (U, V)*. Za takav se rub kaže da se pojavljuje na vrhovima *U* i *V*. U slučaju da je povezan

sa svakim rubom ono može predstavljati težinu troška (cijene). Za grafove čiji rubovi imaju težine navodi se da su ponderirani.

Slijed vertikala $v_1, v_2, \ldots, v_n$ tvori put duljine $n-1$ ako postoje rubovi od v_i do v_{i+1} za $1 \leq i < n$. Put je jednostavan ako su svi vrhovi na samom putu različiti. Duljina putanje predstavlja broj rubova koje ujedno i sadrži. Pojam ciklusa objašnjava put duljine tri (ili više) koji povezuje neki vrh v_1 sa samim sobom. Konkretno, ciklus je jednostavan ako je put jednostavan, osim što su prvi i zadnji vrhovi isti.

Podgraf S formira se iz grafa G odabirom podskupa V_S čvorova i podskupa E_S G- ovih rubova tako da za svaki rub E u E_S oba E- vrha su u V_S.

Neusmjereni graf je povezan ako postoji barem jedan put od bilo kojeg toka do bilo kojeg drugog. Maksimalno povezani podgrafi neusmjerenog grafa nazivaju se povezane komponente.

Graf koji ne sadrži cikluse se naziva acikličan. Prema tome, usmjereni graf bez ciklusa naziva se usmjereni aciklički graf (engl. directed acyclic graph, DAG).

U slučaju da je slobodno stablo povezano, isti se zove neusmjereni graf bez jednostavnih ciklusa. Ekvivalentna definicija je da je slobodno stablo povezano i ima $|V|-1$ rubova.

Postoje dvije najčešće korištene metode za prikazivanje grafova. Prva metoda je matrica susjedstva. Ona je za graf definirana kao $|V| * |V|$ polje. Pretpostavlja se da je $|V| = n$ i da su vrhovi označeni od v_0 do v_n-1. Red i matrice susjedstva tako sadrže zapise za vrh v_i. Stupac j u redu i označen je ako postoji rub od v_i do v_j te nije označen na drugi način. Prema tome, matrica susjedstva zahtijeva jedan bit na svakom položaju. Alternativno, ako se želi pridružiti broj svakim rubom, kao što je težina ili udaljenost između dvaju vrhova, tada svaka pozicija matrice mora pohraniti taj broj. U oba slučaja, prostorni zahtjevi za matricu susjedstva imaju složenost $\Theta(|V|^2)$.

Druga uobičajena metoda grafova je lista susjedstva. Ista predstavlja niz povezanih popisa. Niz $|V|$ se koristi za dugačke stavke, s položajem i pohranjivanjem pokazivača na povezani popis rubova za vrh v_i. Ova povezana lista predstavlja rubove po vrhovima koji su susjedni uz vrh v_i. Lista susjedstva stoga je generalizacija „prikaza djece“ za stabla.

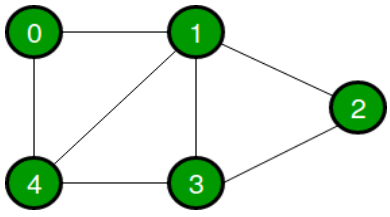
Zahtjevi za pohranu liste susjedstva ovise o broju rubova te broju vrhova u grafu. Naime, mora postojati unos niza za svaki vrh (čak i u slučaju da vrh nije susjedan s bilo kojim drugim vrhom i stoga nema elemenata na svom povezanom popisu), a svaki rub se mora pojaviti na jednom od popisa. Dakle, složenost je $\Theta(|V| + |E|)$.

Matrica susjedstva i lista susjedstva se mogu koristiti za pohranjivanje usmjerenih, odnosno neusmjerenih grafova. Tako je svaki rub neusmjerenog grafa koji povezuje vrhove *U* i *V* predstavljen s dva usmjerena ruba: jedan od *U* do *V* i jedan od *V* do *U*.

Grafički prikaz glede više prostora ovisi o broju rubova u grafu. Konkretno, lista susjedstva pohranjuje informacije samo za one rubove koji se zapravo pojavljuju u grafu, dok matrica susjedstva zahtijeva prostor za svaki potencijalni rub, bilo da postoji ili ne. Međutim, matrica susjedstva ne zahtijeva nikakvo opterećenje, što može predstavljati značajan trošak, pogotovo ako su jedini podaci pohranjeni za jedan rub (jedan bit) koji potom ukazuju na njegovo postojanje. Kako graf postaje sve gušći, matrica susjedstva postaje relativno više učinkovit prostor. Za razliku od gustog grafa, rijetki će imati prikaz liste susjedstva u pogledu više prostora.

Matrica susjedstva često zahtijeva viši asimptotski trošak za algoritam nego što bi to bio slučaj kada bi se koristila lista susjedstva. Razlog tome je što je uobičajeno da grafički algoritam posjećuje svakog susjeda (svakog vrha). Pomoću liste susjedstva ispituju se samo stvarni rubovi koji povezuju vrh s njegovim susjedima. Međutim, matrica susjedstva mora gledati na svaki od svojih $|V|$ potencijalnih rubova, čime se dobiva ukupni trošak vremena $\Theta(|V^2|)$. Inače bi algoritam zahtijevao vrijeme $\Theta(|V|+|E|)$. To predstavlja značajan nedostatak kada je graf oskudan (ne kada je pun).

Slika 1.: Prikaz primjera grafa



Izvor: geeksforgeeks_2, n_d

1.2 Logička razina

Binarna stabla pružaju vrlo koristan način predstavljanja odnosa gdje postoji hijerarhija. To znači da na čvor upućuje najviše jedan drugi čvor (njegov roditelj), a svaki čvor ukazuje na najviše dva druga čvora.

Ako se ukloni ograničenje da svaki čvor može imati samo jedan roditeljski čvor, dolazi do podatkovne strukture koja se naziva graf. Taj graf se sastoji od skupa čvorova koji se nazivaju rubovi (ili lukovi) koji povezuju čvorove. (Dale, 2003., str. 547).

Skup rubova opisuje odnose između vrhova. Primjerice, ako su vrhovi nazivi gradova, rubovi onda povezuju te vrhove i predstavljaju ceste između para gradova. Tako kada cesta koja vodi od primjerice, grada Požege do Osijeka nema smjer, ono se naziva neusmjerenim grafom. Međutim, ako postoji smjer između ta dva grada, onda se takav prikaz naziva usmjerenim grafom.

Iz perspektive programera, vrhovi predstavljaju ono što je predmet istraživanja (primjerice, ljude, kuću, gradove, itd.). Matematički, vrhovi su nedefinirani koncept na kojem počiva teorija grafova. Zapravo, veliki dio formalne matematike je povezan s grafovima. U drugim računalnim tečajevima vjerojatno će se prije analizirati grafovi te dokazivati teoremi o njima.

Da bi se odredio skup točaka, isti se navode u skupu notacija unutar *{* i *}* zagrada (primjerice, *V (Graf 1)= {A, B, C, D}*).

Skup rubova određuje se navođenjem niza rubova. Za označavanje svakog ruba, napišu se imena dvaju vrhova koji se povezuju u zagradama, sa zarezom između njih (primjerice, *E (Graf 1)= {(A, B), (A,D), (B, C), (B, D)}*).

Budući da je prethodno navedeni graf neusmjeren, redoslijed vrhova je u svakom rubu nevažan (primjerice, *E (Graf 1)= {(B, A), (D, A), (C, B), (D, B)}*).

Ako je graf digraf, smjer ruba je prvo označen s kojim je vrhom.

Ako su dva vrha u grafu povezana rubom, isti se nazivaju susjedni. Primjerice, u grafu *1*, vrhovi *A* i *B* su susjedni, ali vrhovi *A* i *C* nisu. U slučaju da su vrhovi povezani s usmjerenim rubom, tada se kaže da je prva točka susjedna drugoj, za koju se kaže da je susjedna od prve.

Stablo je poseban slučaj usmjerenog grafa, u kojem svaki vrh može biti susjedan samo od jednog drugog vrha (njegov roditeljski čvor), a jedan vrh (korijen) nije susjedan od bilo kojeg drugog vrha.

Put od jednog vrha do drugog sastoji se od niza vrhova koji ih povezuju. Konkretno, da bi put postojao, neprekinuti slijed rubova mora početi od prvog, preko bilo kojeg broja vrha, do drugog vrha. Jedinstveni put postoji od korijena pa do svakog drugog čvora u stablu.

U potpunom grafu svaka se točka nalazi uz svaki drugi vrh. Ako postoji *N* vrhova, bit će *N*(N -1)* rubova u potpunom usmjerenom grafu i *N*(N -1) /2* rubova u potpunom neusmjerenom grafu.

U ponderiranom grafu svaki rub nosi vrijednost. Isti se mogu koristiti za predstavljanje aplikacija u kojima je vrijednost veze između vrhova važna, a ne samo postojanje veze. Primjerice, u ponderiranom grafu, vrhovi predstavljaju gradove, a rubovi označavaju letove Air Busters Air linije koji povezuju gradove. Same težine pričvršćene za rubove predstavljaju zračne udaljenosti između parova gradova.

Primjerice, put iz grada Denvera do grada Washingtona. Ovdje se traži put između ta dva grada. Ako je ukupna udaljenost putovanja određena zbrojem udaljenosti između svakog para gradova na putu, može se izračunati udaljenost dodajući „uteg“ pričvršćen za rubove koji čine put između njih. Valja imati na umu da višestruki putevi mogu povezati dva vrha.

Graf na apstraktnoj razini je opisan kao skup vrhova i skup rubova koji povezuju neke ili sve točke jednu s drugom. Postavlja se pitanje: „Koje vrste operacija su definirane na grafu“? U ovom se koraku navodi i implementira mali skup korisnih grafičkih operacija. Mnoge druge operacije na grafu mogu se definirati. Prvo, međutim, moraju se definirati operacije koje su potrebne za obradu grafova.

Naposljetku, potrebno je dodati vrhove i rubove, odrediti težinu na rubu i dobiti pristup vrhovima koji su susjedni od vrha.

1.3 Aplikacijska razina

Specifikacija apstraktnog tipa podatka (engl. abstract data type) uključuje samo najosnovnije operacije. Graf ovdje može preći u druge oblike. Kao rezultat toga, smatra se da je ukrštanje aplikacija graf, a ne sama urođena operacija. Osnovne operacije dane u samoj specifikaciji omogućuju da se implementiraju različiti prijelazi neovisno o tome kako je sam graf implementiran.

Strategija spuštanja grane do njezine najdublje točke i kretanja prema gore naziva se strategija dubine (pretraživanje u dubinu). Postoji još jedan način da se posjeti svaki vrh u stablu: prvo se posjeti svaki vrh na razini 0 (korijen), zatim svaki vrh na razini 1, zatim svaki vrh na razini 2, itd. Posjećivanje svakog vrha po razini na ovaj način naziva se strategija širine (pretraživanje u širinu). Uz same grafove, korisne su strategije pretraživanja u dubinu te pretraživanja u širinu (primjer zrakoplova).

Osim što aplikacijska razina obuhvaća strategije dubine i širine, ista obuhvaća i problem najkraćeg puta s jednim izvorom. Višestruki putevi mogu povezati jedan vrh s drugim (primjerice, pronalazak najkraćeg puta od grada Požege do svakog drugog grada kako bi se došlo do grada Osijeka). Pod najkraćim putem podrazumijeva se put čije se rubne vrijednosti (ili utezi), kada se zbrajaju, daju najmanju sumu.

1.4 Implementacijska razina

Implementacijska razina obuhvaća implementaciju na bazi polja. Jednostavan način za predstavljanje *V* (graf), vrhova u grafu, je s nizom gdje su elementi tipa vrha (*VertexType*). Primjerice, ako bi vrhovi predstavljali gradove, *VertexType* bi bio neka reprezentacija nizova. Također, jednostavan način predstavljanja *E* (graf), rubova u grafu, je s matricom susjedstva, dvodimenzionalnim nizom vrijednosti rubova (težina). Tako se graf sastoji od *numVerticsa* (člana podatka) i dvodimenzionalnih rubova polja.

Ista razina obuhvaća i povezanu implementaciju. Prednosti predstavljanja rubova u grafu s matricom susjedstva odnose se na njegovu brzinu i jednostavnost. S obzirom na indekse dvaju točaka, određivanje postojanja (ili težine) ruba između njih sadrži *O(1)* operacija. Problem s matricama susjedstva je u tome što je njihovo korištenje

prostora $O(N^2)$, gdje je N maksimalni broj vrhova u grafu. U slučaju da je maksimalni broj vrhova u grafu velik, matrice susjedstva tada mogu trošiti puno prostora.

U prošlosti se pokušavalo uštedjeti na prostoru dodjeljivanjem memorije onako kako je bilo potrebno u vrijeme izvođenja, koristeći povezane strukture. Naravno, može se koristiti sličan pristup pri implementaciji grafova. Formalnije, liste susjedstva spadaju u povezane popise, jedan popis po vrhu, koji identificiraju vrhove kojima je povezan svaki vrh. Isti se mogu implementirati na nekoliko različitih načina.

Prvo, vrhovi mogu biti pohranjeni u nizu (listi). Svaka komponenta ovog niza sadrži pokazivač na povezani popis rubnih čvorova. Svaki čvor u tim povezanim popisima sadrži broj indeksa, težinu i pokazivač na sljedeći čvor u listi susjedstva.

Drugo, implementacija ne koristi nikakve nizove. Umjesto toga, popis vrhova je implementiran kao povezani popis. Ovdje svaki čvor u listi susjedstva sadrži pokazivač na podatke o vršcima, a ne indeks samog vrha. Budući da se mnogi od tih pokazivača pojavljuju, za iste se koristi tekst da bi se opisali vrhovi gdje ih svaki pokazivač označava umjesto da ih se crta u obliku strelica.

2. Grafički prikazi

Iako ovo ima prednost krajnje jednostavnosti, prostorni zahtjev traje $\Theta |V|^2$, što može biti preveliko ako graf ima puno rubova. Matrica susjedstva je ovdje prikladna reprezentacija ako je graf gust $(|E| = \Theta (|V|^2))$. Pretežito, prethodno navedeno nije točno kod većine aplikacija. Primjerice, može se pretpostaviti da graf predstavlja kartu ulica (orijentacija poput Manhattana, gdje gotovo sve ulice prolaze ili na sjever ili na jug). Stoga je bilo koje raskrižje vezano za četiri ulice, tako da ako je graf usmjeren i sve ulice su dvosmjerne, tada je $|E|$ približno jednako $4\ |V|$. U slučaju da ima 3 000 raskrižja, onda postoji graf s 3 000 vrhova s 12 000 rubnih ulaza, što bi zahtijevalo niz od 9 000 000. Većina tih unosa bi sadržavala nulu. To je zapravo intuitivno loše, jer se želi da podatkovna struktura pohranjuje podatke koji su prisutni, a ne podatke koji nisu prisutni.

Ako graf nije gust, drugim riječima, ako je graf oskudan (ili rijedak), bolje rješenje je prikaz liste susjedstva. Za svaki vrh se tada drži lista svih susjednih vrhova. Zahtjev za

prostorom je O |E| + |V|, što je linearno. Ako rubovi imaju utege, onda se te dodatne informacije spremaju u liste susjedstva.

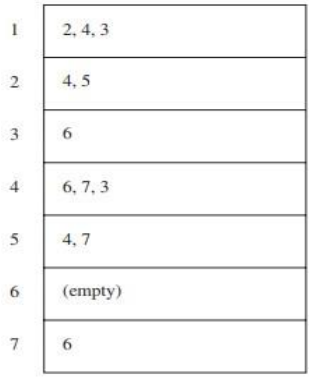
Liste susjedstva predstavljaju standardni način prikaza grafova. Neusmjereni grafovi se mogu prikazati na sličan način; svaki rub (*u, v*) pojavljuje se na dvije liste, tako da se prostorni prostor u osnovi udvostručuje. Stoga, uobičajeni zahtjev u algoritmima grafova je pronaći sve točke koje graniče s nekim danim vrhom *v*, a to se može učiniti u vremenu proporcionalnom broju takvih pronađenih vrhova, tj. jednostavnim skeniranjem na odgovarajućoj listi susjedstva.

Postoji nekoliko alternativa za održavanje liste susjedstva. Prvo, valja primijetiti da se same liste mogu održavati u bilo kojem vektoru. Međutim, za oskudne grafove, u slučaju da se koriste vektori, programer će možda morati inicijalizirati svaki vektor manjeg kapaciteta od zadanog; inače bi mogao postojati značajno veliki gubitak prostora.

Budući da je veoma važno biti u mogućnosti brzo dobiti popis susjednih vrhova za bilo koji vrh, dvije osnovne opcije su korištenje karte u kojoj su ključevi vrhovi, a vrijednosti su liste susjedstva (ili za održavanje svake liste susjedstva kao podatka, primjerice člana *Vertex* klase). Prva opcija je vjerojatno jednostavnija, dok druga opcija može biti brža, jer se izbjegavaju ponovljena pretraživanja na karti.

U drugom slučaju, ako je vrh znakovni niz (primjerice, naziv zračne luke ili naziv raskrižja ulica), tada se može koristiti karta u kojoj je ključno ime vrha, a vrijednost je *Vertex* (pretežno pokazivač na *Vertex*), a svaki *Vertex* objekt čuva listu (pokazivača) susjednih vrhova, a možda i izvornog imena niza.

Slika 2.: Prikaz susjedne liste grafa



Izvor: Weiss, 2014., str.382

3. Algoritmi elementarnih grafova

Ovdje se javlja pojam slijepog pretraživanja (ili neinformiranog), čije su sljedeće karakteristike:

1. Ne koriste dodatne informacije o stanjima osim onih definiranih samim problemom.
2. Sistematski napreduju kroz prostor stanja pretražujući čvorove (nasljednike) trenutnih.
3. Razlikuju ciljne od ne- ciljnih stanja.

Kao i kod stabla, kod grafova postoje dvije osnovne metode pretraživanja (elementarni neinformirani algoritmi pretraživanja prostora stanja koji su “preuzeti” iz teorije grafova), a to su:

1. Prvo u dubinu (engl. Depth- First Search- DFS).
2. Prvo u širinu (engl. Breadth- First Search- BFS).

3.1 Pretraživanje u širinu

Ova vrsta pretraživanja se još naziva pretraživanje širenjem fronte pretraživanja. Konkretno, kreće se od jednog vrha, nakon čega se uzimaju njegovi susjedi, pa njihovi susjedi, itd., dok se ne pretraži cijeli graf.

Ovdje se koriste sljedeće strukture podataka:

1. Polje za čuvanje informacija o nepretraženim vrhovima.
2. Red za čuvanje informacija potrebnih metodi pretraživanja s vraćanjem (engl. backtracking).

Složenost ove vrste pretraživanja grafa je *O(n)*.

Slijedi prikaz pseudokoda pretraživanja u širinu:

```
BFS {
        Staviti početni čvor na početak reda
        Sve dok red nije prazan {
```

Uzeti čvor s početka reda

Ako je trenutni čvor konačan, rezultat= trenutni čvor

Ako je trenutni čvor u listi posjećenih, nastaviti s idućim

Staviti sve susjedne čvorove trenutnog čvora koji nisu posjećeni na kraj reda

}

Rezultat nije pronađen

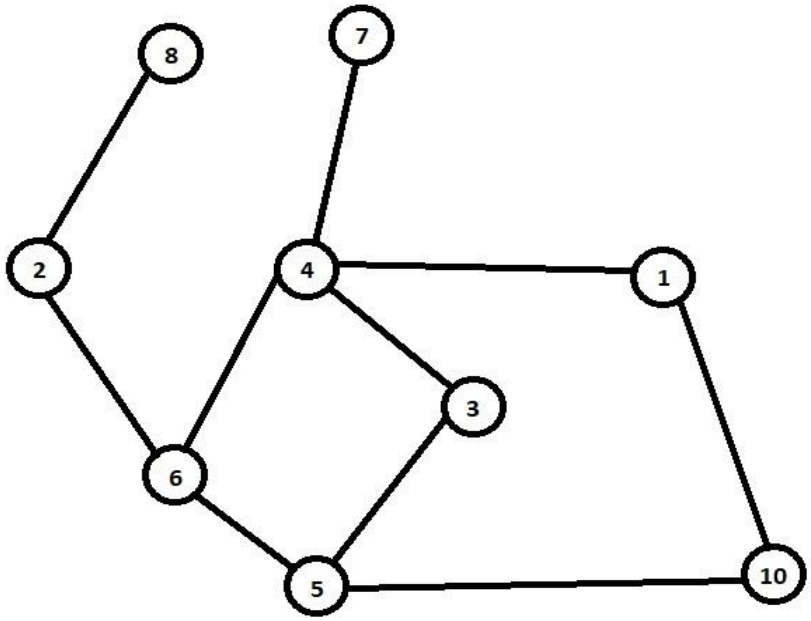
}

Ova vrsta pretraživanja se smatra potpunom te optimalnom.

3.1.1 Primjer

Zadatak nalaže da je potrebno pretražiti dolje prikazan graf putem algoritma pretraživanja u širinu.

Slika 3.: Prikaz primjera grafa pretraživanja u širinu



Izvor: Vlastiti rad

Algoritam pretraživanja u širinu nalaže sljedeći obilazak (počevši od broja 2): 2, 8, 6, 4, 5, 1, 3, 7 i 10.

3.1.2 Implementacija u Pythonu

Ovdje je prikazana implementacija u Pythonu pomoću reda.

Prikaz rješenja:

Datoteka main.py:

```
graph={1:[4,10],
       2:[8,6],
       3:[4,5],
       4:[7,1,6,3],
       5:[6,3,10],
       6:[2,4,5],
       7:[4],
       8:[2],
       10:[1,5]}

def bfs(graph, start, path=[]):
    queue = [start]
    while queue:
        vertex = queue.pop(0)
        if vertex not in path:
            path.append(vertex)
            queue.extend(set(graph[vertex]) - set(path))
    return path

if __name__ == "__main__":

 print("Pretrazivanje u sirinu, pocevsi od vrha 2.")
 print("Rezultat je:")
 print(bfs(graph, 2))
```

Izvor: Anonymous_30, 2014.

Opis koda:

Prvi dio koda sadrži graf koji je implementiran kao rječnik, odnosno isti se sastoji od vrhova (prva kolona) te od rubova (brojevi u uglatim zagradama). Nadalje, funkcija *def bfs(graph, start, path=[])* sadrži sve vrhove (čvorove) grafa (povezane komponente) pritom koristeći pretraživanje u širinu. Sljedeći red, *queue = [start],* prati čvorove koji se provjeravaju. Potom se petlja *while queue:* izvršava sve dok se ne provjere svi

čvorovi. Red *vertex = queue.pop(0)* briše prvi čvor iz reda čekanja. Sljedeće, ako čvor nije istražen *(if vertex not in path:),* dodaje se čvor u popis označenih čvorova *(path.append(vertex)).* Potom se proširuje red sa svim čvorovima koji se još ne nalaze na putu, pa se pri tom koristi operacija skupa (*queue.extend(set(graph[vertex]) - set(path))).* Zaključno, vraća se dati put pomoću *return path*. Nakon toga je definirana funkcija *main()* unutar koje se nalazi pozivanje funkcije *((bfs(graph, 2)))* te ispis rješenja. Prva varijabla koju prima dana funkcija je prethodno definirani graf te vrh (čvor) od kojeg počinje sami algoritam pretraživanja u širinu.

Rješenje primjera:

```
Pretrazivanje u sirinu, pocevsi od vrha 2.

Rezultat je:

[2, 8, 6, 4, 5, 1, 3, 7, 10]
```

3.1.3 Implementacija pomoću vektora

Ovdje je prikazana implementacija u programskom jeziku C++ pomoću strukture podatka vektor.

Datoteka bfs.cpp:

```
#include <iostream>
#include <queue>
#include <vector>
using namespace std;

struct Edge {
  int src, dest;
};

class Graph
{
public:

  vector<vector<int>> adjList;

  Graph(vector<Edge> const &edges, int N)
  {
    adjList.resize(N);
```

```
    for (auto &edge: edges)
    {
      adjList[edge.src].push_back(edge.dest);
      adjList[edge.dest].push_back(edge.src);
    }
  }
};

void BFS(Graph const &graph, int v, vector<bool> &discovered)
{
  queue<int> q;

  discovered[v] = true;

  q.push(v);

  while (!q.empty())
  {
    v = q.front();
    q.pop();
    cout << v << " ";

    for (int u : graph.adjList[v])
      if (!discovered[u])
      {
        discovered[u] = true;
        q.push(u);
      }
  }
}

int main()
{
  vector<Edge> edges = {
    {2, 8}, {2, 6}, {6, 4}, {6, 5}, {5, 3}, {5, 10},
    {3, 4}, {10, 1}, {1, 4}, {4, 7}
  };

  int N = 11;

  Graph graph(edges, N);

  vector<bool> discovered(N, false);

  for (int i = 2; i < N; i++) {
    if (discovered[i] == false) {
      BFS(graph, i, discovered);
    }
  }
```

```
  return 0;
}
```

Izvor: Anonymous_31, n_d

Opis rješenja:

Prvo su uključene biblioteke <*queue*> i <*vector*> za rad s tim strukturama podataka. Potom je implementirana struktura podataka za pohranjivanje rubova grafa (*struct Edge*), pa klasa koja predstavlja graf objekt (*class Graph*). Njezin javni dio sadrži vektor vektora koji predstavlja popis susjednosti (*vector<vector<int>> adjList*) te konstruktor grafa koji mijenja veličinu vektora (*resize*) na *N* elemenata tipa *vector<int> (adjList.resize(N))* gdje se dodaju rubovi u neusmjerenom grafu. Funkcija BFS izvodi isti na grafu počevši od vrha *v* pa stvara red (*queue<int> q*) koji se koristi za izradu pretraživanja u širinu (označava izvorni vrh otkrivenim (*discovered[v] = true*), gura izvorni vrh u red (*q.push(v)),* pokreće isto dok red nije prazan (*while*), izbacuje prvi vrh iz reda te ga ispisuje (*v = q.front();, q.pop();, cout << v << " ")).* Za svaki rub (v->u) se isti označava otkrivenim te ga se stavlja u red. Unutar funkcije *int main()* je definirano iterativno pretraživanje u širinu gdje su unutar vektora definirani rubovi grafa (*int N = 11),* potom se isti definiraju (*Graph graph(edges, N)),* pa se kreira sami graf s tim prethodno definiranim rubovima. Idući podatak tipa vektor provjerava je li su vrhovi otkriveni ili nisu (*vector<bool> discovered(N, false)),* dok petlja *for* čini pretraživanje u širinu iz svih neotkrivenih čvorova kako bi se obuhvatile sve nepovezane komponente grafa. Na kraju se vrši pretraživanje u širinu (poziv funkcije), počevši od vrha dva *i* (*BFS(graph, i, discovered)).*

Prikaz rješenja:

```
2 8 6 4 5 3 1 7 10
```

3.2 Pretraživanje u dubinu

Ovo se pretraživanje temelji na metodi pretraživanja s vraćanjem. Također, kod ove vrste pretraživanja valja paziti na izbjegavanje ciklusa. Isti predstavlja skup vrhova $v_{i1},\ldots,v_{ik}$ za koje vrijedi da su bridovi $(v_{i1}, v_{i2}),\ldots, (v_{i(k-1)}, v_{ik}), (v_{ik}, v_{i1}) \in E$, gdje se k naziva duljina ciklusa.

Kako bi se izbjeglo da algoritam završi u beskonačnoj petlji, potrebno je pamtiti vrhove kroz koje se već prođe te se ne vraćati u njih. Stoga, ovaj algoritam se može izvesti rekurzivno. Kako bi se napravio što efikasniji algoritam, pretraživanje u dubinu valja izvesti pomoću klase te izravno djelovati na samu strukturu podataka, a ne preko definiranih operacija.

Neka su v_i i v_j dva vrha grafa G. U tom slučaju se kaže da postoji put $v_{i1},\ldots, v_{ik}$ u grafu G od vrha v_i do vrha v_j ako vrijedi da je $v_{i1}= v_i$, $v_{ik}= v_j$ *i* $(v_{il}, v_{i(l+1)}) \in E$ za svaki $l=1, \ldots, k-1$. Ovdje se k naziva duljina puta.

Graf G se smatra povezanim ako za svaka dva vrha v_i i v_j postoji put od vrha v_i do vrha v_j. Ako graf nije povezan, postoje podgrafovi $G_1, \ldots, G_m \cup < G$ sa svojstvima da je svaki G_i povezan graf i da za bilo koji $i, j =1, \ldots, m$, $i \neq j$ vrijedi da ne postoji brid (v_a, v_b) takav da je $va \in G_i$ i $v_b \in G_j$. Grafovi $G_1, \ldots, G_m$ se nazivaju komponente povezanosti grafa G.

Da bi se pretražio cijeli graf, mora se pretražiti svaka njegova komponenta povezanosti. Konkretno, pretraživanje treba nastaviti sve dok skup S nepretraženih vrhova nije prazan. Kako bi pretraživanje polja S bila vremenski zahtjevna operacija, jednostavno treba prebrojiti vrhove grafa, te svaki puta taj broj smanjiti za jedan kada je neki vrh do kraja obrađen. Samo će se pretraživanje odvijati sve dok brojač ne dođe na nulu.

Složenost ove vrste pretraživanja grafa je $O(n)$.

Slijedi prikaz pseudokoda pretraživanja u dubinu:

```
DFS {
        Staviti početni čvor na stog
        Sve dok stog nije prazan {
```

Uzeti čvor sa stoga

Ako je trenutni čvor konačan, rezultat= trenutni čvor

Ako je trenutni čvor u listi posjećenih, nastaviti s idućim

Staviti sve susjedne čvorove trenutnog čvora koji nisu prethodno posjećeni na stog

}

Rezultat nije pronađen

}

Pretraživanje u dubinu ne pruža uvijek potpunost te optimalnost. Nasuprot tome, isti je pogodan kada ima puno ciljnih čvorova (a da su svi optimalni). On koristi stog (rekurzivna implementacija) te pronalazi razgranato stablo grafa (isto kao i pretraživanje u širinu).

Postavlja se pitanje: „Kada koristiti pretraživanje u dubinu, a kada pretraživanje u širinu“? Što se tiče odgovora na to pitanje, ono ovisi o samom problemu koji nas zanima, ali se smatra da je pretraživanje u dubinu pogodno koristiti za:

1. Pretraživanje stabala.
2. Grafove s gustim rješenjima.

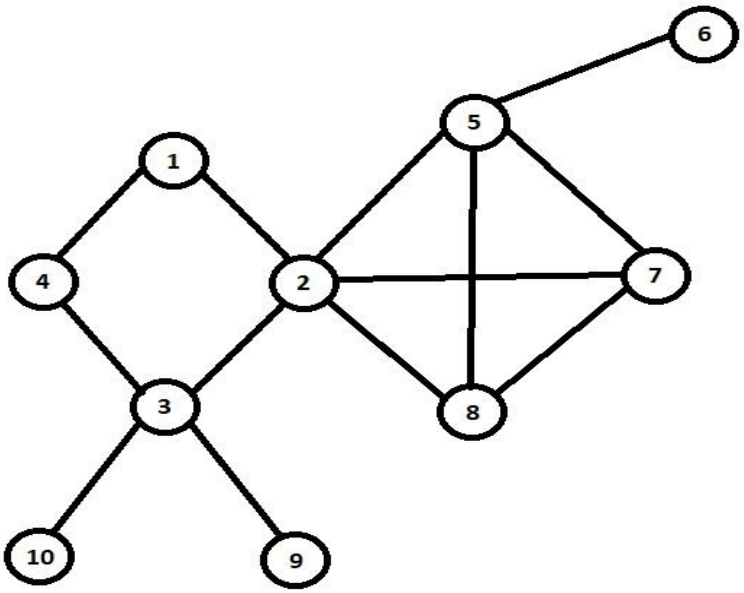
Za razliku od pretraživanja u dubinu, pretraživanje u širinu je pogodno koristiti za:

1. Beskonačne grafove.
2. Grafove s rijetkim rješenjima.
3. Traženje optimalnog rješenja (graf u granicama izračunatosti).

3.2.1 Primjer

Zadatak nalaže da je potrebno pretražiti dolje prikazan graf putem algoritma pretraživanja u dubinu.

Slika 4.: Prikaz primjera grafa pretraživanja u dubinu



Izvor: Vlastiti rad

Algoritam pretraživanja u dubinu nalaže sljedeći obilazak (počevši od broja 1): 1, 4, 3, 2, 5, 6, 7, 8, 10 i 9.

3.2.2 Implementacija u Pythonu

Ovdje je prikazana implementacija u Pythonu koja koristi strukturu podataka stog.

Prikaz rješenja:

Datoteka main.py:

```
graph = {1:[4,2],
         2:[1,3,5,7,8],
         3:[4,2,10,9],
         4:[1,3],
         5:[6,2,7,8],
         6:[5],
         7:[5,2,8],
         8:[2,5,7],
         9:[3],
         10:[3]}
```

```
def dfs(graph,start):
    path = []
    stack = [start]
    while stack != []:
        v = stack.pop()
        if v not in path:
            path.append(v)
        for w in reversed(graph[v]):
            if w not in path:
                stack.append(w)
    return path
```

Izvor: Anonymous_32, 2013.

```
if __name__ == „__main__“:

 print(„Pretrazivanje u dubinu, pocevsi od vrha 1.“)
 print(„Rezultat je:“)
 print(dfs(graph,1))
```

Opis koda:

Prvi dio koda prikazuje u obliku matrice susjedstva definirani graf (brojevi definirani u koloni sadrže vrhove, dok brojevi definirani u uglatim zagradama sadrže rubove). Funkcija *def dfs(graph,start)* sadrži sami graf te početak (prvi vrh od kojeg se počinje izvršavanje algoritma pretraživanja u dubinu). Potom je varijabla *path = [],* stavljena na početku kao prazna (matrica susjedstva), a varijabla *stack = [start]* je postavljena na početak stoga. Sve dok stog *(while stack != [])* nije prazan, izvršava se petlja, a potom se briše prvi čvor s vrha stoga. Sljedeće, ako čvor nije istražen *(if v oti n path* 😊*,* dodaje se čvor u popis označenih čvorova *(path.append(v)).* Potom se isto radi i za obrnutu situaciju *(for w in reversed(graph[v]))* i ako čvor nije istražen *(if w oti n path* 😊*.* Zaključno, vraća se dati put algoritma pretraživanja u dubinu pomoću *return path*. Nakon toga je definirana funkcija *main()* unutar koje se nalazi pozivanje funkcije *((dfs(graph, 1)))* te ispis rješenja. Prva varijabla koju prima dana funkcija je prethodno definirani graf te vrh (čvor) od kojeg počinje sami algoritam pretraživanja u dubinu.

Rješenje primjera:

```
Pretrazivanje u dubinu, pocevsi od vrha 1.
Rezultat je:
[1, 4, 3, 2, 5, 6, 7, 8, 10, 9]
```

3.2.3 Implementacija pomoću vektora

Ovdje je prikazana implementacija pomoću strukture podatka vektor u programskom jeziku C++.

Datoteka dfs.cpp:

```
#include <iostream>
#include <vector>
using namespace std;

struct Edge {
  int src, dest;
};

class Graph
{
public:
  vector<vector<int>> adjList;

  Graph(vector<Edge> const &edges, int N)
  {
    adjList.resize(N);

    for (auto &edge: edges)
    {
      adjList[edge.src].push_back(edge.dest);
      adjList[edge.dest].push_back(edge.src);
    }
  }
  };

void DFS(Graph const &graph, int v, vector<bool> &discovered)
{
  discovered[v] = true;

  cout << v << „ „;

  for (int u : graph.adjList[v])
  {
    if (!discovered[u])
      DFS(graph, u, discovered);
  }
}

int main()
{
  vector<Edge> edges = {
```

```
    {1, 4}, {1, 2}, {4, 3}, {2, 3}, {3, 10}, {3, 9},
    {2, 5}, {2, 7}, {2, 8}, {8, 5}, {8, 7}, {5, 7},
    {5, 6}
  };

  int N = 11;

  Graph graph(edges, N);

  vector<bool> discovered(N);

  for (int i = 1; i < N; i++)
    if (discovered[i] == false)
      DFS(graph, i, discovered);

  return 0;
}
```

Izvor: Anonymous_33, n_d

Opis koda:

Prvo je uključena biblioteka vector kako bi se moglo raditi sa strukturom podatka tipa vektor *(#include<vector>).* Potom je definirana struktura podatka za pohranjivanje rubova grafa (*struct Edge* koja ima dva podatka tipa *int src* i *dest*). Ta dva podatka označavaju početni/ završni vrh u grafu. Također je i definirana klasa imena *Graph* koja predstavlja graf objekt. Ista sadrži javni dio gdje se konstruira vektor vektora koji će predstavljati popis (listu) susjednosti (*vector<vector<int>> adjList*). Klasa sadrži još i konstruktor jednakog imena (*Graph(vector<Edge> const &edges, int N*)) koji definira promjenu veličine vektora na *N* elemenata tipa vektora *<int> (adjList.resize(N)).* Osim toga, klasa sadrži još i dodavanje rubova kod neusmjerenog grafa pomoću petlje *for* (*for (auto &edge: edges))* i (*adjList[edge.src].push_back(edge.dest*), *adjList[edge.dest].push_back(edge.src)).* Nakon te klase je definirana funkcija za izvođenje pretraživanja u dubinu (*void DFS(Graph const &graph, int v, vector<bool> &discovered*)) koja označava trenutni čvor otkrivenim (*discovered[v] = true*) te ga ispisuje (*cout << v << " ").* Ista sadrži još i petlju *for* (*for (int u : graph.adjList[v]))* koja za svaki rub (*v->u*), ako *u* nije otkriven (*if (!discovered[u]))* vrši pretraživanje u dubinu (*DFS(graph, u, discovered*)). Unutar funkcije *int main()* je definirana rekurzivna implementacija pretraživanja u dubinu. Ista sadrži vektor rubova grafa (*vector<Edge> edges = {…}*), broj čvorova u grafu (*int N = 11*, od *0-10*), stvaranje grafa iz danih rubova (*Graph graph(edges, N)),* pohranjivanje vrha (čvora) koji je otkriven (nije otkriven)

(*vector<bool> discovered(N)*) i pretraživanje u dubinu sa svim neotkrivenim čvorovima kako bi se obuhvatile sve nepovezane komponente grafa (petlja *for*).

Prikaz rješenja:

```
1 4 3 2 5 8 7 6 10 9
```

3.2.4 Neusmjereni grafovi

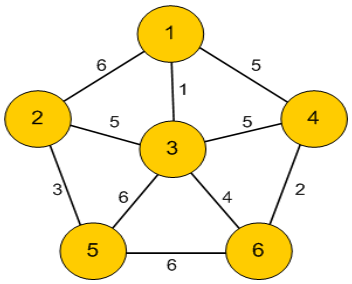
Neusmjereni matematički graf predstavlja par $G= (V, E)$, gdje je konačan V skup vrhova grafa, a $E \leq V^* V$ neuređeni skup parova iz V, koji se nazivaju bridovi grafa G.

Ako je $G= (V, E)$ neusmjereni matematički graf, tada su $v_i, v_j \in V$ i brid je $e_{ij} = \{v_i, v_j\} \in E$. U tom slučaju za vrhove v_i i v_j se kaže da su susjedni, a za brid e_{ij} se kaže da je incidentan s vrhovima v_i i v_j.

Neka je $G= (V, E)$ neusmjereni matematički graf i neka je $|V|= n$. U tom slučaju se opisuje matrica susjednosti kao matrica oblika $n*n$ te vrijedi da je $a_{ij}= 0$ za $\{v_i, v_j\} \in E$ i $a_{ij}= 1$ za $\{v_i, v_j\} \notin E$.

Neka je $G= (V, E)$ neusmjereni matematički graf i neka je $|V|= n$ i $|E|=m$. Tada se definira matrica incidencije kao matrica oblika $m*n$ za koju vrijedi sljedeće: $a_{ij}=0$, ako e_i nije incidentan vrhu v_j i $a_{ij}=1$, ako je e_i incidentan vrhu v_j.

Slika 5.: Prikaz neusmjerenog grafa



Izvor: Anonymous_1, 2014.

3.2.5 Bikonektivnost

Za graf se kaže da je bikonektivan ako je: (Jaimini, n_d)

1. Povezan (moguće je doći do svakog vrha iz svakog drugog vrha, jednostavnim putem).
2. Čak i nakon uklanjanja bilo kojeg vrha, isti ostaje povezan.

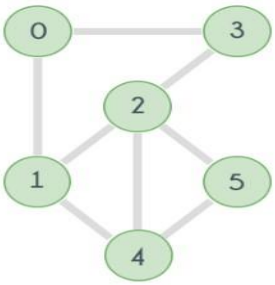
Povezani neusmjereni graf je spojen (engl. biconnected) ako nema vrhove čije uklanjanje prekida ostatak grafa. Ako su čvorišta računala, a rubovi veze, to računalo kada se spusti ne utječe na mrežnu poštu, osim naravno, što utječe na donje računalo. Slično tome, ako je sustav javnog prijevoza povezan, korisnici imaju uvijek alternativnu rutu ako bi terminal bio prekinut. (Weiss, 2014., str. 421 i 422).

Drugim riječima, za bikonektivan graf se može reći da je isti ako postoji ciklus između bilo koje dvije točke.

Može se reći da je graf *G* bikonektivan ako je povezan te ako nema točaka artikulacije ili presjeka. Da bi se riješio navedeni problem, koristi se pretraživanje u dubinu. Koristeći pretraživanje u dubinu, pokušava se pronaći postoji li točka artikulacije ili ne. Također, provjerava se da li pretraživanje u dubinu posjećuje sve točke ili ne, ako se ne može reći da graf nije povezan. (Samual, 2018.).

Preciznije, počinje se s bilo kojim vrhom i obavlja se prolaz pomoću pretraživanja u dubinu. U prijelazu pomoću pretraživanja u dubinu provjerava se je li postoji točka artikulacije. Ako se ne pronađe ni jedna točka artikulacije, onda je graf bikonektivan. Konačno, moraju se provjeriti da li su svi vrhovi dostupni kod pretraživanja u dubinu ili nisu. Ako svi vrhovi nisu bili dostupni, onda graf nije ni povezan. (geeksforgeeks_3, n_d).

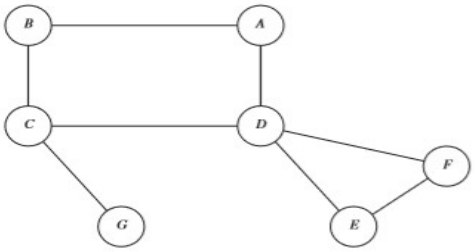
Slika 6.: Prikaz bikonektivnog grafa



Izvor: Jaimini, n_d

Taj graf je jasno povezan. Primjerice, ako se želi ukloniti bilo koji od vrhova ne povećava se broj povezanih komponenti. Iz tog razloga je prethodni graf biokonektivan.

Slika 7.: Prikaz grafa koji nije bikonektivan (graf s točkama artikulacije *C* i *D*)



Izvor: Weiss, 2014., str.422

Ako graf nije spojen, vrhovi čije uklanjanje prekida graf poznati su kao točke artikulacije. Ti su čvorovi kritični u mnogim primjenama. Tako prethodno prikazan graf nije spojen. Konkretno, *C* i *D* su točke povezivanja. Naime, kada bi se uklonio vrh *C* ono bi odspojilo *G*, a uklanjanje *D* bi odspojilo *E* i *F* od ostatka grafa.

Pretraživanje u dubinu omogućuje linearno vrijeme za pronalaženje svih točaka artikulacije u spojenom (povezanom) grafu. Prvo, počevši od bilo kojeg vrha, obavlja se pretraživanje u dubinu i obilaze se čvorovi (broje se). Za svaki vrh *v*, isti se naziva *Num (v).* Zatim, za svaki vrh *v*, kod pretraživanja u dubinu, izračuna se najniže označeni vrh, koji se naziva *Low (v)*, koji je dostupan iz *v* uzimajući nulu ili više rubova stabala te moguće do zadnjeg ruba (tim redoslijedom).

Vrh najnižeg broja koji se može dosegnuti s *A, B* i *C* je vrh *1 (A)*, jer svi mogu uzeti rubove stabla u *D*, a zatim slijedi jedan povratni rub natrag na *A*. Može se učinkovito izračunati *Low* tako što će se dobiti prijelaz dubine (prvo razgranato stablo). Po definiciji *Low (v)* je minimum od:

1. *Num (v).*
2. Najniži *Num (v)* među svim stražnjim rubovima *(v, w).*
3. Najniži *Low (w)* među svim rubovima stabla *(v, w).*

Prvi uvjet je mogućnost neuzimanja rubova, drugi način je da se ne odabere rub stabla i stražnji rub, dok je treći da se odaberu neki rubovi stabla i možda stražnji rub. Ova treća metoda je sažeto opisana putem rekurzivnog poziva. Budući da se treba ocijeniti *Low* za svu djecu *v* prije nego što se može procijeniti *Low (v),* isti obuhvaća obilazak. Za bilo koji rub *(v, w),* može se odrediti je li to rub stabla ili stražnji rub samo provjerom *Num (v)* i *Num (w).* Stoga je lako izračunati *Low (v).* Stoga, valja samo skenirati popis *v* (susjedstvo), primijeniti ispravno pravilo te pratiti minimum. Ako je sve što se radi dobro provedeno, potrebno je vrijeme *O (|E|+|V|).*

Sve što preostaje je iskoristiti ove informacije za pronalaženje artikulacijskih točaka. Korijen je točka artikulacije ako i samo ako ima više od jednog djeteta, jer ako ima dvoje djece, uklanjanje korijena raskida čvorove u različitim podređenicama. U slučaju da ima jedno dijete, uklanjanje korijena samo isključuje korijen. Bilo koji vrh *v* je točka artikulacije ako i samo ako *v* ima neko dijete *w* takvo da je *Low (w)* ≥ *Num (w).* Također, valja primijetiti da je ovo stanje uvijek zadovoljeno u korijenu, zbog čega postoji potreba za posebnim testom.

Ako je dio dokaza jasan kada se ispituju artikulacijske točke koje algoritam određuje, onda se zapravo određuju točke *C* i *D*. *D* ima dijete *E*, a *Low (E)* ≥ *Num (D)*. Dakle, postoji samo jedan način da *E* dođe do bilo kojeg čvora iznad *D*, a to je prolazak kroz *D*. Slično tome, *C* je točka artikulacije, jer *Low (G)* ≥ *Num (C).* Da bi se dokazalo da je ovaj algoritam točan, mora se dokazati da je dio tvrdnje istinit (takve točke se nazivaju točke izražavanja).

Slijedi algoritam bikonektivnog grafa. Postoji funkcija *isArticulation (start, visited, disc, low, parent).* Kao ulaz se uzima početni vrh (*start*), posjećeni niz (*visited*) koji služi za označavanje kada je neki čvor posjećen. Disk (*disc*) će zadržati vrijeme otkrivanja vrha. Nizak (*low*) čvor će zadržavati informacije o podređenima. Nadređeni (*parent*) će držati

roditelja trenutačnog čvora. Kod izlaza se javlja istina ako se pronađe bilo koja točka artikulacije.

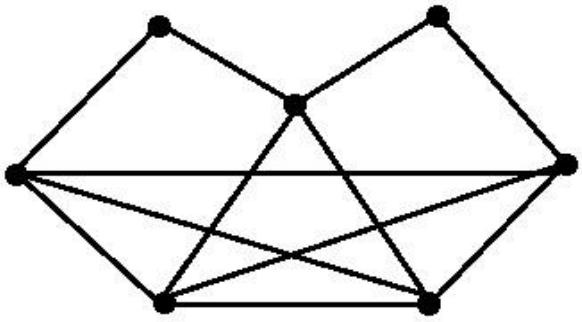
Druga funkcija koja postoji je *isBiconnected (graph).* Kao ulaz ista uzima zadani graf, dok se kao izlaz javlja istina ako je graf dvostruko povezan.

3.2.6 Eulerovi grafovi

Pojam ciklusa $v_{i1}, \ldots, v_{ik}$ u grafu $G=(V, E)$ je Eulerov ciklus ako vrijedi da za svaki brid $e \in E$ postoji $j \in \{1,\ldots, k-1\}$ takav da je $e=\{v_{ij}, v_{i(j+1)}\}$. Odnosno, graf G je Eulerov graf ako u njemu postoji Eulerov ciklus.

Teorem glede Eulera: Graf G je Eulerov graf ako i samo ako je G povezan te ako su svi vrhovi grafa G parnog stupnja. Neformalno, koliko ulaza ima, toliko izlaza ima u svaki vrh, te to sve skupa treba činiti paran broj. Ovo se smatra začetkom teorije grafova.

Slika 8.: Prikaz primjera Eulerovog grafa



Izvor: Anonymous_2, n_d

Primjerice, Fleuryjev algoritam:

1. Provjerava da li je graf Eulerov. Ako nije, mora se zaustaviti.
2. Stavlja se $T[0]=0$, v= bilo koji vrh grafa, $C=v$.
3. Ponavljanje toga sve dok graf G ne postane *null* graf: ako postoji brid koji je incidentan vrhu v koji nije most treba se uzeti on, a ako ne postoji treba se uzeti bilo koji most. Neka je uzet brid (v, u). $C= C||u$. $E=E\setminus\{u, v\}$.

Fleuryjev algoritam konstruira Eulerovu turu tako da počinje u nekom vrhu i u svakom koraku bira vezni brid neprijeđenog podgrafa samo da nema alternative.

Eulerova tura je zatvorena Eulerova staza. (Jurašić, n_d, str.1).

Brid *e* povezanog grafa *G= (V, E)* je most ako graf *G'=(V, E\ {e})* nije povezan. Treba provjeriti je li graf povezan. Uzme se opisani algoritam pretraživanja u dubinu (ne gleda da li je pretraživanjem jedne komponente pretražen cijeli graf). Posebice, nakon završetka pretraživanja valja provjeriti je li broj pretraženih vrhova jednak |*V*|. U slučaju da je, onda je graf povezan, inače nije.

Formalno, treba provjeriti da li je neki brid most. To se provjerava tako da se može obrisati dani brid i provjeriti da li je graf još uvijek povezan.

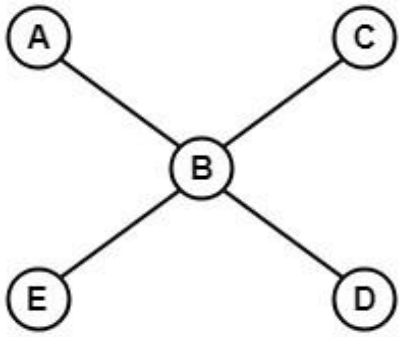
Postupak uzimanja mosta:

1. Ako se time dobiva više komponenti povezanosti (sastoje se od dva ili više vrha), onda se nikako neće moći proći jedna od njih, iz razloga jer među njima nema bridova.
2. Konkretno, most se uzima samo u slučaju ako je jedan od vrhova incidentnih s bridom postao stupnja 0. Izuzetak je posljednji korak vrha *u* koji ne smije biti stupnja 0, jer inače se iz njega ne može ići dalje.

Problem koji nastaje je u tome što će se dogoditi ako je jedan vrh grafa u potpunosti riješen jer tada on neće više biti povezan s ostalim vrhovima. Ono što je potrebno napraviti je brisanje završenih čvorova. Pod tim se misli na čvor iz kojeg se izlazi da mu se time obriše posljednji brid koji treba obrisati iz grafa.

Složenost operacija je: *IsConnected je O(n), IsEulerian O(n)* i *EulerCycle O(n^3).*

Slika 9.: Prikaz grafa koji nije Eulerov



Izvor: Singhal, 2018.

3.2.7 Hamiltonovi grafovi

Ovdje se događa Hamiltonov ciklus koji je jednostavan ciklus u grafu kroz sve vrhove grafa. Također, ovaj ciklus u grafu je jednostavan ako prolazi kroz svaki vrh najviše jednom.

Usmjereni Hamiltonov put u digrafu (usmjereni graf) *D* sadrži usmjereni put (diput) koji ima sve vrhove iz *D* (teorem o turnirima).

Međutim, za problem nalaženja Hamiltonovog ciklusa ne postoji efikasno rješenje, već se treba raditi pretraživanje s vraćanjem, tj. treba primijeniti metodu grananja i ograničenja.

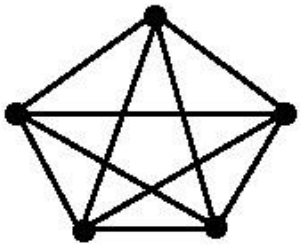
Što se tiče Hamiltonovog algoritma, ono se temelji na pretraživanju s vraćanjem koji kreće od proizvoljnog vrha i bira redom susjedne vrhove. Ako se dođe u situaciju da se više ne može ići dalje, a da se nisu obišli svi vrhovi, onda se treba vratiti u prethodno posjećeni vrh.

Kao ulaz se uzima graf $G=(V,E)$, $|V|=n$, dok se za izlaz uzima niz $v_1, \ldots, v_n$ takav da je $(v_i, v_{i+1}) \in E$, ako Hamiltonov ciklus grafa postoji, inače prazno.

Ovdje se može navesti primjer problema trgovačkog putnika. Točnije, trgovački putnik treba obići neke gradove te se vratiti na polazno (početno) mjesto. Tijekom tog putovanja treba proći samo jednom kroz svaki grad, odnosno svakom cestom najviše

jednom. U vidu se ovdje igra igra “put oko svijeta” na grafu dodekaedra. Pretpostavka je da su gradovi vrhovi dodekaedra, a bridovi ceste između gradova. Dodekaedar predstavlja pravilni poliedar u čijem se svakom vrhu sastaju tri pravilna peterokuta (20 vrhova, 30 bridova te 12 peterokuta).

Slika 10.: Prikaz primjera Hamiltonovog grafa



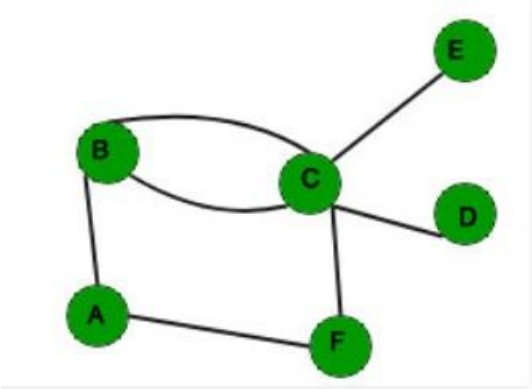
Izvor: Anonymous_3, n_d

Ovaj algoritam se temelji na metodi pretraživanja s vraćanjem. Odnosno, ovdje se kreće od proizvoljnog vrha te se prolazi kroz svaki od njih najviše jednom.

Problem nalaženja Hamiltonovih ciklusa je teži od problema Eulerove staze. Valja zaključiti da Eulerovi i Hamiltonovi grafovi nemaju direktne veze jedni s drugima. Jedan od najvećih problema koji nisu rješivi kod teorije grafova je dati netrivijalni te zadovoljavajuće nužan i dovoljan uvjet da graf bude Hamiltonov. Točnije, problem određivanja se krije u tome da je Hamiltonov graf zapravo *NP*- potpun.

Da bi se ustanovilo ima li neki graf Hamiltonov ciklus ili Hamiltonov put, tada se pretežito treba primijeniti neka ad- hoc metoda.

Slika 11.: Prikaz primjera grafa koji nije Hamiltonov



Izvor: geeksforgeeks_4, n_d

3.3 Usmjereni grafovi

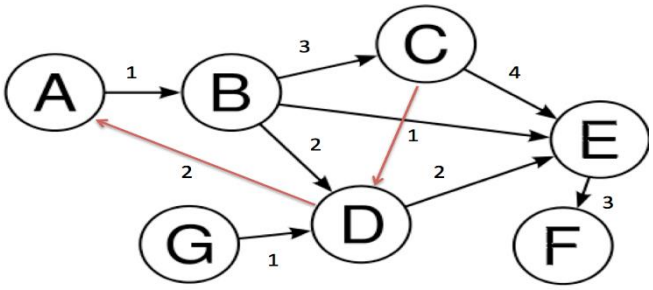
Usmjereni (ili neteţinski) matematički graf je par *G= (V, E)*, gdje je konačan *V* skup vrhova grafa, a *E ≤ V * V* skup uređenih parova iz *V*, koji se nazivaju bridovi grafa *G*.

Konkretno, kod usmjerenih grafova bridovi su, u odnosu na neusmjereni graf, usmjereni pa se prikazuju kao uređeni parovi vrhova. U slučaju da je *(u, v)* brid, onda se kaže da brid izlazi iz vrha *u* i ulazi u vrh *v*.

Također, kod usmjerenog matematičkog grafa se mogu koristiti iste strukture podataka kao i kod neusmjerenog. Zapravo, pretežito strukture podataka modeliraju usmjereni graf, samo što se uz svaki brid *(u, v)* uvijek u strukturu podataka mora dodati i brid *(v, u).*

Metode pretraživanja u dubinu i pretraživanja u širinu se mogu primijeniti i na usmjerene grafove, koristeći pritom bridove za prijelaz iz vrha u vrh. Valja naglasiti da, krene li se iz proizvoljnog vrha kada algoritam ne može više dalje, a da pri tome nije pregledao sve vrhove grafa, nije nužno da se mora eliminirati cijela jedna komponenta povezanosti.

Slika 12.: Prikaz usmjerenog grafa

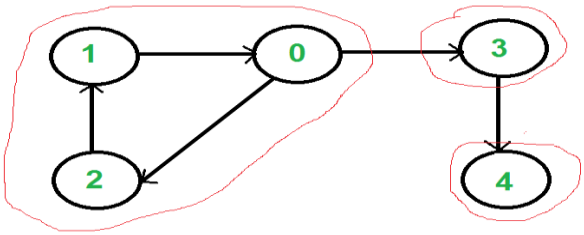


Izvor: Anonymous_4, n_d

3.3.1 Pronalaženje jakih komponenti

Usmjereni graf je snažno (jako) povezan ako postoji put između svih parova točaka. Preciznije, ako postoji put od svakog vrha u grafu do svakog drugog vrha. Jako povezana komponenta (engl. strongly connected component) usmjerenog grafa je maksimalno snažan povezan podgraf.

Slika 13.: Prikaz grafa s tri jako povezane komponente



Izvor: geeksforgeeks_5, n_d

Sve jako povezane komponente se izvršavaju u vremenu *O (V+E)* koji se izvršava pomoću algoritma Kosaraju.

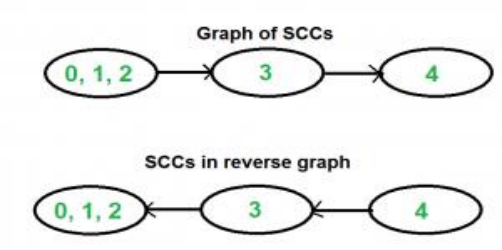
Slijedi detaljan algoritam Kosarajua: (geeksforgeeks_5, n_d).

1. Treba napraviti prazan stog *"S"* i izvršiti provođenje pretraživanja u dubinu na grafu. U prijelazu kod pretraživanja u dubinu, nakon pozivanja rekurzivnog za susjedne vrhove vrha, stavlja se vrh u stog (primjerice, u gornjem grafu, ako se započne pretraživanje u dubinu od vrha 0, dobivaju se točke u stogu 1,2,4,3 i 0).
2. Ako se koristi obrnuti smjer lukova, onda se dobiva transponirani graf.
3. Briše se jedan po jedan vrh iz *S*, sve dok *S* nije prazan. Primjerice, ako se uzme vrh *v* kao izvor, onda se radi pretraživanje u dubinu (poziva se funkcija). Pretraživanje u dubinu polazi od *v* te ispisuje jako povezanu komponentu *v*. Tako bi u gore prikazanom grafu se obrađivali vrhovi redom 0, 3, 4, 2, 1 (jedan po jedan se izbacuju iz stoga).

Prethodno definirani algoritam se temelji na pretraživanju u dubinu (pretraživanje u dubinu se radi dva puta). Isti proizvodi jedno stablo ako su svi vrhovi dostupni iz početne točke pretraživanja u dubinu. U protivnom, pretraživanje u dubinu stvara šumu. Tako graf sa samo jednom jako povezanom komponentom uvijek proizvodi stablo. Valja naglasiti da pretraživanje u dubinu može proizvesti stablo ili šumu kada postoji više od jedne jako povezane komponente, ovisno o odabranoj početnoj točki. Primjerice, u gornjem grafu, ako se pokrene pretraživanje u dubinu iz točaka 0, 1 ili 2, dobiva se stablo kao izlaz. Dok u protivnom (ako se krene od točke 3 ili 4), dobiva se šuma. Da bi se pronašle i ispisale sve jako povezane komponente, valja pokrenuti

pretraživanje u dubinu od vrha 4, zatim se pomaknuti na 3 (skup bez 4), i na kraju bilo koji od preostalih vrhova (0, 1, 2). Nažalost, ne postoji izravan način za dobivanje prethodno napisanog slijeda. Međutim, ako se radi pretraživanje u dubinu i pohrane se vrhovi u skladu s njihovim vremenom završetka, onda će vrijeme završetka vrha koji se povezuje s drugim jako povezanim komponentama (drugi će onda biti s njegovom vlastitom jako povezanom komponentom) uvijek biti veće od vremena završetka vrhova u drugoj jako povezanoj komponenti. Primjerice, u gornjem grafu, vrijeme završetka od vrha 0 je uvijek veće od 3 i 4 (bez obzira na redoslijed točaka koji se uzimaju u obzir za pretraživanje u dubinu). Vrijeme završetka od vrha 3 je uvijek veće od vrha 4. Pretraživanje u dubinu ne jamči za druge vrhove isto, primjerice vrijeme završetka od 1 i 2 može biti manje ili veće od 3 i 4, ovisno o redoslijedu vrhova koji se uzimaju u obzir za pretraživanje u dubinu. Dakle, da bi se koristilo ovo svojstvo, radi se zaokruživanje cijelog grafa i gura se svaki završeni vrh na stog. U stogu se 3 uvijek pojavljuje nakon 4, a 0 se pojavljuje nakon oba (nakon 3 i 4). U sljedećem se koraku preokrene graf. Ovdje valja razmotriti graf jako povezanih komponenti. U preokrenutom grafu rubovi koji povezuju dvije komponente su preokrenute. Tako jako povezana komponenta {0, 1, 2} postaje “ponor”, dok jako povezana komponenta {4} postaje “izvor”. Kao što je gore rečeno, u stog, uvijek je prvo 0 prije 3 i 4. Preciznije, ako se radi pretraživanje u dubinu preokrenutog grafa koristeći slijed redova u stog, obrađuju se vrhovi od ponora do izvora (kod preokrenutog grafa). To je zapravo ono što se želi postići, tj. to je sve potrebno za ispis jedne po jedne jako povezane komponente.

Slika 14.: Prikaz nastavka prethodnog primjera (jako povezane komponente)



Izvor: geeksforgeeks_5, n_d

Vremenska složenost glede gornjeg algoritma poziva pretraživanje u dubinu, potom se ista poziva preokrenuto. Tada je složenost pretraživanja u dubinu *O (V+E).* To se odnosi na graf koji je predstavljen pomoću liste susjedstva. Preokretanje grafa također traje *O (V+E).* Za preokretanje grafa, jednostavno se prelaze sve liste susjedstva.

Jako povezane komponente (algoritmi) mogu se koristiti kao prvi korak kod mnogih grafičkih algoritama koji rade samo na jako povezanom grafu. U društvenim mrežama, skupina ljudi je općenito jako povezana (primjerice, učenici razreda ili bilo kojeg drugog zajedničkog mjesta). Mnogi ljudi u tim skupinama uglavnom vole neke zajedničke stranice ili igraju zajedno igre. Jako povezane komponente se mogu koristiti za pronalaženje takvih grupa i sugerirati stranice ili igre koje se najčešće sviđaju ljudima u grupi koji vole tu određenu stranicu ili vole igrati određenu igru.

Gornji algoritam je asimptotički najbolji algoritam, ali postoje i drugi algoritmi kao što je Tarjanov algoritam koji se temelji na istoj vremenskoj složenosti, ali pronalazi jako povezane komponente koje koriste samo jedno pretraživanje u dubinu.

Tarjan algoritam se temelji na sljedećim činjenicama: (geeksforgeeks_6, n_d).

1. Pretraživanje u dubinu proizvodi samo jedno stablo ili šumu.
2. Jako povezane komponente oblikuju podstavke stabla kod pretraživanja u dubinu.
3. Ako se može pronaći glavni od podstabala, mogu se ispisati ili pohraniti svi čvorovi u tom podstablu (uključujući i glavni) koji će biti jedna jako povezana komponenta.
4. Nema povratnog ruba od jedne jako povezane komponente do druge (mogu biti križni rubovi, ali križni rubovi neće se koristiti tijekom obrade grafa). Da bi se pronašla “glava” (glavni) jako povezana komponenta, izračuna se niz (kao što je isto učinjeno za artikulacijsku točku, most i biokonektivnu komponentu). Konkretno, *low [u]* označava najranije posjećeni vrh (vrh s minimalnim vremenom otkrivanja) koji se može postići iz podstabla ukorijenjenog s *u*. Čvor *u* je glava ako je *disc [u]=low [u].*

Mala (engl. low) vrijednost *u*, može promijeniti dva slučaja:

1. Prvi slučaj (engl. Tree Edge): Ako se čvor *v* već ne posjećuje, nakon što je pretraživanje u dubinu kod *v* dovršen, tada će se minimalna vrijednost *[u]* i *low [v]* ažurirati na nisku vrijednost *[u].*
2. Drugi slučaj (engl. Back Edge): Kada je dijete *v* već posjećeno, tada će minimalna vrijednost *[u]* i *disc [v]* biti ažurirana na nisku vrijednost *[u].*

Vremenska složenost gornjeg algoritma koji poziva pretraživanje u dubinu je *O (V+E).* To se odnosi na graf koji koristi susjede.

3.4 Topološki poredak

Kod topološkog poretka, zadan je usmjereni neciklički graf. Ovdje je potrebno vrhove grafa poredati u niz tako da se ni jedan vrh u grafu u kojeg ulazi brid iz drugog vrha ne nalazi u nizu prije vrha iz kojeg brid izlazi.

Kao ulaz ovdje se nalazi usmjereni aciklički graf *G= (V, E),* dok se kao izlaz javlja niz $[v_{i1}, \ldots, v_{in}]$ tako da za svaki $j, k= 1, \ldots, n, j<k$ u grafu ne postoji brid (v_{ij}, v_{ik}).

Naposljetku, kada se govori o ciklusima u usmjerenom grafu, zapravo se misli na usmjerene cikluse. Preciznije, usmjereni graf tada može biti acikličan, iako u neusmjerenom grafu s istim vrhovima i bridovima postoji ciklus.

Također, ovdje valja reći da je jasno da u slučaju da usmjereni graf sadrži usmjereni ciklus topološko sortiranje, odnosno topološki poredak ne postoji.

Kod samog pretraživanja u dubinu, važno je da on kreće od određenog vrha te ide u neko njegovo dijete, slijedeći put sve dok ne dođe do vrha iz kojeg više ne može ići nikamo, te na taj način isti provodi klasično pretraživanje s vraćanjem. Gledajući na to kao na usmjereni aciklički graf to znači da iz tog vrha nema izlaznih bridova.

Neka je zadana propozicija: *G= (V, E)* kao usmjereni aciklički graf. Ako je $v_{i1}, \ldots, v_{in}$ niz koji vraća pretraživanje u dubinu, onda je $v_{in}, \ldots, v_{i1}$ topološki sort.

Usmjereni težinski matematički graf je par *G= (V, E),* gdje je konačan *V* skup vrhova grafa, a $E \leq V^* V * R$ skup uređenih parova iz *V*, koji se nazivaju bridovi grafa *G*.

Konkretno, pri tome je svakom bridu e_i dodana realna vrijednost w_i, koja se naziva težina brida e_i.

Što se tiče samih implementacija usmjerenog težinskog grafa, iste se izvode iz implementacija neusmjerenog težinskog grafa tako da se dozvoli da matrica susjednosti bude prezentirana dvodimenzionalnim poljem ili listama (nesimetrična).

4. Povezivanje

U mnogim problemima, vlada zainteresiranost za pronalaženje putanje u grafu od jednog vrha do bilo kojeg drugog vrha. Za neusmjerene grafove to znači da ne postoje odvojeni dijelovi, dok za usmjerene (digrafe), to znači da na grafu postoje mjesta na koja se može doći iz nekih smjerova, ali se ne moraju nužno vratiti na početne točke.

4.1 Povezivanje u neusmjerenim grafovima

Neusmjeren graf je povezan kada postoji put između bilo koje dvije točke grafa. Algoritam za pretraživanje u dubinu može se koristiti za prepoznavanje da li je graf povezan (pod uvjetom da se radi o petlji). Konkretno, dok postoji vrh *v* takav da je *num (v) ==0*, tj. dok je isti uklonjen. Potom, kada je algoritam završen, moraju se provjeriti rubovi popisa (liste) na način da se vidi da li isti uključuju sve vrhove grafa. Preciznije, jednostavno se treba provjeriti je li parametar *i* jednak broju vrhova.

Povezivanje dolazi u stupnjevima (graf može biti manje povezan i ovisi o broju različitih puteva između bilo koja dva vrha). Točnije, graf se naziva *n*- spojen ako postoji barem *n* različitih puteva između bilo koja dva vrha (postoji *n* putanja između bilo koje dvije točke koje nemaju vrhove je uobičajena). Nadalje, postoji poseban tip grafa, a to je *2*-vezan (dvosmjerni graf) za kojeg postoje barem dva različita puta između bilo koja dva vrha. Konkretno, graf se ne povezuje ako se može pronaći vrh koji uvijek mora biti uključen u putanju između najmanje dva vrha (*a* i *b*). Drugim riječima, ako je isti vrh uklonjen iz grafa (zajedno s incidentnim rubovima), onda ne postoji način da se pronađe put od *a* do *b*, što znači da je graf podijeljen na dva odvojena podgrafa. Takvi vrhovi se nazivaju točkama artikulacije (izrezani vrhovi) od točke *a* do točke *b*.

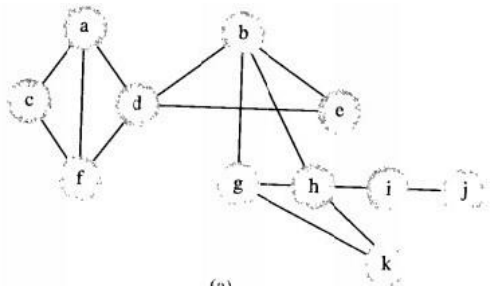
Ako rub uzrokuje da se graf podijeli na dva podgrafa, isti se naziva mostom.

Podgrafi koji proizlaze iz uklanjanja točke artikulacije ili mosta se nazivaju blokovi ili bikonektirane komponente. Naime, važno je znati kako se graf dekomponira na bikonektirane komponente.

Artikulacijske točke se mogu otkriti proširivanjem pomoću algoritma za pretraživanje u dubinu. Ovaj algoritam stvara stablo s prednjim rubovima (rubovi grafa su uključeni u stablo) i stražnjim rubovima (rub nije uključen). Vrh *v* u tom stablu je artikulacijska točka ako ima najmanje jedno podstablo koje nije povezano s bilo kojim od svojih prethodnika na stražnjem rubu. Iz razloga jer je to stablo, nijedno od njegovih prethodnika nije dostupno od bilo kojeg njegovog nasljednika putem prednje veze.

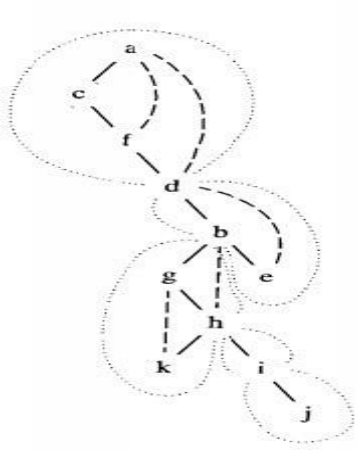
Slika 15.: Prikaz grafa koji se pretvara u stablo pomoću algoritma za pretraživanje u dubinu (Slika 16.). Dato stablo ima četiri točke artikulacije *b, d, h i i,* budući da nema nikakvog zadnjeg ruba od bilo kojeg čvora ispod *d* na bilo koji čvor koji je iznad njega u stablu i bez zadnjeg ruba od bilo kojeg vrha u desnom podstablu od *h* do bilo kojeg vrha iznad *h*. Također, vrh *g* ne može biti točka artikulacije jer je njegov nasljednik *h* povezan s vrhom iznad njega. Četiri vrha dijele graf na pet blokova točkastim linijama (Slika 16.).

Slika 15.: Prikaz grafa



Izvor: Drozdek, 2001., str. 398

Slika 16.: Prikaz prethodnog grafa koji se pretvara u stablo pomoću algoritma za pretraživanje u dubinu

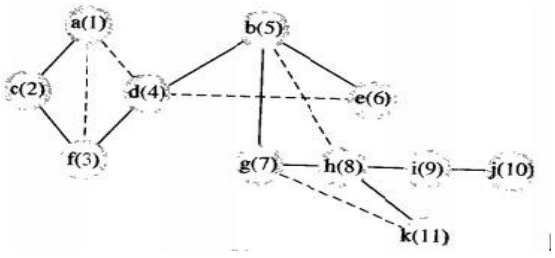


Izvor: Drozdek, 2001., str. 398

Poseban slučaj za točku artikulacije je kada je vrh zapravo korijen s više od jednog potomka. Na slici 17. je stoga odabran vrh kao korijen (*a*), te ima tri incidentna ruba, ali samo jedan od njih postaje prednji rub, iz razloga jer su druga dva obrađena pretraživanjem u dubinu. Stoga, ako ovaj algoritam ponovno rekurzivno dosegne *a*, neće postojati neisprobani rub. Ako je *a* točka artikulacije, postojat će barem jedan takav neisprobani rub, a to ukazuje da je *a rez*. Dakle, to nije točka artikulacije. Točnije, kaže se da je vrh artikulacijska točka: (Drozdek, 2001., str. 399)

1. Ako je *v* korijen stabla za traženje dubine te ako *v* ima više od potomka u tom stablu.
2. Ako najmanji (jedan od) podstabla *v* ne uključuje vrh koji je povezan sa stražnjim rubom s bilo kojim prethodnim od *v*.

Slika 17.: Prikaz odabranog vrha kao korijena (*a*)



Izvor: Drozdek, 2001., str. 398

Da bi se pronašle točke artikulacije, koristi se parametar *pred(v)*, koji je definiran kao *min (num(v), num(v_1),…, num(v_k))*, gdje su v_1,…, v_k vrhovi povezani s rubom (potomkom) *v* ili sa samim *v*. Naime, što je veći *v* (prethodnik), to je niži njegov broj (birajući minimalni broj se odabire najviši prethodnik). Za prethodno prikazanu sliku stoga vrijedi: *pred (c) =pred (d)=1, pred ()=4, i pred (k)=7.*

Algoritam koristi stog za pohranu svih trenutno obrađenih rubova. Nakon što je identificirana točka artikulacije, ispisuju se rubovi koji odgovaraju bloku grafa.

Slika 18.: Prikaz algoritma koji koristi stog za pohranu svih trenutno obrađenih rubova

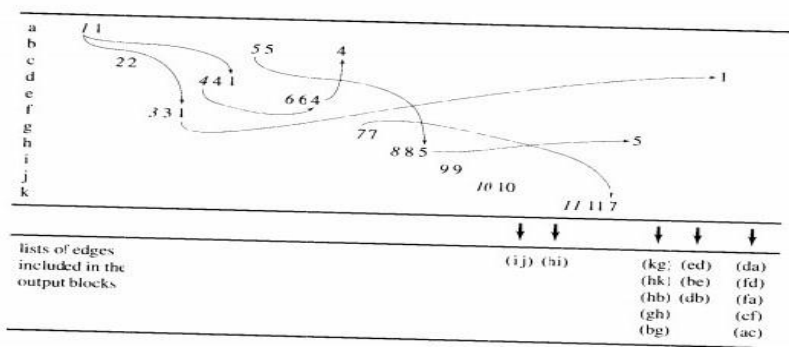
```
blockDFS(v)
   pred(v) = num(v) = i++;
   for all vertices u adjacent to v
      if edge(uv) has not been processed
         push(edge(uv));
      if num(u) is 0
         blockDFS(u);
         if pred(u) ≥ num(v)                  // if there is no edge from u to a
            e = pop();                        // vertex above v, output a block
            while e ≠ edge(vu)                // by popping all edges off the
               output e;                      // stack until edge(vu) is
               e = pop();                     // popped off;
            output e;                         // e == edge(vu);
         else pred(v) = min(pred(v),pred(u)); // take a predecessor higher up in
      else if u is not the parent of v        // tree;
         pred(v) = min(pred(v),num(u));       // update when back edge(vu) is
                                              // found;
blockSearch()
   for all vertices v
      num(v) = 0;
   i = 1;
   while there is a vertex v such that num(v) == 0
      blockDFS(v);
```

Izvor: Drozdek, 2001., str. 399

Primjer izvođenja prethodno prikazanog algoritma se primjenjuje kod grafa koji je bio prikazan na slici 15. U tablici su navedene sve promjene *(pred(v))* za vrhove *v* koji su obrađeni algoritmom, dok strelice pokazuju izvor novih vrijednosti *(pred(v)).* Konkretno, za svaki vrh *v*, *blockDFS(v)* prvo dodjeljuje dva broja: *num(v),* prikazan kurzivom, i *pred(v)*, koji se mogu promijeniti tijekom izvođenja funkcije *blockDFS(v).* Primjerice, *a* se prvo obrađuje s *num(a) i pred(a)*, koji se postavljaju na jedan. Rub *(ac)* se gura u stog, a budući da je *num(c)* nula, algoritam se poziva za *c*. U tom trenutku, *num(c)* i *pred(c)* su postavljeni na dva. Pored toga, algoritam je pozvan za *f* (potomak *c*), tako da su *num(f)* i *pred(f)* postavljeni na tri. Zatim se isti poziva za *a*, koji je potomak od *f*. Budući da *num(a)* nije nula, a *a* nije roditelj od *f*, *pred(f)* je postavljen na *1=min (pred(f), num(a))=min (3, 1).*

Ovaj algoritam se također izrađuje za rubove kod otkrivenih blokova, a ti su rubovi prikazani na idućoj slici (u trenutku kada su isti izašli iz stoga).

Slika 19.: Prikaz rubova kod otkrivenih blokova



Izvor: Drozdek, 2001., str. 398

4.2 Povezivanje u usmjerenim grafovima

Za usmjerene grafove, povezanost se može definirati na dva načina, ovisno o tome uzima li se u obzir smjer rubova. Točnije, usmjereni graf je slabo povezan ako je povezan neusmjereni graf s istim vrhovima i istim rubovima. Za razliku od slabo povezanog grafa, usmjereni graf je snažno (jako) povezan ako za svaki par vrhova postoji put između njih u oba smjera. Cijeli digraf nije uvijek snažno povezan, ali može biti sastavljen od snažno povezanih komponenti, koje su definirane kao podskupovi vrhova grafa tako da svaki od tih podskupova inducira jako povezan graf.

Određivanje jako povezane komponente, također upućuje na algoritam pretraživanja u dubinu. Pretpostavlja se da je vrh *v* prvi vrh jako povezane komponente za koji se primjenjuje prvo pretraživanje u dubinu. Takav se vrh naziva korijen jako povezane *w* komponente. Naime, svaki vrh u toj jako povezanoj komponenti može se dostići iz *v (num(v)< num(u)),* i tek nakon što su svi takvi vrhovi *u* posjećeni, pretraživanje u dubinu prvo se vraća u *v*. U tom slučaju, koji prepoznaje činjenicu da je *pred(v)= num(v),* jako povezana komponenta je dostupna iz korijena koji može predstavljati izlaz. (Drozdek, 2001., str. 400).

Sada se javlja problem kako pronaći sve takve korijene usmjerenog grafa, što je analogno pronalaženju artikulacijskih točaka u neusmjerenom grafu. U tu svrhu se također koristi parametar *pred(v),* gdje je *pred(v)* donji broj izabran iz *num(v)* i *pred(u),* gdje je *u* vrh dostignut iz *v* te pripada istoj jako povezanoj komponenti, kao *v*. Nadalje, postavlja se pitanje: „Kako se može utvrditi pripadaju li dva vrha istoj jako povezanoj

komponenti prije određivanja same jako povezane komponente“? Prividna kružnost je stoga riješena pomoću stoga koji pohranjuje sve vrhove koji pripadaju jako povezanoj komponenti koji su u izgradnji. Konkretno, najviši vrhovi na stogu pripadaju trenutno analiziranoj jako povezanoj komponenti. Iako konstrukcija nije dovršena, barem se zna koji su vrhovi već uključeni u jako povezanu komponentu.

Slika 20.: Prikaz Tarjan algoritma (pripisuje prethodno navedeno)

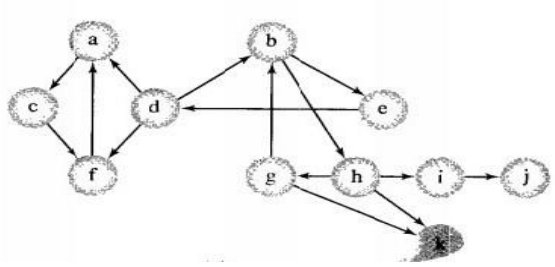
```
strongDFS(v)
  pred(v) = num(v) = i++;
  push(v);
  for all vertices u adjacent to v
    if num(u) is 0
      strongDFS(u);
      pred(v) = min(pred(v),pred(u));          // take a predecessor higher up in
    else if num(u) < num(v) and u is on stack  // tree; update if back edge found
      pred(v) = min(pred(v),num(u));           // to vertex u is in the same SCC;
  if pred(v) == num(v)                         // if the root of a SCC is found,
    w = pop();                                 // output this SCC, i.e.,
    while w != v                               // pop all vertices off the stack
      output w;                                // until v is popped off;
      w = pop();
    output w;                                  // w == v;
```

```
stronglyConnectedComponentSearch()
  for all vertices v
    num(v) = 0;
  i = 1;
  while there is a vertex v such that num(v) == 0
    strongDFS(v);
```

Izvor: Drozdek, 2001., str. 400

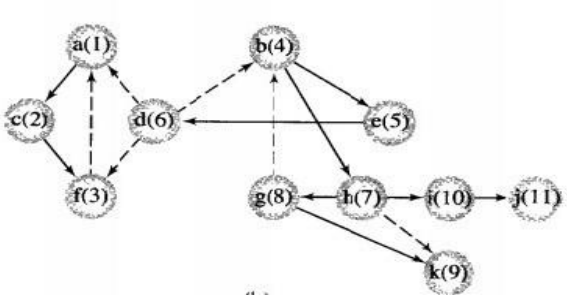
Sljedeći primjer prikazuje izvršenje Tarjanova algoritma. Konkretno, usmjereni graf na idućoj slici (slika 21.) obrađuje niz poziva funkcije *strongDFS(),* koji dodjeljuje vrhove *a*, dok su *k* brojevi prikazani u zagradama kao na slici 22.

Slika 21.: Prikaz usmjerenog grafa koji obrađuje niz poziva funkcije *strongDFS()*



Izvor: Drozdek, 2001., str. 401

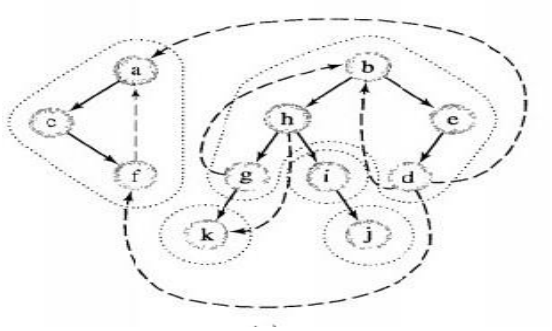
Slika 22.: Prikaz *k* brojeva koji su prikazani u zagradama



Izvor: Drozdek, 2001., str. 401

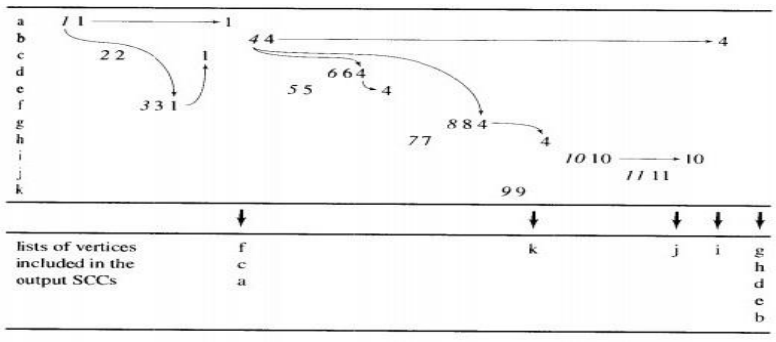
Tijekom tog procesa, otkriveno je pet jako povezanih komponenti: *{a, c, f}, {b, d, e, g, h}, {i}, {j} i {k}*. Slika 23. sadrži stablo koje sadrži pretraživanje u dubinu koje je stvorio taj proces. Valja imati na umu da su dva stabla stvorena tako da broj stabala ne mora odgovarati broju jako povezanih komponenti, budući da broj stabala ne odgovara broju blokova u slučaju neusmjerenih grafova.

Slika 23.: Prikaz stabla koji sadrži pretraživanje u dubinu



Izvor: Drozdek, 2001., str. 401

Slika 24.: Prikaz, tj. označavanje kurzivom brojeva dodijeljenim *num(v)* i sve promjene parametara *pred(v)* za sve vrhove *v* na grafu.



Izvor: Drozdek, 2001., str. 401

Također, ova slika prikazuje izlaz jako povezane komponente tijekom obrade grafa.

5. Uvod u NP potpunost

The Clay Mathematics Institute of Cambridge, Massachusetts (CMI) odabrao je sedam važnih, a dugo vremena neriješenih problema, ponudivši nagradu po milijun dolara za rješenje svakog od njih. Ti se problemi zovu Milenijski. (Šego, 2009., str.1).

P=NP predstavlja problem iz teorije algoritama, a koji znači "postoje li, za naizgled složene probleme, jednostavni algoritmi koji ih rješavaju?".

Pitanje da li je *P=NP* za sada nema odgovora, ali se mogu identificirati problemi u *NP* koji su u određenom smislu "najteži". Takvi se prethodno navedeni problemi zovu *NP*-potpuni. Naime, isti imaju sljedeće svojstvo:

1. Ako postoji polinomni algoritam za jednog od njih, onda i postoji polinomni algoritam za bilo koji problem iz klase *NP*.

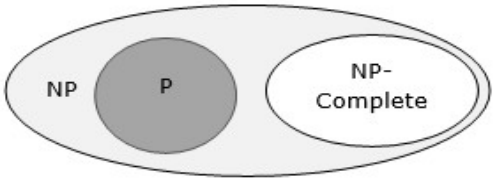
Samo traženje odgovora na pitanje da li je *P=NP* se može usmjeriti na traženje polinomnog algoritma za bilo koji od *NP*- potpunih problema.

Također, ovdje se javlja pojam redukcije. Isti služi za rješavanje problema koji se mogu prikazati kao prepoznavanje (odlučivanje) jezika. Upravo zbog prethodno navedenog razloga se redukcija može definirati kao preslikavanje između jezika. Konkretno, ako su *A* i *B* primjeri nekih jezika, onda se kaže da je *A* reducibilan na *B* (oznaka $A \leq_m B$), ako postoji *T*- izračunata funkcija *f* takva da za bilo koju riječ *w* vrijedi sljedeće svojstvo: $w \in A$ ako i samo ako $f(w) \in B$. Traženje odgovora na pitanje o *A* može se svesti na traženje odgovora na pitanje o *B*. Da bi se ustanovilo da li je $w \in A$ dovoljno je samo ustanoviti da li je $f(w) \in B$.

Ovdje valja naglasiti da se pojam *NP*- potpunosti gradi na prethodno spomenutom pojmu, redukcije. Tako funkcija *f* (koja preslikava riječ u riječ) je izračunata (može se izračunati) u polinomnom vremenu u slučaju da postoji konstanta *(k)* i *TS* (deterministički Turingov stroj, matematička funkcija gdje ponašanje stroja ovisi isključivo o njegovom stanju i znaku koji je pročitan na traci) koji za argument duljine *n* računa vrijednost od *f* u vremenu $O(n^k)$. Također, jezik *A* je reducibilan u polinomnom

vremenu na jezik *B* preko funkcije *f* koja je izračunata u polinomnom vremenu (oznaka: $A \leq_p B$).

Slika 25.: Prikaz NP- potpunog problema



Izvor: Anonymous_5, 2019.

Ako postoji algoritam polinomnog vremena za bilo koji od ovih problema, svi problemi u *NP* bi se mogli riješiti polinomno (*NP*- potpuni/ kompletni). Fenomen *NP*- potpunosti važan je iz teorijskih i praktičnih razloga.

Postoje dva teorema:

1. Ako je $A \leq_p B$ i *B* je u *P*, tada je naravno i *A* u *P*.
2. Ako je $A \leq_p B$ i *A* nije u *P*, tada ni *B* nije u *P*.

Dakle, jezik *S* je *NP*- potpun ako:

1. Jezik *S* pripada *NP*.
2. Za sve *A* koji su u *NP* vrijedi da je $A \leq_p S$.

Prethodno navedeni teorem tako predstavlja sljedeće:

1. Ako je jezik *S NP*- potpun i *S* je u *P*, tada je *P=NP*.
2. Ako je jezik *S NP*- potpun, *T* je u *NP* i $S \leq_p T$, tada je i *T NP*- potpun.

Standardna metoda dokazivanja *NP*- potpunosti je da se uzme problem za koji se zna da je *NP*- potpun te da ga se reducira u polinomnom vremenu na problem iz *NP* za kojeg se želi dokazati da je *NP*- potpun. Dakle, primjenjuje se prethodno navedena tvrdnja (ona pod 2).

Nadalje, definira se Cook- Levinov teorem. Isti kaže da jezik *SAT*, dakle problem zadovoljivosti Booleovske formule u konjuktivnoj normalnoj formi, je *NP*- potpun.

Sama ideja dokaza teorema je da je *SAT* u *NP*. Tako je jezik *A* iz *NP* koji se može reducirati na *SAT* u polinomnom vremenu. Budući da je *A* iz *NP*, postoji nedeterministički Turingov stroj *M* koji odlučuje *A* u polinomnom vremenu. Ovdje

redukcija *A* na *SAT* uzima zadanu riječ *w* i stvara Booleovsku formulu u *KNF* koja simulira rad od *M* na ulazu *w*. Konkretno, ako *M* prihvaća riječ *w*, tada je formula zadovoljiva, a odgovarajuće pridruživanje istine (ili laži) varijablama od formule koja opisuje granu računanja od *M* vodi u prihvatljivo stanje. Pri tome, ako *M* ne prihvaća riječ *w*, tada ne postoji pridruživanje za koje je formula istinita. To znači da je *w* u *A* ako i samo ako je formula zadovoljiva. Stoga se sama pretvorba *M* i *w* u formulu odvija u polinomnom vremenu. Točnije, ako *n* predstavlja duljinu od *w*, a *M* odlučuje *w* u polinomnom vremenu *p(n),* tada vrijeme za konstrukciju formule iznosi *O (p od M*) koja vodi u prihvatljivo stanje. Pri tome, ako *M* ne prihvaća riječ *w*, tada ne postoji pridruživanje za koje je formula istinita. To znači da je *w* u *A* ako i samo ako je formula zadovoljiva. Stoga se sama pretvorba *M* i *w* u formulu odvija u polinomnom vremenu (*N* od *M* koja vodi u prihvatljivo stanje). Pri tome, ako *M* ne prihvaća riječ *w*, tada ne postoji pridruživanje za koje je formula istinita. To je *z* od *M* koja vodi u prihvatljivo stanje. Prema tome, ako *M* ne prihvaća riječ *w*, tada ne postoji pridruživanje za koje je formula istinita. To znači da je *w* u *A* ako i samo ako je formula zadovoljiva. Stoga se sama pretvorba *M* i *w* u formulu odvija u polinomnom vremenu. Točnije, ako *n* predstavlja duljinu od *w*, a *M* odlučuje *w* u polinomnom vremenu *p(n),* tada vrijeme za konstrukciju formule iznosi $O\ (p^3(n)*\ log\ n)$.

Postupak kako se iz *M* i *w* konstruira formula je prilično kompliciranа te sadrži veliki broj tehničkih detalja. Kada se ustanovi da je *SAT NP*- potpun, radi se sljedeće:

1. *SAT* se reducira na razne druge probleme iz *NP* te se time dokazuju da su i ti drugi problemi *NP*- potpuni.
2. Drugi problemi se reduciraju na treće probleme iz *NP* i dokazuje se da su i ti treći problemi *NP*- potpuni, itd.

Konkretno, valja zaključiti da se skup *NP*- potpunih problema širi.

Također, postoji nekoliko tisuća problema za koje se zna da su *NP*- potpuni. Naime, popis tih problema je stalno u porastu.

Slijedi nekoliko primjera problema za koje vrijedi sljedeće:

1. Jasno je da su iz *NP* jer postoji očigledni certifikat.
2. Redukcijom je dokazano da su *NP*- potpuni.

Primjeri NP- potpunih problema su:

1. 3SAT.
2. CLIQUE.
3. DOMINATION.
4. VERTEX_COVER.
5. HAMCYCLE.
6. HAMPATH.
7. COLOR.
8. EXACT_COVER.
9. SUBSET_SUM.
10. PARTITION.

5.1 Lagani i teški problemi

NP- potpuni problemi su se bavili problemima odlučivanja koji su kao izlaz davali odgovor da ili ne na postavljeno pitanje. U stvarnom životu se više upotrebljavaju problemi optimizacije, koji kao izlaz daju minimalnu ili maksimalnu vrijednost zadane funkcije cilja. Naime, mnogi od problema optimizacije spadaju u klasu *NP*- teških problema, što može značiti da su vrlo zahtjevni za rješavanje. Konkretno, svaki *NP*- teški problem optimizacije je u bliskoj vezi s odgovarajućim i sličnim *NP*- potpunim problemom odlučivanja.

Valja naglasiti da je problem *NP*- težak ako postojanje polinomnog algoritma za njegovo rješavanje povlači da je *P=NP*. Zbog vjerovanja da je $P \neq NP$, također se vjeruje da za *NP*- teške probleme ne postoje polinomni algoritmi. Upravo zbog prethodno navedene tvrdnje se intenzivno proučavaju i brzi približni načini rješavanja *NP*- teških problema.

NP- teških problema također ima na tisuće. Stoga, odabrani problemi spadaju u kombinatornu optimizaciju, odnosno bave se:

1. Raspoređivanjem.
2. Optimizacijom na grafovima.

Neki *NP*- teški problemi su:

1. Problem ranca.
2. Problem raspoređivanja poslova na identične strojeve.
3. Problem optimalnog bojenja grafa.
4. Problem trgovačkog putnika.

Dokazivanje *NP*- težine se izvodi metodom redukcije u polinomnom vremenu (slično kao dokazivanje *NP*- potpunosti). Upravo da bi za neki "novi" problem dokazali da je *NP*- težak, treba raditi sljedeće:

1. Naći neki "stari" problem za koji se već zna da je *NP*- potpun ili *NP*- težak.
2. Pokazati da se "stari" problem može u polinomnom vremenu reducirati na "novi" problem.

Navedeni dokaz je valjan zbog novog problema koji se rješava u polinomnom vremenu, pa se tada i stari problem primjenom redukcije rješava u polinomnom vremenu (zbog *NP*- potpunosti ili *NP*- težine starog problema će vrijediti *P=NP*).

Slijede četiri odabrana problema optimizacije koja su *NP*- teška (teorema):

1. *NP*- potpuni problem *PARTITION* se reducira u polinomnom vremenu na problem ranca.
2. *NP*- potpuni problem *PARTITION* reducira se u polinomnom vremenu na problem raspoređivanja poslova na identične strojeve.
3. *NP*- potpuni problem *COLOR* se reducira u polinomnom vremenu na problem optimalnog bojenja grafa.
4. *NP*- potpuni problem *HAMCYCLE* reducira se u polinomnom vremenu na problem trgovačkog putnika.

Postoje tri metode rješavanja *NP*- teških problema:

1. Egzaktni algoritmi.
2. Aproksimacijski algoritmi.
3. Heuristike.

Egzaktni algoritmi daju optimalno rješenje, no zbog velike računske složenosti mogu se primijeniti samo na relativno male primjerke problema. Za razliku od egzaktnih algoritama, aproksimacijski daju u polinomnom vremenu približno (ili sub- optimalno)

rješenje, te naravno i garanciju da je to rješenje “blizu” optimalnom. Posljednja metoda, heuristike, brzim i jednostavnim računanjem daje približno rješenje, koje se često pokazuje zadovoljavajućim unatoč nedostatku garancije “dobrote”. Međutim, moguće je konstruirati i primjere za koje se dugo izvršavaju i/ ili daju rješenja koja su daleko od optimalnog.

Također, postoji nekoliko tipova egzaktnih algoritama:

1. Dinamičko programiranje (engl. Dinamic programming).
2. Pretraživanje s vraćanjem (engl. Backtracking).
3. Grananje i ograničavanje (engl. Branch and Bound).
4. Algoritam odsijecajućih ravnina.

Pošto nema efikasnih egzaktnih algoritama, pribjegava se raznim “slabijim” algoritmima koji: (Šego, 2009., str.12)

1. Rješavaju samo neku usku podklasu problema koji se proučava.
2. Ne garantiraju optimalnost rješenja (ali za gotovo sve ulazne podatke daju rješenje blisko optimalnom), ali garantiraju brzinu.
3. Garantiraju da je dobiveno rješenje uvijek proizvoljno blizu optimalnom, ali ne garantiraju brzinu (tj. u pravilu su brzi, ali je moguće konstruirati ulazne podatke za koje nisu).

Vrste aproksimacijskih algoritama su:

1. Apsolutne aproksimacije.
2. Relativne aproksimacije.
3. Aproksimacijske sheme.

5.2 Klasa NP

NP- problemi se definiraju slično *P*- problemima, uz dodatak *“N”* koji znači “nedeterministički”. Dakle, *NP*- problemi (ili problemi klase *NP*) su oni problemi za koje je moguće definirati nedeterminističke Turingove strojeve koji ih rješavaju u polinomnom vremenu. (Šego, 2009., str.7).

Nedeterministički stroj kada ima “dvojbu” ispravno pogađa što će se od ponuđenih akcija napraviti. Kako bi se pokušao implementirati takav stroj, mora se programirati isprobavanje svih mogućnosti, pa bi tada broj koraka jako narastao (primjerice, eksponencijalno).

5.3 NP kompletni problem

Komplement klase *NP* se označava s *co- NP*. Sama klasa problema *NP* je asimetrična:

1. *NP* se bavi traženjem potvrdnog (“da”) odgovora za neki problem odlučivanja.
2. *Co- NP* se bavi pronalaženjem opovrgavajućeg (“ne”) odgovora za isti, ali komplementarni problem odlučivanja.

Ako postoji problem *X*, njegov komplement je identičan problemu kojem su odgovori da i ne preokrenuti. Tako problem *X* pripada klasi *co- NP* ako i samo ako njegov komplement pripada klasi *NP*.

Problemi odlučivanja za koje postoje certifikati u polinomnom vremenu: SAT, HAMILTON- CYCLE i COMPOSITE. Komplementi za prethodno navedene probleme odlučivanja su: TAUTOLOGY, NO- HAMILTON- CYCLE i PRIMES.

6. Minimalno razgranato stablo

Pronalaženje minimalnog razgranatog stabla je bitno jer omogućava optimalno iskorištavanje resursa kojima se raspolaže, pa zbog toga ima mnogostruke primjene u praksi.

Neka je *G= (V, E)* povezani neusmjereni graf. Postoji razgranato stablo grafa *G* koje predstavlja podgraf *T= (V, E’)* koji je uz to povezan i neciklički. Dakle, on povezuje sve vrhove.

Neka je *G= (V, E)* povezani neusmjereni težinski graf. Ima minimalno razgranato stablo *T* grafa *G* koje je razgranato stablo čija je ukupna težina bridova minimalna. U definiciji se pretpostavlja da je zadani graf označen u smislu da su svakomu bridu dane težine koji su pozitivni realni brojevi.

Razgranato stablo je stablo koje se dobiva uklanjanjem određenog broja bridova iz grafa, a da pritom dobiveni graf ostane povezan.

Minimalno razgranato stablo je stablo koje ima minimalnu sumu težine bridova.

Očito je da graf može imati puno razgranatih stabala. Općenitije, bilo koji neusmjereni graf ima minimalnu šumu (engl. minimum forests), koja je unija minimalnih stabala za svoje povezane komponente.

Minimalno razgranato stablo ima izravnu primjenu u projektiranju mreža. Koristi se u algoritmima koji aproksimiraju problem trgovačkog putnika, multi- terminalnom minimalnom problemu i minimalnom troškovnom ponderiranom savršenom podudaranju. Ostale praktične primjene su:

1. Klaster analiza.
2. Prepoznavanje rukopisa.
3. Segmentacija slike. (Abdelnabi, 2019a.)
4. Ostalo (primjerice, pronalazak cestovne mreže kod satelitskih te zračnih snimki, smanjivanje pohrane podataka u sekvenciranju amino kiselina u proteinu, međudjelovanje čestica u turbulentnim strujama tekućine, premošćivanje Etherneta kako bi se izbjegli ciklusi u mreži, algoritmi aproksimacije za *NP* teške probleme, dizajn mreže (komunikacija, hidraulika) i sl. (Sedgewick i sur., n_d, str. 13).

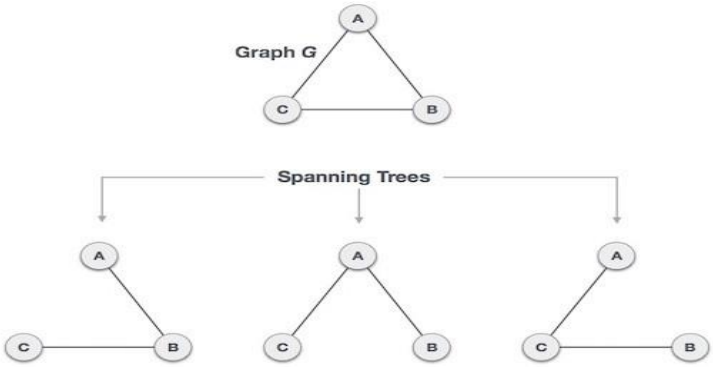
Matematička svojstva razgranatog stabla (Anonymous_6, 2019.):

1. Razgranato stablo ima $n-1$ rubova, gdje je n broj čvorova (vrhova).
2. Iz cijelog grafa, uklanjanjem maksimalnih $e-n+1$ rubova, može se izgraditi razgranato stablo.
3. Cijeli graf može imati najviše n^{n-2} broja razgranatih stabala.

Dakako, iz svega navedenoga, može se reći da su razgranata stabla podskup povezanih grafova *G* te da nepovezani grafovi nemaju razgranato stablo.

Što se tiče sljedećeg grafa, postoje tri stabla koja su nastala od jednog potpunog grafa. Potpuno neusmjereni graf može imati maksimalan broj n^{n-2} stabla u rasponu, gdje je pritom n broj čvorova. U dolje navedenom primjeru (slika 26.), n je 3, dakle moguće je $3^{3-2}=3$ razgranatih stabala.

Slika 26.: Primjer razgranatog stabla



Izvor: Anonymous_6, 2019.

Postoje tri poznata algoritma koja služe za pronalaženje minimalnog razgranatog stabla, a to su:

1. Primov algoritam.
2. Kruskalov algoritam.
3. Bourivkin algoritam.

Primov algoritam ima složenost *O(E log V).* Kruskalov ima također složenost *O(E log V)*, kao i Bourivkin algoritam. Naime, njegove karakteristike su da je on pohlepni algoritam te da su sve težine bridova različite. (Bujanović, 2016., str. 10).

Osnovne razlike između prethodno navedena tri algoritma: (Anonymous_7, n_d)

1. Bourivkin algoritam u svakom koraku spaja sva najbliža stabla s novom granom.
2. Primov algoritam u svakom koraku proširuje označeno stablo s najbližim čvorom.
3. Kruskalov algoritam u svakom koraku spaja dva najbliža stabla s novom granom.

Kruskalov te Bourivkin algoritam se mogu unaprijediti tako da njihova složenost bude *O (m*L(n)),* gdje je *L* inverzna Ackermanova funkcija.

6.1 Rast minimalnog razgranatog stabla

Ovdje se pretpostavlja da postoji povezani i neusmjereni graf *G=(V,E)* s težinskom funkcijom *w:E->R*. Naime, ovdje se želi postići, tj. pronaći minimalno razgranato stablo za graf *G*. Dva algoritma koja se ovdje razmatraju koriste pohlepni pristup u rješavanju određenog problema te se razlikuju u načinu na koji primjenjuju taj pristup.

Ta pohlepna strategija se opisuje sljedećim „generičkim“ algoritmom, koji uzdiže minimalno razgranato stablo jednog ruba u isto vrijeme. Konkretno, taj algoritam upravlja skupom rubova *A*, održavajući sljedeću petlju nepromjenjivom: (Cormen i sur., 2001.).

1. Prije svake iteracije, *A* je podskup nekih minimalnih razgranatih stabala.

Preciznije, na svakom koraku se određuje rub *(u, v)* koji se može dodati na *A* bez kršenja (nepromjenjivog), u smislu da je *A {(u, v)}* također podskup minimalnog razgranatog stabla. Takav rub se naziva sigurnim rubom za *A*, budući da se sigurno može dodati *A*, a zadržati nepromjenjivost.

Slika 27.: Prikaz (dolje) opisane nepromjenjivosti petlje

```
GENERIC-MST(G, w)
1  A ← ∅
2  while A does not form a spanning tree
3      do find an edge (u, v) that is safe for A
4         A ← A ∪ {(u, v)}
5  return A
```

Izvor: Cormen i sur., 2001.

Koristi se nepromjenjiva petlja: (Cormen i sur., 2001.).

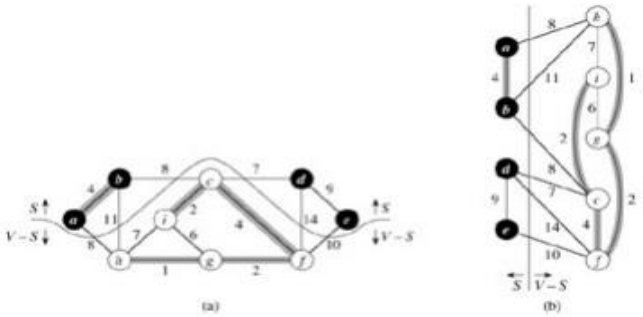
1. Inicijalizacija: nakon prvog retka, skup *A* trivijalno zadovoljava nepromjenjivu petlju.
2. Održavanje: petlja u crtama *2- 4* održava nepromjenjivost pritom dodajući samo sigurne rubove.
3. Prekid: svi rubovi koji su dodani u *A* (minimalnom razgranatom stablu). Stoga se skup *A* vraća u petom retku i isti mora biti minimalno razgranato stablo.

Zanosnim dijelom se smatra pronalaženje sigurnog ruba u trećem retku. Jedan rub stoga mora postojati, jer kada se izvodi treća linija, nepromjenjivost (invarijant) diktira

da postoji razgranato stablo *T (A T).* Unutar tijela petlje *A*, mora postojati odgovarajući podskup od *T* i rub *(u, v) T* takav da je *(u, v)* i *A* i *(u, v)* siguran za *A*.

Najprije trebaju neke definicije. *Rez (S, V - S)* neusmjerenog grafa *G = (V, E)* je particija *V*. Sljedeća slika ilustrira taj pojam. Kaže se da rub *(u, v) E* prelazi *rez (S, V - S)* ako je jedna od njegovih krajnjih točaka u *S*, a druga je u *V - S*. Isto tako kaže se da *rez* poštuje skup *A* rubova ako bez ruba u *A* prelazi *rez*. Rub je lagani rub koji prelazi *rez* ako je težina minimalna od bilo kojeg ruba koji prelazi *rez*. Valja imati na umu da u slučaju veza može biti više od jednog svjetlosnog ruba koji prelazi *rez*. Općenitije, kaže se da je rub svijetli rub koji zadovoljava dano svojstvo ako je njegova težina minimalna od bilo kojeg ruba koji zadovoljava svojstvo.

Slika 28.: Prikaz *Rez (S, V - S)* neusmjerenog grafa *G = (V, E)* koji je particija *V*



Izvor: Cormen i sur., 2001.

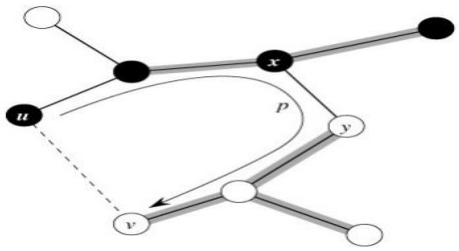
Prethodna slika prikazuje dva načina gledanja *reza (S, V - S)* grafa s prve slike (slika a), vrha u skupu *S* prikazana su crnom bojom, a ona u *V - S* bijelom. Rubovi koji prelaze *rez* su oni koji povezuju bijele vrhove s crnim vrhovima. Rub *(d, c)* je jedinstveni svjetlosni rub koji prelazi *rez*. Podskup *A* rubova je zasjenjen, stoga valja napomenuti da *rez (S, V - S)* poštuje *A*, budući da niti jedan rub *A* ne prelazi *rez*. Druga slika (slika b) prikazuje isti graf s vrhovima koji postavlja *S* s lijeve strane i vrhove u skupu *V - S* na desnu stranu. Rub presijeca *rez* ako povezuje vrh s lijeve strane s vrhom na desnoj strani.

Pravilo za prepoznavanje sigurnih rubova dano je sljedećim teoremom. Neka je *G = (V, E)* spojeni, neusmjereni graf s realnom vrijednosnom funkcijom *w* definiranom na *E.* Neka je *A* ⊆ *E* koja je uključena u neko minimalno razgranato stablo za *G*, neka je *(S, V - S)* bilo koji rez od *G* koji poštuje *A*, i neka je *(u, v)* svjetlosni prijelaz *(S, V - S).* Tada je rub *(u, v)* siguran za *A.*

Dokaz: Neka je *T* minimalno razgranato stablo koje uključuje *A* i pretpostavlja se da *T* ne sadrži rub svjetla *(u, v),* jer ako se to dogodi, to se upravo učinilo. Konstruirat će se još jedno minimalno razgranato stablo *T'* koje uključuje *A {(u, v)}* pomoću tehnike rezanja i lijepljenja (engl. cut- and- past), čime se pokazuje da je *(u, v)* siguran rub za *A.*

Rub *(u, v)* tvori ciklus s rubovima na putu *p* od *u* do *v* u *T*, kao što je prikazano na sljedećoj slici. Budući da su *u* i *v* na suprotnim stranama *reza (S, V - S),* postoji najmanje jedan rub u *T* na putu *p* koji također prelazi *rez*. Neka je *(x, y)* bilo koji takav rub. Rub *(x, y)* nije u *A*, jer je *rez rez*. *A*. Budući da je *(x, y)* na jedinstvenom putu od *u* do *v* u *T*, uklanjanje *(x, y)* razbija *T* na dvije komponente. Dodavanjem *(u, v)* ponovno ih spaja da bi se stvorilo novo razgranato stablo *T' = T - {(x, y)} {(u, v)}.*

Slika 29.: Prikaz ruba *(u, v)* koji tvori ciklus s rubovima na putu *p* od *u* do *v* u *T*



Izvor: Cormen i sur., 2001.

Slika prikazuje dokaz teorije. Vrhovi u *S* su crni, a vrhovi u *V - S* su bijeli. Prikazani su rubovi minimalnog razgranatog stabla *T*., ali rubovi u grafu *G* nisu. Rubovi u *A* su zasjenjeni, a *(u, v)* je lagani rub koji prelazi *rez (S, V - S)*. Rub *(x, y)* je rub na jedinstvenom putu *p* od *u* do *v* u *T*. Minimalno razgranato stablo *T'* koje sadrži *(u, v)* nastaje uklanjanjem ruba *(x, y)* iz *T* i dodavanjem ruba *(u, v).*

Zatim će se pokazati da je *T'* minimalno razgranato stablo. Budući da je *(u, v)* svjetlosni prijelaz *(S, V - S)* i *(x, y),* isti također prelazi ovaj *rez*, *w (u, v) ≤ w (x, y).*

Stoga je,

w (T ') = w (T - w (x, y) + w (u, v)) ≤ w (T).

T je minimalno razgranato stablo, tako da je *w (T) ≤ w (T').* Dakle, *T'* također mora biti minimalno razgranato stablo.

Ostaje pokazati da je *(u, v)* zapravo siguran rub za *A*. Postoji *A T'*, budući da su *A T* i *(x, y); A*; dakle, *A {(u, v)} T'*. Prema tome, budući da je *T'* minimalno razgranato stablo, *(u, v)* je siguran za *A.*

Teorem daje bolje razumijevanje djelovanja GENERIC-MST algoritma na povezanom grafu *G = (V, E).* Kako se algoritam nastavlja, skup *A* je uvijek acikličan, u suprotnom, minimalno razgranato stablo uključujući *A* bi sadržavalo ciklus, što je kontradikcija. U bilo kojem trenutku izvođenja algoritma, graf *GA = (V, A)* je šuma, a svaka od povezanih komponenti *GA* je stablo. (Neka stabla mogu sadržavati samo jedan vrh, kao što je slučaj, primjerice, kada počinje algoritam: *A* je prazan, a šuma sadrži |*V*| stabla, jedan za svaki vrh.) Štoviše, svaki siguran rub *(u, v)* za *A* povezuje različite komponente *GA*, jer *A {(u, v)}* mora biti acikličan.

Izvodi se petlja u crtama *2-4* GENERIC-MST |*V*|- *1* puta kao svaki od |*V*| - *1* rubovi minimalnog razgranatog stabla se sukcesivno određuju. U početku, kada je *A = Ø,* postoje |*V*| stabla u *GA*, a svaka iteracija taj broj smanjuje za jedan. Kada šuma sadrži samo jedno stablo, algoritam se završava.(Cormen i sur., 2001.).

Posljedica teorema. Neka je *G = (V, E)* spojeni, neusmjereni graf s realnom vrijednosnom funkcijom *w* definiranom na *E*. Neka je *A⊂E* koja je uključena u neko minimalno razgranato stablo za *G*, i neka je *C = (VC)* , *EC)* povezana komponenta (stablo) u šumi *GA = (V, A).* Ako je *(u, v)* lagani rub koji povezuje *C* s nekom drugom komponentom u *GA*, tada je *(u, v)* siguran za *A.*

Dokaz: *Rez (VC, V - VC)* poštuje *A*, a *(u, v)* je lagani rub za taj *rez*. Stoga je *(u, v)* siguran za *A*.

6.2 Primov algoritam

Primov algoritam je algoritam minimalnog razgranatog stabla koji uzima graf kao ulaz i pronalazi podskup rubova tog grafa koji formira stablo. Dakle, isti uključuje svaki vrh koji ima minimalni zbroj težina među svim stablima koja se mogu formirati iz grafa. (Anonymous_8, n_d).

Ova vrsta algoritma se temelji na metodi pohlepe. Konkretno, u svakom se trenutku uzima brid najmanje težine koji povezuje neki od već iskorištenih vrhova s vrhom koji

još nije iskorišten. Naposljetku, taj algoritam u svakom trenutku povećava skup iskorištenih vrhova za jedan i isto tako smanjuje broj neiskorištenih vrhova za jedan. Međutim, može se naglasiti da su vrhovi podijeljeni na dva skupa i da se traži najmanji brid koji se nalazi na granici tih dvaju skupova.

Kod ovog algoritma se prvo popunjava vektor iskorištenih i slobodnih vrhova. Potom se za svaki iskorišteni vrh gledaju svi njegovi susjedi u skupu slobodnih vrhova i traži se najmanji. Zaključno, ne smiju se zaboraviti obrisati vektori vrhova koji su se koristili u proceduri.

Ovaj algoritam služi za rješavanje problema heuristike pri izradi lokalno optimalnog izbora u svakoj fazi s nadom da će se pronaći globalni optimum. Dakle, on učinkovito pronalazi minimalno razgranato stablo za povezane neizravne grafove.

Opis algoritma:

1. Počinje se od dodavanja u stablo proizvoljnog vrha (u svakom krugu stablo se „uzgaja“ za jedan vrh).
2. Dodaje se u stablo brid koji ima svojstvo da dodaje novi vrh u stablo i da pri tome ima minimalnu težinu (povezuje stablo s vrhovima koji još nisu dodani).
3. Postupak je gotov kad su svi vrhovi u stablu.

Najučinkovitiji način za implementaciju algoritma je korištenje reda čekanja. Taj red čuva sve čvorove koji su posjećeni, ali još nisu dodani stablu. Redoslijed čvorova u redu čekanja određuje se pomoću udaljenosti svakog čvora od stabla. Dakle, u svakom krugu vrh reda koji je povezan sa stablom (najjeftinijim) je uklonjen iz reda čekanja. (Anonymous_9, 2014.).

Algoritam pomoću reda čekanja se može opisati korak po korak:

1. Prvi je dio inicijalizacija (proizvoljni čvor je odabran kao korijen stabla).
2. Niz udaljenosti prati minimalni ponderirani vrh koji povezuje svaki vrh sa stablom.
3. Korijen čvor ima udaljenost nula (nema ruba za ostale vrhove), pa je njihova udaljenost postavljena na beskonačnost.
4. Korijen čvor je dodan u red.

U svakom krugu se izvlači čvor (kod reda čekanja). U početku to može biti samo korijenski čvor. Zatim se razmatraju svi susjedni vrhovi uklonjenog čvora s njihovim

odgovarajućim rubovima. Svaki susjed se provjerava je li u redu, a njegov spojni rub teži manje od trenutne vrijednosti udaljenosti za tog susjeda. Ako je to slučaj, trošak tog susjeda može se svesti na novu vrijednost. Inače, ako čvor još nije posjećen, tada ga se ubacuje u red, tako da se rubovi koji izlaze iz njega mogu razmotriti kasnije. Gornja iteracija se nastavlja sve dok u redu čekanja nema više čvorova. Konkretno, tada algoritam završava i kao rezultat vraća sve rubove korištene za izgradnju minimalnog razgranatog stabla.

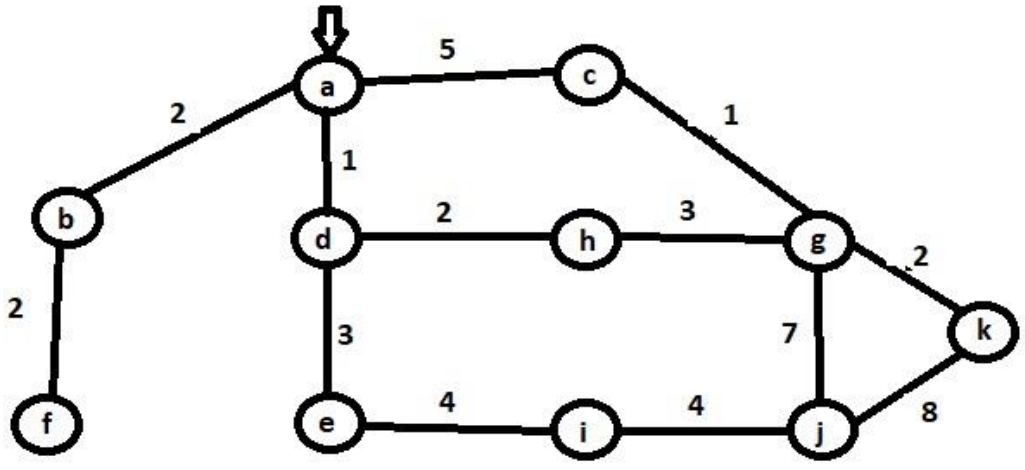
Složenost ovog algoritma je $O(n^3)$.

Iako se ovaj algoritam može modificirati kako bi se pronašla minimalna razgranata šuma za nepovezane grafove, preporučuju se druge metode kao što je to primjerice, Kruskalov algoritam.

6.2.1 Primjer

Zadatak nalaže da se mora pronaći minimalni put od vrha *a* pa do kraja grafa (svi vrhovi moraju biti obuhvaćeni), pritom koristeći Primov algoritam.

Slika 30.: Prikaz primjera Primovog algoritma



Izvor: Vlastiti rad

Strelica je prikazana na vrhu *a* jer počinje od *a* izvršenje Primovog algoritma (izračun minimalnog puta počevši od vrha *a,* pa skroz dok ne budu svi vrhovi obuhvaćeni).

Rješenje ovog primjera nalaže da se zbroje težine koje se nalaze na grafu (brojevi), tako da pritom zbroj bude minimalan.

Rješenje je sljedeće:

Dakle, postoji 11 vrhova, a minimalno razgranato stablo je |*E*|=*10.*

Rub *a, d =1*

Rub *a, b =2*

Rub *b, f= 2*

Rub *d, e= 2*

Rub *h, g= 3*

Rub *g, c= 1*

Rub *g, k= 2*

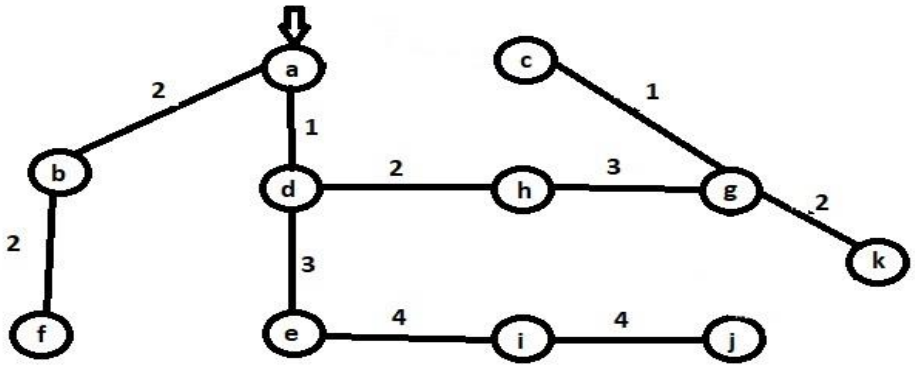
Rub *d, e= 3*

Rub *e, i= 4*

Rub *e, j= 4*

Iznos toga je 24, što je minimalno, što se može postići kada se obuhvate svi vrhovi.

Slika 31.: Prikaz rješenja Primovog primjera



Izvor: Vlastiti rad

6.2.2 Implementacija u Pythonu

Prikaz primjera grafa u programskom jeziku Python.

Datoteka main.py:

```
from collections import defaultdict
import heapq

def create_spanning_tree(graph, starting_vertex):
    mst = defaultdict(set)
    visited = set([starting_vertex])
    edges = [
        (cost, starting_vertex, to)
        for to, cost in graph[starting_vertex].items()
    ]
    heapq.heapify(edges)

    while edges:
        cost, frm, to = heapq.heappop(edges)
        if to not in visited:
            visited.add(to)
            mst[frm].add(to)
            for to_next, cost in graph[to].items():
                if to_next not in visited:
                    heapq.heappush(edges, (cost, to, to_next))

    return mst
```

Izvor: Anonymous_34, n_d

```
if __name__ == "__main__":

     graph = {
    'A': {'B': 2, 'D': 1, 'C': 5},
    'B': {'A': 2, 'F': 2},
    'C': {'A': 5, 'G': 1},
    'D': {'A': 1, 'H': 2, 'E': 3},
    'E': {'D': 3, 'I': 4},
    'F': {'B': 2},
    'G': {'C': 1, 'H': 3, 'K': 2, 'J': 7},
    'H': {'D': 2, 'G': 3},
    'I': {'E': 4, 'J': 4},
    'J': {'I': 4, 'G': 7, 'K': 8},
    'K': {'G': 2, 'J': 8},
}

print("Primov algoritam.")
print("Ispis:", create_spanning_tree(graph, 'A'))
```

Opis koda:

Prvi red *from collections import defaultdict* jednostavno stvara stavke kojima se pokušava pristupiti (pod uvjetom da još uvijek ne postoje). Da bi se stvorila takva "zadana" stavka, ona poziva funkcijski objekt koji se prosljeđuje konstruktoru (uključuje funkcije i tipove objekata). Konkretno, ovdje se vraćaju podaci koje sadrži skup. Sljedeći red *import heapq* pruža implementaciju algoritma reda hrpe, također poznatog i kao algoritam prioritetnog reda. Funkcija *def create_spanning_tree(graph, starting_vertex)* sadrži graf (sa svojim vrhovima te rubovima) te početni vrh od kojeg se izvršava Primov algoritam. Red *mst = defaultdict(set)* stvara stavku tipa skup te ju sprema u varijablu mst (minimalno razgranato stablo). Potom red, *visited = set([starting_vertex]),* sadrži vrhove koji se nalaze u skupu (početni vrh). Sljedeći red, *heapq.heapify(edges)*, pretvara popis rubova u hrpu, u linearnom vremenu. Zatim *cost, frm, to = heapq.heappop(edges)* vraća najmanji predmet iz hrpe (u ovom slučaju rubove). Unutar funkcije *main()* je definiran poziv funkcije *(create_spanning_tree (graph, 'A')))* te ispis iste. Taj poziv funkcije sadrži varijablu graf (sastoji se od vrhova te rubova) i početni vrh od kojeg počinje izvršenje Primovog algoritma.

Rješenje primjera:

```
Primov algoritam.
Ispis: defaultdict(<class 'set'>, {'A': {'B', 'D'}, 'B': {'F'}, 'D': {'E', 'H'},
'H': {'G'}, 'G': {'K', 'C'}, 'E': {'I'},'I': {'J'}})
```

6.2.3 Implementacija pomoću vektora

Prikaz primjera grafa u programskom jeziku C++.

Datoteka primov.cpp:

```
#include<bits/stdc++.h>
#include<vector>

using namespace std;
# define INF 0x3f3f3f3f

typedef pair<int, int> iPair;

void addEdge(vector <pair<int, int> > adj[], int u,
```

```
                                    int v, int wt)
{
    adj[u].push_back(make_pair(v, wt));
    adj[v].push_back(make_pair(u, wt));
}

void primMST(vector<pair<int,int> > adj[], int V)
{
    priority_queue< iPair, vector <iPair> , greater<iPair> > pq;

    int src = 0;

    vector<int> key(V, INF);

    vector<int> parent(V, -1);

    vector<bool> inMST(V, false);

    pq.push(make_pair(0, src));
    key[src] = 0;

    while (!pq.empty())
    {
        int u = pq.top().second;
        pq.pop();

        inMST[u] = true;

        for (auto x : adj[u])
        {
            int v = x.first;
            int weight = x.second;

            if (inMST[v] == false && key[v] > weight)
            {
                key[v] = weight;
                pq.push(make_pair(key[v], v));
                parent[v] = u;
            }
        }
    }

    for (int i = 1; i < V; ++i)
        cout<<parent[i] <<" "<< i <<endl;
}

int main()
{
    int V = 11;
```

```
    vector<iPair > adj[V];

    addEdge(adj, 0, 1, 2);
    addEdge(adj, 0, 3, 1);
    addEdge(adj, 0, 2, 5);
    addEdge(adj, 1, 5, 2);
    addEdge(adj, 3, 7, 2);
    addEdge(adj, 3, 4, 3);
    addEdge(adj, 2, 6, 1);
    addEdge(adj, 7, 6, 3);
    addEdge(adj, 4, 8, 4);
    addEdge(adj, 8, 9, 4);
    addEdge(adj, 6, 10, 2);
    addEdge(adj, 6, 9, 7);
    addEdge(adj, 10, 9, 8);

    primMST(adj, V);

    return 0;
}
```

Izvor: geeksforgeeks_17, n_d

Opis koda:

Valja naglasiti da se ovdje umjesto slova (*a…k* koriste brojevi *0…10*). Prvo je uključena biblioteka vector *(#include<vector>)* kako bi se moglo raditi sa strukturom podatka tipa vektor. Nakon toga je definiran *Integer Pair (typedef pair<int, int> iPair)*, pa funkcija za dodavanje rubova (*void addEdge(vector <pair<int, int> > adj[], int u, int v, int wt*)) te funkcija za ispisivanje najkraćeg puta od izvora, pa do svih ostalih vrhova (*void primMST(vector<pair<int,int> > adj[], int V)).* Ista funkcija sadrži pohranjivanje vrhova pomoću prioritetnog reda (*priority_queue< iPair, vector <iPair> , greater<iPair> > pq*), uzimanje vrha 0 kao izvorište (*int src = 0*), stvaranje vektora za ključeve i inicijaliziranje istih kao beskonačne (*vector<int> key(V, INF)),* spremanje matičnog (roditeljskog) niza koji zauzvrat pohranjuje minimalno razgranato stablo *(vector<int> parent(V, -1)),* praćenje vrhova koji su uključeni u minimalno razgranato stablo (*vector<bool> inMST(V, false)),* umetanje izvorišta u prioritetni red i inicijaliziranje njegovog ključa kao 0 (*pq.push(make_pair(0, src)) i key[src] = 0*) te izvršenje petlje dok prioritetni red nije prazan (*while (!pq.empty())).* U toj petlji, prvi vrh se smatra minimalnim pa ga se izdvoji u prioritetni red (prvi par), a oznaka vrha se pohranjuje u drugom paru (to se mora učiniti na način da se ključ sortira uz pomoć vrhova, ključ mora biti prva stavka u paru) (*int u = pq.top().second i pq.pop()).* Nakon toga se uključuje vrh u minimalnom razgranatom stablu (*inMST[u] = true*), pretražuju se svi vrhovi (*for (auto x : adj[u])* i

dobivaju se vrhovi i težine koji su susjedni od *u* (*int v = x.first* i *int weight = x.second*). Osim toga, ako *v* nije u minimalnom razgranatom stablu, a težina (u, v) je manja od trenutne (*if (inMST[v] == false && key[v] > weight)),* onda se ažurira ključ od *v* (*key[v] = weight, pq.push(make_pair(key[v], v))* i *parent[v] = u*). Na kraju toga se ispisuju rubovi minimalnog razgranatog stabla pomoću nadređenog niza (*for (int i = 1; i < V; ++i), cout<<parent[i] <<" "<< i <<endl*). Unutar funkcije *int main()* se definira testiranje klase *Graph* gdje se prikazuje broj vrhova (*0-10*) (*int V = 11;)* koji se spremaju unutar vektora (*vector<iPair > adj[V])* i izrada samog grafa (primjerice, *addEdge(adj, 8, 9, 4))* te poziv funkcije (*primMST(adj, V)).*

Prikaz rješenja:

```
0 1

6 2

0 3

3 4

1 5

7 6

3 7

4 8

8 9

6 10
```

6.3 Kruskalov algoritam

Ovaj algoritam se također temelji na metodi pohlepe, s razlikom što ima drugačiju koncepciju nego Primov algoritam. Isti kreće od situacije u kojoj svaki vrh grafa čini poseban skup vrhova. Tako se u svakom koraku povezuju dva skupa vrhova za koje se to može učiniti uz najmanju cijenu. Konačno, algoritam završava kada se svi vrhovi grafa nalaze u istom skupu.

Opis algoritma:

1. Svaki čvor se na početku smatra zasebnim stablom.
2. Dodaje se u stablo brid koji ima svojstvo da spaja sva stabla i da ima minimalnu težinu.
3. Postupak je gotov kad su svi vrhovi u stablu.

Kruskalov algoritam: (Bujanović, 2016., str. 13)

1. Neka je *F* šuma koja se sastoji od čvorova stabla gdje svaki čvor predstavlja zasebno stablo.
2. Neka je *S* skup svih bridova u grafu.
3. Ako je *S* neprazan i *F* nije razgranato stablo učini sljedeće:
4. Makni brid koji ima najmanju težinu u *S*
5. Ako taj brid povezuje dva stabla u *F*, dodaj ga u *F*
6. Završi petlju.

Vremenska složenost ovog algoritma je $O(n^3)$.

Kruskalov algoritam je još jedan od popularnih algoritama minimalnog razgranatog stabla koji koristi drugačiju logiku za pronalaženje najkraćeg puta kroz graf. Umjesto da krene od vrha, Kruskalov algoritam razvrstava sve rubove od najmanjeg prema najvećem (stalno dodaje najniže rubove), ignorirajući one rubove koji stvaraju ciklus.

Slijedi prvi dokaz korektnosti Kruskalovog algoritma. Naime, tvrdnja je da je dobiveni podgraf razgranato stablo početnog grafa. Dokaz iste nalaže da je *T* podgraf dobiven Kruskalovim algoritmom. Tako očito *T* ne može imati ciklus, jer svaki brid koji se dodaje povezuje dva (pritom različita) podstabla. *T* ne može biti nepovezan, iz razloga jer bi to značilo da ne postoji brid koji bi povezao neke dvije komponente, a što bi bilo u kontradikciji s pretpostavkom da je početni graf povezan.

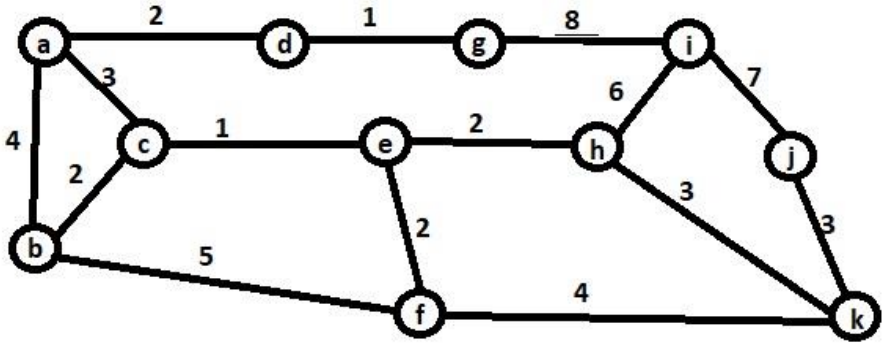
Ovdje je definiran drugi dokaz korektnosti Kruskalovog algoritma. Tvrdnja kaže da je dobiveno razgranato stablo minimalno. Dokaz koji potkrepljuje tu tvrdnju je matematička indukcija. Ista dokazuje sljedeću propoziciju *P*. Ako je *E* skup tih odabranih bridova u nekom koraku algoritma, tada postoji minimalno razgranato stablo koji sadrži *E*. Sama baza rješavanja ove tvrdnje je da *P* vrijedi na početku jer svaki graf sadrži minimalno razgranato stablo. Slijedi korak rješavanja prethodno navedene tvrdnje. Pretpostavlja se da je *P* istinit u nekom (ali ne konačnom) koraku algoritma, te

neka je *T* minimalno razgranato stablo koji sadrži *E*. U slučaju da je sljedeći odabrani brid *e* također u *T*, tada je *P* točna za *E*+*e*, dok je u suprotnom *T*+*e* ciklus *C* (sadrži ga) i postoji brid *f* koji je u *C*, ali nije u *E*. Tada je *T* –*f*+*e* stablo, i ima istu težinu kao i *T*, iz razloga što je *T* imao najmanju težinu, a *f* ima težinu koja je veća ili jednaka težini od *e*. Dakle, *T* –*f*+*e* je minimalno razgranato stablo koji sadrži *F*+*e*, pa *P* vrijedi.

6.3.1 Primjer

Zadatak nalaže da se mora pronaći minimalni put od vrha *a* pa do kraja grafa (svi vrhovi moraju biti obuhvaćeni), pritom koristeći Kruskalov algoritam.

Slika 32.: Prikaz primjera Kruskalovog algoritma



Izvor: Vlastiti rad

Rješenje ovog primjera nalaže da se zbroje težine koje se nalaze na grafu (brojevi), tako da pritom zbroj bude minimalan.

Rješenje je sljedeće:

Dakle, postoji 11 vrhova, a minimalno razgranato stablo je |*E*|=*10.*

Rub *d, g =1*

Rub *c, e =1*

Rub *a, d= 2*

Rub *e, f= 2*

Rub *e, h= 2*

Rub b*, c= 2*

Rub *j, k= 3*

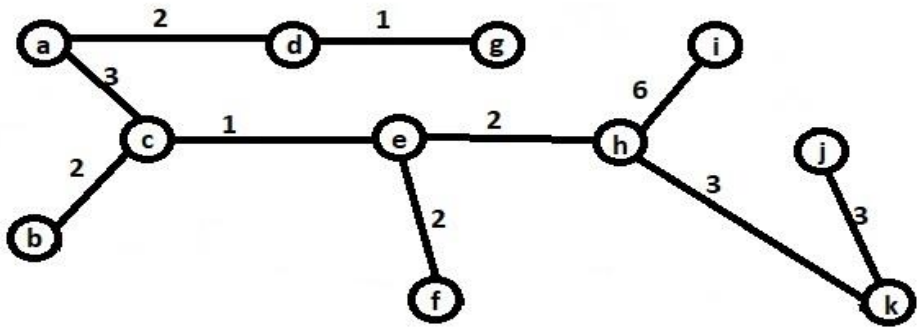
Rub *h, k= 3*

Rub *a, c= 3*

Rub *h, i= 6*

Iznos toga je 25.

Slika 33.: Prikaz rješenja Kruskalovog primjera



Izvor: Vlastiti rad

6.3.2 Implementacija u Pythonu

Prikaz rješenja u Pythonu:

Datoteka main.py:

```
from operator import itemgetter

class DisjointSet(dict):
    def add(self, item):
        self[item] = item

    def find(self, item):
        parent = self[item]

        while self[parent] != parent:
            parent = self[parent]

        self[item] = parent
        return parent
```

```
    def union(self, item1, item2):
        self[item2] = self[item1]
```

Izvor: Anonymous_35, 2010.

```
def kruskal(nodes, edges):
    forest = DisjointSet()
    mst = []
    for n in nodes:
        forest.add(n)

    sz = len(nodes) - 1

    for e in sorted(edges, key=itemgetter(2)):
        n1, n2, _ = e
        t1 = forest.find(n1)
        t2 = forest.find(n2)
        if t1 != t2:
            mst.append€
            sz -= 1
            if sz == 0:
                return mst

            forest.union(t1, t2)
```

Izvor: Anonymous_36, 2010.

```
if __name__ == „__main__“:

 nodes = list(„ABCDEFGHIJK“)
 edges = [(„A“, „D“, 2), („A“, „C“, 3), („A“, „B“, 4),
          („B“, „C“, 2), („B“, „F“, 5),
          („C“, „E“, 1),
          („D“, „G“, 1),
          („E“, „H“, 2), („E“, „F“, 2),
          („F“, „K“, 4),
          („G“, „I“, 8),
          („H“, „K“, 3), („H“, „I“, 6),
          („I“, „J“, 7),
          („J“, „K“, 3)]

print(„Kruskalov algoritam.“)
print(„Ispis:“, kruskal(nodes, edges))
```

Opis rješenja:

Prvo je definirano dodavanje paketa *from operator import itemgetter* koja vraća objekt koji se može pozivati i koji uzima stavku iz njenog operanda pomoću operandove *__getitem__()* metode. Ako je specificirano više stavki, vraća se zbir vrijednosti pretraživanja. Potom je definirana klasa *class DisjointSet(dict):* koja sadrži riječnik i funkcije *add(), find() i union().* Prethodno definirane operacije su pomoćne, kako bi se omogućilo izvršenje Kruskalovog algoritma. Funkcija add() dodaje stavke u riječnik. Operacija find() pronalazi roditelja (početni čvor) od kojeg će se početi izvršavati Kruskalov algoritam, dok funkcija union() spaja dvije stavke jer je potrebno da se u konačnici dobije stablo (Union se provodi izmjenom roditelja korijenskog čvora). Potom je implementirana funkcija *def kruskal(nodes, edges):* koja sadrži čvorove te rubove. Podaci se spremaju u skup, a minimalno razgranato stablo je u početku prazno. Nakon toga se dodaju čvorovi u šumu (koji se spremaju u skup), a potom se smanjuju isti za jedan. Kod funkcije *kruskal()* se još sortiraju težine (rubovi) od najmanje prema najvećoj vrijednosti. Pošto se radi o neusmjerenom grafu, gleda se i jedna i druga težina, tj. gledaju se dva čvora. Kada se nađe najmanja vrijednost, ona se dodaje u stablo. Nakon toga se poziva funkcija koja spaja sve čvorove (njihove težine) u stablo. Postupak je gotov kada se svi čvorovi nalaze u stablu. Unutar funkcije *if __name__ == „__main__“:* su definirani čvorovi (nodes) u obliku liste te rubovi (edges). Nakon toga slijedi prikaz ispisa Kruskalovog algoritma te pozivanje funkcije *kruskal* koja sadrži čvorove te rubove, koji su prethodno definirani.

Prikaz rješenja:

```
Kruskalov algoritam.

Ispis: [('C', 'E', 1), ('D', 'G', 1), ('A', 'D', 2), ('B', 'C', 2), ('E', 'H', 2), ('E', 'F', 2), ('A', 'C', 3), ('H', 'K', 3), ('J', 'K', 3), ('H', 'I', 6)]
```

6.3.3 Implementacija pomoću vektora

Prikaz implementacije pomoću vektora u programskom jeziku C++.

Datoteka kruskal.cpp:

```
#include <iostream>
#include <vector>
#include <utility>
#include <algorithm>
using namespace std;

const int MAX = 1000;
int id[MAX], nodes, edges;
pair <long long, pair<int, int> > p[MAX];

void init()
 {
     for(int i = 0;i < MAX;++i)
         id[i] = i;
 }

int root(int x)
 {
     while(id[x] != x)
     {
         id[x] = id[id[x]];
         x = id[x];
     }
     return x;
 }

void union1(int x, int y)
 {
     int p = root(x);
     int q = root(y);
     id[p] = id[q];
 }

long long kruskal(pair<long long, pair<int, int> > p[])
 {
     int x, y;
     long long cost, minimumCost = 0;
     for(int i = 0;i < edges;++i)
     {
         x = p[i].second.first;
         y = p[i].second.second;
         cost = p[i].first;
```

```
        if(root(x) != root(y))
        {
            minimumCost += cost;
    cout<<x<<“ --→ „<<y<<“ :“<<p[i].first<<endl;
            union1(x, y);
        }
    }
    return minimumCost;
 }

int main()
{
    int x, y;
    long long weight, cost, minimumCost;
    init();
    cout <<“Unesite cvorove i rubove:“<<endl;
    cin >> nodes >> edges;

    for(int i = 0;i < edges;++i)
    {
        cout<<“Unesite vrijednost X, Y i rubove:“<<endl;
    cin >> x >> y >> weight;
        p[i] = make_pair(weight, make_pair(x, y));
    }

    sort(p, p + edges);
    minimumCost = kruskal(p);
    cout <<“Minimalni trosak je: „<< minimumCost << endl;
    return 0;
 }
```

Izvor: Anonymous_37, n_d

Opis koda:

U ovom primjeru su umjesto slova korišteni brojevi što se tiče čvorova. Konkretno, umjesto A, B, C, D, E, F, G, H, I, J i K su upotrjebljeni 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 i 10. Prvo su uključene biblioteke *vector, utility, algorithm* te *iostream* kako bi se moglo raditi s vektorima (*vector*), parovima (*make_pair* te *pair*) te s rasponima elemenata (*algorithm,* funkcija *max*). Što se tiče parova to su zapravo objekti koji predstavljaju dvije vrijednosti različitih vrsta. Sama funkcija *make_pair* konstruira parni objekt, dok *pair* predstavlja vrijednost para. Rasponi elemenata se prikazuju putem iteratora ili pokazivača. Redak *int id[MAX], nodes, edges;* označava provjeru nadređenog čvora (prvog, roditelja). Funkcija *void init()* inicijalizira nadređeni niz, dok u funkciji *root()* redak *while(id[x] != x)* označava slučaj kada *x* nije sam roditelj, pa se mora ažurirati nadređeni. Na kraju te funkcije se onda vraća nadređeni pomoću naredbe *return x;.*

Sljedeća funkcija *void union1(int x, int y)* služi za uniju, dok funkcija *long long kruskal(pair<long long, pair<int, int> > p[])* služi za pronalaženje rubova kod minimalnog razgranatog stabla te njegovog troška (cijene). Redak u toj funkciji *cout<<x<<" ----> "<<y<<" :"<<p[i].first<<endl;* služi za ispis rubova koji se nalaze u minimalnom razgranatom stablu. Potom, unutar *int main()* (unutar *for*), se unose čvorovi te troškovi rubova. Pomoću *sort(p, p + edges);* se sortiraju rubovi prema njihovoj cijeni, pa se ispiše rezultat pomoću *cout <<"Minimum cost is "<< minimumCost << endl;*.

Prikaz rješenja:

```
Unesite cvorove i rubove:

11

15

Unesite vrijednost X, Y i rubove:

0

3

2

Unesite vrijednost X, Y i rubove:

0

2

3

Unesite vrijednost X, Y i rubove:

0

1

4

Unesite vrijednost X, Y i rubove:

1

2

2
```

```
Unesite vrijednost X, Y i rubove:
1
5
5
Unesite vrijednost X, Y i rubove:
2
4
1
Unesite vrijednost X, Y i rubove:
3
6
1
Unesite vrijednost X, Y i rubove:
4
7
2
Unesite vrijednost X, Y i rubove:
4
5
2
Unesite vrijednost X, Y i rubove:
5
10
4
Unesite vrijednost X, Y i rubove:
6
8
```

```
8
Unesite vrijednost X, Y i rubove:
7
10
3
Unesite vrijednost X, Y i rubove:
7
8
6
Unesite vrijednost X, Y i rubove:
8
9
7
Unesite vrijednost X, Y i rubove:
9
10
3
2 ----> 4 :1
3 ----> 6 :1
0 ----> 3 :2
1 ----> 2 :2
4 ----> 5 :2
4 ----> 7 :2
0 ----> 2 :3
7 ----> 10 :3
9 ----> 10 :3
7 ----> 8 :6
```

```
Minimalni trosak je: 25
```

7. Algoritmi najkraćeg puta

Problem najkraćeg puta se krije u pronalaženju putanje između dva vrha u grafu tako da je ukupni zbroj težina minimalan. (Abdelnabi, 2019b.).

U algoritme najkraćeg puta spadaju:

1. Bellman Ford algoritam.
2. Dijkstrin algoritam.
3. Floyd- Warshall algoritam.
4. Johnson algoritam.

7.1 Najmanji (najkraći) putevi

Većina ljudi je svjesna problema najkraćeg puta, ali njihovo poznavanje počinje i završava kada se razmatra najkraći put između dvije točke, *A* i *B*. Međutim, za računalne znanstvenike taj problem ima drugačiji preokret, budući da mogu biti potrebni različiti algoritmi u rješavanju različitih problema (prethodno navedeni).

Za jednostavnost, algoritmi najkraćeg puta djeluju na graf koji je sastavljen od vrhova i rubova koji ih povezuju (primjerice, usmjereni graf, ponderirani graf, itd.). Te razlike određuju koji će algoritam funkcionirati bolje od drugog za određene vrste grafa. (Anonymous_10, 2019.).

Algoritmi najkraćeg puta imaju različite namjene (primjerice, softver za planiranje rute, Google Maps).

7.2 Problemi algoritma najkraćeg puta

U ulazu se definira usmjereni težinski graf $G=(V, E)$, $u, v \in E$, a u izlazu se definira iznos najkraćeg puta od vrha *u* do vrha *v*.

Ovdje postoje dva konkretna različita slučaja, a to su:

1. Kada je graf acikličan.
2. Kada graf nema negativne cikluse.

U slučaju da je graf acikličan, onda se u njemu može definirati topološko sortiranje te se mogu gledati redom po topološkom sortu od vrha *u* pa do vrha *v*.

Ulaz kod acikličnog grafa: $G=(V, E), u, v \in V$. Izlaz je iznos najkraćeg puta od vrha *u* do vrha *v*.

Pseudokod:

Treba napraviti topološki sort grafa *G* i pospremiti ga u polje *v*;

Svakom vrhu *s* se treba dodati oznaka $o[s]=+\infty$;

$O(u)=0;$

Za svaki vrh *s* od *u* do *v* treba raditi sljedeće;

Za svaki vrh *t* takav da je $(s, t) \in E$ treba raditi sljedeće;

Ako je $o(t)>o(s)+|s,t|$ onda je $o(t)=o(s)+|s, t|;$

Na kraju treba vratiti *o(v).*

Složenost problema najkraćeg puta je $O(n^2)$.

Kod problema najkraćeg puta postoji graf s pozitivnim težinama bridova. Naravno, ovaj graf može imati cikluse, ali su mu sve težine bridova pozitivne. Tada se trebaju gledati kroz koje se sve čvorove prošlo te se ne vraćati u njih. Za prethodno navedeni scenarij se koristi Dijkstrin algoritam.

Kao ulaz se koristi usmjereni težinski graf $G=(V, E), u, v \in E$ s pozitivnim težinama bridova, dok se za izlaz koristi iznos najkraćeg puta od vrha *u* do vrha *v*.

7.2.1 Najkraći putevi s jednim izvorom

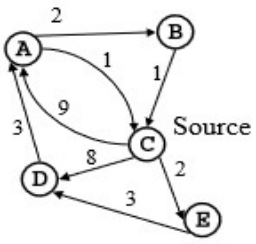
Algoritam najkraćeg puta s jednim izvorom (problem najkraćeg puta s jednim izvorom) pripada matematičkoj grani koja se zove teorija grafova. Isti se algoritam primjenjuje najčešće u matematici iz razloga jer je čest kod problema u stvarnom životu.

Ovdje je dat graf *G=(V,E)* i "izvorni" vrh *s* u *V*. Kao izlaz treba pronaći puteve minimalne cijene od *s* do svakog vrha u *V*.

Preciznije, algoritam za najkraći put jednog izvora izračunava najkraću (ponderiranu) putanju od čvora do svih ostalih čvorova u grafu.

Algoritam najkraćeg puta s jednim izvorom je postao istaknut u isto vrijeme kao i algoritam najkraćeg puta. Dijkstrin algoritam može djelovati kao implementacija za oba problema.

Slika 34.: Prikaz grafa s jednim izvorom



Izvor: Wolfman, 2000., str. 2

Open Shortest Path First (OSPF) je protokol usmjeravanja za IP mreže. Isti koristi Dijkstrin algoritam za pomoć u otkrivanju promjena u topologiji (primjerice, neuspjesi veze). Ovdje dolazi do nove strukture usmjeravanja u sekundama.

Delta stepping ne podržava negativne težine. Algoritam pretpostavlja da dodavanje odnosa putanji nikada ne može napraviti kraći put (narušen negativnim težinama). (Anonymous_11, n_d).

7.2.2 Bellman- Ford algoritam

Bellman- Ford algoritam traži najkraći put za jedan početni čvor u težinski usmjerenom grafu.

Bellman- Ford algoritam radi s negativnim vrijednostima. Isti je jednostavniji od Dijkstrinog algoritma i dobro se slaže s distribuiranim sustavima.

Kao i Dijkstrin algoritam, Bellman- Fordov se zasniva na metodi iteracije (opušta procjene pronalaženjem novih puteva koji su kraći od prethodno precijenjenih puteva), po kome se aproksimacija točne udaljenosti postepeno zamjenjuje točnijom

vrijednošću, sve dok se konačno ne dostigne optimalno rješenje. U oba algoritma, približna udaljenost do svakog čvora je uvijek precijenjena vrijednost točne udaljenosti (precjenjuje duljinu puta od početnog vrha do svih ostalih vrhova), i zamjenjuje se minimumom svoje stare vrijednosti i dužine novootkrivenog puta. Radeći to više puta za sve vrhove, može se jamčiti da je krajnji rezultat optimiziran.

Rubovi s negativnim težinama se mogu naći u mnogim primjenama grafova, što ovaj algoritam čini veoma korisnim. Međutim, ukoliko graf sadrži negativni ciklus (primjerice, ciklus čiji rubovi imaju negativan zbroj), onda ne postoji najkraći put, zato što u tom slučaju svaki put može postati kraći ako se još jednom prođe kroz negativni ciklus. U takvim slučajevima, Bellman- Fordov algoritam može otkriti negativne cikluse i ukazati na njihovo postojanje, ali ne može precizno prikazati najkraći put ukoliko se do negativnog ciklusa može doći iz početnog čvora. (Bačlija, n_d).

Vremenska složenost Bellman- Forda je *O(VE),* što je više od Dijkstrinog algoritma. (geeksforgeeks_7, n_d).

Ovaj algoritam, kao i drugi problemi s dinamičkim programiranjem, izračunava najkraći put krećući se od dolje prema gore. Točnije, prvo izračunava najkraće udaljenosti koje imaju najviše jedan rub na putu. Sljedeće, izračunava najkraće puteve s najviše dva ruba, itd. Nakon *i*-te iteracije vanjske petlje, izračunavaju se najkraći putevi s najviše *i* rubova. Tako valja naglasiti da će najviše biti $|V|-1$ rubova na putu, zato se vanjska petlja pokreće $|v|-1$ puta. Stoga, ideja se krije u tome da nema negativnog ciklusa težine, ako se izračuna najkraći put s najviše *i* rubova. Također, iteracija će tada nad svim rubovima jamčiti najkraći put s najviše $(i+1)$ rubova.

Slijede dva svojstva Bellman- Fordovog algoritma:

1. Negativne težine nalaze se u različitim primjenama grafa (primjerice, umjesto da se plaćaju troškovi nekog puta, može se dobiti neka prednost ako se slijedi put).
2. Bellman- Ford algoritam radi bolje od Dijkstrinog algoritma kod distribuiranih sustava. Za razliku od Dijkstrinog algoritma, gdje se mora pronaći minimalna vrijednost svih rubova, kod Bellman- Forda se rubovi gledaju jedan po jedan kada se pronalazi minimalna vrijednost.

Negativni rubovi težine u početku mogu izgledati beskorisno, ali mogu objasniti mnoge pojave (primjerice, novčani tok, toplinu koja se oslobađa ili apsorbira u kemijskoj reakciji, i sl.). Isto tako, ako postoje različiti načini za dosezanje od jedne kemikalije *A* do druge kemikalije *B*, svaka metoda će tada imati podreakcije koje uključuju toplinu i apsorpciju. Ako se želi pronaći skup reakcija gdje je potrebna minimalna energija, tada će se morati biti sposoban faktorirati apsorpciju topline kao negativne težine i dispipaciju topline kao pozitivne težine.

Rubovi negativne težine mogu stvoriti negativne cikluse težine (misli se na ciklus koji može smanjiti ukupnu udaljenost puta kada se vraća na istu točku). Algoritam najkraćeg puta (primjerice, Dijkstrin algoritam) koji ne mogu otkriti takav ciklus mogu dati pogrešan rezultat jer mogu proći kroz negativni ciklus i pritom smanjiti dužinu puta.

Što se tiče pseudokoda, mora se održavati udaljenost svakog puta. Konkretno, valja pohraniti u nizu veličinu *v*, gdje je *v* broj vrhova. Također, cilj je dobiti najkraći put, a ne samo znati dužinu najkraćeg puta. Za to se mapira svaki vrh na vrh koji je posljednji put ažurirao svoju dužinu puta. Kada je algoritam gotov, može se vratiti od odredišnog toka do izvorne točke kako bi se pronašla putanja.

Slika 35.: Prikaz pseudokoda Bellman- Fordovog algoritma

```
function bellmanFord(G, S)
        for each vertex V in G
                        distance[V] <- infinite
                        previous[V] <- NULL
        distance[S] <- 0
        for each vertex V in G
                for each edge (U,V) in G
                        tempDistance <- distance[U] + edge_weight(U, V)
                        if tempDistance < distance[V]
                            distance[V] <- tempDistance
                            previous[V] <- U

        for each edge (U,V) in G
                If distance[U] + edge_weight(U, V) < distance[V}
                        Error: Negative Cycle Exists

        return distance[], previous[]
```

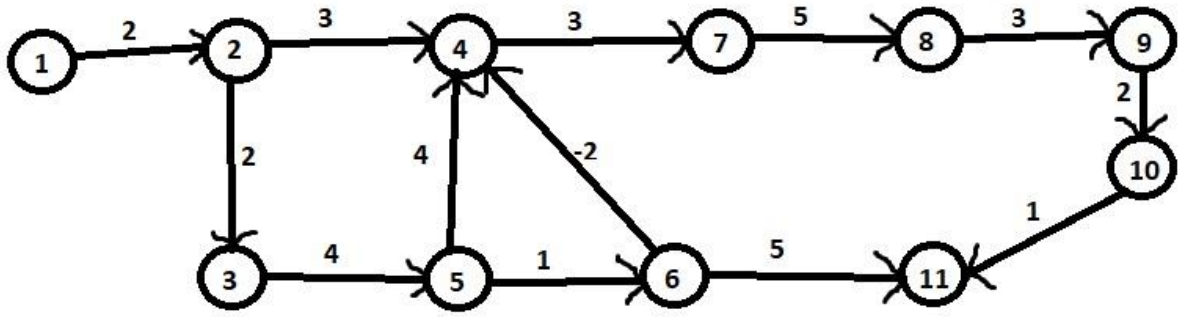
Izvor: Anonymous_12, n_d

Ovaj algoritam se prvenstveno koristi kod mreža, kako bi se odredio optimalni put mrežnih paketa u nekoj računalnoj mreži.

7.2.2.1 Primjer

Zadatak nalaže da se mora pronaći minimalni put od vrha *1* (zbrajanje istih od početnog vrha do krajnjeg) pa do kraja grafa (svi vrhovi moraju biti obuhvaćeni), pritom koristeći Bellman- Ford algoritam.

Slika 36.: Prikaz primjera Bellman- Ford algoritma



Izvor: Vlastiti rad

Rješenje ovog primjera nalaže da se zbroje težine koje se nalaze na grafu (brojevi), tako da pritom zbroj bude minimalan (gleda se najkraći put).

Rješenje je sljedeće:

Dakle, postoji 11 vrhova, a minimalno razgranato stablo je |*E*|=*11.*

Rub *1, 1 =0*

Rub *1, 2 = 0+2=2*

Rub *1, 2, 3= 0+2+2=4*

Rub *1, 2, 4= 0+2+3=5*

Rub *1, 2, 3, 5= 0+2+2+4=8*

Rub 1, 2, 3, 5, 6= 0+2+2+4+1=9

Rub *1, 2, 4, 7= 0+2+3+3=8*

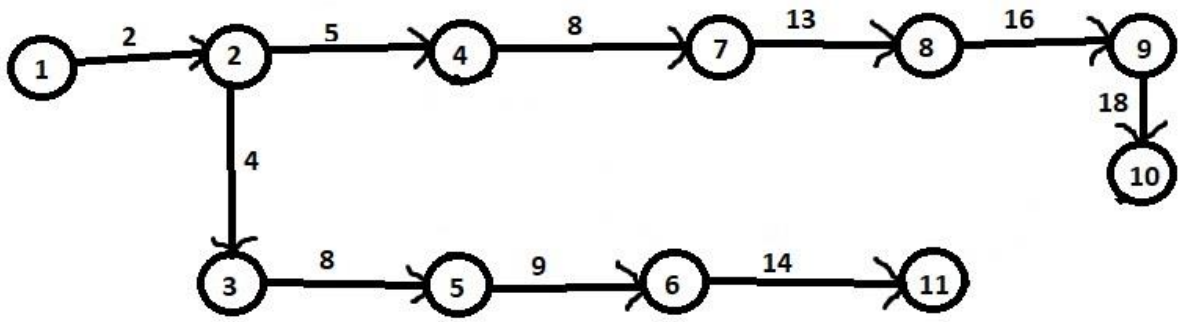
Rub *1, 2, 4, 7, 8= 0+2+3+3+5= 13*

Rub *1, 2, 4, 7, 8, 9= 0+2+3+3+5+3= 16*

Rub *1, 2, 4, 7, 8, 9, 10= 0+2+3+3+5+3+2= 18*

Rub *1, 2, 3, 5, 6, 11= 0+2+2+4+1+5= 14*

Slika 37.: Prikaz rješenja Bellman- Ford algoritma



Izvor: Vlastiti rad

7.2.2.2 Implementacija u Pythonu

Zadatak je, a ujedno i problem, pronaći najkraću udaljenost od svih vrhova iz izvorne točke u ponderiranom usmjerenom grafu koji može imati negativne rubne težine. Da bi problem bio dobro definiran, ne bi trebalo biti nikakvih ciklusa u grafu s negativnom ukupnom težinom.

Ovdje je prikazana implementacija Bellman- Ford algoritma u Pythonu.

Datoteka main.py:

```
class Graph:
    def __init__(self):
        self.vertices = {}

    def add_vertex(self, key):
        vertex = Vertex(key)
        self.vertices[key] = vertex

    def get_vertex(self, key):
        return self.vertices[key]

    def __contains__(self, key):
        return key in self.vertices
```

```
    def add_edge(self, src_key, dest_key, weight=1):
        self.vertices[src_key].add_neighbour(self.vertices[dest_key], weight)

    def does_edge_exist(self, src_key, dest_key):
        return
self.vertices[src_key].does_it_point_to(self.vertices[dest_key])

    def __len__(self):
        return len(self.vertices)

    def __iter__(self):
        return iter(self.vertices.values())
```

Izvor: Anonymous_38, n_d

```
    def remove_vertex(self, n):
      for n1 in self.get_(n):
        if n1 != key:
          self.graph[n1].get_neighbours.remove(n)
          del self._weight[self._edge(n, n1)]
          del self.graph[n]
```

Izvor: Anonymous_39, 2015.

```
    def remove_edge(self, n1, n2):
      try:
        if self != None:
          del g[n1][n2]
          print ('Brisanje\n', n1[0], n1[1])
          return 1
      except:
            return None

    def is_empty(self, key):

      if self.vertices[key]==0:
        return 1
      else:
        return 0

class Vertex:
    def __init__(self, key):
        self.key = key
        self.points_to = {}

    def get_key(self):
        return self.key

    def add_neighbour(self, dest, weight):
        self.points_to[dest] = weight

    def get_neighbours(self):
        return self.points_to.keys()
```

Izvor: Anonymous_38, n_d

```
    def get_weight(self, dest):
        return self.points_to[dest]

    def does_it_point_to(self, dest):
        return dest in self.points_to

def bellman_ford(g, source):
    distance = dict.fromkeys(g, float('inf'))
    distance[source] = 0

    for _ in range(len(g) - 1):
        for v in g:
            for n in v.get_neighbours():
                distance[n] = min(distance[n], distance[v] + v.get_weight(n))

    return distance

g = Graph()
print('Izbornik')
print('dodaj vrh <key>')
print('dodaj rub <src> <dest> <weight>')
print('ukloni vrh <n>')
print('ukloni rub <n1> <n2>')
print('prazan <key>')
print('bellman-ford <source vertex key>')
print('prikaz')
print('izlaz')

while True:
    do = input('Sto zelite uciniti? ').split()

    operation = do[0]
    if operation == 'dodaj':
        suboperation = do[1]
        if suboperation == 'vrh':
            key = int(do[2])
            if key not in g:
                g.add_vertex(key)
            else:
                print('Vrh vec postoji.')
        elif suboperation == 'rub':
            src = int(do[2])
            dest = int(do[3])
            weight = int(do[4])
            if src not in g:
                print('Vrh {} ne postoji.'.format(src))
            elif dest not in g:
                print('Vrh {} ne postoji.'.format(dest))
```

```
            else:
                if not g.does_edge_exist(src, dest):
                    g.add_edge(src, dest, weight)
                else:
                    print('Rub vec postoji.')

    elif operation=='prazan':
      if key in g:
        g.is_empty(key)
        print('Graf nije prazan.')
      else:
        print('Graf je prazan.')

    elif operation == 'bellman-ford':
        key = int(do[1])
        source = g.get_vertex(key)
        distance = bellman_ford(g, source)
        print('Udaljenost od {}: '.format(key))
        for v in distance:
            print('Udaljenost do {}: {}'.format(v.get_key(), distance[v]))
        print()

    elif operation == 'prikaz':
        print('Vrhovi: ', end='')
        for v in g:
            print(v.get_key(), end=' ')
        print()

        print('Rubovi: ')
        for v in g:
            for dest in v.get_neighbours():
                w = v.get_weight(dest)
                print('(src={}, dest={}, weight={}) '.format(v.get_key(),
                                                             dest.get_key(),
w))
        print()

    elif operation == 'izlaz':
        break
```

Izvor: Anonymous_38, n_d

Opis koda:

Prvo je definirana klasa *Graph()* koja sadrži sljedeće funkcije: *def __init__(self), def add_vertex(self, key), def get_vertex(self, key), def __contains__(self, key), def add_edge(self, src_key, dest_key, weight=1), def does_edge_exist(self, src_key, dest_key), def __len__(self), def __iter__(self), def remove_vertex(self, n), def remove_edge(self, n1, n2) i def is_empty(self, key).* Prva spomenuta funkcija sadrži rječnik koji ukazuje na odgovarajući vrh objekta (inicijalizira prazan graf na početku). Druga dodaje vrh s odgovarajućom vrijednosti (1,…, 11). Nadalje, treća funkcija vraća vrh s odgovarajućom vrijednosti. Potom *add_edge()* dodaje rub od *src_key* do *dest_key* zadane težine. Sljedeća funkcija vraća istinu (1) ako postoji rub od *src_key* do *dest_key*. Pomoćne funkcije *__len__ i __iter__* vraćaju dužinu vrha, odnosno iterator na same vrijednosti vrha. Funkcije *remove_vertex* i *remove_edge*, uklanjaju vrh i sve rubove koji su s njim incidentni ili brišu brid iz grafa. Posljednja definirana funkcija *is_empty* provjerava da li je dani graf prazan ili nije, te ispisuje poruku. Druga definirana klasa je Vertex() koja ima sljedeće operacije: *def __init__(self, key), def get_key(self), def add_neighbour(self, dest, weight), def get_neighbours(self), def get_weight(self, dest) i def does_it_point_to(self, dest).* Na kraju je definiran algoritam Bellman- Ford. Prva operacija definira prazan vrh (vrh koji nema nikakve vrijednosti). Potom sljedeća funkcija vraća vrijednost koja odgovara samom vrhu. Nakon toga se stvara vrh točke (*dest*) s danom težinom ruba. Što se tiče sljedeće operacije, ista vraća sve vrhove koji su povezani s danim vrhom. Drugim riječima, ona vraća susjede danog vrha. Sljedeća pomoćna funkcija vraća težinu ruba vrha (*dest*). Posljednja pomoćna funkcija vraća istinu ako je vrh dane točke (*dest*). Algoritam Bellman- Ford vraća udaljenost, gdje je udaljenost *[v]* minimalna udaljenost od izvora do *r*. To će potom vratiti rječničku udaljenosti. *G* ovdje predstavlja graf objekt koji može imati negativne rubne težine. Izvor je sami *Vertex* objekt u *g*. Na kraju je prikazan kratki izbornik gdje je korisniku omogućeno izvršavanje određenih operacija kod samog grafa (primjerice, dodavanje vrha/ ruba, prikaz tih podataka, provjera je li graf prazan ili nije, izvršenje samog algoritma Bellman- Ford, itd.).

Prikaz primjera rješenja:

```
Izbornik

dodaj vrh <key>

dodaj rub <src> <dest> <weight>

ukloni vrh <n>

ukloni rub <n1> <n2>

prazan <key>

bellman-ford <source vertex key>

prikaz

izlaz

Sto zelite uciniti? dodaj vrh 1

Sto zelite uciniti? dodaj vrh 2

Sto zelite uciniti? dodaj vrh 3

Sto zelite uciniti? dodaj vrh 4

Sto zelite uciniti? dodaj vrh 5

Sto zelite uciniti? dodaj vrh 6

Sto zelite uciniti? dodaj vrh 7

Sto zelite uciniti? dodaj vrh 8

Sto zelite uciniti? dodaj vrh 9

Sto zelite uciniti? dodaj vrh 10

Sto zelite uciniti? dodaj vrh 11

Sto zelite uciniti? dodaj rub 1 2 2

Sto zelite uciniti? dodaj rub 2 3 2

Sto zelite uciniti? dodaj rub 2 4 3

Sto zelite uciniti? dodaj rub 3 5 4

Sto zelite uciniti? dodaj rub 5 4 4
```

```
Sto zelite uciniti? dodaj rub 5 6 1

Sto zelite uciniti? dodaj rub 6 4 -2

Sto zelite uciniti? dodaj rub 4 7 3

Sto zelite uciniti? dodaj rub 7 8 5

Sto zelite uciniti? dodaj rub 8 9 3

Sto zelite uciniti? dodaj rub 9 10 2

Sto zelite uciniti? dodaj rub 10 11 1

Sto zelite uciniti? dodaj rub 6 11 5

Sto zelite uciniti? bellman-ford 1

Udaljenost od 1:

Udaljenost do 1: 0

Udaljenost do 2: 2

Udaljenost do 3: 4

Udaljenost do 4: 5

Udaljenost do 5: 8

Udaljenost do 6: 9

Udaljenost do 7: 8

Udaljenost do 8: 13

Udaljenost do 9: 16

Udaljenost do 10: 18

Udaljenost do 11: 14


Sto zelite uciniti? prikaz

Vrhovi: 1 2 3 4 5 6 7 8 9 10 11

Rubovi:

(src=1, dest=2, weight=2)

(src=2, dest=3, weight=2)
```

```
(src=2, dest=4, weight=3)

(src=3, dest=5, weight=4)

(src=4, dest=7, weight=3)

(src=5, dest=4, weight=4)

(src=5, dest=6, weight=1)

(src=6, dest=4, weight=-2)

(src=6, dest=11, weight=5)

(src=7, dest=8, weight=5)

(src=8, dest=9, weight=3)

(src=9, dest=10, weight=2)

(src=10, dest=11, weight=1)


Sto zelite uciniti? izlaz
```

7.2.2.3 Implementacija pomoću vektora

Ovdje je prikazana implementacija pomoću strukture podatka vektor u programskom jeziku C++.

Datoteka BellmanFord.cpp:

```cpp
#include <iostream>
#include <vector>
#include <iomanip>
#include <climits>
using namespace std;

struct Edge
{
  int source, dest, weight;
};

void printPath(vector<int> const &parent, int v)
{
  if (v < 0)
```

```
    return;

  printPath(parent, parent[v]);
  cout << v << " ";
}

void BellmanFord(vector<Edge> const &edges, int source, int N)
{
  int E = edges.size();

  vector<int> distance (N, INT_MAX);
  distance[source] = 0;

  vector<int> parent (N, -1);

  int u, v, w, k = N;

  while (--k)
  {
    for (int j = 0; j < E; j++)
    {
      u = edges[j].source, v = edges[j].dest;
      w = edges[j].weight;

      if (distance[u] != INT_MAX && distance[u] + w < distance[v])
      {
        distance[v] = distance[u] + w;
        parent[v] = u;
      }
    }
  }

  for (int i = 0; i < E; i++)
  {
    u = edges[i].source, v = edges[i].dest;
    w = edges[i].weight;

    if (distance[u] != INT_MAX && distance[u] + w < distance[v])
    {
      cout << "Pronadjen je negativan ciklus!!";
      return;
    }
  }

  for (int i = 0; i < N; i++)
  {
    cout << "Udaljenost od pocetnog vrha (1) do " << i << " je "
       << setw(2) << distance[i] << ". Put je [ ";
    printPath(parent, i); cout << "]" << '\n';
```

```
  }
}

int main()
{
  vector<Edge> edges =
  {
    { 1, 2, 2 }, { 2, 4, 3 }, { 4, 7, 3 }, { 7, 8, 5 },
    { 8, 9, 3 }, { 9, 10, 2 }, { 10, 11, 1 }, { 2, 3, 2 },
    { 3, 5, 4 }, { 5, 4, 4 }, { 5, 6, 1 }, { 6, 4, -2 },
    { 6, 11, 5 }
  };

  int N = 12;

  int source = 1;

  BellmanFord(edges, source, N);

  return 0;
}
```

Izvor: Anonymous_40, n_d

Opis koda:

Ovdje su uključene tri biblioteke kako bi se moglo raditi s danim strukturama *(#include<vector>, #include<iomanip> i #include<climits>).* Biblioteka vector služi za rad s vektorima, biblioteka iomanip služi za rad s parametričnim manipulatorima, dok climits služi za konstante s ograničenjima osnovnih integralnih tipova za konkretni primijenjeni sustav i prevoditelj (*INT_MAX*). Potom je definirana struktura podatka za prikaz rubova grafa (*struct Edge*) koja sadrži tri podatka (*int source, dest, weight*). Osim toga je prikazana funkcija koja služi za ispis puta zadanog vrha *v* iz izvora (*void printPath(vector<int> const &parent, int v*)) te funkcija za pokretanje samog algoritma Bellman-Ford (*void BellmanFord(vector<Edge> const &edges, int source, int N)*). Ova druga spomenuta funkcija broji rubove koji su prisutni u grafu (*int E = edges.size())* i *distance()* i *parent()* pohranjuju informacije o najkraćem putu (najmanje cijene/ puta). Stoga u početku svi vrhovi, osim izvora, imaju težinu beskonačnosti i nemaju roditelja (*vector<int> distance (N, INT_MAX), distance[source] = 0, vector<int> parent (N, -1), int u, v, w, k = N*). Naime, slijedi korak (*while*) koji se vrši *V-1* puta (*while (--k)).* Isti se vrši od ruba *u* do ruba *v* koji sadrži težinu *w* (*u = edges[j].source, v = edges[j].dest, w = edges[j].weight*) te ako se udaljenost do odredišta *v* može skratiti uzimanjem ruba

(*u->v*), onda se ažurira udaljenost na novu nižu vrijednost (*distance[v] = distance[u] + w*) te se postavlja *v* kao roditelj *u* (*parent[v] = u*). Nakon toga, unutar petlje *for*, se izvršava korak (N-ti put) kako bi se provjerilo ima li ciklusa s negativnom težinom (*for (int i = 0; i < E; i++)).* Dano definira rub od *u* do *v* koji ima *težinu w (u = edges[i].source, v = edges[i].dest, w = edges[i].weight*). Ako se udaljenost do odredišta *u* može skratiti uzimanjem ruba (*u->v*), onda je pronađen ciklus negativne težine te se ispisuje udaljenost vrhova te put. Unutar funkcije *int main()* je definiran vektor rubova (*vector<Edge> edges*), potom rub od *x* do *y* s težinom (primjerice, *{ 1, 2, 2 }),* postavljanje maksimalnog broja vrhova na grafu (*int N = 12*), postavljanje izvora na 1 (*int source = 1*) i izvođenje algoritma Bellman- Ford iz navedenog izvora (*BellmanFord(edges, source, N)).*

Prikaz rješenja:

```
Udaljenost od pocetnog vrha (1) do 1 je  0. Put je [ 1 ]

Udaljenost od pocetnog vrha (1) do 2 je  2. Put je [ 1 2 ]

Udaljenost od pocetnog vrha (1) do 3 je  4. Put je [ 1 2 3 ]

Udaljenost od pocetnog vrha (1) do 4 je  5. Put je [ 1 2 4 ]

Udaljenost od pocetnog vrha (1) do 5 je  8. Put je [ 1 2 3 5 ]

Udaljenost od pocetnog vrha (1) do 6 je  9. Put je [ 1 2 3 5 6]

Udaljenost od pocetnog vrha (1) do 7 je  8. Put je [ 1 2 4 7 ]

Udaljenost od pocetnog vrha (1) do 8 je 13. Put je [ 1 2 4 7 8]

Udaljenost od pocetnog vrha (1) do 9 je 16. Put je [ 1 2 4 7 8 9 ]

Udaljenost od pocetnog vrha (1) do 10 je 18. Put je [ 1 2 4 7 8 9 10 ]

Udaljenost od pocetnog vrha (1) do 11 je 14. Put je [ 1 2 3 5 6 11 ]
```

7.2.3 Najkraći putevi s jednim izvorom u usmjerenim acikličkim grafovima

Rubovi ponderiranog usmjerenog acikličkog grafa *G = (V, E)* prema topološkom tipu njegovih vrhova, se mogu izračunati pomoću najkraćih puteva iz jednog izvora u *Θ (V + E)* vremenu. Najkraći putevi su uvijek dobro definirani u usmjerenom acikličkom grafu, jer čak i ako postoje negativni rubovi, ne mogu postojati negativni ciklusi.

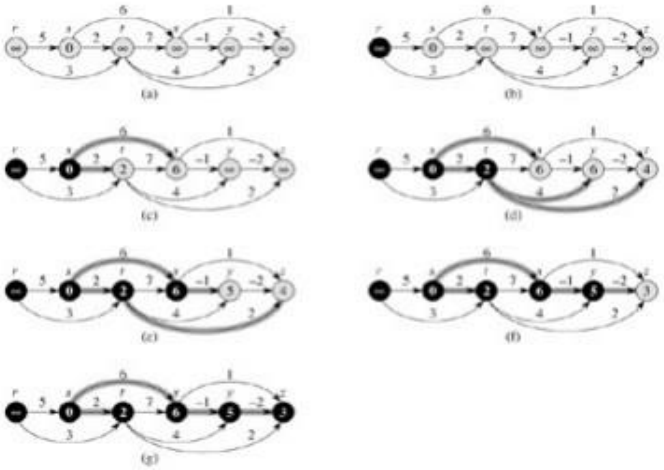
Algoritam započinje topološkim sortiranjem usmjerenog acikličkog grafa kako bi se nametnuo linearni poredak na vrhovima. Ako postoji put od vrha *u* do vrha *v*, onda *u* prethodi *v* u topološkom obliku. Izrađuje se samo jedan prelazak vrhova u topološki poredanom poretku. Kako se obrađuje svaki vrh, opušta se svaki rub koji napušta vrh.

Slika 38.: Prikaz algoritma koji započinje topološkim sortiranjem usmjerenog acikličkog grafa

```
DAG-SHORTEST-PATHS(G, w, s)
1  topologically sort the vertices of G
2  INITIALIZE-SINGLE-SOURCE(G, s)
3  for each vertex u, taken in topologically sorted order
4       do for each vertex v   Adj[u]
5             do RELAX(u, v, w)
```

Izvor: Cormen i sur., 2001.

Slika 39.: Prikaz izvršenja topološkog sortiranja usmjerenog acikličkog grafa



Izvor: Cormen i sur., 2001.

Slika prikazuje izvođenje algoritma za najkraće puteve u usmjerenom acikličkom grafu. Stoga su vrhovi topološki razvrstani s lijeva na desno. Izvorna točka je *s*. Vrijednosti *d* prikazane su unutar vrhova, a osjenčani rubovi označavaju *π* vrijednosti. Situacija prije prve iteracije petlje *(a) (for 3-5. (b) - (g)).* Situacija nakon svake iteracije petlje *(for 3-5).* Novo pocrnjeli vrh u svakoj iteraciji korišten je kao u toj iteraciji. Vrijednosti prikazane u dijelu *(g)* su konačne vrijednosti.

Vrijeme rada tog algoritma je lako analizirati. Topološka vrsta linije (jedan) može se izvesti u vremenu *Θ (V + E).* Poziv *INITIALIZESINGLE-SOURCE* u retku dva traje *Θ (V)*. Postoji jedna iteracija po vrhu u petlji *(for 3-5).* Za svaki vrh, svaki od rubova koji napuštaju vrh, ispituje se točno jednom. Dakle, postoji ukupno |*E*| iteracija za petlju

redova *4-5* (ovdje se koristi analiza agregata.) Budući da svaka iteracija unutarnje *for* petlje traje *Θ(1),* ukupno vrijeme rada je *Θ (V + E)*, što je linearno u veličini reprezentacije popisa susjedstva grafa.

Sljedeći teorem pokazuje da *DAG SHORTEST-PATHS* postupak ispravno izračunava najkraće puteve. (Cormen i sur., 2001.).

Teorem: ako ponderirani, usmjereni graf $G = (V, E)$ ima izvorni vrh s i bez ciklusa, onda na završetku *DAG-SHORTEST-PATHS* postupka, $d[v] = \delta(s, v)$ za sve vrhove $v \, V$, a prethodni podgraf G_π je stablo najkraćeg puta.

Dokaz: prvo se pokazuje da je $d[v] = \delta(s, v)$ za sve točke $v \, V$ pri završetku. Ako v nije dostupno iz s, tada $d[v] = \delta(s, v) = \infty$ po svojstvu bez puta. Pretpostavlja se da je v dostupno iz s, tako da postoji najkraći put $p = v_0, v_1, ..., v_k,$ gdje je $v_0 = s$ i $v_k = v$. Budući da se obrađuju vrhovi topološki poredanog reda, rubovi se obrađuju (opisani su redom $(v_0, v_1), (v_1, v_2), ..., (v_{k-1}, v_k)$). Svojstvo relaksacije puta implicira da $d[v_i] = \delta(s, v_i)$ na završetku za $i = 0, 1, ..., k$. Konačno, svojstvo prethodnika-podgrafa, G_π je stablo najkraćih puteva.

Zanimljiva primjena ovog algoritma nastaje u određivanju kritičnih putova u Pert analizi. Rubovi predstavljaju zadatke koje treba izvesti, a utezi (težine) rubova predstavljaju vremena potrebna za izvođenje određenih poslova. Ako rub (u, v) ulazi u vrh v i rub (v, x) napušta v, tada se posao (u, v) mora izvršiti prije posla (v, x). Put kroz ovaj usmjereni graf predstavlja slijed poslova koji se moraju izvršiti u određenom redoslijedu. Kritična staza je najduža staza kroz usmjereni graf koja odgovara najduljem vremenu za izvođenje redoslijeda zadataka. Težina kritične staze je donja granica ukupnog vremena za obavljanje svih poslova. Može se pronaći i kritički put: (Cormen i sur., 2001.).

1. Negiranje težina rubova i pokretanje *DAG-SHORTEST-PATH*-ova.
2. Pokretanje *DAG-SHORTEST-PATHS*, s izmjenom koja zamjenjuje *"∞"* s *"-∞"* u drugom retku *INITIALIZE-SINGLE-SOURCE* i *">"* za *"<"* u *RELAX* postupku.

7.3 Dijkstrin algoritam

Često se u čvoru mora naći najkraći put između dva čvora, stoga se za to koristi Dijkstrin, odnosno Floydov algoritam. Vrh grafa mogu primjerice biti, gradovi, a rubovi mogu sadržavati udaljenosti između njih.

Dijkstrin algoritam je zapravo proširenje Primovog algoritma. Ovaj algoritam gradi stablo koje se sastoji od bridova koji čine minimalne puteve od početnog vrha pa do ostalih vrhova. U svakom koraku se dodaje jedan vrh u stablo, i to onaj vrh koji ima minimalnu udaljenost od početnog vrha.

Dakle, ovaj algoritam može izračunati najkraće udaljenosti između jednog grada i svih drugih gradova. Rubovi mogu opisivati troškove, udaljenosti ili neku drugu mjeru koja je korisna za korisnika. (Anonymous_13, 2014.).

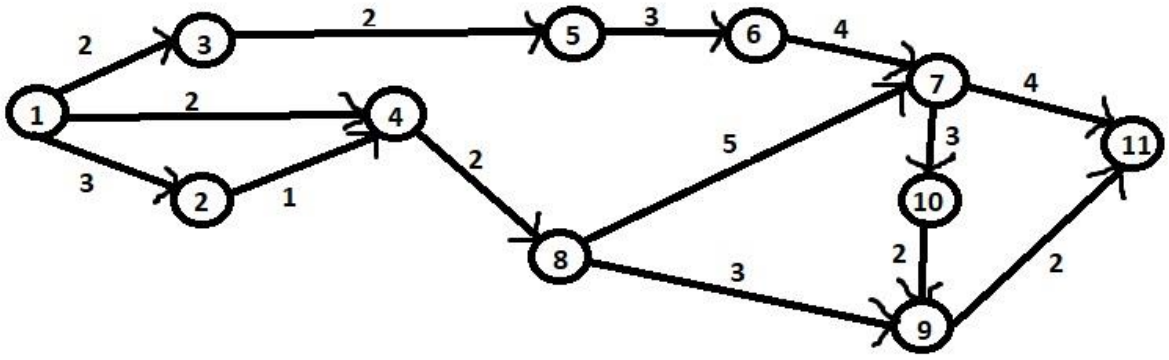
Važno ograničenje Dijkstrinog algoritma je da svi rubni troškovi (ili rubne oznake) ne smiju biti negativne.

Ovim algoritmom se jednostavno dobivaju najmanje udaljenosti od jednog vrha do svih ostalih vrhova u grafu. Međutim ako su potrebne vrijednosti najmanjih udaljenosti između svaka dva vrha u grafu trebao bi se ovaj algoritam ponoviti onoliko puta koliko ima vrhova u grafu. Nije teško primijetiti da je to veoma zahtjevan posao i zato se koristi drugi algoritam iz teorije grafova. Riječ je o Floydovom algoritmu s pomoću kojeg se veoma jednostavno mogu pronaći vrijednosti najmanjih udaljenosti između svih parova vrhova u grafu. (Marinović, n_d, str. 18).

7.3.1 Primjer

Zadatak nalaže da se mora pronaći minimalni put od vrha *a* pa do kraja grafa (svi vrhovi moraju biti obuhvaćeni), pritom koristeći Dijkstrin algoritam.

Slika 40.: Prikaz primjera Dijkstrinog algoritma



Izvor: Vlastiti rad

Rješenje ovog primjera nalaže da se zbroje težine (od početnog do završnog vrha) koje se nalaze na grafu (brojevi), tako da pritom zbroj bude minimalan.

Rješenje je sljedeće:

Dakle, postoji 11 vrhova, a minimalno razgranato stablo je |*E*|*=11.*

Rub *1, 1 =0*

Rub *1, 3 =0+2=2*

Rub *1, 4=0+2=2*

Rub *1, 2=0+3=3*

Rub *1, 3, 5=0+2+2=4*

Rub *1, 4, 8=0+2+2=4*

Rub *1, 3, 5, 6=0+2+2+3=7*

Rub *1, 4, 8, 9=0+2+2+3=7*

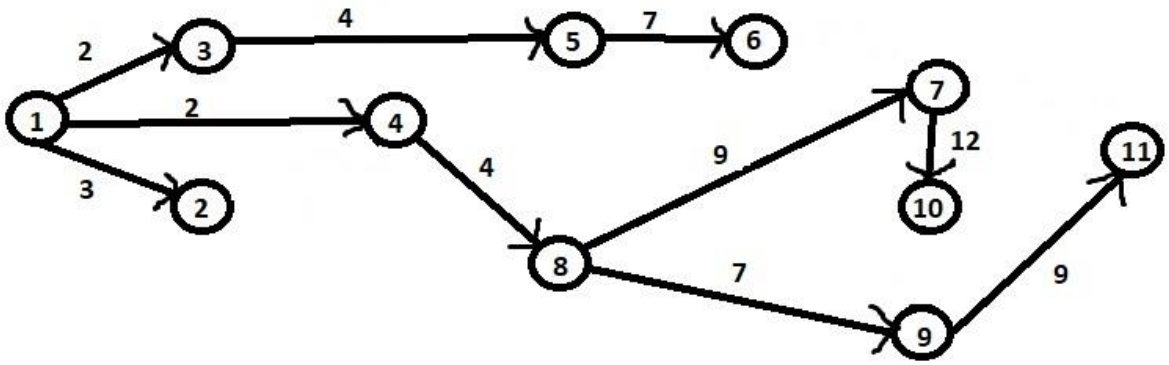
Rub *1, 4, 8, 7=0+2+2+5=9*

Rub *1, 4, 8, 9, 11=0+2+2+3+2=9*

Rub *1, 4, 8, 7, 10=0+2+2+5+3=12*

Iznos toga je 59.

Slika 41.: Prikaz rješenja Dijkstrinog algoritma



Izvor: Vlastiti rad

7.3.2 Implementacija u Pythonu

Slijedi prikaz implementacije Dijkstrinog algoritma u programskom jeziku Python.

Datoteka main.py:

```
class Graph:
    def __init__(self):
        self.vertices = {}

    def add_vertex(self, key):
        vertex = Vertex(key)
        self.vertices[key] = vertex

    def get_vertex(self, key):
        return self.vertices[key]

    def __contains__(self, key):
        return key in self.vertices

    def add_edge(self, src_key, dest_key, weight=1):
        self.vertices[src_key].add_neighbour(self.vertices[dest_key], weight)

    def does_edge_exist(self, src_key, dest_key):
        return
self.vertices[src_key].does_it_point_to(self.vertices[dest_key])

    def __len__(self):
        return len(self.vertices)
```

```
    def __iter__(self):
        return iter(self.vertices.values())
```

Izvor: Anonymous_41, n_d

```
    def remove_vertex(self, n):
      for n1 in self.get_(n):
        if n1 != key:
          self.graph[n1].get_neighbours.remove(n)
          del self._weight[self._edge(n, n1)]
          del self.graph[n]
```

Izvor: Anonymous_39, 2015.

```
    def remove_edge(self, n1, n2):
      try:
        if self != None:
          del g[n1][n2]
          print ('Brisanje\n', n1[0], n1[1])
          return 1
      except:
            return None

    def is_empty(self, key):

      if self.vertices[key]==0:
        return 1
      else:
        return 0

class Vertex:
```

Izvor: Anonymous_41, n_d

```
    def __init__(self, key):
        self.key = key
        self.points_to = {}

    def get_key(self):
        return self.key

    def add_neighbour(self, dest, weight):
        self.points_to[dest] = weight

    def get_neighbours(self):
        return self.points_to.keys()

    def get_weight(self, dest):
        return self.points_to[dest]

    def does_it_point_to(self, dest):
        return dest in self.points_to

def dijkstra(g, source):
    unvisited = set(g)
    distance = dict.fromkeys(g, float('inf'))
```

```
    distance[source] = 0

    while unvisited != set():
        closest = min(unvisited, key=lambda v: distance[v])
        unvisited.remove(closest)

        for neighbour in closest.get_neighbours():
           if neighbour in unvisited:
               new_distance = distance[closest] +
closest.get_weight(neighbour)
               if distance[neighbour] > new_distance:
                   distance[neighbour] = new_distance

    return distance

g = Graph()
print('Izbornik')
print('dodaj vrh <key>')
print('dodaj rub <src> <dest> <weight>')
print('ukloni vrh <n>')
print('ukloni rub <n1> <n2>')
print('prazan <key>')
print('dijkstra <source vertex key>')
print('prikaz')
print('izlaz')

while True:
    do = input('Sto zelite uciniti? ').split()

    operation = do[0]
    if operation == 'dodaj':
        suboperation = do[1]
        if suboperation == 'vrh':
            key = int(do[2])
            if key not in g:
                g.add_vertex(key)
            else:
                print('Vrh vec postoji.')
        elif suboperation == 'rub':
            src = int(do[2])
            dest = int(do[3])
            weight = int(do[4])
            if src not in g:
                print('Vrh {} ne postoji.'.format(src))
            elif dest not in g:
                print('Vrh {} ne postoji.'.format(dest))
            else:
                if not g.does_edge_exist(src, dest):
                    g.add_edge(src, dest, weight)
```

```
                else:
                    print('Rub vec postoji.')

    elif operation=='prazan':
      if key in g:
        g.is_empty(key)
        print('Graf nije prazan.')
      else:
        print('Graf je prazan.')

    elif operation == 'dijkstra':
        key = int(do[1])
        source = g.get_vertex(key)
        distance = dijkstra(g, source)
        print('Udaljenost od {}: '.format(key))
        for v in distance:
            print('Udaljenost do {}: {}'.format(v.get_key(), distance[v]))
        print()

    elif operation == 'prikaz':
        print('Vrhovi: ', end='')
        for v in g:
            print(v.get_key(), end=' ')
        print()

        print('Rubovi: ')
        for v in g:
            for dest in v.get_neighbours():
                w = v.get_weight(dest)
                print('(src={}, dest={}, weight={}) '.format(v.get_key(),
                                                             dest.get_key(),
w))
        print()

    elif operation == 'izlaz':
        break
```

Izvor: Anonymous_41, n_d

Opis koda:

Prvo je definirana klasa *Graph()* koja sadrži sljedeće funkcije: *def __init__(self), def add_vertex(self, key), def get_vertex(self, key), def __contains__(self, key), def add_edge(self, src_key, dest_key, weight=1), def does_edge_exist(self, src_key, dest_key), def __len__(self), def __iter__(self), def remove_vertex(self, n), def remove_edge(self, n1, n2) i def is_empty(self, key).* Prva spomenuta funkcija sadrži rječnik koji ukazuje na odgovarajući vrh objekta (inicijalizira prazan graf na početku). Druga dodaje vrh s odgovarajućom vrijednosti (1,…, 11). Nadalje, treća funkcija vraća

vrh s odgovarajućom vrijednosti. Potom *add_edge()* dodaje rub od *src_key* do *dest_key* zadane težine. Sljedeća funkcija vraća istinu (1) ako postoji rub od *src_key* do *dest_key*. Pomoćne funkcije *__len__ i __iter__* vraćaju dužinu vrha, odnosno iterator na same vrijednosti vrha. Funkcije *remove_vertex* i *remove_edge*, uklanjaju vrh i sve rubove koji su s njim incidentni ili brišu brid iz grafa. Posljednja definirana funkcija *is_empty* provjerava da li je dani graf prazan ili nije, te ispisuje poruku. Druga definirana klasa je Vertex() koja ima sljedeće operacije: *def __init__(self, key), def get_key(self), def add_neighbour(self, dest, weight), def get_neighbours(self), def get_weight(self, dest) i def does_it_point_to(self, dest).* Na kraju je definiran Dijkstrin algoritam. Prva operacija definira prazan vrh (vrh koji nema nikakve vrijednosti). Potom sljedeća funkcija vraća vrijednost koja odgovara samom vrhu. Nakon toga se stvara vrh točke (*dest*) s danom težinom ruba. Što se tiče sljedeće operacije, ista vraća sve vrhove koji su povezani s danim vrhom. Drugim riječima, ona vraća susjede danog vrha. Sljedeća pomoćna funkcija vraća težinu ruba vrha (*dest*). Posljednja pomoćna funkcija vraća istinu ako je vrh dane točke (*dest*). Dijkstrin algoritam vraća udaljenost na kojoj je udaljenost *[v]* minimalna udaljenost od izvora do *v*. To će vratiti rječničku udaljenost. *g* je *Graph()* objekt. Izvor je *Vertex* objekt u *g*. Konkretno, prvo se nalazi Vertex s minimalnom udaljenosti, a potom se označavaju posjećeni čvorovi (oni koji to jesu ili nisu). Nakon toga se ažurira udaljenost između vrhova. Na kraju je prikazan kratki izbornik gdje je korisniku omogućeno izvršavanje određenih operacija kod samog grafa (primjerice, dodavanje vrha/ ruba, prikaz tih podataka, provjera je li graf prazan ili nije, izvršenje Dijkstrin algoritma, itd.).

Prikaz primjera rješenja:

```
Izbornik

dodaj vrh <key>

dodaj rub <src> <dest> <weight>

ukloni vrh <n>

ukloni rub <n1> <n2>

prazan <key>

dijkstra <source vertex key>

prikaz
```

```
izlaz

Sto zelite uciniti? dodaj vrh 1

Sto zelite uciniti? dodaj vrh 2

Sto zelite uciniti? dodaj vrh 3

Sto zelite uciniti? dodaj vrh 4

Sto zelite uciniti? dodaj vrh 5

Sto zelite uciniti? dodaj vrh 6

Sto zelite uciniti? dodaj vrh 7

Sto zelite uciniti? dodaj vrh 8

Sto zelite uciniti? dodaj vrh 8

Vrh vec postoji.

Sto zelite uciniti? dodaj vrh 9

Sto zelite uciniti? dodaj vrh 10

Sto zelite uciniti? dodaj vrh 11

Sto zelite uciniti? dodaj rub 1 3 2

Sto zelite uciniti? dodaj rub 1 4 2

Sto zelite uciniti? dodaj rub 1 2 3

Sto zelite uciniti? dodaj rub 2 4 1

Sto zelite uciniti? dodaj rub 3 5 2

Sto zelite uciniti? dodaj rub 5 6 3

Sto zelite uciniti? dodaj rub 6 7 4

Sto zelite uciniti? dodaj rub 4 8 2

Sto zelite uciniti? dodaj rub 8 9 3

Sto zelite uciniti? dodaj rub 8 7 5

Sto zelite uciniti? dodaj rub 7 10 3

Sto zelite uciniti? dodaj rub 10 9 2

Sto zelite uciniti? dodaj rub 7 11 4
```

```
Sto zelite uciniti? dodaj rub 9 11 2

Sto zelite uciniti? dijkstra 1

Udaljenost od 1:

Udaljenost do 1: 0

Udaljenost do 2: 3

Udaljenost do 3: 2

Udaljenost do 4: 2

Udaljenost do 5: 4

Udaljenost do 6: 7

Udaljenost do 7: 9

Udaljenost do 8: 4

Udaljenost do 9: 7

Udaljenost do 10: 12

Udaljenost do 11: 9


Sto zelite uciniti? prikaz

Vrhovi: 1 2 3 4 5 6 7 8 9 10 11

Rubovi:

(src=1, dest=3, weight=2)

(src=1, dest=4, weight=2)

(src=1, dest=2, weight=3)

(src=2, dest=4, weight=1)

(src=3, dest=5, weight=2)

(src=4, dest=8, weight=2)

(src=5, dest=6, weight=3)

(src=6, dest=7, weight=4)

(src=7, dest=10, weight=3)
```

```
(src=7, dest=11, weight=4)

(src=8, dest=9, weight=3)

(src=8, dest=7, weight=5)

(src=9, dest=11, weight=2)

(src=10, dest=9, weight=2)


Sto zelite uciniti? izlaz
```

7.3.3 Implementacija pomoću vektora

Ovdje je prikazana implementacija pomoću strukture podatka vektor u programskom jeziku C++.

Datoteka Dijkstrin.cpp:

```
#include <iostream>
#include <vector>
#include <queue>
#include <climits>
using namespace std;

struct Edge {
  int source, dest, weight;
};

struct Node {
  int vertex, weight;
};

class Graph
{
public:
  vector<vector<Edge>> adjList;

  Graph(vector<Edge> const &edges, int N)
  {
    adjList.resize(N);

    for (Edge const &edge: edges)
    {
```

```
      adjList[edge.source].push_back(edge);
    }
  }
};

void print_route(vector<int> const &prev, int i)
{
  if (i < 0)
    return;

  print_route(prev, prev[i]);
  cout << i << " ";
}

struct comp
{
  bool operator()(const Node &lhs, const Node &rhs) const
  {
    return lhs.weight > rhs.weight;
  }
};

void shortestPath(Graph const& graph, int source, int N)
{
  priority_queue<Node, vector<Node>, comp> min_heap;
  min_heap.push({source, 0});

  vector<int> dist(N, INT_MAX);

  dist[source] = 0;

  vector<bool> done(N, false);
  done[0] = true;

  vector<int> prev(N, -1);

  while (!min_heap.empty())
  {
    Node node = min_heap.top();
    min_heap.pop();

    int u = node.vertex;

    for (auto i : graph.adjList[u])
    {
      int v = i.dest;
      int weight = i.weight;

      if (!done[v] && (dist[u] + weight) < dist[v])
```

```
      {
        dist[v] = dist[u] + weight;
        prev[v] = u;
        min_heap.push({v, dist[v]});
      }
    }

    done[u] = true;
  }

  for (int i = 1; i < N; ++i)
  {
    cout << "Put od vrha 1 do vrha " << i << " je minimalan, "
        "a cijena mu je " << dist[i] << " i ruta je [ ";
    print_route(prev, i);
    cout << "]\n";
  }
}

int main()
{
  vector<Edge> edges =
  {
    {1, 3, 2}, {1, 4, 2}, {1, 2, 3}, {2, 4, 1}, {3, 5, 2},
    {5, 6, 3}, {6, 7, 4}, {4, 8, 2}, {8, 9, 3}, {8, 7, 5},
    {7, 10, 3}, {10, 9, 2}, {7, 11, 4}, {9, 11, 2}
  };

  int N = 12;

  Graph graph(edges, N);

  shortestPath(graph, 1, N);

  return 0;
}
```

Izvor: Anonymous_42, n_d

Opis koda:

Prvo su uključene tri biblioteke *(#include<vector>, #include<queue> i #include<climits>*). Biblioteka vector služi za rad sa strukturom podatka tipa vektor, queue služi za rad sa strukturom podatka tipa red, dok climits (limits.h) služi za definiranje konstante s ograničenjima osnovnih integralnih tipova za konkretni primijenjeni sustav i prevoditelj (*INT_MAX*). Postoje dvije strukture, a to su struktura podataka za pohranjivanje rubova grafa (*struct Edge*) i struktura podataka za

spremanje čvorova (vrhova) (*struct Node*). Prva struktura sadrži tri podatka (*int source, dest, weight*), dok druga sadrži dva podatka (*int vertex, weight*). Potom je definirana klasa *Graph* koja ima javni dio gdje se konstruira vektor vektora iz strukture *Edge* koji predstavlja popis susjednosti (*vector<vector<Edge>> adjList*). Ista sadrži i vlastiti konstruktor (*Graph(vector<Edge> const &edges, int N)*) koji mijenja veličinu vektora na *N* elemenata tipa vektor *Edge* (*adjList.resize(N)),* dodaje rubove danom grafu (*for (Edge const &edge: edges))* te ih umeće na kraju (*adjList[edge.source].push_back(edge)).* Nakon toga je definirana funkcija koja ispisuje put (*void print_route(vector<int> const &prev, int i))* te objekt koji služi za usporedbu kod hrpe vezano za težinu (*struct comp*). Na kraju je definiran Dijkstrin algoritam pomoću funkcije (*void shortestPath(Graph const& graph, int source, int N))* koji stvara minimalnu hrpu i dodaje istom izvor čvora s udaljenosti nula (*priority_queue<Node, vector<Node>, comp> min_heap, min_heap.push({source, 0});).* Isti algoritam postavlja beskonačnu udaljenost od izvora *v* do samog početka (*vector<int> dist(N, INT_MAX)),* pa je udaljenost od izvora do samog sebe jednaka nuli (*dist[source] = 0*). Potom je unutar tog algoritma definiran *boolean* niz za praćenje vrhova za koje je već pronađen najmanji trošak (*vector<bool> done(N, false), done[0] = true*), pohranjivanje prethodnika vrha (za ispis puta) (*vector<int> prev(N, -1)),* izvršavanje minimalne hrpe sve dok nije prazna (*while (!min_heap.empty())),* uklanjanje i vraćanje vrha (*Node node = min_heap.top(), min_heap.pop()),* dohvaćanje broja vrha (*int u = node.vertex*), izvršenje za svakog susjeda *v* do *u* (*for (auto i : graph.adjList[u]), int v = i.dest, int weight = i.weight*), označavanje vršne točke *u* kao gotove tako da se više ista ne bira (*done[u] = true*) te ispis puta i troška. Unutar funkcije *int main()* su inicijalizirani rubovi (*vector<Edge> edges*), definiran je broj čvorova u grafu (*int N = 12*), konstruiran je graf (*Graph graph(edges, N))* i poziva se funkcija (*shortestPath(graph, 1, N)).*

Prikaz rješenja:

```
Put od vrha 1 do vrha 1 je minimalan, a cijena mu je 0 i ruta je [ 1 ]

Put od vrha 1 do vrha 2 je minimalan, a cijena mu je 3 i ruta je [ 1 2 ]

Put od vrha 1 do vrha 3 je minimalan, a cijena mu je 2 i ruta je [ 1 3 ]

Put od vrha 1 do vrha 4 je minimalan, a cijena mu je 2 i ruta je [ 1 4 ]

Put od vrha 1 do vrha 5 je minimalan, a cijena mu je 4 i ruta je [ 1 3 5 ]
```

```
Put od vrha 1 do vrha 6 je minimalan, a cijena mu je 7 i ruta je [ 1 3 5 6 ]

Put od vrha 1 do vrha 7 je minimalan, a cijena mu je 9 i ruta je [ 1 4 8 7 ]

Put od vrha 1 do vrha 8 je minimalan, a cijena mu je 4 i ruta je [ 1 4 8 ]

Put od vrha 1 do vrha 9 je minimalan, a cijena mu je 7 i ruta je [ 1 4 8 9 ]

Put od vrha 1 do vrha 10 je minimalan, a cijena mu je 12 i ruta je [ 1 4 8 7 10
]

Put od vrha 1 do vrha 11 je minimalan, a cijena mu je 9 i ruta je [ 1 4 8 9 11
]
```

7.4 Razlika između ograničenja i najkraćeg puta

Ovdje se istražuje poseban slučaj linearnog programiranja koji se može svesti na pronalaženje najkraćih puteva iz jednog izvora. Problemi s najkraćim putevima s jednim izvorom čiji rezultati mogu biti riješeni pomoću algoritma Bellman Ford, čime se rješavaju i problemi linearnog programiranja.

Konkretno, u općem problemu linearnog programiranja, dana je matrica $m \times n$, m-vektor b i n-vektor c. Želi se pronaći vektor x od n elemenata koji maksimizira objektivnu funkciju ovisno o ograničenjima m danim od $A_x \leq b$.

Iako simpleksni algoritam ne teče uvijek u vremenskom polinomu u veličini njegova unosa, postoje i drugi algoritmi linearnog programiranja koji rade u polinomnom vremenu. Postoji nekoliko razloga zbog kojih je važno razumjeti postavljanje problema linearnog programiranja. Prvo, znajući da se zadani problem može izvesti kao problem linearnog programiranja veličine polinoma odmah znači da postoji problem polinomnog vremena. Drugo, postoje mnogi posebni slučajevi linearnog programiranja za koje postoje brži algoritmi. Primjerice, jedan izvor problema najkraćih puteva je poseban slučaj linearnog programiranja. Ostali problemi koji se mogu pretvoriti u linearno programiranje uključuju problem najmanjih puteva s jednim parom i problem maksimalnog protoka. (Cormen i sur., 2001.).

Ponekad se zapravo ne staje do funkcije cilja, već se samo želi pronaći bilo koje moguće rješenje, tj. bilo koji vektor x koji zadovoljava $A_x \leq b$, ili odrediti da ne postoji izvedivo rješenje.

U sustavu razlika ograničenja, svaki red matrice linearnog programiranja sadrži jedan (1) i jedan (-1), a svi ostali unosi od *A* su 0. Dakle, ograničenja koja daje $A_x \leq b$ su skup *m* ograničenja razlike koja uključuje *n* nepoznanica, u kojima je svako ograničenje jednostavna linearna nejednakost forme:

$X_j - x_i \leq b_k$,

gdje je $1 \leq i, j \leq n$ i $1 \leq k \leq m$.

Slika 42.: Problem pronalaženja 5-vektora ($x = (x_i)$) koji zadovoljava sljedeće

$$\begin{pmatrix} 1 & -1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & -1 \\ 0 & 1 & 0 & 0 & -1 \\ -1 & 0 & 1 & 0 & 0 \\ -1 & 0 & 0 & 1 & 0 \\ 0 & 0 & -1 & 1 & 0 \\ 0 & 0 & -1 & 0 & 1 \\ 0 & 0 & 0 & -1 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{pmatrix} \leq \begin{pmatrix} 0 \\ -1 \\ 1 \\ 5 \\ 4 \\ -1 \\ -3 \\ -3 \end{pmatrix}.$$

Izvor: Cormen i sur., 2001.

Slika 43.: Prikaz (tog) problema koji je jednak pronalaženju nepoznanica x_i, *za* $i = 1, 2, ..., 5$, tako da su zadovoljena sljedeća 8 ograničenja razlike

$$\begin{aligned}
&(24.3)\ x_1 - x_2 \leq 0, \\
&(24.4)\ x_1 - x_5 \leq -1, \\
&(24.5)\ x_2 - x_5 \leq 1, \\
&(24.6)\ x_3 - x_1 \leq 5, \\
&(24.7)\ x_4 - x_1 \leq 4, \\
&(24.8)\ x_4 - x_3 \leq -1, \\
&(24.9)\ x_5 - x_3 \leq -3, \\
&(24.10)\ \quad x_5 - x_4 \leq -3.
\end{aligned}$$

Izvor: Cormen i sur., 2001.

Jedno rješenje ovog problema je da je $x = (-5, -3, 0, -1, -4)$, što se može izravno provjeriti provjerom svake nejednakosti. Zapravo, postoji više od jednog rješenja ovog problema. Drugi je $x' = (0, 2, 5, 4, 1)$. Ova dva rješenja su povezana: svaka komponenta od x' je 5 puta veća od odgovarajuće komponente x. Ta činjenica nije puka slučajnost.

Lema: Neka je $x = (x_1, x_2, ..., x_n)$ rješenje sustava $A_x \leq b$ razlika ograničenja, i neka *d* bude bilo koja konstanta. Tada je $x + d = (x_1 + d, x_2 + d, ..., x_n + d)$ također rješenje za $A_x \leq b$. Dokaz: Za svaki x_i i x_j imamo $(x_j + d) - (x_i + d) = x_j - x_i$. Dakle, ako *x* zadovoljava $A_x \leq b$, to znači i $x + d$.

Sustavi razlika ograničenja javljaju se u mnogim različitim aplikacijama. Primjerice, nepoznanice x_i mogu biti vremena u kojima će se događaji dogoditi. Svako ograničenje može se smatrati tvrdnjom da između dva događaja mora postojati barem određeno vrijeme, ili najviše određeno vrijeme. Možda su događaji zadaci koji se trebaju obaviti tijekom sastavljanja proizvoda. Ako se primjerice, primijeni ljepilo koje traje 2 sata za postavljanje u vrijeme od x_1, tada se mora pričekati da se postavi za instaliranje dijela u vremenu x_2, tada postoji ograničenje da je $x_2 \geq x_1 + 2$ ili, ekvivalentno, da je $x_1 - x_2 \leq -2$. Alternativno, može se zahtijevati da se dio postavi nakon nanošenja ljepila, ali ne kasnije od vremena kada je ljepilo postavljeno na pola puta. U ovom slučaju se dobiva par ograničenja $x_2 \geq x_1$ i $x_2 \leq x_1 + 1$ ili, ekvivalentno, $x_1 - x_2 \leq 0$ i $x_2 - x_1 \leq 1$.

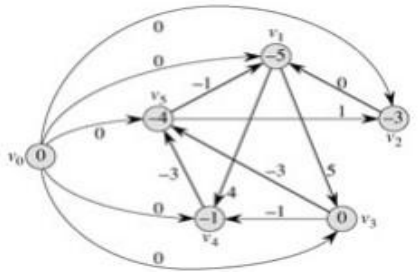
Kod grafova ograničenja, korisno je tumačiti sustave razlika ograničenja sa stajališta teorije grafova. Ideja je da se u sustavu $A_x \leq b$ razlika ograničenja, matrica linearnog programiranja $m \times n$ može promatrati kao transponiranje matrice incidencije za graf s n vrhova i m rubova. Svaka točka v_i u grafu, za $i = 1, 2, ..., n,$ odgovara jednoj od n nepoznatih varijabli x_i. Svaki usmjereni rub u grafu odgovara jednoj od nejednakosti m koja uključuje dvije nepoznanice.

Formalno, s obzirom na sustav $A_x \leq b$ razlike ograničenja, odgovarajući graf ograničenja je ponderirani, usmjereni graf $G = (V, E),$ gdje je:

$V = \{v_0, v_1, ..., v_n\}$ *i* $E = \{(v_i, v_j): x_j - x_i \leq b_k$ je ograničenje$\}$ $\{(v_0, v_1), (v_0, v_2), (v_0, v_3), ..., (v_0, v_n)\}.$

Dodatni vrh v_0 ugrađen je, kao što će se vidjeti u kratkim crtama, radi osiguranja da svaki drugi vrh bude dostupan iz njega. Dakle, skup vrha V sastoji se od vrha v_i za svaku nepoznatu x_i, plus dodatni vrh v_0. Set rubova E sadrži rub za svako ograničenje razlike, plus rub (v_0, v_i) za svaku nepoznatu vrijednost x_i. Ako je $x_j - x_i \leq b_k$ razlika u ograničenju, tada je težina ruba (v_i, v_j) w $(v_i, v_j) = b_k$. Težina svakog ruba koji napušta v_0 je 0.

Slika 44.: Prikaz grafa ograničenja za sustav ograničenja razlike



Izvor: Cormen i sur., 2001.

Slika prikazuje graf ograničenja koji odgovara sustavu ograničenja razlike. Vrijednost $\delta(v_0, v_i)$ prikazana je u svakoj točki v_i. Moguće rješenje sustava je $x = (-5, -3, 0, -1, -4)$.

Sljedeći teorem pokazuje da se može pronaći rješenje za sustav razlike ograničenja pronalazeći težine najkraćeg puta u odgovarajućem grafu ograničenja.

Teorem: S obzirom na sustav $A_x \leq b$ razlike ograničenja, neka je $G = (V, E)$ odgovarajući graf ograničenja.

Slika 45.: Prikaz slučaja gdje G ne sadrži cikluse negativne težine

$$x = (\delta(v_0, v_1), \delta(v_0, v_2), \delta(v_0, v_3), \ldots, \delta(v_0, v_n))$$

Izvor: Cormen i sur., 2001.

Ovo je izvedivo rješenje za sustav. Ako G sadrži ciklus s negativnom težinom, tada ne postoji izvedivo rješenje za sustav.

Dokaz: Prvo se pokazuje da ako graf ograničenja ne sadrži negativno-teške cikluse, tada jednadžba (prethodno napisana) daje izvedivo rješenje. Valja razmotriti svaki rub $(v_i, v_j) E$. Nejednakost trokuta, $\delta(v_0, v_j) \leq \delta(v_0, v_i) + w(v_i, v_j)$ ili, ekvivalentno, $\delta(v_0, v_j) - \delta(v_0, v_i) \leq w(v_i, v_j)$. Dakle, ostavljajući $x_i = \delta(v_0, v_i)$ i $x_j = \delta(v_0, v_j)$ zadovoljava se razlikovno ograničenje $x_j - x \leq w(v_i, v_j)$ koje odgovara rubu (v_i, v_j).

Preciznije, ako graf ograničenja sadrži ciklus negativno-težinski, onda sustav razlike ograničenja nema izvedivog rješenja. Bez gubitka općenitosti, neka negativni ciklus bude $c = v_1, v_2, \ldots, v_k$, gdje je $v_1 = v_k$ (točka v_0 ne može biti na ciklusu c, jer nema ulaznih rubova.).

Slika 46.: Prikaz *C* ciklusa koji odgovara sljedećim ograničenjima razlike:

$$
\begin{aligned}
x_2 - x_1 &\le w(v_1, v_2), \\
x_3 - x_2 &\le w(v_2, v_3), \\
&\vdots \\
x_k - x_{k-1} &\le w(v_{k-1}, v_k), \\
x_1 - x_k &\le w(v_k, v_1).
\end{aligned}
$$

Izvor: Cormen i sur., 2001.

Pretpostavlja se da postoji rješenje za *x* koji zadovoljava svaku od ovih *k* nejednakosti. Ovo rješenje također mora zadovoljiti nejednakost koja nastaje kada se zbrajaju *k* nejednakosti zajedno. Ako se sumiraju lijeve strane, svaka nepoznata vrijednost x_i dodaje se jednom i oduzima se jednom, tako da je lijeva strana zbroja 0. Desna strana zbraja se u *w (c),* i tako se dobiva $0 \le w(c)$. Ali budući da je *c* ciklus s negativnom težinom, $w(c) < 0$, dobiva se da je $0 \le w(c) < 0$.

Rješavanje sustava razlika ograničenja teorema govori da se može koristiti Bellman-Fordov algoritam za rješavanje sustava razlika ograničenja. Budući da postoje rubovi od izvorne točke v_0 do svih ostalih točaka u grafu ograničenja, svaki negativni-ciklus u grafu ograničenja je dostupan iz v_0. Ako Bellman-Fordov algoritam vrati istinu, tada su težine najkraće putanje moguće rješenje sustava. Na prethodnoj slici, primjerice, težine najkraće putanje pružaju izvedivo rješenje $x = (-5, -3, 0, -1, -4)$, a prema lemi, $x = (d - 5, d - 3, d, d - 1, d - 4)$ također je izvedivo rješenje za bilo koju konstantu *d*. Ako Bellman-Fordov algoritam vraća laž, ne postoji izvedivo rješenje za sustav ograničenja razlike. (Cormen i sur., 2001.).

Sustav razlika ograničenja s *m* ograničenjima na *n* nepoznanica stvara graf s $n + 1$ vrhova i $n + m$ rubova. Tako, koristeći Bellman-Fordov algoritam, može se riješiti sustav u $O((n + 1)(n + m)) = O(n^2 + nm)$ vrijeme.

7.5 Dokazi u svojstvima najkraćih puteva

Argumenti ispravnosti oslanjali su se na nejednakost trokuta, svojstvo gornje granice, svojstvo bez puta, svojstvo konvergencije, svojstvo relaksacije puta i svojstvo prethodnika-podgrafa.

Prvo svojstvo je nejednakost trokuta. U proučavanju pretraživanja po dubini, dokazano je kao lema jednostavno svojstvo najkraćih udaljenosti u netežinskim grafovima. Nejednakost trokuta generalizira svojstvo na ponderirane grafove.

Lema: Neka je $G = (V, E)$ ponderirani, usmjereni graf s težinskom funkcijom $w: E \rightarrow R$ i izvornim vrhom s. Zatim, za sve rubove (u, v) E postoji $\delta(s, v) \leq \delta(s, u) + w(u, v)$. Dokaz: pretpostavlja se da postoji najkraći put p od izvora s do vrha v. Tada p nema više težine od bilo kojeg drugog puta od s do v. Konkretno, put p nema više težine od određenog puta koja uzima najkraći put od izvora s na vrh u i zatim uzima rub (u, v).

Drugo svojstvo su učinci relaksacije na procjenu najkraćeg puta. Sljedeća grupa lema opisuje kako na najmanju procjenu utječu kada se izvršava slijed relaksacijskih koraka na rubovima ponderiranog, usmjerenog grafa koji je inicijaliziran od *INITIALIZE-SINGLE-SOURCE.*

Lema (Gornja granica svojstva): Neka je $G = (V, E)$ ponderirani, usmjereni graf s težinskom funkcijom $w: E \rightarrow R$. Neka je s V izvorna točka i neka se graf inicijalizira pomoću *INITIALIZE-SINGLE-SOURCE (G, s).* Zatim, slijedi da je $d[v] \geq \delta(s, v)$ za sve v V, pa se ta invarijanta održava iznad bilo kojeg slijeda relaksacijskih koraka na rubovima G. Osim toga, jednom se kod $d[v]$ postiže donja granica $\delta(s, v)$ koja se nikad ne mijenja.

Dokaz: dokazuje se invarijantna $d[v] \geq \delta(s, v)$ za sve točke v V indukcijom preko broja koraka relaksacije. Za osnovu, $d[v] \geq \delta(s, v)$ je svakako točna nakon inicijalizacije, budući da $d[s] = 0 \geq (s, s)$ (valja imati na umu da je $\delta(s, s)$ $-\infty$ ako je s na negativnom težinskom ciklusu i 0 inače) i $d[v] = \infty$ podrazumijeva $d[v] \geq \delta(s, v)$ za sve v $V - \{s\}$.

Za induktivni korak se razmatra opuštanje ruba (u, v). Prema induktivnoj hipotezi, $d[x] \geq \delta(s, x)$ za sve x V prije relaksacije. Jedina d vrijednost koja se može promijeniti je $d[v]$. Ako se promijeni, postoji i tako se nepromjenjivo održava.

$D[v] = d[u] + w(u, v)$

≥ *δ (s, u) + w (u, v)* (pomoću induktivne hipoteze)

≥ *δ (s, v)* (nejednakosti trokuta). (Cormen i sur., 2001.)

Da bi se vidjelo da se vrijednost *d [v]* nikad ne mijenja jednom *d [v] = δ (s, v),* valja imati na umu da nakon što se postigne donja granica, *d [v]* se ne može smanjiti jer se upravo pokazalo da je *d [v] ≥ δ (s, v)*, i ne može se povećati jer koraci relaksacije ne povećavaju *d* vrijednosti.

Posljedica tog je pretpostavka da u ponderiranom, usmjerenom grafu *G = (V, E)* s funkcijom težine *w: ER*, nijedan put ne povezuje izvornu točku *s V* s danom točkom *v V*. Zatim, nakon što se graf inicijalizira pomoću *INITIALIZE-SINGLE IZVOR (G, s),* postoji *d [v] = δ (s, v) = ∞,* i ta jednakost se održava kao invarijantna u odnosu na bilo koji slijed relaksacijskih koraka na rubovima *G*. Dokaz gornjim graničnim svojstvom je da uvijek postoji: *∞ = δ(s, v) ≤ d[v], pa d[v] = ∞ = δ(s, v).*

Lema: Neka je *G = (V, E)* ponderirani, usmjereni graf s funkcijom težine *w: E → R*, i *(u, v) E*. Zatim, odmah nakon relaksirajućeg ruba *(u, v)* izvršava se *RELAX-a (u, v) , w),* pa postoji *d [v] ≤ d [u] + w (u, v).* Dokaz: Neposredno prije relaksirajućeg ruba *(u, v)* postoji *d [v]> d [u] + w (u, v),* zatim *d [v] = d [u] + w (u, v)* poslije. Ako, umjesto toga, *d [v] ≤ d [u] + w (u, v)* neposredno prije relaksacije, tada se ni *d [u]* ni *d [v]* ne mijenjaju, pa *d [v] ≤ d [u] + w (u, v)* nakon toga.

Lema svojstva konvergencije: Neka je *G = (V, E)* ponderirani, usmjereni graf s težinskom funkcijom *w: E → R*, neka je *V* izvorna točka, i neka je najkraći put u *G* za neke vrhove *u, v V*. Pretpostavlja se da je *G* inicijaliziran pomoću *INITIALIZE SINGLE-SOURCE (G, s),* a zatim se na rubovima *G* izvodi redoslijed relaksacijskih koraka koji uključuje poziv *RELAX (u, v, w). s, u)* u bilo koje vrijeme prije poziva, tada je *d [v] = δ (s, v)* u svakom trenutku nakon poziva.

Dokaz gornjim graničnim svojstvom, ako je *d [u] = δ (s, u)* u nekoj točki prije relaksirajućeg ruba *(u, v),* onda se ta jednakost nakon toga pridržava. Posebno, nakon relaksirajućeg ruba *(u, v),* postoji:

d [v] ≤ d [u] + w (u, v) (prema lemi)

= δ (s, u) + w (u, v)

= δ (s, v) (prema lemi).

S gornjim graničnim svojstvom, $d[v] \geq \delta(s, v)$, iz kojeg se zaključuje da je $d[v] = \delta(s, v)$, ta se jednakost održava nakon toga.

Lema svojstva relaksacije puta: Neka je $G = (V, E)$ ponderirani, usmjereni graf s funkcijom težine $w: E \rightarrow R$ i neka je V izvorna točka. Razmatra se svaki najkraći put $p = v_0, v_1, \ldots, v_k$ od $s = v_0$ do v_k. Ako je G inicirana pomoću *INITIALIZE-SINGLE-SOURCE* (G, s), a zatim slijedi slijed koraka relaksacije koji uključuje redom, relaksacije rubova $(v_0, v_1), (v_1, v_2), \ldots, (v_{k-1}, v_k)$, zatim $d[v_k] = \delta(s, v_k)$ nakon tih relaksacija i uvijek nakon toga. Ista se drži bez obzira na sve (pojavljuju se i druge relaksacije rubova, uključujući i relaksacije koje se miješaju s relaksacijama rubova p).

Dokaz: Pokazuje se indukcijom da se nakon i-tog ruba puta p vidi da je $d[v_i] = \delta(s, v_i)$. Za bazu, $i = 0$, i prije nego što su svi rubovi p opušteni, iz inicijalizacije postoji $d[v_0] = d[s] = 0 = \delta(s, s)$. Prema svojstvu gornje granice, vrijednost $d[s]$ se nikada ne mijenja nakon inicijalizacije.

Za induktivni korak pretpostavlja se da je $d[v_{i-1}] = \delta(s, v_{i-1})$, te se ispituje relaksacija ruba (v_{i-1}, v_i). S obzirom na svojstvo konvergencije, nakon toga, postoji $d[v_i] = \delta(s, v_i)$, a ta jednakost se održava u svakom trenutku nakon toga.

Sljedeći dokaz su stabla opuštanja i najkraći putevi. Sada se pokazuje da jednom kad je slijed relaksacija uzrokovao da se najkraći put procjene konvergira u težine najkraćeg puta, prethodnik podgraf G_π induciran je rezultirajućim π vrijednostima (stablo najkraćeg puta za G). Počinje se sa sljedećom lemom, što pokazuje da prethodni podgraf uvijek formira ukorijenjeno stablo čiji je izvor izvor.

Lema: Neka je $G = (V, E)$ ponderirani, usmjereni graf s funkcijom težine $w: E \rightarrow R$, neka je V izvorna točka, i pretpostavlja se da G ne sadrži negativno-težinske cikluse koji su dostupni iz s. Zatim, nakon što je graf inicijaliziran pomoću *INITIALIZE-SINGLE-SOURCE* (G, s), prethodni podgraf G_π formira ukorijenjeno stablo s korijenom s, a bilo koji slijed koraka relaksacije na rubovima G održava ovu osobinu kao invarijantnu.

Dokaz: u početku, jedini vrh u G_π je izvorna točka, a lema je trivijalno istinita. Razmatra se prethodni podgraf G_π koji nastaje nakon niza koraka relaksacije. Prvo će se dokazati da je G_π acikličan. Pretpostavlja se zbog kontradikcije da neki korak relaksacije stvara ciklus u grafu G_π. Neka ciklus bude $c = v_0, v_1, \ldots, v_k$, gdje je $v_k = v_0$. Tada, $\pi[v_i] = v_{i-1}$

za i = 1, 2, …, k i, bez gubitka općenitosti, može se pretpostaviti da je relaksacija ruba (v_{k-1}, v_k) stvorila ciklus u G_π.

Tvrdi se da su svi vrhovi na ciklusu *c* dostupni iz izvora *s*. Postavlja se pitanje zašto? Svaki vrh na *c* ima ne-NIL prethodnika, pa je svaki vrh na *c* dodijeljen konačnoj procjeni najkraćeg puta kada je dodijeljena njena ne-NIL π vrijednost. Prema svojstvu gornje granice, svaki vrh na ciklusu *c* ima konačnu težinu najkraće putanje, što znači da je dostupno iz *s*.

Ispitat će se procjene najkraće putanje na *c* neposredno prije poziva *RELAX* (v_{k-1}, v_k, w) i pokazati da je *c* ciklus s negativnom težinom, čime se proturječi pretpostavka da *G* ne sadrži negativne težinske cikluse koji su dostupni iz izvora. Neposredno prije poziva postoji $\pi[v_i] = v_{i-1}$ *za* $i = 1, 2, \ldots, k-1$. Dakle, za $i = 1, 2, \ldots, k-1$, posljednje ažuriranje do $d[v_i]$ bio je po zadatku $d[v_i] \leftarrow d[v_{i-1}] + w(v_{i-1}, v_i)$. Ako se $d[v_{i-1}]$ od tada promijenio, smanjio se.

Slika 47.: Prikaz gdje se pokazuje da neposredno prije poziva *RELAX* (v_{k-1}, v_k, w), postoji:

$$d[v_i] \geq d[v_{i-1}] + w(v_{i-1}, v_i) \qquad \text{for all } i = 1, 2, \ldots, k-1 .$$

Izvor: Cormen i sur., 2001.

Budući da se pozivom mijenja $\pi[v_k]$, neposredno prije toga postoji i stroga nejednakost: $d[v_k] > d[v_{k-1}] + w(v_{k-1}, v_k)$.

Slika 48.: Prikaz sumirajuće stroge nejednakost s $k-1$ s nejednakostima gdje se dobiva zbroj procjena najkraće putanje oko ciklusa *c*

$$\begin{aligned} \sum_{i=1}^{k} d[v_i] &> \sum_{i=1}^{k} (d[v_{i-1}] + w(v_{i-1}, v_i)) \\ &= \sum_{i=1}^{k} d[v_{i-1}] + \sum_{i=1}^{k} w(v_{i-1}, v_i) . \end{aligned}$$

But

$$\sum_{i=1}^{k} d[v_i] = \sum_{i=1}^{k} d[v_{i-1}] ,$$

Izvor: Cormen i sur., 2001.

Slika 49.: Prikaz svakog vrha u ciklusu *c* koji se pojavljuje točno jednom u svakom zbrajanju. Ta jednakost podrazumijeva:

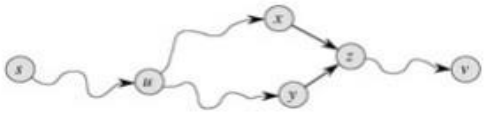
$$0 > \sum_{i=1}^{k} w(v_{i-1}, v_i) .$$

Izvor: Cormen i sur., 2001.

Dakle, zbroj težina oko ciklusa *c* je negativan, što daje željenu kontradikciju. Dokazalo se da je G_π usmjereni, aciklički graf. Da bi se pokazalo da on formira ukorijenjeno stablo s korijenom *s*, dovoljno je dokazati da za svaki vrh *v* V_π postoji jedinstveni put od *s* do *v* u G_π. Najprije se mora pokazati da postoji put od *s* za svaku točku u V_π. Vrh u V_π su ona s ne-NIL π vrijednostima, plus *s*. Ideja je da se indukcijom dokaže da postoji put od *s* do svih točaka u V_π.

Da bi se dovršio dokaz leme, mora se pokazati da za bilo koji vrh *v* V_π postoji najviše jedan put od *s* do *v* u grafu G_π. Pretpostavlja se drugačije, tj. pretpostavlja se da postoje dva jednostavna puta od s do neke točke *v: p1*, koje se mogu dekomponirati na $s \rightsquigarrow u \rightsquigarrow x \rightarrow z \rightsquigarrow v$, koje se mogu dekomponirati na $s \rightsquigarrow u \rightsquigarrow y \rightarrow z \rightsquigarrow v$, gdje je $x \neq Y$. Ali onda, $\pi[z] = x$ i $\pi[z] = y$, što implicira kontradikciju da je $x = y$. Zaključuje se da postoji jedinstveni jednostavan put u G_π od *s* do *v*, pa G_π formira ukorijenjeno stablo s korijenom *s*. (Cormen i sur., 2001.).

Slika 50.: Prikaz dokaza leme koja pokazuje da za bilo koji vrh *v* V_π postoji najviše jedan put od *s* do *v* u grafu G_π



Izvor: Cormen i sur., 2001.

Prethodna slika pokazuje da je put u G_π od izvora *s* do točke *v* jedinstven. Ako postoje dva puta $p_1 (s \rightsquigarrow u \rightsquigarrow x \rightarrow z \rightsquigarrow v)$ i $p_2 (s \rightsquigarrow u \rightsquigarrow y \rightarrow z \rightsquigarrow v)$, gdje $x \neq y$, onda je $\pi[z] = x$ i $\pi[z] = y$, kontradikcija. (Cormen i sur., 2001.).

Sada se može pokazati da ako, nakon što se provede niz koraka relaksacije, svim točkama su dodijeljene njihove prave težine najkraće putanje, pa je tada prethodni podgraf G_π stablo najkraćeg puta.

Lema svojstva prethodnika-podgrafa: Neka je $G = (V, E)$ ponderirani, usmjereni graf s funkcijom težine $w: E \rightarrow R$, neka je V izvorna točka, i pretpostavlja se da G ne sadrži negativno-težinske cikluse koji su dostupni iz s. Tako postoji *INITIALIZE-SINGLE-SOURCE (G, s)* i zatim se izvršava bilo koji slijed koraka relaksacije na rubovima G koji proizvodi $d[v] = \delta(s, v)$ za sve v V. Tada je prethodni podgraf G_π stablo najkraćeg puta ukorijenjeno u s.

Dokaz: mora se dokazati da tri svojstva stabala najkraćih puteva vrijede za G_π. Da bi se pokazalo prvo svojstvo, mora se pokazati da je V_π skup točaka do kojih se može doći iz s. Prema definiciji, težina najkraće putanje $\delta(s, v)$ konačna je ako i samo ako je v dostupno iz s, i stoga su točke koje se mogu doseći iz s točno one s konačnim d vrijednostima. Ali vrh v $V - \{s\}$ dodijeljen je konačnoj vrijednosti za $d[v]$ ako i samo ako je $\pi[v]$ različit od NIL. Dakle, vrhovi u V_π su upravo oni koji se mogu dostići iz s.

Drugo svojstvo slijedi izravno iz leme. Ostaje, dakle, dokazati posljednje svojstvo stabala najkraćih puteva: za svaki vrh v V_π jedinstvena jednostavna staza u G_π je najkraći put od s do v u G. Neka je $p = v_0, v_1, \ldots, v_k$, gdje $v_0 = s$ i $v_k = v$. *Za* $i = 1, 2, \ldots, k$ postoji i $d[v_i] = \delta(s, v_i)$ i $d[v_i] \geq d[v_{i-1}] + w(v_{i-1}, v_i)$, iz kojih se zaključuje da je $w(v_{i-1}, v_i) \leq \delta(s, v_i) - \delta(s, v_{i-1})$.

Slika 51.: Zbrajanje težina duž prinosa p puta:

$$
\begin{aligned}
w(p) &= \sum_{i=1}^{k} w(v_{i-1}, v_i) \\
&\leq \sum_{i=1}^{k} (\delta(s, v_i) - \delta(s, v_{i-1})) \\
&= \delta(s, v_k) - \delta(s, v_0) && \text{(because the sum telescopes)} \\
&= \delta(s, v_k) && \text{(because } \delta(s, v_0) = \delta(s, s) = 0) \ .
\end{aligned}
$$

Izvor: Cormen i sur., 2001.

Dakle, $w(p) \leq \delta(s, v_k)$. Budući da je $\delta(s, v_k)$ donja granica na težini bilo kojeg puta od s do v_k, zaključuje se da je $w(p) = \delta(s, v_k)$, pa je p najkraći put od s do $v = v_k$.

8. Najkraći putevi svih parova

Problem najmanjeg puta svih parova je određivanje najkraćih udaljenosti između svakog para vrhova u danom grafu.(Anonymous_14, n_d).

Isti, prethodno definirani problem, može se riješiti primjenom Dijkstrinog algoritma i Floyd- Warshallovog algoritma. Potonji algoritam također radi u slučaju ponderiranog grafa gdje rubovi imaju negativne težine. Matrica svih udaljenosti između parova vrhova naziva se matrica udaljenosti (ponekad i matrica najkraćeg puta svih parova).

Neki parovi čvorova možda nisu međusobno dostupni, tako da ne postoji najkraći put između tih parova.

Obzirom na usmjereni graf *G=(V,E),* gdje svaki rub *(v, w)* ima nenegativni trošak *C[v, w],* vrijedi za sve parove točaka *(v, w).* Konkretno, prethodno rečeno se uzima kao ulaz, dok se kao izlaz uzima trošak najnižeg troškovnog puta od *v* do *w*.

Ovdje se javlja pojam generalizacije problema s jednim izvorom i najkraćom putanjom. Isto tako ovdje se koristi Dijkstrin algoritam, mijenjajući izvorni čvor među svim čvorovima na grafu. (Anonymous_15, n_d).

Glede algoritma, valja pronaći put najniže cijene između svakog para vrhova. Također, mora se obnoviti sam put, a ne samo cijena puta.

Algoritam najkraćeg puta svih parova se koristi u sljedećim situacijama: (Anonymous_16, n_d).

1. Algoritam za najkraći put svih parova koristi se u problemima gradskog servisnog sustava, kao što je lokacija urbanih objekata, distribucija ili isporuka robe. Jedan takav primjer je određivanje očekivanog prometnog opterećenja na različitim segmentima transportne mreže.
2. Najmanji put svih parova koristi se kao dio algoritma za projektiranje podatkovnog centra Rewire koji pronalazi mrežu s maksimalnom propusnošću i minimalnom latencijom.

Za svaki par vrhova *u* i *v* se žele izračunati sljedeće informacije:

1. *Dist (u,v)* je duljina najkraćeg puta od *u* do *v*.
2. *Pred (u,v)* je drugi na zadnji vrh (engl. second-to-last) na najkraćem putu od *u* do *v*.

Ove prethodno spomenute definicije isključuju nekoliko graničnih slučajeva, a to su: (Anonymous_17, n_d, str. 310)

1. Ako nema puta od *u* do *v*, tada ne postoji najkraći put od *u* do *v*. U ovom slučaju se definira *dist (u,v)*= ∞ i *pred (u,v)=Null.*
2. Ako postoji negativni ciklus između *u* i *v*, onda postoji put od *u* do *v* s proizvoljno negativnom duljinom. U ovom se slučaju definira *dist (u,v)*=- ∞ i *pred (u,v)=Null.*
3. Ako *u* ne leži na negativnom ciklusu, onda najkraći put od *u* do samog sebe nema rubova i stoga nema posljednjeg ruba. U ovom se slučaju definira *dist (u,u)=0* i *pred (u,u)=Null.*

Željeni rezultat problema najkraćeg puta svih parova je par niza *V*V*, od kojih jedan pohranjuje V^2 najkraće udaljenosti, a drugi pohranjuje prethodnike od V^2. Polje prethodnika, gdje se mogu izračunati stvarni najkraći putevi, izračunavaju se ali samo s manjim modifikacijama.

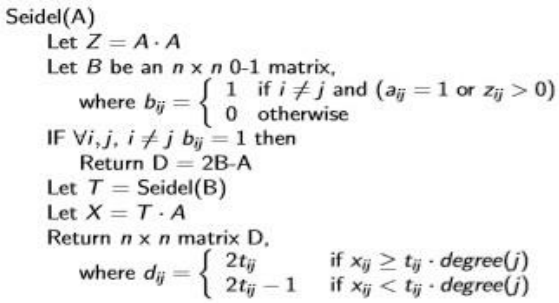
8.1 Najkraći putevi i množenje matrica

Problem koji se ovdje javlja je naći najkraći put između svih točaka gdje je $v \in V$ za graf $G=(V, E)$.

Postoje dvije definicije (prva se odnosi na ulaz, druga na izlaz):

1. Seidelov algoritam. *A* predstavlja matricu susjedstva, a *n*n* je booleova matrica gdje a_1 predstavlja rub između čvora *i* čvora *j* u grafu *G.*
2. Seidelov algoritam. *D* je matrica udaljenosti, a *n*n* je cjelobrojna matrica s d_{ij} koja predstavlja duljinu najkraćeg puta od vrha *i* do vrha *j* u grafu *G.*

Slika 52.: Prikaz pseudokoda Seidelovog algoritma



Izvor: Burfield, 2013., str. 5

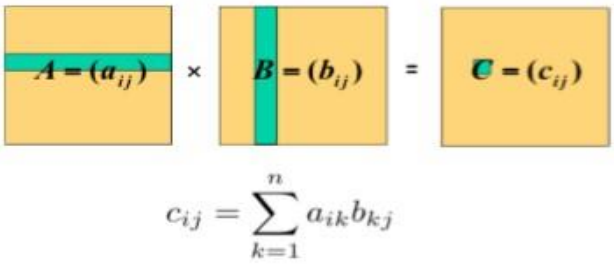
Seidelov algoritam se može koristiti na neodređenim (engl. unweighted) i neusmjerenim grafovima.

Vremenska složenost je *O (M(n) logn),* gdje *M (n)* označava vrijeme potrebno da se množe dvije *n*n* cjelobrojne matrice.

Kod množenja kvadratne matrice postoje tri različite vremena trajanja složenosti: (Burfield, 2013., str. 12)

1. Naivno kvadratno matrično množenje koje traje O(n^3).
2. Strassenov algoritam koji traje O ($n^{2.807}$).
3. Najbrži algoritam (brza matrica) koji radi u O ($n^{2.373}$).

Slika 53.: Prikaz naivnog kvadratnog matričnog množenja



Izvor: Burfield, 2013., str. 13

Slika 54.: Prikaz osam operacija množenja i četiri operacije zbrajanja (nastavak prethodnog primjera)

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

$$\begin{aligned} C_{11} &= A_{11}B_{11} + A_{12}B_{21} \\ C_{12} &= A_{11}B_{12} + A_{12}B_{22} \\ C_{21} &= A_{21}B_{11} + A_{22}B_{21} \\ C_{22} &= A_{21}B_{12} + A_{22}B_{22} \end{aligned}$$

8 multiplications
4 additions

Izvor: Burfield, 2013., str. 14

Prethodni primjer prikazuje 8 množenja i 4 zbrajanja. Složenost je T(*n)=8T (n/2) + O (n*^{2}). Točnije, složenost je *O (n*$^{log8/log2}$*)=O(n*^{3}*).* Dakle, složenost je jednaka kao u prethodnom primjeru, tj. nije došlo do promjene, odnosno do poboljšanja složenosti.

Slika 55.: Prikaz Strassenovog algoritma za 2*2 matrice

2 x 2 Matrices:

$$\begin{aligned} C_{11} &= A_{11}B_{11} + A_{12}B_{21} \\ C_{12} &= A_{11}B_{12} + A_{12}B_{22} \\ C_{21} &= A_{21}B_{11} + A_{22}B_{21} \\ C_{22} &= A_{21}B_{12} + A_{22}B_{22} \end{aligned}$$

$$\begin{aligned} C_{11} &= M_1 + M_4 - M_5 + M_7 \\ C_{12} &= M_3 + M_5 \\ C_{21} &= M_2 + M_4 \\ C_{22} &= M_1 - M_2 + M_3 + M_6 \end{aligned}$$

$$\begin{aligned} M_1 &= (A_{11} + A_{22})(B_{11} + B_{22}) \\ M_2 &= (A_{21} + A_{22})B_{11} \\ M_3 &= A_{11}(B_{12} - B_{22}) \\ M_4 &= A_{22}(B_{21} - B_{11}) \\ M_5 &= (A_{11} + A_{12})B_{22} \\ M_6 &= (A_{21} - A_{11})(B_{11} + B_{12}) \\ M_7 &= (A_{12} - A_{22})(B_{21} + B_{22}) \end{aligned}$$

7 multiplications
18 additions/subtractions

Izvor: Burfield, 2013., str. 15

Dakako, postoji i Strassenov algoritam za *n*n* matrica. Ovdje se može zamisliti matrica s *n*n* kao *2*2* matrica čiji su elementi *n/2** *n/2* matrice. Isti primjenjuje algoritam *2*2* rekurzivno.

Vremenska složenost je *T(n)= 7T (n/2) + O(n*^{2}*).* Preciznije, *O(n*$^{log7/log2}$*)=O(n*$^{2.807}$*)*, što znači da se složenost poboljšala u odnosu na prethodni algoritam.

Usporedba Seidelovog algoritma, Strassenovog i brze matrice:

1. Seidelov algoritam radi u *O(M(n) logn).*
2. Koristeći Strassenov algoritam, umjesto naivnog algoritma, smanjuje se vrijeme izvođenja iz *O(n*^{3} *logn)* u *O(n*$^{2.81}$*logn).*
3. Ako se koristi najbrži algoritam (brza matrica), može se postići vrijeme izvođenja u *O (n*$^{2.373}$*logn).*

8.2 Floyd- Warshall algoritam

Pri razmatranju udaljenosti između lokacija, primjerice u logistici (ovdje se često spominje problem pronalaženja najkraćeg puta). U takvim situacijama, lokacije i putanje (smjerovi) mogu se modelirati kao vrhovi i rubovi grafa.

U mnogim postavkama problema potrebno je pronaći najkraće puteve između svih parova čvorova u grafu i odrediti njihovu duljinu. Floyd- Warshallov algoritam rješava prethodno navedeni problem i može se izvoditi na bilo kojem grafu, pod uvjetom da ne sadrži ni jedan ciklus negativne težine ruba. Inače, ti se ciklusi mogu koristiti za konstruiranje puteva koji su proizvoljno kratki (negativna duljina) između određenih parova čvorova (algoritam ne može pronaći optimalno rješenje). (Anonymous_18, 2015.).

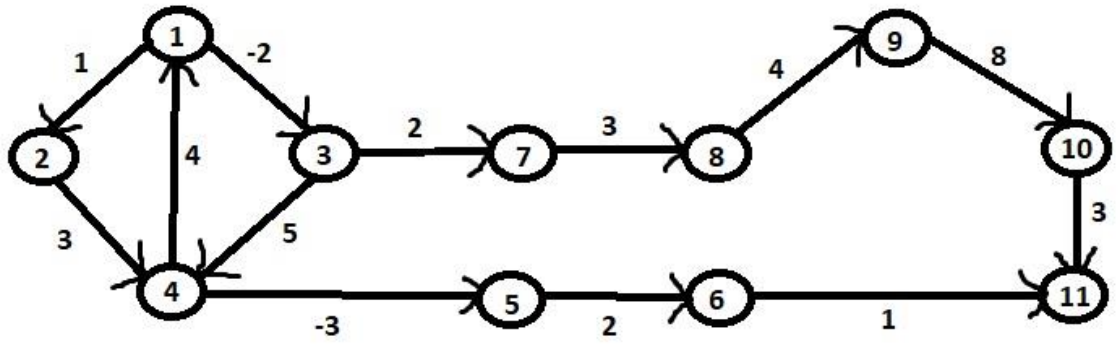
Preciznije, ovdje se pronalaze najmanje udaljenosti između svih parova vrhova grafa. Ideja algoritma se krije u ispitivanju svih mogućih puteva u grafu, ali se pri takvom ispitivanju koristi upravo činjenica da isti ima optimalnu substrukturu i da se do ukupnog minimuma dolazi spajanjem minimuma problema nekog manjeg reda.

Ovaj algoritam je tipičan primjer dinamičkog programiranja, a dokaz da je rezultat izvođenja algoritma ispravan je i veoma složen. (Marinović, n_d, str. 18).

8.2.1 Primjer

Zadatak je, a ujedno i problem, pronaći najkraću udaljenost između svih parova vrhova (točaka) u ponderiranom usmjerenom grafu koji može imati negativne rubne težine. Da bi problem bio dobro definiran, ne bi trebalo biti nikakvih ciklusa u grafu s negativnom ukupnom težinom.

Slika 56.: Prikaz primjera Floyd- Warshallovog grafa



Izvor: Vlastiti rad

Prikaz rješenja Floyd- Warshall algoritma (prikaz za sve rubove):

Od 1 do 2: 1 -> 2 = udaljenost 1,

Od 1 do 3: 1 -> 3 = udaljenost -2,

Od 1 do 4: 1 -> 3 -> 4 = -2+5 = udaljenost 3,

Od 1 do 5: 1 -> 3 -> 4 -> 5 = -2+5-3 = udaljenost 0,

Od 1 do 6: 1 -> 3 -> 4 -> 5 -> 6 = -2+5-3+2 = udaljenost 2,

Od 1 do 7: 1 -> 3 -> 7 = -2+2 = udaljenost 0,

Od 1 do 8: 1 -> 3 -> 7 -> 8 = -2+2+3 = udaljenost 3,

Od 1 do 9: 1 -> 3 -> 7 -> 8 -> 9 = -2+2+3+4 = udaljenost 7,

Od 1 do 10: 1 -> 3 -> 7 -> 8 -> 9 -> 10 = -2+2+3+4+8 = udaljenost 15,

Od 1 do 11: 1 -> 3 -> 4 -> 5 -> 6 -> 11 = -2+5-3+2+1 = udaljenost 3,

Od 2 do 1: 2 -> 4 -> 1 = 3+4 = udaljenost 7,

Od 2 do 3: 2 -> 4 -> 1 -> 3 = 3+4-2 = udaljenost 5,

Od 2 do 4: 2 -> 4 = 3 = udaljenost 3,

Od 2 do 5: 2 -> 4 -> 5 = 3-3 = udaljenost 0,

Od 2 do 6: 2 -> 4 -> 5 -> 6 = 3-3+2 = udaljenost 2,

Od 2 do 7: 2 -> 4 -> 1 -> 3 -> 7 = 3+4-2+2 = udaljenost 7,

Od 2 do 8: 2 -> 4 -> 1 -> 3 -> 7 -> 8 = 3+4-2+2+3 = udaljenost 10,

Od 2 do 9: 2 -> 4 -> 1 -> 3 -> 7 -> 8 -> 9 = 3+4-2+2+3+4 = udaljenost 14,

Od 2 do 10: 2 -> 4 -> 1 -> 3 -> 7 -> 8 -> 9 -> 10 = 3+4-2+2+4+8 = udaljenost 22,

Od 2 do 11: 2 -> 4 -> 5 -> 6 -> 11 = 3-3+2+1 = udaljenost 3,

Od 3 do 1: 3 -> 4 -> 1 = 5+4 = udaljenost 9,

Od 3 do 2: 3 -> 4 -> 1 -> 2 = 5+4+1 = udaljenost 10,

Od 3 do 4: 3 -> 4 = 5 = udaljenost 5,

Od 3 do 5: 3 -> 4 -> 5 = 5-3 = udaljenost 2,

Od 3 do 6: 3 -> 4 -> 5 -> 6 = 5-3+2 = udaljenost 4,

Od 3 do 7: 3 -> 7 = 2 = udaljenost 2,

Od 3 do 8: 3 -> 7 -> 8 = 2+3 = udaljenost 5,

Od 3 do 9: 3 -> 7 -> 8 -> 9 = 2+3+4 = udaljenost 9,

Od 3 do 10: 3 -> 7 -> 8 -> 9 -> 10 = 2+3+4+8 = udaljenost 17,

Od 3 do 11: 3 -> 4 -> 5 -> 6 -> 11 = 5-3+2+1 = udaljenost 5,

Od 4 do 1: 4 -> 1 = 4 = udaljenost 4,

Od 4 do 2: 4 -> 1 -> 2 = 4+1 = udaljenost 5,

Od 4 do 3: 4 -> 1 -> 3 = 4-2 = udaljenost 2,

Od 4 do 5: 4 -> 5 = -3 = udaljenost -3,

Od 4 do 6: 4 -> 5 -> 6 = -3+2 = udaljenost -1,

Od 4 do 7: 4 -> 1 -> 3 -> 7 = 4-2+2 = udaljenost 4,

Od 4 do 8: 4 -> 1 -> 3 -> 7 -> 8 = 4-2+2+3 = udaljenost 7,

Od 4 do 9: 4 -> 1 -> 3 -> 7 -> 8 -> 9 = 4-2+2+3+4 = udaljenost 11,

Od 4 do 10: 4 -> 1 -> 3 -> 7 -> 8 -> 9 -> 10 = 4-2+2+3+4+8 = udaljenost 19,

Od 4 do 11: 4 -> 5 -> 6 -> 11 = -3+2+1 = udaljenost 0,

Od 5 do 6: 5 -> 6 = 2 = udaljenost 2,

Od 5 do 11: 5 -> 6 -> 11 = 2+1 = udaljenost 3,

Od 6 do 11: 6 -> 11 = 1 = udaljenost 1,

Od 7 do 8: 7 -> 8 = 3 = udaljenost 3,

Od 7 do 9: 7 -> 8 -> 9 = 3+4 = udaljenost 7,

Od 7 do 10: 7 -> 8 -> 9 -> 10 = 3+4+8 = udaljenost 15,

Od 7 do 11: 7 -> 8 -> 9 -> 10 -> 11 = 3+4+8+3 = udaljenost 18,

Od 8 do 9: 8 -> 9 = 4 = udaljenost 4,

Od 8 do 10: 8 -> 9 -> 10 = 4+8 = udaljenost 12,

Od 8 do 11: 8 -> 9 -> 10 -> 11 = 4+8+3 = udaljenost 15,

Od 9 do 10: 9 -> 10 = 8 = udaljenost 8,

Od 9 do 11: 9 -> 10 -> 11 = 8+3 = udaljenost 11 i

Od 10 do 11: 10 -> 11 = 3 = udaljenost 3.

8.2.2 Implementacija u Pythonu

Slijedi prikaz implementacije Floyd- Warshall algoritma u programskom jeziku Python.

Datoteka main.py:

```
class Graph:
    def __init__(self):
        self.vertices = {}

    def add_vertex(self, key):
        vertex = Vertex(key)
        self.vertices[key] = vertex

    def get_vertex(self, key):
        return self.vertices[key]

    def __contains__(self, key):
```

```
        return key in self.vertices

    def add_edge(self, src_key, dest_key, weight=1):
        self.vertices[src_key].add_neighbour(self.vertices[dest_key], weight)

    def does_edge_exist(self, src_key, dest_key):
        return
self.vertices[src_key].does_it_point_to(self.vertices[dest_key])

    def __len__(self):
        return len(self.vertices)

    def __iter__(self):
        return iter(self.vertices.values())
```

Izvor: Anonymous_43, n_d

```
    def remove_vertex(self, n):
      for n1 in self.get_(n):
        if n1 != key:
          self.graph[n1].get_neighbours.remove(n)
          del self._weight[self._edge(n, n1)]
          del self.graph[n]

    def remove_edge(self, n1, n2):
      try:
        if self != None:
          del g[n1][n2]
          print ('Brisanje\n', n1[0], n1[1])
          return 1
      except:
            return None

    def is_empty(self, key):

      if self.vertices[key]==0:
        return 1
      else:
        return 0

class Vertex:
    def __init__(self, key):
        self.key = key
        self.points_to = {}

    def get_key(self):
        return self.key

    def add_neighbour(self, dest, weight):
        self.points_to[dest] = weight
```

Izvor: Anonymous_39, 2015.

Izvor: Anonymous_43, n_d

```
    def get_neighbours(self):
        return self.points_to.keys()

    def get_weight(self, dest):
        return self.points_to[dest]

    def does_it_point_to(self, dest):
        return dest in self.points_to

def floyd_warshall(g):

    distance = {v:dict.fromkeys(g, float('inf')) for v in g}
    next_v = {v:dict.fromkeys(g, None) for v in g}

    for v in g:
        for n in v.get_neighbours():
            distance[v][n] = v.get_weight(n)
            next_v[v][n] = n

    for v in g:
         distance[v][v] = 0
         next_v[v][v] = None

    for p in g:
        for v in g:
            for w in g:
                if distance[v][w] > distance[v][p] + distance[p][w]:
                    distance[v][w] = distance[v][p] + distance[p][w]
                    next_v[v][w] = next_v[v][p]

    return distance, next_v

def print_path(next_v, u, v):

    p = u
    while (next_v[p][v]):
        print('{} -> '.format(p.get_key()), end='')
        p = next_v[p][v]
    print('{} '.format(v.get_key()), end='')


g = Graph()
print('Izbornik')
print('dodaj vrh <key>')
print('dodaj rub <src> <dest> <weight>')
print('ukloni vrh <n>')
print('ukloni rub <n1> <n2>')
print('prazan <key>')
print('floyd-warshall')
```

```
print('prikaz')
print('izlaz')

while True:
    do = input('Sto zelite uciniti? ').split()

    operation = do[0]
    if operation == 'dodaj':
        suboperation = do[1]
        if suboperation == 'vrh':
            key = int(do[2])
            if key not in g:
                g.add_vertex(key)
            else:
                print('Vrh vec postoji.')
        elif suboperation == 'rub':
            src = int(do[2])
            dest = int(do[3])
            weight = int(do[4])
            if src not in g:
                print('Vrh {} ne postoji.'.format(src))
            elif dest not in g:
                print('Vrh {} ne postoji.'.format(dest))
            else:
                if not g.does_edge_exist(src, dest):
                    g.add_edge(src, dest, weight)
                else:
                    print('Rub vec postoji.')

    elif operation=='prazan':
      if key in g:
        g.is_empty(key)
        print('Graf nije prazan.')
      else:
        print('Graf je prazan.')

    elif operation == 'floyd-warshall':
        distance, next_v = floyd_warshall(g)
        print('Najkraće udaljenosti:')
        for start in g:
            for end in g:
                if next_v[start][end]:
                    print('Od {} do {}: '.format(start.get_key(),
                                                 end.get_key()),
                          end = '')
                    print_path(next_v, start, end)
                    print('(udaljenost {})'.format(distance[start][end]))

    elif operation == 'prikaz':
```

```
        print('Vrhovi: ', end='')
        for v in g:
            print(v.get_key(), end=' ')
        print()

        print('Rubovi: ')
        for v in g:
            for dest in v.get_neighbours():
                w = v.get_weight(dest)
                print('(src={}, dest={}, weight={}) '.format(v.get_key(),
                                                            dest.get_key(),
w))
        print()

    elif operation == 'izlaz':
        break
```

Izvor: Anonymous_43, n_d

Opis koda:

Prvo je definirana klasa *Graph()* koja sadrži sljedeće funkcije: *def __init__(self), def add_vertex(self, key), def get_vertex(self, key), def __contains__(self, key), def add_edge(self, src_key, dest_key, weight=1), def does_edge_exist(self, src_key, dest_key), def __len__(self), def __iter__(self), def remove_vertex(self, n), def remove_edge(self, n1, n2) i def is_empty(self, key).* Prva spomenuta funkcija sadrži rječnik koji ukazuje na odgovarajući vrh objekta (inicijalizira prazan graf na početku). Druga dodaje vrh s odgovarajućom vrijednosti (1,…, 11). Nadalje, treća funkcija vraća vrh s odgovarajućom vrijednosti. Potom *add_edge()* dodaje rub od *src_key* do *dest_key* zadane težine. Sljedeća funkcija vraća istinu (1) ako postoji rub od *src_key* do *dest_key*. Pomoćne funkcije *__len__ i __iter__* vraćaju dužinu vrha, odnosno iterator na same vrijednosti vrha. Funkcije *remove_vertex* i *remove_edge*, uklanjaju vrh i sve rubove koji su s njim incidentni ili brišu brid iz grafa. Posljednja definirana funkcija *is_empty* provjerava da li je dani graf prazan ili nije, te ispisuje poruku. Druga definirana klasa je Vertex() koja ima sljedeće operacije: *def __init__(self, key), def get_key(self), def add_neighbour(self, dest, weight), def get_neighbours(self), def get_weight(self, dest) i def does_it_point_to(self, dest).* Na kraju je definiran Floyd-Warshall algoritam. Prva operacija definira prazan vrh (vrh koji nema nikakve vrijednosti). Potom sljedeća funkcija vraća vrijednost koja odgovara samom vrhu. Nakon toga se stvara vrh točke (*dest*) s danom težinom ruba. Što se tiče sljedeće operacije, ista vraća sve vrhove koji su povezani s danim vrhom. Drugim riječima, ona

vraća susjede danog vrha. Sljedeća pomoćna funkcija vraća težinu ruba vrha (*dest*). Posljednja pomoćna funkcija vraća istinu ako je vrh dane točke (*dest*). Floyd- Warshall algoritam vraća udaljenost rječnika i *next_v*. Udaljenost *[u] [v]* je najkraća udaljenost od vrha *u* do *v*. *Next_v [u] [v]* je sljedeća točka nakon točke *v* na najkraćem putu od *u* do *v*. *"None"* se definira u slučaju ako nema puta između njih. Stoga, *next_v [u] [u]* bi trebao biti *None* za sve *u*. *g* je *Graph()* objekt koji može imati negativne rubne težine. Nakon Floyd- Warshall algoritma je definirana pomoćna funkcija koja ispisuje najkraći put od vrha *u* do vrha *v*. *Next_v* je rječnik gdje je *next_v [u] [v]* sljedeći vrh nakon vrha *u* na najkraćem putu od *u* do *v*. To je *None* ako nema puta između njih. *next_v [u] [u]* treba biti *None* za sve *u*. *u* i *v* predstavljaju *Vertex* objekte. Na kraju je prikazano stvaranje instance *Grapha()* i kratki izbornik gdje je korisniku omogućeno izvršavanje određenih operacija kod samog grafa (primjerice, dodavanje vrha/ ruba, prikaz tih podataka, provjera je li graf prazan ili nije, izvršenje Floyd- Warshall algoritma, itd.). Da bi se pronašao najkraći put između parova vrhova, Floyd- Warshall algoritam dohvaća udaljenost, putem rječnika, pomoću *next_v. print_path* se poziva putem rječnika, pomoću *next_v* za svaki par vrhova za ispis puta, a udaljenost rječnika se koristi za ispis udaljenosti između svakog para.

Prikaz primjera rješenja:

```
Izbornik

dodaj vrh <key>

dodaj rub <src> <dest> <weight>

ukloni vrh <n>

ukloni rub <n1> <n2>

prazan <key>

floyd-warshall

prikaz

izlaz

Sto zelite uciniti? dodaj vrh 1

Sto zelite uciniti? dodaj vrh 2

Sto zelite uciniti? dodaj vrh 3
```

```
Sto zelite uciniti? dodaj vrh 4

Sto zelite uciniti? dodaj vrh 5

Sto zelite uciniti? dodaj vrh 6

Sto zelite uciniti? dodaj vrh 7

Sto zelite uciniti? dodaj vrh 8

Sto zelite uciniti? dodaj vrh 9

Sto zelite uciniti? dodaj vrh 10

Sto zelite uciniti? dodaj vrh 11

Sto zelite uciniti? dodaj rub 1 2 1

Sto zelite uciniti? dodaj rub 1 3 -2

Sto zelite uciniti? dodaj rub 2 4 3

Sto zelite uciniti? dodaj rub 3 4 5

Sto zelite uciniti? dodaj rub 4 1 4

Sto zelite uciniti? dodaj rub 3 7 2

Sto zelite uciniti? dodaj rub 7 8 3

Sto zelite uciniti? dodaj rub 8 9 4

Sto zelite uciniti? dodaj rub 9 10 8

Sto zelite uciniti? dodaj rub 10 11 3

Sto zelite uciniti? dodaj rub 4 5 -3

Sto zelite uciniti? dodaj rub 5 6 2

Sto zelite uciniti? dodaj rub 6 11 1

Sto zelite uciniti? floyd-warshall

Najkraće udaljenosti:

Od 1 do 2: 1 -> 2 (udaljenost 1)

Od 1 do 3: 1 -> 3 (udaljenost -2)

Od 1 do 4: 1 -> 3 -> 4 (udaljenost 3)

Od 1 do 5: 1 -> 3 -> 4 -> 5 (udaljenost 0)
```

```
Od 1 do 6: 1 -> 3 -> 4 -> 5 -> 6 (udaljenost 2)

Od 1 do 7: 1 -> 3 -> 7 (udaljenost 0)

Od 1 do 8: 1 -> 3 -> 7 -> 8 (udaljenost 3)

Od 1 do 9: 1 -> 3 -> 7 -> 8 -> 9 (udaljenost 7)

Od 1 do 10: 1 -> 3 -> 7 -> 8 -> 9 -> 10 (udaljenost 15)

Od 1 do 11: 1 -> 3 -> 4 -> 5 -> 6 -> 11 (udaljenost 3)

Od 2 do 1: 2 -> 4 -> 1 (udaljenost 7)

Od 2 do 3: 2 -> 4 -> 1 -> 3 (udaljenost 5)

Od 2 do 4: 2 -> 4 (udaljenost 3)

Od 2 do 5: 2 -> 4 -> 5 (udaljenost 0)

Od 2 do 6: 2 -> 4 -> 5 -> 6 (udaljenost 2)

Od 2 do 7: 2 -> 4 -> 1 -> 3 -> 7 (udaljenost 7)

Od 2 do 8: 2 -> 4 -> 1 -> 3 -> 7 -> 8 (udaljenost 10)

Od 2 do 9: 2 -> 4 -> 1 -> 3 -> 7 -> 8 -> 9 (udaljenost 14)

Od 2 do 10: 2 -> 4 -> 1 -> 3 -> 7 -> 8 -> 9 -> 10 (udaljenost 22)

Od 2 do 11: 2 -> 4 -> 5 -> 6 -> 11 (udaljenost 3)

Od 3 do 1: 3 -> 4 -> 1 (udaljenost 9)

Od 3 do 2: 3 -> 4 -> 1 -> 2 (udaljenost 10)

Od 3 do 4: 3 -> 4 (udaljenost 5)

Od 3 do 5: 3 -> 4 -> 5 (udaljenost 2)

Od 3 do 6: 3 -> 4 -> 5 -> 6 (udaljenost 4)

Od 3 do 7: 3 -> 7 (udaljenost 2)

Od 3 do 8: 3 -> 7 -> 8 (udaljenost 5)

Od 3 do 9: 3 -> 7 -> 8 -> 9 (udaljenost 9)

Od 3 do 10: 3 -> 7 -> 8 -> 9 -> 10 (udaljenost 17)

Od 3 do 11: 3 -> 4 -> 5 -> 6 -> 11 (udaljenost 5)

Od 4 do 1: 4 -> 1 (udaljenost 4)
```

```
Od 4 do 2: 4 -> 1 -> 2 (udaljenost 5)

Od 4 do 3: 4 -> 1 -> 3 (udaljenost 2)

Od 4 do 5: 4 -> 5 (udaljenost -3)

Od 4 do 6: 4 -> 5 -> 6 (udaljenost -1)

Od 4 do 7: 4 -> 1 -> 3 -> 7 (udaljenost 4)

Od 4 do 8: 4 -> 1 -> 3 -> 7 -> 8 (udaljenost 7)

Od 4 do 9: 4 -> 1 -> 3 -> 7 -> 8 -> 9 (udaljenost 11)

Od 4 do 10: 4 -> 1 -> 3 -> 7 -> 8 -> 9 -> 10 (udaljenost 19)

Od 4 do 11: 4 -> 5 -> 6 -> 11 (udaljenost 0)

Od 5 do 6: 5 -> 6 (udaljenost 2)

Od 5 do 11: 5 -> 6 -> 11 (udaljenost 3)

Od 6 do 11: 6 -> 11 (udaljenost 1)

Od 7 do 8: 7 -> 8 (udaljenost 3)

Od 7 do 9: 7 -> 8 -> 9 (udaljenost 7)

Od 7 do 10: 7 -> 8 -> 9 -> 10 (udaljenost 15)

Od 7 do 11: 7 -> 8 -> 9 -> 10 -> 11 (udaljenost 18)

Od 8 do 9: 8 -> 9 (udaljenost 4)

Od 8 do 10: 8 -> 9 -> 10 (udaljenost 12)

Od 8 do 11: 8 -> 9 -> 10 -> 11 (udaljenost 15)

Od 9 do 10: 9 -> 10 (udaljenost 8)

Od 9 do 11: 9 -> 10 -> 11 (udaljenost 11)

Od 10 do 11: 10 -> 11 (udaljenost 3)

Sto zelite uciniti? prikaz

Vrhovi: 1 2 3 4 5 6 7 8 9 10 11

Rubovi:

(src=1, dest=2, weight=1)

(src=1, dest=3, weight=-2)
```

```
(src=2, dest=4, weight=3)

(src=3, dest=4, weight=5)

(src=3, dest=7, weight=2)

(src=4, dest=1, weight=4)

(src=4, dest=5, weight=-3)

(src=5, dest=6, weight=2)

(src=6, dest=11, weight=1)

(src=7, dest=8, weight=3)

(src=8, dest=9, weight=4)

(src=9, dest=10, weight=8)

(src=10, dest=11, weight=3)


Sto zelite uciniti? Izlaz
```

8.2.3 Implementacija pomoću vektora

Treba pronaći najkraću udaljenost između svakog para vrha na danom grafu. Ovaj je problem poznat i kao problem svih parova s najkraćim putem.

Jedno rješenje tog problema je primjena Dijkstrin algoritma za sve vrhove. No ako graf sadrži rubove s negativnom težinom, algoritam Dijkstrin ne može se primijeniti. U tom slučaju se koristi algoritam Floyda Warshalla. Započinje se s matricom susjednosti grafa. Ako nema puta od vrha *i* do *j*, ćelija *(i, j)* u matrici bit će inicijalizirana u *INFINITY*. I svaka ćelija *(i, i)* biti će inicijalizirana na *0*. Zatim se smatra da svaki vrh (jedan po jedan) bude kao intermedijski. Primjerice, ako je međuprostorni vrh *i*, a želi se doseći od vrha *j* do vrha *k*, prvo će se preći iz vrha *i* u vrh *j*, a zatim iz vrha *j* u vrh *k*, umjesto da se izravno ide iz *j* u *k.* To je algoritam koji traje $O(V^3)$, gdje je *V* broj vrhova.

Ovdje je prikazana implementacija pomoću strukture podatka vektor u programskom jeziku C++.

Datoteka FloydWarshall.cpp:

```
#include<bits/stdc++.h>
#include<vector>
using namespace std;

int main()
{
    int i,j,k;
    int n,e;
    int s,d,w;

    cout<<“Unesi broj vrhova „;
    cin>>n;

    vector<vector<int> > distMat(n,vector<int>(n,INT_MAX));

    for(i=0;i<n;i++)
    {
        distMat[i][i]=0;
    }

    cout<<“Unesi broj rubova „;
    cin>>e;

    cout<<“Unesi pocetnu vrijednost, zavrsnu i tezinu“<<endl;
    for(i=0;i<e;i++)
    {
        cin>>s>>d>>w;

        distMat[s-1][d-1]=w;
    }

    for(k=0;k<n;k++)
    {
        for(i=0;i<n;i++)
        {
            for(j=0;j<n;j++)
            {
                if(distMat[i][k]!=INT_MAX &&
                   distMat[k][j]!=INT_MAX &&
                   distMat[i][k]+distMat[k][j]<distMat[i][j])
                    {
                        distMat[i][j]=distMat[i][k]+distMat[k][j];
                    }

            }
        }
    }
```

```
    cout<<endl<<“Duljina izmedju svih parova prikazana u matrici je „<<endl;

    cout<<“       „;

    for(i=0;i<n;i++)
    {
        cout<<i+1<<“ „;
    }

    cout<<endl;

    for(i=0;i<6*n;i++)
    {
        cout<<“_“<< „ „;
    }

    cout<<“_______“<<endl;

    for(i=0;i<n;i++)
    {
        cout<<“ |“<<i+1<<“ „;
        for(j=0;j<n;j++)
        {
            if(distMat[i][j]==INT_MAX)
                cout<<“   INF“;

            else
                cout<<distMat[i][j]<<“ „;
        }

        cout<<endl;
    }

    return 0;
}
```

Izvor: Anonymous_44, n_d

Opis koda:

Prvo je uključena biblioteka vector *(#include<vector>)* kako bi se moglo raditi sa strukturom podatka tipa vektor. Nakon toga su unutar funkcije *int main()* definirani podatci tipa *int* koji označavaju vrhove (*int n*), rubove (*int e*), korištenje u petljama *for* (*int i, j, k*), početni vrh (*int s*), završni vrh (*int d*) i težinu (*int w*). Osim toga, korisnik prvo unosi broj vrhova (*cout << "Unesite broj vrhova“, cin >> n*), pa je definirana matrica susjednosti inicijalizirana u beskonačnost (*vector<vector<int>* >

distMat(n,vector<int>(n,INT_MAX))) te prva petlja *for* gdje se postavlja da je udaljenost vrha od samog sebe uvijek jednaka nuli (*for(i=0;i<n;i++), distMat[i][i]=0*). Nadalje se unosi broj rubova koje korisnik ima na grafu (*cout << "Unesite broj rubova", cin >> e*), početni i završni čvor te težina svakog ruba (*cout<<"Unesi pocetnu vrijednost, zavrsnu i tezinu"<<endl, for(i=0;i<e;i++), cin>>s>>d>>w*), dodaje se isto na popis susjednosti (*distMat[s-1][d-1]=w*), stvara se matrica susjednosti gdje se na osnovu toga stvara *n* matrica smatrajući pritom sami vrh kao intermedijarno područje (srednji, *for(k=0;k<n;k++)),* (izvorište, *for(i=0;i<n;i++)),* (odredište, *for(j=0;j<n;j++)),* izračun udaljenosti (*if(distMat[i][k]!=INT_MAX && distMat[k][j]!=INT_MAX && distMat[i][k]+distMat[k][j]<distMat[i][j]), distMat[i][j]=distMat[i][k]+distMat[k][j]*), ispisivanje matrice svih para najkraćih udaljenosti (*cout<<endl<<"Duljina izmedju svih parova prikazana u matrici je "<<endl, cout<<" "*). Ugnježđena petlja izračunava sve najkraće puteve para grafa. Na kraju se ispisuju svi vrhovi (početni i završni vrh) te težine u obliku matrice susjednosti (*for(i=0;i<n;i++), cout<<i+1<<" ", cout<<endl, for(i=0;i<6*n;i++), cout<<"_"<< " ", cout<<"_____"<<endl, for(i=0;i<n;i++), cout<<" |"<<i+1<<" ", for(j=0;j<n;j++), if(distMat[i][j]==INT_MAX), cout<<" INF", else, cout<<distMat[i][j]<<" ", cout<<endl*).

Prikaz rješenja:

```
Unesi broj vrhova 11

Unesi broj rubova 13

Unesi pocetnu vrijednost, zavrsnu i tezinu

1 2 1

1 3 -2

2 4 3

3 4 5

4 1 4

3 7 2

7 8 3

8 9 4

9 10 8
```

```
10 11 3

4 5 -3

5 6 2

6 11 1


Duljina izmedju svih parova prikazana u matrici je
       1 2 3 4 5 6 7 8 9 10 11
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _______
 |1 0 1 -2 3 0 2 0 3 7 15 3
 |2 7 0 5 3 0 2 7 10 14 22 3
 |3 9 10 0 5 2 4 2 5 9 17 5
 |4 4 5 2 0 -3 -1 4 7 11 19 0
 |5    INF   INF   INF   INF0 2    INF   INF   INF   INF3
 |6    INF   INF   INF   INF   INF0    INF   INF   INF   INF1
 |7    INF   INF   INF   INF   INF   INF0 3 7 15 18
 |8    INF   INF   INF   INF   INF   INF   INF0 4 12 15
 |9    INF   INF   INF   INF   INF   INF   INF   INF0 8 11
 |10    INF   INF   INF   INF   INF   INF   INF   INF   INF0 3
 |11    INF   INF   INF   INF   INF   INF   INF   INF   INF   INF0
```

8.3 Johnsonov algoritam za rijetke grafove

Johnsonov algoritam pripada algoritmima najkraćeg puta i bavi se problemom najkraćeg puta glede svih parova. Problem svih parova kod algoritama najkraćeg puta uzima graf s vrhovima i rubovima. Kao izlaz se uzima najkraći put između svakog para vrhova u grafu. Johnsonov algoritam je vrlo sličan Floyd- Warshallovom algoritmu. Međutim, Floyd- Warshall algoritam je najučinkovitiji za guste grafove (koji imaju puno

rubova), dok je Johnsonov algoritam najučinkovitiji za rijetke grafove (koji imaju samo nekoliko rubova).

Razlog zbog kojeg je Johnsonov algoritam bolji za rijetke grafove je da njegova vremenska složenost ovisi o broju rubova u grafu, dok kod Floyd- Warshallovog to nije tako. Johnsonov algoritam radi u $O(V^2 log(V)+ VE)$ vremenu. Dakle, ako je broj rubova mali (graf je rijedak), on će se izvoditi brže od Floyd- Warshallovog algoritma (vremenska složenost kod njega je $O(V^3)$). (Anonymous_19, 2019.).

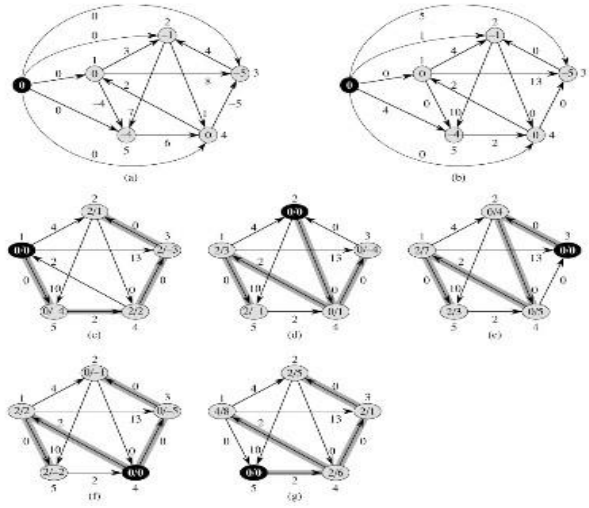
Johnsonov algoritam je zanimljiv jer koristi dva druga algoritma najkraćeg puta kao potprograme. Koristi Bellman- Ford algoritam kako bi preradio ulazni graf (uklanjanje negativnih rubova i otkrivanje negativnih ciklusa). S tim novim, izmijenjenim grafom, koristi Dijkstrin algoritam za izračunavanje najkraćeg puta između svih parova točaka. Izlaz algoritma je tada skup najkraćih putova u izvornom grafu.

Johnsonov algoritam radi, kao i mnogi algoritmi najkraćeg puta, na ulaznom grafu *G*. Ovaj graf ima skup vrhova *V*, koji se preslikavaju na skup rubova *E*. Svaki rub, *(u,v),* pri tome ima funkciju težine (*w*=udaljenost *(u,v)).* Johnsonov algoritam djeluje na usmjerene ili ponderirane grafove. To dopušta da rubovi imaju negativne težine, ali isto tako ne dopušta da postoje negativni ciklusi težine (jer tada ne postoji najkraći put za točke do kojih se može doći tim ciklusom). Također, zanimljivo je primijetiti da Johnsonov algoritam može podržavati negativne grafove težine jer primjerice, Dijkstrin algoritam ne može.

Johnsonov algoritam ima tri glavna koraka: (Anonymous_19, 2019.)

1. U graf se dodaje novi vrh, koji se povezuje rubovima (nula) težine sa svim ostalim vrhovima u grafu.
2. Svi rubovi prolaze kroz proces ponovnog “vaganja” koji eliminira negativne rubove težine.
3. Dodani vrh iz prvog koraka se uklanja i Dijkstrin algoritam se izvodi na svakom čvoru u grafu.

Slika 57.: Prikaz prethodno objašnjena tri koraka, odnosno Johnsonovog algoritma. Prva slika prikazuje prvi korak. druga prikazuje drugi korak, dok treća slika prikazuje treći korak. Dijkstrin algoritam se izvodi na svakom od pet vrhova na grafu.



Izvor: Anonymous_19, 2019.

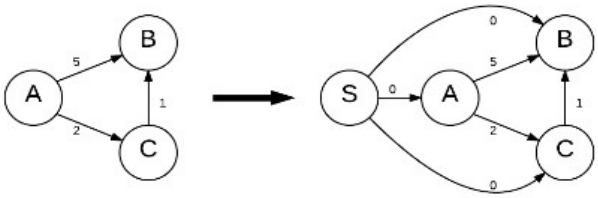
Slika 58.: Prikaz pseudokoda Johnsonovog algoritma

```
Johnson(G)
    1.
    create G` where G`.V = G.V + {s},
        G`.E = G.E + ((s, u) for u in G.V), and
        weight(s, u) = 0 for u in G.V
    2.
    if Bellman-Ford(s) == False
        return "The input graph has a negative weight cycle"
    else:
        for vertex v in G`.V:
            h(v) = distance(s, v) computed by Bellman-Ford
        for edge (u, v) in G`.E:
            weight`(u, v) = weight(u, v) + h(u) - h(v)
    3.
        D = new matrix of distances initialized to infinity
        for vertex u in G.V:
            run Dijkstra(G, weight`, u) to compute distance`(u, v) for all v in G.V
            for each vertex v in G.V:
                D_(u, v) = distance`(u, v) + h(v) - h(u)
        return D
```

Izvor: Anonymous_19, 2019.

Slijedi opis algoritma, tj. tri glavna dijela algoritma. U prvom koraku se dodaje osnovni vrh. Stoga, prvi dio algoritma je prilično jednostavan. Dodavanje novog vrha *(s)* grafu i njegovo povezivanje sa svim drugim točkama s rubom nulte težine lako je dati bilo kojoj metodi prikazivanja grafova.

Slika 59.: Prikaz prethodno opisanog prvog koraka algoritma, odnosno dodavanja osnovnog vrha



Izvor: Anonymous_19, 2019.

Drugi korak algoritma se temelji na ponovnom “vaganju” rubova. Taj korak se smatra najnovijim dijelom ovog algoritma. Osnovni koncept je sljedeći: ako su svi rubovi u grafu nenegativni, izvođenje Dijkstrinog algoritma na svakom vrhu je najbrži način za rješavanje problema najkraćeg puta svih parova. Međutim, ako neki rubovi imaju negativne težine, može ih se ponovo “izvagati” tako da bi vrijedila sljedeća definicija.

Ponovno “vaganje” je proces kojim se mijenja težina rubova kako bi se zadovoljila sljedeća dva svojstva:

1. Za sve parove točaka *(u,v)* u grafu, ako je određeni najkraći put između tih vrhova prije ponovnog vaganja, ista također mora biti najkraća putanja između tih vrhova nakon ponovnog vaganja.
2. Za sve rubove *(u,v)* u grafu, težina *(u,v)* mora biti nenegativna.

Ovaj prethodni korak koristi Bellman- Ford algoritam kako bi pomogao “ojačati” rubove. Koristeći Bellman- Ford algoritam u tom koraku, Johnsonov algoritam je također u stanju detektirati (pronaći) negativne cikluse težina.

Treći korak algoritma služi za pronalaženje svih parova najkraćeg puta. Konačno, Dijkstrin algoritam se izvodi na svim točkama kako bi se pronašao najkraći put. To je moguće jer su težine pretvorene u nenegativne težine. Naposljetku, važno je transformirati te težine putanje natrag u izvorne težine putanja tako da se na kraju algoritmu vrati precizna težina puta. To se radi na kraju algoritma jednostavnim preokretanjem procesa vaganja. Obično se vraća struktura podataka kao što je matrica. Na svakoj ćeliji *(i,j)* ove matrice se nalazi najkraći put od vrha *i* do vrha *j*.

Da bi se dokazalo da je strategija ponovnog vaganja (ponderiranja) održiva, obje se osobine ponovnog vaganja moraju dokazati kao istinite. Prvo svojstvo je očuvanje

najkraćih putova. To svojstvo navodi da najkraći put između bilo koje dvije točke *(u,v)* treba ostati najkraća putanja nakon ponovnog vaganja. Funkcija *(h(v))* je funkcija koja mapira vrh na broj (težina *(u,v)*=težina *(u,v)*+ *h(u)- h(v).* Neka je $p=\{v_0, v_1,...,v_k\}$ put od v_0 do v_k. Dakle, *p* je najkraći put s izvornom težinskom funkcijom ako i samo ako je najkraći put s "vagajućom" funkcijom. Postoji i težina *(p)*= težina *(p)*+ $h(v_0)$- $h(v_k)$, *sum* $\{i\}^k$ *težina* $^{v\_s}$ = $\{i\}^k$, težina $(v\ \{i-1\}, v_i)$ + $h(v\ \{i-1\} - h(v_i))$, suma= težina *(p)* + $h(v_0)$ - $h(v_k)$. Dakle, svaki put od v_0 do v_k koji je najkraći put koristi izvornu težinu (težinsku funkciju, težina *(p)).* Ista predstavlja najkraći put pomoću funkcije ponovnog vaganja (težina^p). To je zato što $(h(v_0))$ i $(h(v_k))$ ne ovise o putu. Isto tako, važno je dokazati da svaki put *(p)* koji ima negativni ciklus težine koristeći izvornu funkciju težine također ima negativni ciklus koristeći funkciju ponovnog vaganja (tako da se mogu otkriti negativni težinski ciklusi nakon transformacije). Ciklus *c* se definira sljedećom funkcijom: *težina*^{c}= *težina c* + $h(v_0)$ - $h(v_k)$, težina *c*. Dakle, svaki ciklus *(c)* koji ima negativnu težinsku vrijednost prije transformacije.

Drugo svojstvo obuhvaća sve precijenjene rubove koji moraju imati nenegativnu vrijednost. Dodatno, valja zapamtiti (kod prvog koraka) konstruiranje novog grafa. Isti je imao dodatni vrh, koji je imao rubove koji idu od nje do svih ostalih vrhova, a svaki od tih rubova je imao nultu težinu. Važna posljedica toga je da će svi najkraći putevi u ovom grafu sadržavati samo *(s)* ako je *(s)* izvor (jer nema rubova u *(s)).* Pretpostavlja se da *G* nema negativne cikluse težine, stoga se može definirati sljedeće: *h(v)=udaljenost (s,v)* za sve *v* u G^V. Prema nejednakosti, dobiva se *h(v) h(u)* + težina *(u, v),* za sve rubove u G^E. Dakle, ako se jednostavno definira jednadžba ponovnog vaganja, dobiva se težina$^{(u,v)}$= težina *(u, v)* + *h(u) - h (v),* pa je onda drugo svojstvo zadovoljeno.

Analiza Johnsonovog algoritma je jednostavna ako se razumije analiza Bellman-Fordovog i Dijkstrinog algoritma. Važno je napomenuti da se često u analizi implementacije Dijkstrinog algoritma koristi Fibonaccijeva hrpa kao svoj prioritetni red (minimalna), jer je to najučinkovitija struktura za tu vrstu algoritma. Iako će čak i korištenje jednostavne minimalne hrpe donijeti bolji rezultat od Floyd- Warshalla. Prvi korak u algoritmu je jednostavan i traje *O (V)* jer treba stvoriti novi rub sa svim točkama na grafu. Drugi korak algoritma, Bellman- Ford, radi u *O (VE)* vremenu, kao i sam Bellman- Ford. Proces ponovnog vaganja grafa traje jednako vremena. Posljednji korak algoritma je Dijkstrin algoritam koji se izvodi na svim *V* vrhovima. Koristeći

Fibonaccijevu hrpu, svaka od tih iteracija traje *O (V+E).* Tako je ukupna složenost *O (V^2log(V)+ VE)*. Valja imati na umu da ako bi se koristila minimalna hrpa kao red prioriteta minimalnog prioriteta u Dijktrinom algoritmu, svaka bi iteracija Dijkstrinog algoritma trajala *O (E log(V)),* pa bi ukupna vremenska složenost bila *O(V E log(V)).*

9. Grafovi s negativnim rubnim troškovima

Ako graf ima negativne troškove ruba, onda Dijkstrin algoritam ne radi. Problem je u tome da se jednom, kada je vrh, *u*, proglašen poznatim, moguće je da iz nekog drugog nepoznatog vrha, *v*, postoji put natrag do *u* koji je vrlo negativan. U takvom slučaju, uzimajući put od *s* do *v* natrag u *u* je bolje nego od *s* do *u* bez korištenja *v*. (Weiss, 2014., str.400)

Primamljivo rješenje toga je dodavanje konstante (delta, Δ) svakom trošku ruba, čime se uklanjaju negativni rubovi, izračunava najkraća putanja na novom grafu, a zatim se taj rezultat koristi na izvorniku. Naivna implementacija ove strategije ne funkcionira jer putevi s mnogo rubova postaju teži od puteva s nekoliko rubova.

Kombinacija ponderiranih i neponderiranih algoritama riješit će problem, ali po cijenu drastičnog povećanja vremena izvođenja. Zaboravlja se na koncept poznatih vrhova, jer taj algoritam mora moći promijeniti svoje mišljenje. Počinje se tako da se stavi *s* na red. Zatim, u svakoj fazi, *dequeue a vertex v*. Nalaze se svi vrhovi *w* koji su susjedni *v* takvi da $d_w > dv + cv, w.$ Ažurira se d_w i p_w i stavlja se *w* na red ako već nije tamo. Također, može biti postavljen za svaki vrh kako bi ukazao na prisutnost u redu. Postupak se ponavlja dok se red ne isprazni.

Iako algoritam funkcionira ako ne postoje negativni ciklusi, više nije točno da se kod unutarnje petlje izvršava jednom po rubu. Svaki se vrh može odvojiti najviše u |*V*| vremena, tako da je vrijeme rada *O (|E| · |V|)* ako se koriste popisi susjedstva. To je prilično povećanje u odnosu na Dijkstrin algoritam, pa je sretno što u praksi troškovi ruba nisu negativni. Ako su prisutni negativni ciklusi, tada će pisani algoritam biti beskonačan. Zaustavljanjem algoritma nakon što je bilo koji vrh uklonjen |*V*| + *1* puta, može se jamčiti završetak.

Slika 60.: Prikaz pseudokoda za algoritam ponderiranog najkraćeg puta s negativnim troškovima ruba

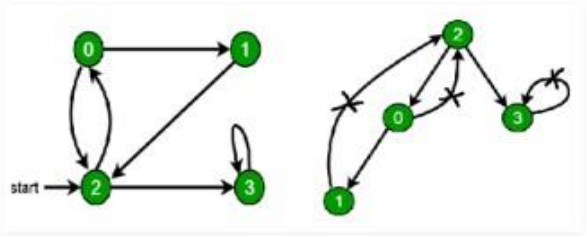
```
void Graph::weightedNegative( Vertex s )
{
    Queue<Vertex> q;

    for each Vertex v
        v.dist = INFINITY;

    s.dist = 0;
    q.enqueue( s );

    while( !q.isEmpty( ) )
    {
        Vertex v = q.dequeue( );

        for each Vertex w adjacent to v
            if( v.dist + cvw < w.dist )
            {
                // Update w
                w.dist = v.dist + cvw;
                w.path = v;
                if( w is not already in q )
                    q.enqueue( w );
            }
    }
}
```

Izvor: Weiss, 2014., str.401

9.1 Otkrivanje ciklusa

Otkrivanje ciklusa kod usmjerenog grafa se provjerava tako da se vidi sadrži li graf uopće ciklus ili ne. Ovdje bi funkcija trebala vratiti istinu ako zadani graf sadrži bar jedan ciklus, inače se vrati laž. Primjerice, sljedeći graf ima tri ciklusa (0->2->0, 0->1->2->0, 3->3), tako da funkcija vrati istinu.

Slika 61.: Prikaz grafa koji sadrži tri ciklusa



Izvor: geeksforgeeks_8, n_d

Pretraživanje u dubinu se može koristiti za otkrivanje ciklusa na grafu. Pretraživanje u dubinu za povezani graf proizvodi stablo. Stoga, u grafu će postojati ciklus samo ako je na grafu prisutan zadnji rub. Zadnji rub predstavlja rub koji je od čvora do samog sebe (engl. self- loop) ili djeluje od jednog od njegovih predaka u stablu koje proizvodi

pretraživanje u dubinu. Točnije, to bi bio rub koji ide od vrha prema samom sebi. Na prethodno prikazanom grafu se nalaze tri stražnja ruba (na slici označena križnim znakom). Valja primijetiti da ta tri stražnja ruba ukazuju na tri ciklusa koja su prisutna u grafu. Ovdje postoji još pojam samostojeće petlje koja predstavlja rub od bilo kojeg potomka pa natrag do vrha.

Algoritam pretraživanja u dubinu obuhvaća sljedeća četiri koraka: (Anonymous_20, 2018.).

1. Treba napraviti pretraživanje u dubinu iz svakog vrha.
2. Za pretraživanje u dubinu iz svakog vrha valja pratiti posjećivanje vrhova u nizu rekurzija.
3. Ako se naiđe na vrh koji je već prisutan u rekurzivnom stogu, onda se pronašao ciklus.
4. Definira se funkcija *visited[]* za pretraživanje u dubinu da bi se pratile već posjećene točke.

Postoje dvije razlike između rekurzivnog stoga te funkcije *visited[]* koje se ovdje koriste, a to su:

1. Funkcija *visited[]* se koristi za praćenje već posjećenih vrhova tijekom pretraživanja u dubinu.
2. Rekurzivni stog se koristi iz praćenja posjeta vrhova tijekom pretraživanja u dubinu iz određenog vrha. Kada se ciklus ne pronađe iz tog vrha, pokušat će se napraviti pretraživanje u dubinu iz drugih vrhova.

Za neusmjereni graf se dobiva pomoću pretraživanja u dubinu šuma kao izlaz. Da bi se ovdje otkrio ciklus, mora se provjeriti ciklus u pojedinačnim stablima provjeravajući rubove. (geeksforgeeks_8, n_d).

Problem otkrivanja u neusmjerenim grafovima se može riješiti na više načina:

1. Topološko sortiranje.
2. Pretraživanje u dubinu.
3. Nepovezani skupovi.

Za otkrivanje zadnjeg ruba, mogu se pratiti točke koje su trenutno u rekurzivnom stogu funkcije za pretraživanje u dubinu. Ako se dođe do vrha koji je već u rekurzivnom stogu,

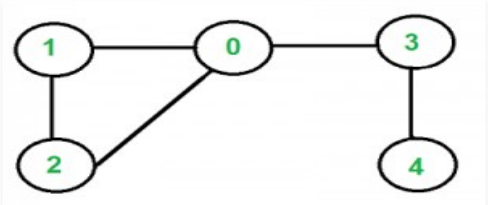
tada će u stablu postojati ciklus. Preciznije, rub koji povezuje trenutni vrh s vrhom u rekurzivnom stogu je zadnji rub.

Vremenska složenost za otkrivanje ciklusa kod usmjerenog grafa je *O (V+ E),* kao i kod pretraživanja u dubinu.

Kao što postoji otkrivanje ciklusa u usmjerenom grafu, tako postoji i otkrivanje ciklusa u neusmjerenom grafu.

Primjerice, sljedeći graf ima ciklus 1- 0- 2- 1.

Slika 62.: Prikaz grafa koji ima ciklus 1- 0- 2- 1



Izvor: geeksforgeeks_9, n_d

Kao i kod usmjerenog grafa, može se koristiti pretraživanje u dubinu za otkrivanje ciklusa u neusmjerenom grafu. Njegova vremenska složenost je *O (V+ E),* tj. ista kao i kod neusmjerenog grafa. Ovdje se pravi prijelaz putem pretraživanja u dubinu za zadani graf. Konkretno, za svaki vrh *v*, ako postoji susjedni *u* takav da je već posjećen i nije roditelj od *v*, onda u grafu postoji ciklus. Ako se ne pronađe prethodni slučaj za bilo koji vrh, onda se zaključuje da nema ciklusa. Pretpostavka tog slučaja je da ne postoje paralelni rubovi između bilo koje dvije točke.

Pretraživanje u dubinu se ostvaruje u tri koraka: (Anonymous_21, 2018.)

1. Prvo se ostvaruje pretraživanje u dubinu iz svakog vrha.
2. Za vrijeme pretraživanja u dubinu, za bilo koju trenutnu točku *“x”* (koja trenutno posjećuje vrh) ako je prisutan susjedni vrh *“y”* koji je već posjećen, a *“y”* nije izravni roditelj od *“x”,* u grafu se nalazi ciklus.
3. Pretpostavlja se da je vrh *x’* i *xy’* i da postoji rub između njih. Sada se ne vrši pretraživanje u dubinu iz *“x”,* nakon što se dođe do *“y”.* Susjedni vrh je *“x”,* budući da je već posjećen trebao bi biti ciklus, ali zapravo ne postoji ciklus jer je *“x”* roditelj od *“y”.* Iz tog razloga treba ignorirati posjećeni vrh jer je on roditelj od trenutnog vrha.

9.1.1 Problem s pronalaženjem unije

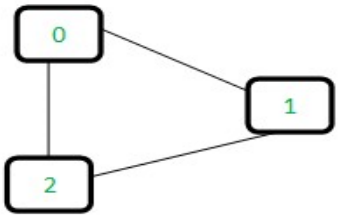
Struktura za disjunktni skup podataka kod neusmjerenog grafa je struktura podataka koja prati skup elemenata koji su podijeljeni u niz podskupina koji se ne presijecaju. Algoritam pronalaženja unije je algoritam koji izvodi dvije korisne operacije na takvoj strukturi podataka.

Ovdje valja odrediti koji je podskup određeni element. To se može koristiti za određivanje jesu li dva elementa u istom podskupu.

Kod unije se spajaju dva podskupa u jedan podskup.

Algoritam za pronalaženje unije se može iskoristiti za provjeru sadrži li neusmjereni graf ciklus ili ne. Ovo je još jedna metoda koja se temelji na otkrivanju unije. Ova metoda pretpostavlja da graf sadrži put iz samog sebe u sebe. Također, mogu se pratiti podskupovi u jednodimenzionalnom nizu, koji se primjerice, može nazvati *roditelj [].*

Slika 63.: Prikaz grafa



Izvor: geeksforgeeks_10, n_d

Za svaki rub se treba napraviti podskup koristeći oba vrha ruba. Ako su oba vrha u istom podskupu, nalazi se ciklus. U početku se sva mjesta roditeljskog polja inicijaliziraju na –1 (znači da postoji samo jedna stavka u svakom podskupu).

U sljedećem koraku se trebaju obraditi svi rubovi (jedan po jedan). Prvo se obrađuje rub *0-1*. Kod njega valja pronaći podskupove u kojima su točke 0 i 1. Budući da se nalaze u različitim podskupovima, uzima ih se putem unije. Za uzimanje unije, treba se stvoriti čvor 0 kao roditelj čvora 1, ili obrnuto. Drugo se obrađuje rub *1- 2*. Ovdje je 1 u podskupu 1, a 2 je u podskupu 2. Dakle, ovdje se uzima unija. Treće se obrađuje rub *0- 2*. Tu je 0 u podskupu 2 i 2 je također u podskupu 2. Dakle, uključujući ta dva ruba formira se ciklus.

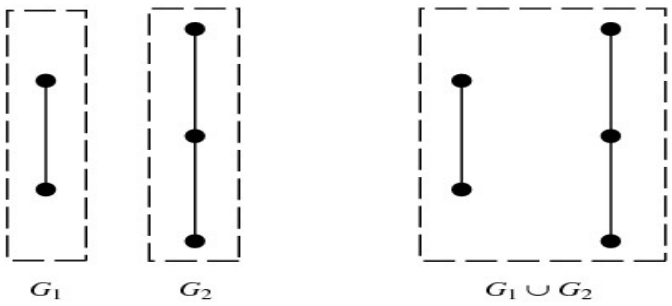
Što se tiče vremenske složenosti same implementacije unije, ista iznosi *O (n)* u najgorem slučaju. Ona se može poboljšati tako da iznosi *O (logn),* na način da se unija klasificira prema rangu ili visini.

Sama opća postavka pronalaženja problema unije se krije u održavanju zbirke nepovezanih skupova $\{S_1,\ldots,S_k\}$ sa sljedećim operacijama (Anonymous_22, n_d, str. 79 i 80):

1. Stvaranje skupa (engl. *MakeSet (x)).* Ovdje se stvara novi skup *{x}.*
2. Unija (engl. Union). Ovdje se zamijeni skup *x* (ili *S*), a skup *y* (*S'*) s jednim skupom, tj. *S U S'*.
3. Pronalaženje (engl. *Find (x*)). Ovdje se vraća jedinstveni ID za skup koji sadrži *x* (ovo je samo neki reprezentativni element tog skupa).

Primjerice, obzirom na prethodne operacije, može se implementirati Kruskalov algoritam na sljedeći način. Skupovi S_i će biti skupovi vrhova različitih stabala u šumi. Postupak započinje s *MakeSet (v)* za sve točke *v* (svaka točka je u svom stablu). Kada se uzme u obzir neki rub *(v, w)* u algoritmu, samo se testira je li *Find (v) jednak Find (w)*. Ako su jednaki, to znači da su *v* i *w* već u istom stablu pa se preskače rub. Ako nisu jednaki, ubaci se rub u šumu i izvodi se *Union (v, w)* operacija. Zajedno se rade *MakeSet* operacije.

Slika 64.: Prikaz unije grafova



Izvor: Anonymous_23, n_d

Unija $G= G_1 U G_2$ grafova G_1 i G_2 sa skupovima nepovezanih točaka V_1 i V_2 te rubnim skupovima X_1 i X_2 je graf s $V= V_1 U V_2$ *i* $X= X_1 U X_2$. Ova se operacija često naziva unija grafova. Unija grafova se primjerice, može izračunati u Wolfram jeziku pomoću unije skupova $[g_1, g_2, \ldots]$. Također, ovdje valja imati na umu da funkcija Wolfram

Language Graph Union *[g_1, g_2]* bi izvršila drugačiju operaciju od uobičajene unije grafova. Unija *n* grafova predstavlja kopiju grafova.

9.2 Aciklički grafovi

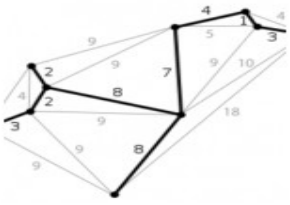
Pridjev ***acikličko*** se koristi za opisivanje grafova koji nemaju ciklusa ili zatvorenog puta. Drugim riječima, to je put bez ponovljenih vrhova (čvorova koji tvore graf, ili veze između vrhova), isključujući početne i završne točke. (Anonymous_24, n_d).

U računalnoj se znanosti pojam "aciklički" koristi pretežito kod usmjerenih grafova. Tehnički, usmjereni aciklički graf je graf koji se formira povezivanjem različitih vrhova s rubovima koji su usmjereni na način na koji se ne dopušta navigacija kroz slijed koji može imati vrh koji prolazi kroz njega više od dva puta (nema zatvorenog puta). Ovi se grafovi još mogu koristiti za otkrivanje zastoja (ilustrira da resursi moraju čekati da se neki proces nastavi).

"Graf" u ovom smislu znači strukturu napravljenu od čvorova i rubova. Čvorovi su obično označeni krugovima ili ovalima (iako tehnički mogu biti prikazani bilo kojim oblikom po vlastitom izboru). Rubovi su veze između čvorova (povezuju dva čvora). Obično su predstavljene linijama ili crtama sa strelicama. Usmjereni aciklički grafovi se temelje na osnovnim acikličkim grafovima. (Anonymous_25, 2016.).

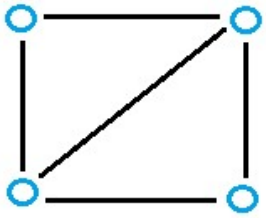
Točnije, aciklički graf je graf bez ciklusa (ciklus predstavlja kompletan krug). On prati graf od čvora do čvora na način da nikada ne posjeti isti čvor dva puta.

Slika 65.: Prikaz grafa (debela crna crta) koji je acikličan (stablo) jer nema ciklusa (kompletne krugove)



Izvor: Anonymous_25, 2016.

Slika 66.: Prikaz grafa koji ima kompletan krug te nije acikličan



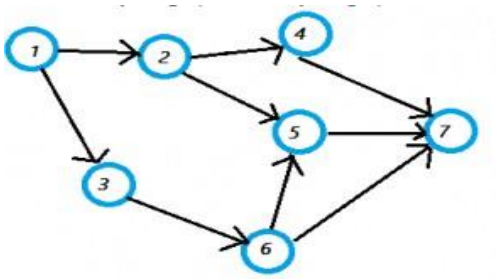
Izvor: Anonymous_25, 2016.

Koncept usmjerenog acikličkog grafa koristi se za dizajn igara riječi poput Scrabblea i znanstvenih istraživanja temeljenih na biologiji i genetici. Isti se također koristi u:

1. Izgradnji modela u matematici.
2. Računalnoj znanosti.
3. Elektroničkim sklopovima.
4. Kompiliranju operacija.
5. Izračunavanju povezanih vrijednosti na obrascima, i sl.

Usmjereni aciklički grafovi se koriste u modelima kako bi ilustrirali protok informacija kroz sustav. Preciznije, ova vrsta algoritma je bolja alternativa drugim tehnikama u strukturi podataka pružajući optimizaciju korištenja memorije i poboljšanja performansi.

Slika 67.: Prikaz usmjerenog acikličkog grafa



Izvor: Anonymous_25, 2016.

U gornjem grafu postoje tri dijela:

1. Cjelobrojne vrijednost (engl. Integer). Oni predstavljaju skup za vrhove.
2. Skup vrhova (engl. Vertices set). Sadrže skup={1, 2, 3, 4, 5, 6, 7}.
3. Skup rubova (engl. Edge set). Sadrže skup={(1,2), (1,3), (2,4), (2,5), (3,6), (4,7), (5,7), (6,7)}.

Usmjereni aciklički graf ima topološki poredak. Konkretno, to znači da su čvorovi uređeni tako da početni čvor ima nižu vrijednost od završnog čvora. On ima jedinstveni topološki poredak ako ima usmjerenu putanju koja sadrži sve čvorove. U tom slučaju redoslijed je isti kao i redoslijed kojim se čvorovi pojavljuju na putu.

Ciklus je put koji prolazi kroz niz točaka, tako da su početni i završni vrhovi ista točka. Ako graf nema takve cikluse, onda se isti naziva acikličkim. Primjerice, valja razmotriti tri točke *(X, Y i Z)* koje su povezane u grafu. Dok se prelazi iz bilo kojeg od triju vrhova kroz njegovu strukturu na različite moguće načine, onda se ne može vratiti natrag na istu početnu točku bez posjete bilo kojem vrhu (bez početne točke), onda je to aciklički graf.

Duljina najkraćeg ciklusa i opseg acikličkog grafa se definira kao beskonačnost. Primjeri acikličkih grafova su stabla i šume. Aciklički i neusmjereni graf s bilo kojim dvjema točkama, koji su povezani samo jednim putem, predstavljaju stablo. Obiteljsko stablo je dobar primjer koncepta usmjerenog acikličkog stabla. Šuma je neusmjereni graf čiji su podskupovi stabla. (Anonymous_24, n_d).

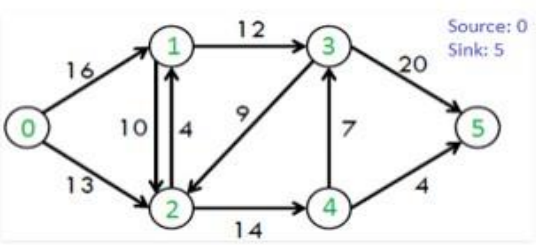
Aciklički grafovi su bipartitni. Ovdje se stablo još može definirati kao povezani aciklički graf, a eventualno odspojeni aciklički graf je poznat kao šuma (skup stabala). Naime, postoji još i pojam unicikličkog grafa koji predstavlja graf s jednim ciklusom. (Anonymous_26, n_d).

Stabla se još definiraju kao aciklički grafovi kojima je jedna ili više “grana” stabla isključena.

10. Maksimalni protok

Problemi s maksimalnim protokom uključuju pronalaženje izvodljivog protoka kroz mrežu s jednim izvorom ili jednostrukim protokom koji je maksimalan.

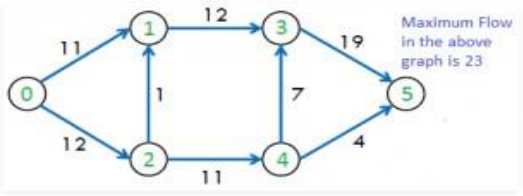
Slika 68.: Prikaz grafa koji objašnjava što gore opisana definicija znači



Izvor: geeksforgeeks_11, n_d

Na prethodnoj slici je svaki rub označen kapacitetom, tj. maksimalnom količinom stvari koju može nositi. Ovdje je cilj otkriti koliko se stvari može povući od vrha (izvora) do drugog vrha (ponora).

Slika 69.: Prikaz nastavka prethodnog grafa (maksimalni protok u grafu je 23)



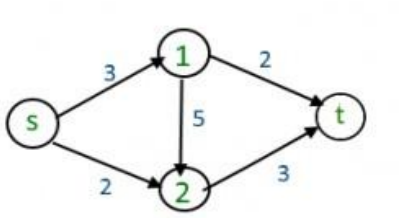
Izvor: geeksforgeeks_11, n_d

Slijede različiti pristupi rješavanju problema:

1. Pristup putem naivnog algoritma pohlepe. Ovaj tip algoritma ne može proizvesti optimalan ili točan rezultat. Poželjan pristup rješavanju problema maksimalnog protoka je započeti s protokom (koji sadrži sve nule) i pohlepno proizvoditi tokove sa sve većom vrijednošću. Drugim riječima, krenuti od najmanje vrijednosti prema najvećoj. Prirodni način prelaska s jednog na drugi je slanje više tokova putem nekog puta (primjerice, od *s* do *t*). Slijedi objašnjenje pseudokoda ovog algoritma. Dakle, *E* predstavlja broj ruba, *f(e)* protok ruba te *C(e)* kapacitet ruba. Prvo se treba inicijalizirati *max_flow=0, f(e)= 0* za svaki rub *'e'* u *E*. Potom treba ponoviti traženje *s-t* puta *P* dok postoji. U ovom koraku se prvo, u slučaju da postoji put, mora pronaći od *s* do *t* put koristeći pretraživanje u širinu ili pretraživanje u dubinu. Postoji put ako je *f(e)< C(e)* za svaki rub na putu. Ako put nije pronađen, onda se treba vratiti *max_flow* ili se treba pronaći minimalna vrijednost za put *P*. Potom može doći do toga da je tijek ograničen

najmanjim preostalim rubom na putu *P* (*protok=min (C(e)- f(e))* za put *P*, *max_flow* += *flow*). Za sve rubove toka se povećava put *(f(e)* += *protok*). Naposljetku, treba se vratiti *max_flow*. Također, traženje puta samo treba odrediti da li postoji li ne put *s-t* u podgrafu rubova *E* s *f(e)* < *C(e).* To se lako radi u linearnom vremenu koristeći pretraživanje u širinu ili pretraživanje u dubinu.

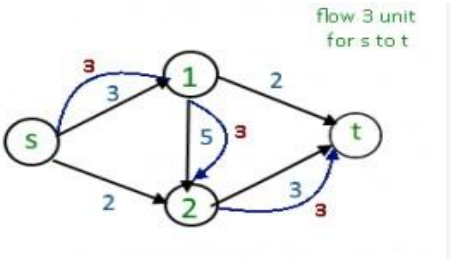
Slika 70.: Prikaz pristupa putem naivnog algoritma pohlepe



Izvor: geeksforgeeks_11, n_d

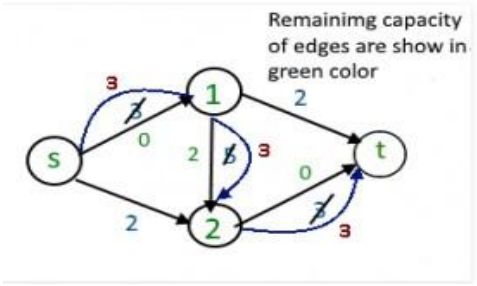
To je put od izvora *(e)* do ponora *(t),* tj. *s* ide u 1, pa u 2 te do *t* (s maksimalnim protokom od tri jedinice, isti je prikazan u plavoj boji), nakon uklanjanja svih beskorisnih rubova u grafu. Za gornji graf ne postoji put od izvora do ponora tako da maksimalni protok iznosi tri jedinice (a maksimalni protok je zapravo pet).

Slika 71.: Prikaz puta od izvora *(e)* do ponora *(t),* tj. *s* ide u 1, pa u 2 te do *t*



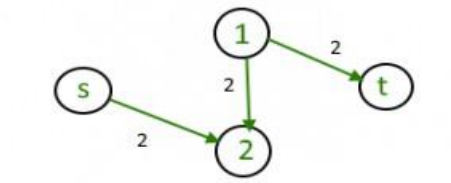
Izvor: geeksforgeeks_11, n_d

Slika 72.: Prikaz maksimalnog protoka koji iznosi 3



Izvor: geeksforgeeks_11, n_d

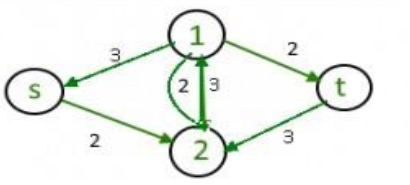
Slika 73.: Uklanjanje beskorisnih rubova u grafu



Izvor: geeksforgeeks_11, n_d

Rezidualni graf se zasniva na ideji proširivanja naivnog pohlepnog algoritma dopuštajući „poništavanje“ operacija. Primjerice, od točke gdje se taj algoritam (gornja slika) želi usmjeriti još dvije jedinice protoka duž ruba *(s, 2),* a zatim natrag uz rub *(1, 2),* poništavajući dvije od tri jedinice koje su preusmjerile prethodnu iteraciju. Posebice, uz rub *(1, t),* unatrag *(f(e))* te prednji rub: *(c(e) – f(e)).*

Slika 74.: Prikaz rezidualne mreže



Izvor: geeksforgeeks_11, n_d

Ovdje je potrebno definirati način da se formalno odrede dopuštene operacije „poništavanja“. To motivira sljedeću jednostavnu ali važnu definiciju rezidualne mreže. Naime, ideja je da, s obzirom na graf *G* i tok *f* u njemu, se formira nova protočna mreža

G_f koja ima isti skup vrha od *G* i koja ima dva ruba za svaki rub *G*. Graničnik *e=(1, 2)* od *G* koji nosi tok *f(e)* i ima kapacitet *c(e)* (gornja slika). Također, ovdje se stvara „prednji rub“ od G_f s kapacitetom *c(e) – f(e)* (preostali) i unatrag s *(2, 1)* od G_f s kapacitetom *f(e)* (količina prethodno usmjerenog toka koji se može poništiti). Izvor *(e)* – ponor *(t)* puta s *f(e)< c(e)* za sve rubove (od naivnog pohlepnog algoritma), koji odgovaraju posebnom slučaju *st* puta G_f koji obuhvaćaju samo rubove koji se nalaze naprijed.

Ideja rezidualnog grafa je korištena u Ford- Fulkersonovom te Dinicevom algoritmu.

10.1 Maksimalni protok minimalne cijene

Iako je moguće mnogo različitih maksimalnih protoka kroz mrežu, bira se onaj koju diktira algoritam koji se trenutno koristi. Primjerice, sljedeća slika ilustrira dva moguća maksimalna toka neke mreže. Valja imati na umu da se u prvom slučaju rub *(ab)* uopće ne koristi; samo u drugom slučaju svi rubovi prenose neki tok. Algoritam po pretraživanju po širini dovodi do prvog maksimalnog protoka i završava potragu za maksimalnim protokom nakon što ga se identificira. Međutim, u mnogim rješenjima to nije dobra odluka. Ako postoji više mogućih maksimalnih tokova, to ne znači da je bilo koja od njih jednako dobra.

Slika 75.: Prikaz dva moguća maksimalna toka neke mreže

Izvor: Drozdek, 2001., str 415

Valja razmotriti sljedeći primjer. Ako su rubovi ceste između nekih mjesta, onda nije dovoljno znati da cesta ima jednu ili dvije trake za odabir odgovarajuće rute. Ako je udaljenost *(a, t)* vrlo duga, a udaljenost *(a, b)* i udaljenost *(b, t)* relativno kratka, onda je bolje razmotriti drugi maksimalni protok (druga slika) kao održivu opciju, a ne kao

prvu (prva slika). Međutim, to možda nije dovoljno. Kraći put ne može imati pločnik: može biti blatnjav, brdovit, blizu lavina, ponekad blokiran kamenjem, i sl. Stoga je korištenje udaljenosti kao jedinog kriterija za odabir ceste nedovoljno. Kružni put nas može dovesti do odredišta brže i jeftinije (primjerice, samo vrijeme i benzin).

Jasno je da je potreban treći parametar za rub: trošak prijenosa jedne jedinice protoka kroz taj rub. Sada je problem kako pronaći maksimalni protok uz minimalne troškove. Formalnije, ako se za svaki rub *e* određuje trošak *(e)* slanja jedne jedinice protoka tako da *cost n * trošak (e)* za prijenos n jedinica toka preko ruba *e*, tada se mora pronaći maksimalni protok *f* minimalnog troška ili toka (sljedeća formula).

Slika 76.: Prikaz formule za pronalazak maksimalnog protoka *f* minimalnog troška

$$cost(f) = \min\{\sum_{e \in E} f(e) \cdot cost(e) : f \text{ is a maximum flow}\}$$

Izvor: Drozdek, 2001., str. 415

Pronalaženje svih mogućih maksimalnih tokova i uspoređivanje njihovih troškova nije izvedivo rješenje jer količina rada za pronalaženje svih takvih tokova može biti previsoka. Potrebni su algoritmi koji nalaze ne samo maksimalni protok već i maksimalni protok uz minimalne troškove.

Jedna strategija temelji se na sljedećem teoremu, dokazanom najprije W.S. Jewell, R.G. Buscaker i P.J. Gowen, te implikacija koju su koristili M. Iri (Ford i Fulkerson 1962.). (Drozdek, 2001., str. 415).

Teorem: Ako je *f* protok minimalnog troška s vrijednošću protoka *v* i *p* je put povećanja minimalnih troškova koji šalje protok vrijednosti jedan iz izvora u odredište, tada je protok *f+p* minimalan i njegova vrijednost protoka je *v+1*.

Teorem bi trebao biti intuitivno jasan. Ako se odredio najjeftiniji način slanja *v* jedinica toka kroz mrežu i nakon toga se pronašao put za slanje jedne jedinice toka od izvora do odredišta, tada se pronašao najjeftiniji način slanja *v+1* jedinica koristeći put koji je kombinacija već određene rute i upravo pronađenog puta. Ako ova povećavajuća putanja omogućuje slanje jedne jedinice za minimalni trošak, ona također omogućuje slanje dvije jedinice uz minimalnu cijenu, a također i tri jedinice, do *n* jedinica, pri čemu je *n* maksimalna količina jedinica koje se mogu slati putem ovog puta, a to je:

Slika 77.: Prikaz formule za slanje jedne jedinice za minimalni trošak

$$n = \min\{capacity(e) - f(e) : e \text{ is an edge in minimum cost augmenting path}\}$$

Izvor: Drozdek, 2001., str. 416

To također sugerira kako se može sustavno nastaviti s pronalaženjem najjeftinije maksimalne rute. Počinje se sa svim protocima postavljenim na nulu. U prvom prolazu pronalazi se najjeftiniji način slanja jedne jedinice, a zatim se šalje što više jedinica po tom putu. Nakon druge iteracije, pronalazi se putanja za slanje jedne jedinice po najnižoj cijeni i šalje se kroz tu stazu onoliko jedinica koliko taj put može držati i tako dalje, sve dok se ne mogu napraviti daljnja slanja iz izvora ili odredišta gdje se ne može prihvatiti više protoka.

Valja napomenuti da je problem pronalaženja maksimalnog protoka minimalnog troška donekle sličan problemu pronalaženja najkraćeg puta, budući da se najkraći put može shvatiti kao put s minimalnim troškom. Stoga je potrebna procedura za pronalaženje najkraćeg puta u mreži tako da se kroz taj put može poslati što je moguće više protoka. Stoga, upućivanje na algoritam koji rješava problem najkraćeg puta ne bi trebao biti iznenađujući. Modificira se Dijkstrin algoritam koji se koristi za rješavanje problema s jednim i samo jednim najkraćim putem.

Slika 78.: Prikaz pseudokoda modificiranog Dijkstrinog algoritma

```
modifiedDijkstraAlgorithm(network, s, t)
   for all vertices u
      f(u) = 0;
      cost(u) = ∞;
   set flows of all edges to 0;
   label(s) = (null,∞,0);
   labeled = null;
   while (1)
      v = a vertex not in labeled with minimal cost(v);
      if v == t
         if cost(t) == ∞ // no path from s to t can be found;
            return failure;
         else return success;
      add v to labeled;
      for all vertices u not in labeled and adjacent to v
         if forward(edge(vu)) and slack(edge(vu)) > 0 and cost(v)+ cost(vu) < cost(u)
            label(u) = (v+, min(slack(v), slack(edge(vu)), cost(v)+ cost(vu))
         else if backward(edge(vu)) and f(edge(uv)) > 0 and cost(v)− cost(uv) < cost(u)
            label(u) = (v−, min(slack(v), f(edge(uv)), cost(v) − cost(uv));

maxFlowMinCostAlgorithm(network with source s and sink t)
   while modifiedDijkstraAlgorithm(network,s,t) is successful
      augmentPath(network,s,t);
```

Izvor: Drozdek, 2001., str. 416

Operacija *modifiedDijkstraAlgorithm ()* prati tri stvari u isto vrijeme tako da je oznaka za svaki vrh trostruka:

Slika 79.: Prikaz formule oznake za svaki vrh koji je trostruk

$$label(u) = (parent(u), flow(u), cost(u))$$

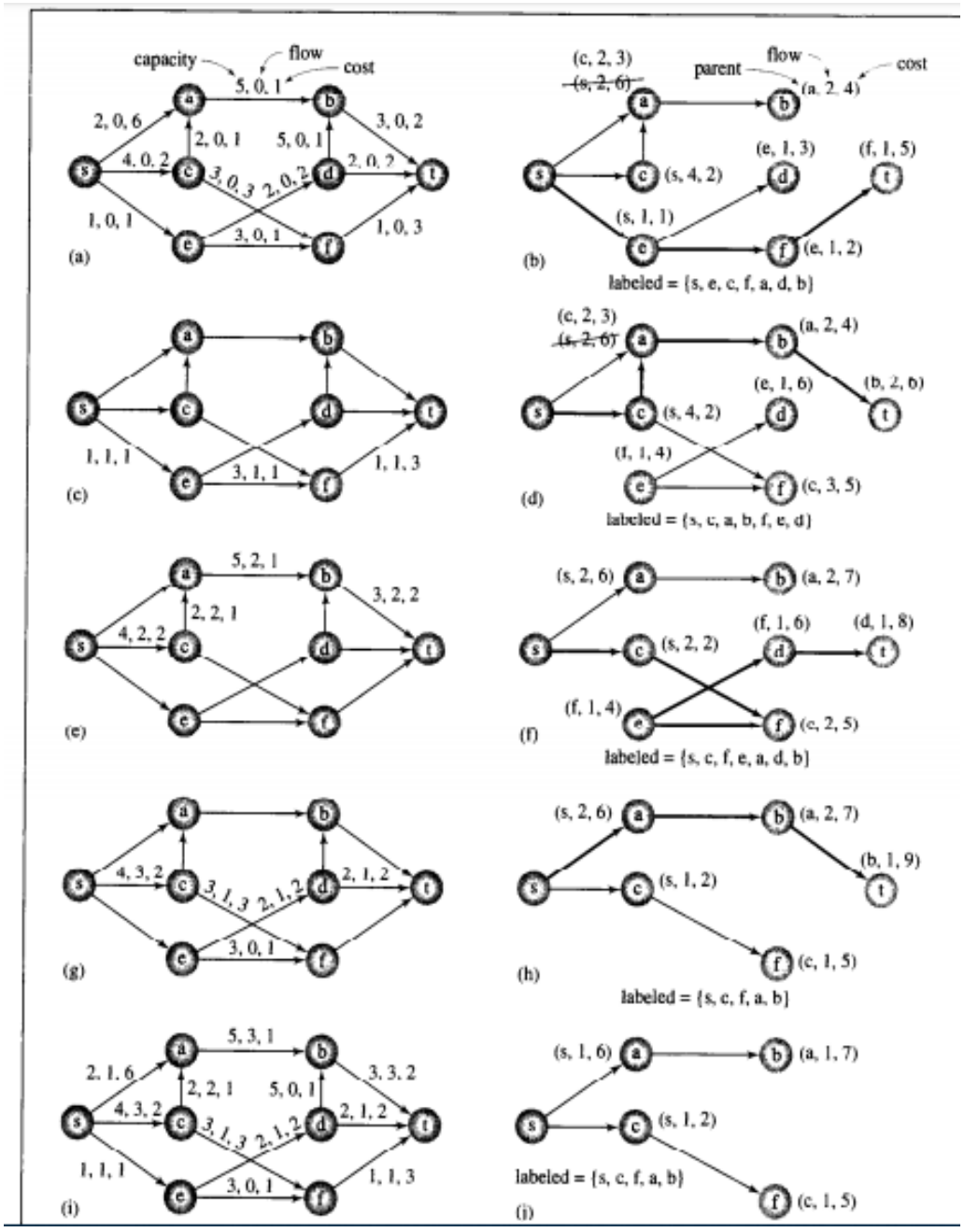
Izvor: Drozdek, 2001., str. 416

Prvo, za svaku točku *u*, zapisuje se prethodnik *v*, vrh preko kojeg je *u* dostupan iz izvornog *s*. Drugo, bilježi se maksimalni iznos protoka koji se može gurnuti kroz put od *s* do *u* i na kraju do *t*. Treće, pohranjuju se troškovi prolaska svih rubova od izvora do *u*. Za prednji rub *(vu),* trošak *(u)* je zbroj već akumuliranih troškova u *v* plus trošak guranja jedne jedinice protoka kroz rub *(vu).* Za unatrag *(vu),* jedinični trošak prolaska kroz ovaj rub se oduzima od troška *(v)* i pohranjuje u trošak *(u).* Također se ažuriraju tokovi rubova uključenih u proširene putanje. Taj zadatak izvodi *augmentPath ().*

Sljedeća slika ilustrira primjer. U prvoj iteraciji petlje *while*, označena su sa *{s},* a tri vrha susjedna sa *s* označena su na sljedeći način: oznaka (engl. label) *(a) = (s, 2,6),*

oznaka *(c) = (s, 4,2),* i oznaka *(e) = (s, 1,1).* Tada se bira vrh s najmanjim troškom. Naime, vrh *e* tada je označen *s = {s, e}.* Dva vrha zahtijevaju nove oznake, oznaku *(d) = (e, 1,3)* i oznaku *(f) = (e, 1,2).* U trećoj iteraciji izabran je vrh *c*, budući da je njegov trošak, dva, minimalan. Vrh *a* dobiva novu oznaku, *(c,2,3),* jer je trošak pristupa od *s* do *c* manji od pristupa izravno iz *s*. Vrh *f*, koji je susjedan *c*, ne dobiva novu oznaku, jer trošak slanja jedne jedinice toka od *s* do *f* do *c, 5* premašuje trošak slanja ove jedinice kroz *e*, što je dva. U četvrtoj iteraciji, *f* je izabran, pa postaje označen sa *{s, e, c, f}* i oznakom (*t) = (f, 1,5).* Nakon izlaska sedme iteracije, odmah nakon odabira odredišta *t*, povećava se put sljedećih oznaka *s, e, f, t* (slika c). Izvršenje se nastavlja, *modifiedDijkstraAlgorithm ()* se poziva još četiri puta, a u zadnjem pozivu ne može se naći niti jedan drugi put od *s* do *t*. Valja imati na umu da su isti putevi pronađeni, iako u drugačijem redoslijedu, koji je zbog cijene tih puteva: pet je cijena prvog otkrivenog puta (slika b), šest je trošak drugog puta (slika d), osam je trošak trećeg (slika f), a devet je trošak četvrtog (slika h). Ali distribucija tokova za pojedine rubove dopušta maksimalni protok koji je malo drugačiji. Na slici ista tri ruba *(sa, sc i ca)* prosljeđuju *1,3 i 2* jedinice.

Slika 80.: Prikaz primjera pronalaženja maksimalnog protoka minimalnih troškova



Izvor: Drozdek, 2001., str. 418

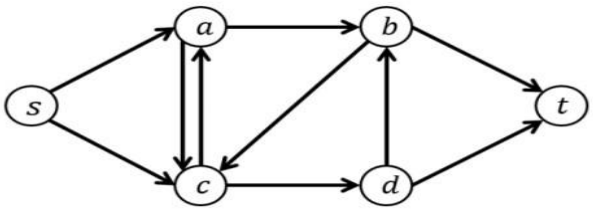
10.2 Mreže protoka

Mreža protoka je usmjereni graf gdje svaki rub ima kapacitet i protok. Obično se koriste za modeliranje problema koji uključuju prijenos predmeta između lokacija, koristeći mrežu ruta s ograničenim kapacitetom. Konkretno, primjeri uključuju modeliranje predmeta na mreži cesta, električne energije u mreži neke komponente, i sl. (Anonymous_27, 2019.).

Primjerice, neka tvrtka želi poslati pošiljke iz Los Angelesa do New Yorka koristeći pri tome kamione za prijevoz između srednjih gradova. Ako postoji samo jedan kamion za rutu koja povezuje par gradova i svaki kamion (maksimalno opterećenje), onda će graf koji opisuje predstavljati rutu kamiona između tih gradova (primjerice, autocesta). Kapacitet za određenu rutu bit će maksimalno opterećenje koje može nositi kamion. Koristeći ovaj model, tvrtka može odlučiti kako podijeliti svoje pakete između kamiona tako da paketi mogu stići na svoje odredište koristeći dostupne rute i kamione. Broj paketa koje tvrtka odluči isporučiti određenom rutom kamiona je protok za tu rutu.

Primjerice, može se zamisliti kurirska služba koja želi isporučiti što je moguće više paketa od grada *s* do grada *t.* Nažalost, nema načina da se paketi pošalju izravno od *s* do *t*, tako da kurirska služba mora isporučiti pakete pomoću srednjih gradova *a, b, c i d.* Naime, pojedini parovi gradova povezani su letovima koji omogućavaju prijevoz paketa između tih gradova. Ova prometna mreža može biti predstavljena sljedećim usmjerenim grafom (prikaz na donjoj slici), gdje čvorovi predstavljaju gradove, a usmjereni rubovi predstavljaju letove između tih gradova.

Slika 81.: Prikaz mogućih letova između gradova

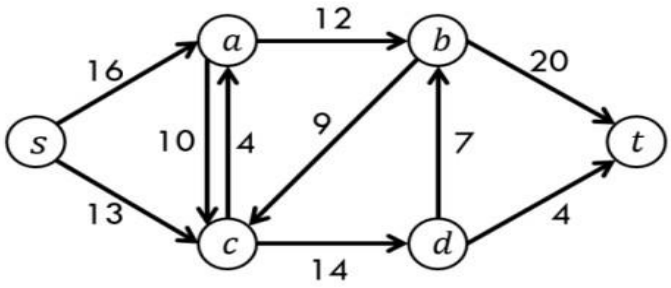


Izvor: Anonymous_27, 2019.

Preciznije, svaki zrakoplov ne može nositi neograničen broj paketa. Dakle, svaki let ima maksimalan broj paketa koje može nositi, ovisno o veličini njegova skladišta. Taj se maksimum naziva kapacitetom za taj let. Ono predstavlja broj povezan sa svakim

rubom u gornjem grafu i označava maksimalan broj paketa koji se mogu prevesti između gradova. Donji graf (prikazan na slici ispod) je gornji graf plus odgovarajući kapaciteti. Primjerice, let od *b* do *c* može nositi maksimalan broj paketa (9), tako da rub *bc* (strelica iznad) ima kapacitet devet.

Slika 82.: Prikaz letova između gradova s odgovarajućim kapacitetom

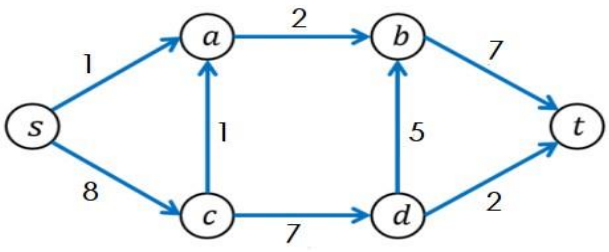


Izvor: Anonymous_27, 2019.

Kada kurirska služba ima prikaz letova koje može koristiti za prijevoz paketa, kao i maksimalni broj paketa koji se mogu premjestiti između gradova, može započeti zadatak odlučivanja koliko je potrebno paketa prevoziti na svaki let. Broj paketa koji se prevozi u svaki let poznat je pod nazivom protok za taj let. Dakle, naivno rješenje bilo bi samo dodijeliti maksimalno mogući broj paketa svakom letu. Unatoč tome, to krši ograničenje zdravog razuma da broj paketa koji se prebacuju u srednji grad mora biti jednak broju paketa koji izlaze iz tog grada. U suprotnom, paketi se ostavljaju iza sebe (to se događa u slučaju da ih je više letjelo nego što nije) ili su stvoreni (u slučaju da je više protjecanja), tj. dva scenarija koji nisu u interesu kurira ili moći. Doista, dodjeljivanje maksimalno mogućih tokova ostavlja pakete (14) ulaskom u grad *d* i samo je tada jedanaest odlaznih elemenata (oni napuštaju tok), što znači da su ti paketi nestali, a ostalo ih je samo tri.

Dakle, za svaki čvor u grafu osim *i*, ukupni protok koji napušta taj čvor mora biti jednak ukupnom protoku koji ulazi u taj čvor. Jedna moguća dodjela koja poštuje ta dva ograničenja (protok jednog ruba ne smije premašiti njegov kapacitet i ukupni protok koji ulazi u čvor mora biti jednak ukupnom protoku koji izlazi iz tog čvora) prikazan je na sljedećoj slici (Anonymous_27, 2019.).

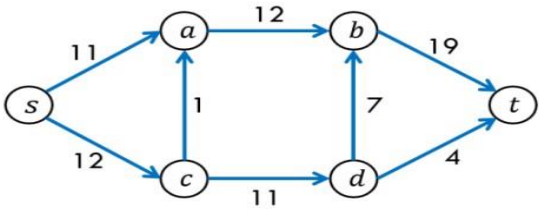
Slika 83.: Prikaz raspodjele tokova za prijevoz paketa između *s* i *t*



Izvor: Anonymous_27, 2019.

Postavlja se pitanje je li to zadatak koji maksimizira broj paketa koji se isporučuju iz grada *s* u grad *t*. Budući da je broj paketa koji se dostavljaju u grad *t* jednostavno broj paketa koji napuštaju *s* ili ulaze u *t*, (zbrajajući ih), jedan od njih daje tu dodjelu protoka koja dopušta isporuku devet paketa. Ustvari, optimalni zadatak, poznat kao maksimalni protok, je 23 paketa, koji je prikazan na donjem grafu. Ovo je jedan od zadatka protoka (nije nužno jedinstven) koji maksimizira cilj dostave kurirske službe, koji je maksimalan broj paketa koje treba poslati od *s* do *t.*

Slika 84.: Prikaz dodjele tokova koja maksimizira prijevoz paketa između *s* i *t*



Izvor: Anonymous_27, 2019.

10.2.1 Problemi s mrežnim protokom

Postoji nekoliko različitih problema koji se mogu modelirati kao protočne mreže. Nekoliko njih je uobičajeno i pojavljuju se na vrlo različitim poljima.

Vrste problema s mrežnim protokom:

1. Problem maksimalnog protoka je problem pronalaženja maksimalno dopuštenog protoka kroz jedinstven izvor (jednu mrežu protoka). Izvorno je formuliran 1954. godine od strane matematičara koji su pokušavali modelirati

protok željezničkog prometa. Dobro poznata rješenja za problem maksimalnog protoka uključuju Ford- Fulkersonov algoritam, Edmonds- Karp algoritam i Diniciev algoritam. Algoritmi maksimalnog protoka imaju velik raspon primjera. Ti primjeri uključuju raspoređivanje zrakoplovne posade, problem potražnje za cirkulacijom (gdje se roba s potražnjom ovisnom o lokaciji mora prevoziti rutama s ograničenim kapacitetom) i određivanje kada se tijekom sportske sezone eliminiraju one ekipe koje su izgubile).

2. Minimalni protok troškova. Ovdje je minimalni trošak problem pronalaženja najjeftinijeg mogućeg načina slanja određene količine protoka kroz mrežu. Ono zahtijeva proširenje protočne mreže tako da svaki rub *e=(u, v)* sada također ima pridruženi trošak *a(e)* po jedinici protoka po rubu. Također, ukupni trošak protoka je tada zbroj *(sum (e ∈E) a(e) cdot f(e)).* Minimalni protok troškova se može riješiti pomoću linearnog programiranja jer je funkcija cilja linearna (kao i ograničenja). Jedna od uobičajenih primjena protoka minimalnih troškova je određivanje najjeftinijeg načina transporta predmeta od točke *A* do točke *B*, s obzirom na neke rute s ograničenim kapacitetom i povezanim troškovima. Primjerice, rute kamiona mogu imati veći kapacitet, ali veće troškove (u smislu vremena), dok avionske rute mogu imati manji kapacitet, ali niže troškove. Stoga će tvrtka koja želi smanjiti troškove prijevoza nastojati modelirati ovaj kompromis između kapaciteta i troškova uz korištenje minimalnog protoka troškova.
3. Maksimalno dvostrano (bipartitno) podudaranje. Isti predstavlja problem određivanja maksimalnog podudaranja za bipartitni graf. U slučaju da je graf modeliran kao protočna mreža (protok iz jednog skupa čvorova u drugi), onda se za njegovo rješavanje mogu koristiti različiti algoritmi protoka. Primjerice, Ford- Fulkersonov algoritam može riješiti bipartitno podudaranje u netežinskim grafovima, kao i Hopcroft- Karp algoritam, koji to čini učinkovitije jer je dizajniran posebno za bipartitne grafove. Za slučaj ponderiranog grafa (nazvan problem zadatka), najpoznatiji algoritam je Mađarski algoritam. Ovdje je najzanimljivija maksimalna bipartitna podudarnost koja se koristi u smislu problema zadatka. Tako je najčešća formulacija problema zadatka da se uz određeni broj agenata i određeni broj zadatka (kao i trošak/ korist za svakog agenta za svaki zadatak), svakom agentu dodijeli točno jedan zadatak tako da trošak/ korist bude minimiziran/ maksimiziran. Stoga se mnogi uobičajeni problemi mogu formulirati

u smislu problema zadatka, tako da su aplikacije iznimno različite. Te aplikacije uključuju raspoređivanje, raspodjelu resursa i segmentaciju prijevoza.

4. Problem prijevoza. Ovaj problem spada u općenite vrste problema. Naime, pokazalo se da je problem prijevoza također specifičnija varijanta problema s minimalnim troškovima. Ovdje valja naglasiti da je problem prijevoza sličan problemu dodjele, ali umjesto da se svakom agentu dodijeli točno jedan zadatak, svaki agent može podijeliti svoje vrijeme između više zadataka, a svaki zadatak može poslušati više agenata. Iz prethodno navedenog razloga, neki agenti i zadaci ne mogu biti dodijeljeni. Problem prijevoza, kao što mu samo ime kaže, se ponekad opisuje u smislu rudnika te tvornica. Ako se rudnici opskrbljuju putem tvornica rudama, problem prijevoza se svodi na određivanje najisplativijeg načina prijevoza rude iz pojedinih rudnika u pojedine tvornice. Upravo se onda na taj način ruda iz jednog rudnika može slati u više tvornica, dok jedna tvornica može koristiti rudu iz mnogih rudnika. U ovom slučaju, funkcija troška će ovisiti o euklidskoj udaljenosti između svakog rudnika i tvornice, kao i troškovima koji su dostupni metodi prijevoza između njih. Stoga je očito da rudarstvo nije jedina industrija čiji se problemi u transportu mogu formulirati na taj način, pa je i sam problem vrlo općenit, kao i njegove primjene.

10.3 Ford- Fulkersenova metoda

Ford- Fulkersenov algoritam (pohlepni algoritam) se koristi za problem najvećeg protoka (protok može značiti bilo što, ali obično označava podatke putem računalne mreže). Prikazani graf predstavlja protočnu mrežu gdje svaki rub ima kapacitet. Također, dana su dva izvora *s* i *t* u grafu.

Ovu metodu su otkrili 1956. godine Ford i Fulkerson. Taj se algoritam ponekad naziva metodom jer dijelovi njegovog protokola nisu u potpunosti specificirani i mogu varirati od implementacije do implementacije. Algoritam se obično odnosi na specifičan protokol za rješavanje problema, dok je metoda općenitiji pristup problemu. (Chumbley i sur., 2019.).

Ideja algoritma je sljedeća:

1. Sve dok postoji put od izvora (početnog čvora) do odredišta (završnog čvora), s raspoloživim kapacitetom na svim rubovima putanje, šalje se protok uz jednu od staza.
2. Onda se nalazi drugi put, itd. Put s dostupnim kapacitetom se naziva putanja povećanja.

Ford- Fulkersonov algoritam kao ulaz ima graf *G* (*G* predstavlja izvor), *s* i vrh *t* (engl. sink vertex). Tako je graf bilo koji prikaz težinskog grafa gdje su vrhovi povezani rubovima specificiranih težina. Moraju postojati dva vrha za razumijevanje početka i kraja protočne mreže (prethodno navedena).

Pronalazak najvećeg mogućeg protoka od *s* do *t* sa sljedećim ograničenjima: (geeksforgeeks_12, n_d).

1. Protok na rubu ne prelazi zadani kapacitet ruba.
2. Dolazni protok je jednak izlaznom protoku za svaki vrh osim *s* i *t.*

Ford- Fulkersonov algoritam (ideja Ford- Fulkersonovog algoritma):

1. Valja početi s početnim protokom (nula).
2. Dok postoji proširen put od izvora do odredišta (treba dodati taj tok u tok).
3. Treba vratiti tok.

Dodavanjem putanja i povećanjem protoka u već uspostavljeni tok u grafu, maksimalni protok će se postići kada se više ne može pronaći putanja povećanja protoka u grafu. Međutim, ne postoji izvjesnost da će se takva situacija ikada postići, tako da je najbolje zajamčiti da će odgovor biti točan ako se algoritam prekine. U slučaju da algoritam radi konstantno (zauvijek), protok se možda neće čak ni približiti maksimalnom protoku. Međutim, ta se situacija događa samo s iracionalnim vrijednostima protoka. Kada su kapaciteti cijeli brojevi, vremenska složenost ovog algoritma je *O (E*f),* gdje je *E* broj rubova u grafu, a *f* je maksimalni protok u grafu. To je zato što se svaka putanja povećavanja može pronaći u *O(E),* gdje vrijeme povećava protok za cijeli broj od najmanje jedan s gornjom granicom *f.*

Kako implementirati gore navedeni algoritam? Najprije treba definirati pojam rezidualnog grafa koji je potreban za razumijevanje provedbe.

Rezidualni graf protočne mreže je graf koji pokazuje dodatni mogući protok. Ako postoji put od izvora do ponora u rezidualnom grafu, tada je moguće dodati tok. Svaki rub rezidualnog grafa ima vrijednost nazvanu zaostali kapacitet koji je jednak izvornom kapacitetu ruba. Preostali kapacitet je u osnovi trenutni kapacitet ruba. Preostali kapacitet je nula ako ne postoji rub između dvaju vrhova rezidualnog grafa. Može se inicijalizirati rezidualni graf kao izvorni graf jer nema početnog toka i početno preostali kapacitet jednak je izvornom kapacitetu. Da bi se pronašao put proširenja, može se napraviti pretraživanje u širinu ili u dubinu rezidualnog grafa. Koristeći pretraživanje u širinu, može se saznati postoji li put od izvora do ponora. Konkretno, taj algoritam (pretraživanje u širinu) također gradi matricu roditelja. Koristeći matricu roditelja, prelazi se pronađena staza i pronalazi se mogući protok kroz tu stazu pronalazeći minimalni preostali kapacitet duž putanje. Kasnije se treba dodati pronađeni tijek toka ukupnom protoku. (geeksforgeeks_12, n_d).

Važno je ažurirati preostale kapacitete u rezidualnom grafu. Također, valja oduzeti tok staze od svih rubova duž staze te dodati tok staze uzduž obrnutih rubova. Mora se dodati tok staze uzduž obrnutih rubova jer će se kasnije morati poslati protok u obrnutom smjeru.

Implementacija Ford- Fulkersonovog algoritma se još naziva Edmonds- Karp algoritam (zajamčen je završetak i vrijeme trajanja je neovisno o maksimalnoj vrijednosti protoka). Ideja Edmonds- Karpa je koristiti pretraživanje u širinu u Ford- Fulkersonovoj implementaciji jer pretraživanje u širinu uvijek odabire put s minimalnim brojem rubova. Kada se koristi pretraživanje u širinu, najgori slučaj vremenske složenosti može se smanjiti na $O(V*E^2)$.

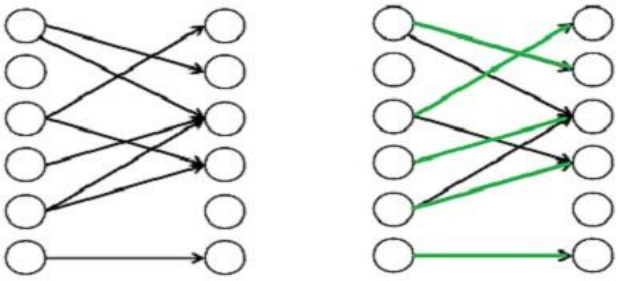
10.4 Maksimalno dvostrano podudaranje

Usklađivanje u dvostranom (bipartitnom) grafu je skup rubova odabranih na takav način da ne postoje dva ruba koja dijele krajnju točku. Naime, maksimalno podudaranje je podudaranje maksimalne veličine (sadrži maksimalni broj rubova). Također, u maksimalnom podudaranju, ako mu se doda bilo koji rub, to više nije podudaranje. Stoga, valja naglasiti da može postojati više od jednog maksimalnog podudaranja za dani bipartitni graf.

Postoje mnogi problemi stvarnog svijeta koji se mogu formirati kao dvostrano podudaranje. Primjerice, može se razmotriti sljedeći primjer: postoji *M* kandidata za posao te *N* radnih mjesta. Ovdje svaki kandidat ima podskup poslova za koje je zainteresiran. Svaki posao tako može prihvatiti samo jednog kandidata, a kandidat za posao može biti imenovan samo za jedan posao. Valja pronaći dodjelu poslova kandidatima tako da što više kandidata dobije posao.

Prva slika (slika 85.) prikazuje podnositelje te poslove, odnosno koji je podnositelj povezan s kojim poslom. Druga slika (slika 86.) prikazuje da najviše pet osoba može dobiti posao (na slici označeno zelenim strelicama).

Slike 85. i 86.: Prikaz podnositelja poslova te prikaz da najviše pet osoba može dobiti posao



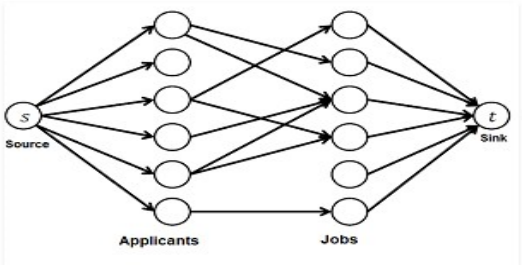
Izvor: geeksforgeeks_13, n_d

Problem maksimalnog dvostranog podudaranja te maksimalnog protoka se može riješiti pretvorbom u protočnu mrežu.

Slijede koraci rješavanja prethodno navedenog problema:

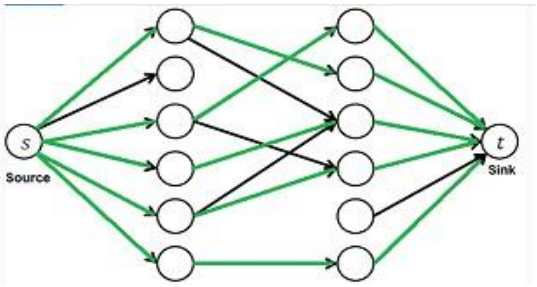
1. Izgrađivanje mreže protoka. U ovom koraku mora postojati izvorište te odredište u protočnoj mreži. Iz tog razloga se dodaje izvor te rubovi od izvora pa do svih podnositelja zahtjeva. Isto tako, dodaju se rubovi iz svih poslova u odredište. Kapacitet svakog ruba je ovdje označen s jednom jedinicom.

Slika 87.: Prikaz izgrađivanja mreže protoka



Izvor: geeksforgeeks_13, n_d

2. Pronalazak maksimalnog protoka. Ovdje se koristi Ford- Fulkersonov algoritam kako bi se pronašao maksimalni protok u protočnoj mreži koja je izgrađena u prvom koraku. Stoga je maksimalni protok zapravo problem maksimalnog dvostranog podudaranja kojeg tražimo.
   Slika 88.: Prikaz maksimalnog protoka od izvorišta do odredišta koji iznosi pet jedinica. Dakle, najviše pet ljudi može dobiti posao.



Izvor: geeksforgeeks_13, n_d

Gore navedeni pristup se ispunjava tako da se najprije definiraju ulazni i izlazni oblici. Ulaz je ovdje u obliku *Edmonds* matrice koja je prikazana dvodimenzionalnim nizom *'bpGraph [M] [N]'* s *M* redova (za *M* kandidata za posao) i *N* stupaca (za *N* poslova). Vrijednost *bpGraph [i] [j]* je jedan ako je sam podnositelj zahtjeva zainteresiran za *j* poslova, u suprotnom je nula. Izlaz predstavlja broj maksimalnog broja ljudi koji mogu dobiti posao.

Jednostavan način da se prethodno navedeno provede je stvoriti matricu koja predstavlja matrično susjedstvo usmjerenog grafa, tj. *M+ N+ 2* vrhova. Primjerice, ista se može nazvati *fordFulkerson()* za matricu. Ta implementacija zahtjeva *O ((M+ N) * (M+N))* dodatnog prostora.

Dodatni prostor se može smanjiti, a kod može biti pojednostavljen korištenjem činjenice da je graf bipartitni, a kapacitet svakog ruba da je nula ili jedan. Ideja je ovdje koristiti pretraživanje u dubinu za pronalaženje posla za podnositelja zahtjeva (slično proširenom putu kod Ford- Fulkersonovog algoritma). Isti se naziva *bpm()* za svakog podnositelja zahtjeva. Naime, ona je zapravo funkcija temeljena na pretraživanju u dubinu koja pokušava sve mogućnosti kako bi dodijelila posao podnositelju zahtjeva. (geeksforgeeks_13, n_d).

U *bpm(),* jedan po jedan se isprobavaju svi poslovi za koje je zainteresiran podnositelj zahtjeva dok se ne nađe posao, ili dok se ne ispitaju svi poslovi. Ako posao nije dodijeljen nikome, jednostavno ga se dodijeli podnositelju zahtjeva te se vrati istina. U slučaju da je zadatak dodijeljen nekom drugom, primjerice *x-u*, onda se rekurzivno provjeri može li *x-u* biti dodijeljen neki drugi posao. Kako bi bili sigurni da *x* više neće dobiti isti posao, isti se označava s *v*, kao što se može vidjeti prije nego što se napravi rekurzivni posao za *x*. Tako da ako *x* može dobiti drugi posao, mijenja se kandidat za posao *v* te se vraća istina. Stoga se koristi matrica *maxR [0,...,N- 1]* koja pohranjuje kandidate dodijeljene različitim poslovima.

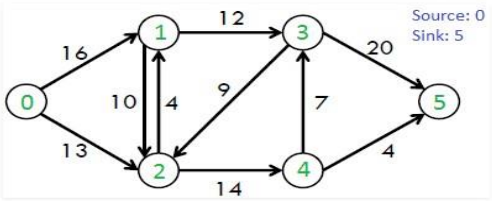
Ako *bpm()* vrati istinu, to znači da postoji put proširenja u protočnoj mreži te se jedna jedinica toka dodaje rezultatu u *maxBPM().*

10.5 Push- relabel algoritmi

Ovdje je prikazan graf koji predstavlja protočnu mrežu gdje svaki rub sadrži kapacitet. Također, ovdje su dana dva izvora *s* i odredište *t* u grafu. Stoga je potrebno pronaći najveći mogući protok od *s* do *t* sa sljedećim ograničenjima:

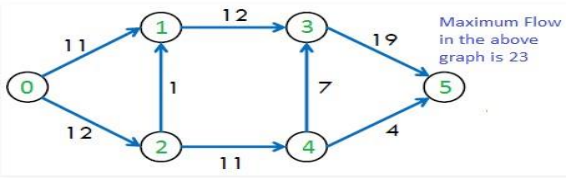
1. Protok na rubu ne prelazi zadani kapacitet ruba.
2. Dolazni protok je jednak izlaznom toku za svaki vrh osim *s* i *t.*

Slika 89.: Prikaz grafa od *s* do *t* s određenim ograničenjima (prethodno navedena)



Izvor: geeksforgeeks_14, n_d

Slika 90.: Prikaz njegovog maksimalnog protoka koji iznosi 23



Izvor: geeksforgeeks_14, n_d

Push- relabel algoritam je učinkovitiji od Ford- Fulkersonovog algoritma. Ovdje se zapravo govori o Goldbergovom "generičkom" algoritmu maksimalnog protoka koji ima vremensku složenost jednaku $O(V^E)$. Ta je složenost vremena bolja od $O(E^V)$, što predstavlja vremensku složenost Edmond- Karp algoritma (implementacija Ford- Fulkersonovog algoritma je temeljena na pretraživanju u širinu). Naravno, postoji algoritam temeljen na push- relabel pristupu koji radi u $O(V^3)$ koji se čak smatra boljim.

Dakako, postoje sličnosti s Ford- Fulkersonovim algoritmom. Kao i Ford- Fulkerson, Push- relabel radi i na rezidualnom grafu (rezidualni graf protočne mreže je graf koji pokazuje dodatni mogući protok). Ako postoji put od izvora do odredišta u rezidualnom grafu, tada je moguće dodati protok.

Razlike između Ford- Fulkersonovog i Push- relabel algoritma: (Cormen i sur., 2001.).

1. Push- relabel algoritam radi više lokalizirano. Umjesto ispitivanja cjelokupne preostale mreže da bi se pronašla proširiva putanja, push- relabel algoritmi djeluju na jednu točku odjednom.
2. U Ford- Fulkersonovom, neto razlika između ukupnog odljeva i priljeva za svaki vrh (osim izvora i odredišta) se odražava s nulom. Push- relabel algoritam

omogućuje da dotok prijeđe izlazni tok prije postizanja konačnog toka. U konačnom toku, neto razlika je nula za sve osim izvora i odredišta.

3. Vrijeme složenosti je učinkovitije.

Intuicija za Push- relabel algoritam (fluidni problem) je da se razmatraju rubovi i čvorovi. Smatra se da je izvor na najvišoj razini koji se šalje prema svim susjednim čvorovima. Jednom kada, primjerice čvor ima višak "vode", on gura "vodu" u čvor manje visine. Ako voda bude lokalno zarobljena na samom vrhu, vrh je *Relabeled* što znači da je njegova visina povećana.

Slijedi nekoliko korisnih činjenica koje treba razmotriti: (geeksforgeeks_14, n_d).

1. Svaki vrh ima pridruženu visinsku varijablu i višak protoka. Visina se koristi za određivanje može li se vršak gurati u susjedni vršak ili ne (vrh može gurati tok samo na manji vrh visine). Višak protoka predstavlja razliku ukupnog protoka koji dolazi u vrh minus ukupni protok koji izlazi iz vrha (višak protoka *u* = ukupni priljev *u* – ukupni odljev iz *u*).
2. Kao Ford- Fulkerson, svaki rub je povezan s protokom (označava trenutni protok) i kapacitetom.

Slijede apstraktni koraci kompletnog, Push- relabel algoritma:

1. Inicijaliziranje predpokretanja (inicijaliziranje tokova i visine).
2. Dok je moguće izvesti *Push()* ili *Relabel()* na vrhu, ili dok postoji vrh koji ima višak protoka *(Push() ili Relabel())*, tada u tom trenutku svi vrhovi imaju *Excess Flow* kao nulu (osim izvora i odredišta).
3. Povratak (engl. return) protoka.

U Push- relabel algoritmu postoje tri glavne operacije:

1. *PreFlow().* Funkcija koja inicijalizira visinu te protok svakog vrha (stavlja ih na nulu). Ista inicijalizira i visinu izvorne točke koja je jednaka ukupnom broju vrhova u grafu te protok svakog ruba stavlja na nulu. Također, za sve vertikale koje su susjedni izvoru *s* i protoku *i*, prekomjerni protok je početno jednak kapacitetu.
2. *Push().* Ova funkcija se koristi za stvaranje toka iz čvora koji ima višak protoka. U slučaju da vrh ima višak protoka i nalazi se susjedno od njega s malom visinom (u rezidualnom grafu), onda se protok gura od vrha do susjeda s nižom

visinom. Količina potisnutog protoka kroz rub je jednaka minimalnom višku protoka te kapacitetu ruba.

3. *Relabel().* Ovo je vrsta operacije koja se koristi kada kod vrha postoji višak protoka te nijedan njegov susjed nije na nižoj visini. Ovdje se u osnovi mora povećati visina ruba tako da se može izvoditi funkcija *push().* Da bi se povećala visina, odabire se minimalna visina koja se nalazi u susjedstvu (u rezidualnom grafu, tj. susjed kojemu se može dodati protok) i doda se u isti.

Ovdje valja naglasiti da se gore prethodno navedene operacije izvode na rezidualnom grafu (kao što je to Ford- Fulkerson). Stoga, valja dobro znati rezidualni graf. Kad god se gura ili dodaje vrh *u* u *v*, slijede ažuriranja u rezidualnom grafu:

1. Oduzima se protok od kapaciteta ruba iz *u*. U slučaju da kapacitet ruba postane nula, tada taj rub neće više postojati u rezidualnom grafu.
2. Dodaje se tok kapacitetu ruba od *v* do *u.*

Primjerice, valja razmotriti dva vrha *u* i *v*. U izvornom grafu *3/ 10 (u-->v), 3* predstavlja tok od *u* do *v*, a *10* kapacitet ruba od *u* do *v*. U rezidualnom grafu odgovaraju dva ruba (*7, u-->v* te *3 u<-- v*). (geeksforgeeks_14, n_d).

Koraci Push- relabel algoritma na primjeru:

1. Prikaz dijagrama toka.
2. Nakon funkcije *PreFlow(),* u rezidualnom grafu postoji rub od *A* do *S* s kapacitetom tri i bez ruba od *S* do *A.*
3. Označeni vrh je ponovno označen (visina postaje jedan) iz razloga što ima višak protoka i nema susjeda s manjom visinom. Nova visina je jednaka minimalnoj visini susjednog plus jedan. U rezidualnom grafu postoje dva susjedna vrha *A (S i B).* Visina *S* je četiri, a visina *B* je nula. Minimum od te dvije visine je nula. Stoga se uzme minimum i doda se jedan.
4. Označeni vrh ima višak protoka te postoji susjed s nižom visinom, tako da se ovdje poziva funkcija *push().* Višak toka *A* je dva, a kapacitet ruba *(A, B)* je jedan. Prema tome, količina potisnutog protoka je jedan (minimalno dvije vrijednosti).
5. Označeni vrh je ponovno označen (visina postaje jedan) jer ima višak protoka te nema susjeda s manjom visinom.

6. Označeni vrh ima višak protoka i postoji susjed s nižom visinom, tako da se protok gura iz *B* u *T.*
7. Isti je ponovno označen (visina postaje pet) jer ima višak protoka i nema susjeda s manjom visinom.
8. Označeni vrh ima višak protoka te postoji susjed s nižom visinom, tako da se javlja funkcija *push().*
9. Označeni vrh je ponovno označen (visina se povećava za jedan) jer ima višak protoka i nema susjeda s manjom visinom.

10.6 Algoritam relabel- to- front

Algoritam Relabel- to- front se koristi za pronalaženje maksimalnog protoka u mreži. Isti je učinkovitiji od generičkog Push- relabel algoritma. U Push- relabel algoritmu se mogu primijeniti osnovne operacije *push* i *relabel* bilo kojim redoslijedom. Za razliku od Push- relabel algoritma, Relabel- to- front pažljivo bira red te učinkovito upravlja mrežnim strukturama podataka.

Prvo se moraju razumjeti osnovne operacije *(push() i relabel()).* Naime, svaki vrh u mreži ima dvije varijable koje su povezane s njom, a to su visinska varijabla *(h)* te višak protoka *(e):* (geeksforgeeks_15, n_d).

1. *Push().* Ako vrh ima višak protoka i postoji susjedni čvor s nižom visinom u rezidualnom grafu, onda se gura protok od vrha do čvora niže visine.
2. *Relabel().* Ako vrh ima višak toka i nema susjednog čvora s nižom visinom, tada se koristi *relabel()* funkcija koja služi za povećanje visine vrha tako da se može izvesti *push()* operacija.

Algoritam Relabel-to- front održava popis točaka u mreži. Isti započinje od početka popisa te opetovano bira preljevnu točku *u* i na njoj izvodi operaciju pražnjenja.

Prethodno spomenuta operacija pražnjenja provodi operaciju guranja i ponovnog označavanja sve dok vrh *u* nema pozitivnog viška protoka *(e).* Ako je vrh ponovno označen, onda se pomiče na prednji dio popisa, a algoritam se skenira ponovno.

Relabel- to- front algoritam: (geeksforgeeks_15, n_d)

1. Inicijaliziranje *preflow* i visine na iste vrijednosti kao u generičkom Push- relabel algoritmu.
2. Inicijaliziranje popisa *L* koji sadrži sve vrhove osim izvorišta i odredišta.
3. Inicijaliziranje trenutnog pokazivača za svaki vrh nesigurnosti do prvog vrha *u* liste susjeda *n*. Popis susjeda *n* stoga sadrži one vrhove za koje postoji rezidualni rub.
4. Dok algoritam dođe do kraja liste *L:*
    a) Treba se odabrati točka *u* s popisa *L* i izvršiti operacija pražnjenja.
    b) Ako je *u* preimenovan u pražnjenje, potrebno je premjestiti *u* na prednji dio popisa.
    c) Ako je *u* premješten na prednji dio popisa, vrh u sljedećoj iteraciji je onaj koji slijedi u novoj poziciji na popisu.

Pseudokod Relabel- to- front algoritma: (Anonymous_28, n_d).

```
RELABEL-TO-FRONT(G, s, t)
 1  INITIALIZE-PREFLOW(G, s)
 2  L ← V[G] - {s, t}, in any order
 3  for each vertex u ∈ V[G] - {s, t}
 4       do current[u] ← head[N[u]]
 5  u ← head[L]
 6  while u ≠ NIL
 7       do old-height ← h[u]
 8          DISCHARGE(u)
 9          if h[u] > old-height
10             then move u to the front of list L
11          u ← next[u]
```

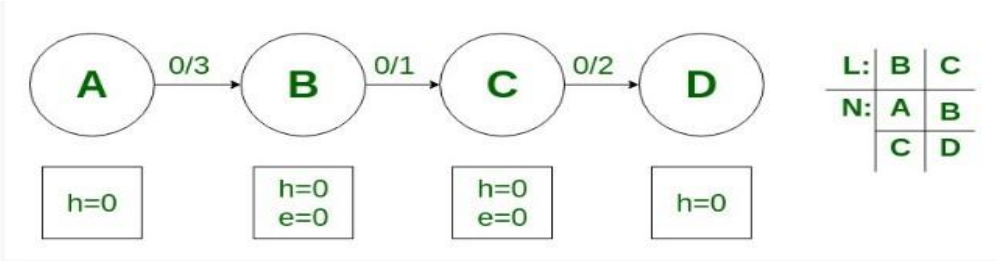
Algoritam Relabel-to-front radi na sljedeći način. *Linija 1* inicira *preflow* i visinu na iste vrijednosti kao u generičkom Push-relabel algoritmu. *Red 2* inicira popis *L* tako da

sadrži sve potencijalno prepune vrhove bilo kojim redoslijedom. *Linije 3 - 4* inicijalizaciju trenutni pokazivač za svaki vrh *ü* do prvog s vrhom u *Ü* „s liste susjeda. U *while* petlji *linije 6 - 11* pokreću se preko liste *L*, te se ispuštaju vrhovi. *Red 5* čini početak s prvim vrhom na popisu. Svaki put kroz petlju, *A* vrh *u* ispušta se u skladu s *8.* Ako je *u relabeled* od pražnjenja, *red 10* premješta se na prednjem dijelu liste *L.* Ova odluka donosi uštedu *u* zbog visine u promjenjivost zbog stare visine prije operacije pražnjenja *(linija 7)* i uspoređuje tu spremljenu visine do *ü 's* visinom poslije *(redak 9). Red 11* čini sljedeću iteraciju tog vremena te petlja koristiti sljedeće popise polukružno u popisu *L.* Ako je *u* premješten na prednji dio popisa, vrh koji se koristi u sljedećoj iteraciji je onaj koji slijedi u novoj poziciji na popisu.

Slijedi primjer Relabel- to- front algoritma:

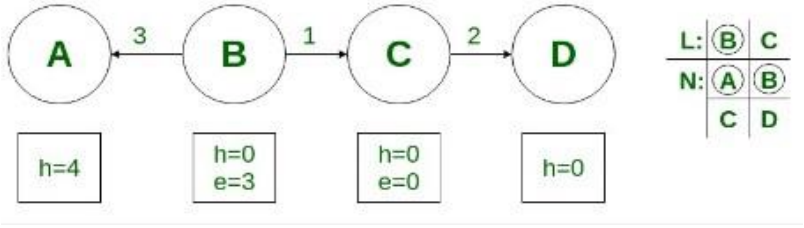
Na početku valja razmotriti mrežu protoka. Ovdje je prikazan početni popis *L=(B,C) (*na desnoj strani) gdje je u početku *u=B*.

Slika 91.: Prikaz početnog popisa *L=(B,C) (*na desnoj strani) gdje je u početku *u=B*.



Izvor: geeksforgeeks_15, n_d

Slika 92.: Prikaz inicijalizacije operacije *preflow()*.

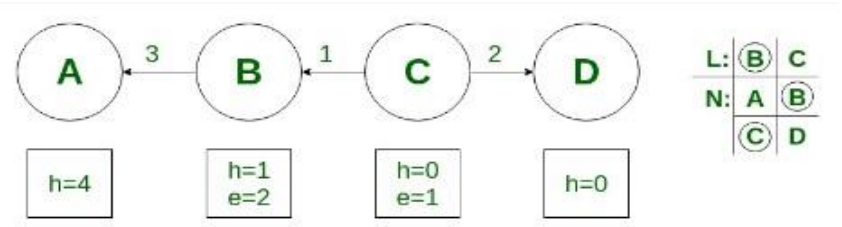


Izvor: geeksforgeeks_15, n_d

Tu se uz svaku točku u popisu *L* nalazi njezin popis susjeda *n* s trenutno zaokruženim susjedom.

Čvor *B* prolazi kroz postupak pražnjenja jer ima višak protoka *(3),* tj. *e=3*. Čvor *B* nema čvor s nižom visinom, tako da radi *relabel()* operaciju *(h=1)* i gura tok 1 u vrh *C*.

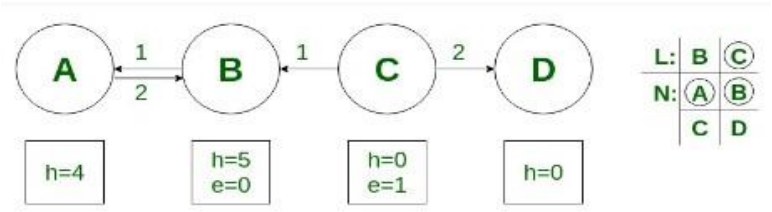
Slika 93.: Prikaz *relabel()* operacije *(h=1)* koja gura tok 1 u vrh *C*.



Izvor: geeksforgeeks_15, n_d

Čvor *B* i dalje ima višak strujanja *(2)*, tj. *e=2*. Iz tog razloga se izvodi *relabel()* operacija *(h=5)* i gura se tijek *2* u vrh *A*. Pošto je vrh *B* ponovno označen, isti ostaje na prednjoj strani popisa. Sada je točka *C* podvrgnuta operaciji pražnjenja jer ima višak protoka *(1),* tj. *e=1*.

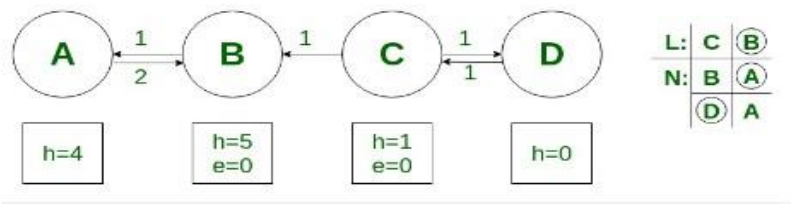
Slika 94.: Prikaz točke *C* koja je podvrgnuta operaciji pražnjenja jer ima višak protoka *(1),* tj. *e=1*



Izvor: geeksforgeeks_15, n_d

Čvor *C* izvodi *relabel()* operaciju *(h=1)* i gura tijek *1* do čvora *D*. Kada je *C* izvršio *relabel()* operaciju, isti se premješta na prednji dio popisa.

Slika 95.: Prikaz guranja tijeka *1* do čvora *D* te izvršavanje *relabel()* operacije



Izvor: geeksforgeeks_15, n_d

Točka *B* sada slijedi vrhu *C* u *L*, ali *B* nema višak protoka. Ovdje je onda Relabel- to-front algoritam došao do kraja popisa *(L).* Također, nema preljevnih vrhova, tako da je pre- tok maksimalni protok (tu je maksimalni protok 1).

Kod popisa susjeda rubovi u relabel-to-front algoritmu organizirani su u "susjedne popise". S obzirom na mrežu toka $G = (V , E)$ je popis susjeda $N [u]$ za vrh koji je $\in V$. To je jednostruko vezan popis susjeda glede nesigurnosti u *G*. Tako se vrh *v* pojavljuje u popisu $N [u]$ ako je $(u , v) \in E$ ili $(v , u) \in E$. Popis susjeda $N [u]$ sadrži točno one vrhove *v* za koje može postojati rezidualni rub *(u, v).* Prvi vrh u $N [u]$ pokazuje na glavu $[N [u]].$ Sljedeći je vrh *v* u listi susjeda. Ukazano je to pomoću sljedećeg-susjeda *[v];* ovaj pokazivač je *NIL* ako je *v* posljednji vrh u popisu susjeda.

Relabel-to-front algoritam kruži kroz svaki popis susjeda u proizvoljnom redoslijedu koji je fiksiran tijekom izvršenja algoritma. Za svaki vrh *u*, struja polja *[u]* ukazuje na vrh koji se trenutno razmatra u $N [u].$ U početku je trenutna *[u]* postavljena na početak $[N [u]].$

Pražnjenje preljevnog vrha *u* je ispražnjen guranjem cijelog njegovog viška protoka kroz dopuštene rubove u susjedne vrhove, ponovno označavanje *u* kao nužnog kako bi uzrokovalo da rubovi koji napuštaju *u* postanu dopušteni.

Slijedi prikaz pseudokoda pražnjenja preljevnog vrha: (Anonymous_28, n_d).

```
DISCHARGE(u)
1  while e[u] > 0
2     do v ← current[u]
3        if v = NIL
4           then RELABEL(u)
5                current[u] ← head[N[u]]
6        elseif cf(u, v) > 0 and h[u] = h[v] + 1
7          then PUSH(u, v)
8        else current[u] ← next-neighbor[v]
```

Vremenska složenost je $O(V^3)$ na mreži $G= (V, E).$ Iz tog razloga se smatra učinkovitijim od generičkog Push- relabel algoritma koji radi u vremenu $O(V^2 E).$

11. Podudaranje

Odgovarajući graf je podgraf grafa gdje nema rubova koji su jedan uz drugi. Jednostavnije, ne smije postojati zajednički vrh između bilo koja dva ruba. (Anonymous_29, n_d).

Drugim riječima, podudaranje u grafu je skup vrhova koji nemaju skup uobičajenih vrhova. (Moore i sur., 2019.).

Neka je *G= (V, E)* graf. Isti se naziva podudarnim *M(G),* ako je svaki vrh *G* povezan s najviše jednim rubom u *M*, tj. *deg(V) <= 1V ∈ G.* To predstavlja vrhove koji bi trebali imati stupanj 1 ili 0, gdje bi rubovi trebali biti incidentni od grafa *G.*

S obzirom na neusmjereni graf, podudaranje je skup vrhova, tako da ne postoje dva ruba koja dijele isti vrh. Smatra se da je vrh povezan ako ima s njim slučaj, inače je slobodan.

Slika 96.: Prikaz mogućih podudaranja *K4*, gdje crveni rubovi označavaju podudaranje

Izvor: geeksforgeeks_16, n_d

Podudaranje u grafu ne treba miješati s izomorfizmom grafa. Izomorfizam grafa provjerava jesu li dva grafa ista, dok podudaranje kod grafa predstavlja poseban podgraf grafa.

Naravno, podudaranje u grafu ima primjene u protočnim mrežama, raspoređivanju i planiranju, modeliranju veza u kemiji, bojanju grafova, problemu stabilnog podudaranja, neuronskim mrežama u umjetnoj inteligenciji, i sl.

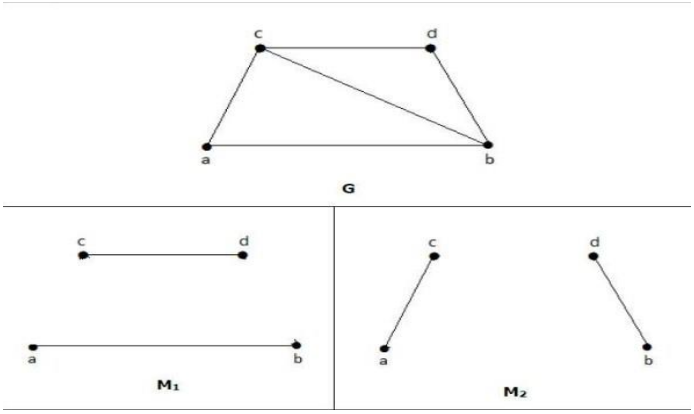
Isto tako postoje i mnogi algoritmi podudaranja grafova upravo kako bi se optimizirali parametri koji su potrebni da bi se riješio određeni problem.

U podudaranju: (Anonymous_29, n_d).

1. Ako je *deg (V)= 1*, tada se kaže da je *(V)* uparen.
2. Ako je *deg (V)= 0*, tada se *(V)* ne podudara.

Maksimalno podudaranje se definira kao maksimalno podudaranje s maksimalnim brojem rubova. Ovdje se broj rubova u maksimalnom podudaranju naziva njegovim odgovarajućim brojem.

Slika 97.: Prikaz primjera maksimalnog podudaranja

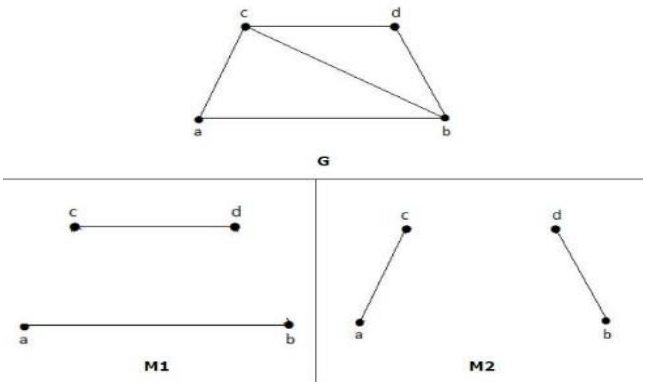


Izvor: Anonymous_29, n_d

Za dane grafove u prethodnom primjeru, M_1 i M_2 predstavljaju maksimalno usklađivanje *G* koje odgovaraju broju dva. Iz tog razloga se može pomoću grafa *G*, formirati podgraf s najviše dva ruba (odgovarajući broj je stoga dva).

Osim što postoji maksimalno podudaranje, postoji i savršeno podudaranje. Ono predstavlja odgovarajuće *(M)* grafa *(G)* ako je svaki vrh grafa *g (G)* incidentan na točno jednom rubu podudaranja *(M),* odnosno *deg (V)= 1*. Konkretno, to znači da svaki vrh u grafu treba imati stupanj jedan.

Slika 98.: Prikaz primjera savršenog podudaranja *(G),* gdje su M_1 i M_2 primjeri savršenog podudaranja

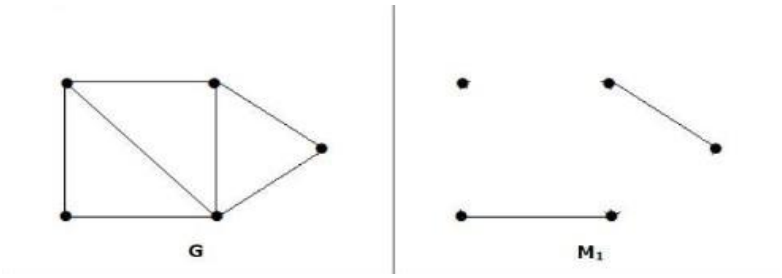


Izvor: Anonymous_29, n_d

Valja naglasiti da je svako savršeno podudaranje grafa maksimalno podudaranje grafa, jer nema šanse za dodavanje još jednog ruba u savršeno podudarnom grafu.

Naime, maksimalno podudaranje grafa ne mora biti savršeno. Ako graf *G* ima savršeno podudaranje, tada je broj vrhova |*V (G)* | ravnomjeran. U slučaju da je neparan, onda se posljednji vrh spaja s drugim vrhom, i na kraju ostaje samo jedan vrh koji se ne može upariti s bilo kojim drugim vrhom za koji stupanj iznosi nula. Stoga to jasno krši savršen princip podudaranja.

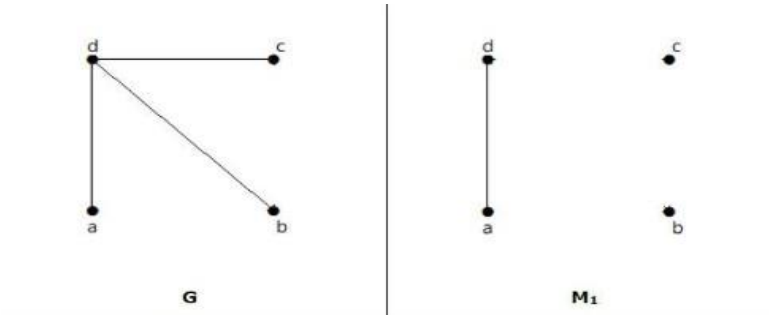
Slika 99.: Prikaz primjera savršenog podudaranja



Izvor: Anonymous_29, n_d

Prethodno definirana izjava ne mora biti istinita. Tako ako *G* ima čak i broj vrhova, onda M_1 ne mora biti savršen.

Slika 100.: Prikaz *G* koji sadrži broj vrhova gdje *M1* ne mora biti savršen



Izvor: Anonymous_29, n_d

Prethodna slika prikazuje podudarnost, koje nije savršeno, iako ima par točaka.

11.1 Problem stabilnog podudaranja

Kod podudaranja postoji graf *G=(V, E).* Konkretno, ovdje se želi postići odgovarajući *M* (maksimizirati neki cilj). Ovdje pojam podudaranja podrazumijeva skup rubova tako da je svaki vrh uključen najviše jednom.

Online dvostrano (bipartitno) podudaranje traži maksimum te je dokazano da je *1- 1/e* složenost u najgorem slučaju.

Postoje sljedeći problemi: (Dickerson, n_d, str. 3)

1. Stabilan brak (bipartitni, jedan vrh na jedan vrh).
2. Problem bolnice/ stanovnika (bipartitni, jedan vrh na više točaka).
3. Problem stabilnih cimera (nebipartitni, jedan vrh na jedan vrh).

Kod problema stabilnog braka se ispunjava bipartitni graf s jednakim stranama (*n* muškaraca i *n* žena). Također, svaki muškarac ima strogu, potpunu prednost u odnosu na žene i obrnuto. Ono što se ovdje želi je stabilno podudaranje.

Stabilno podudaranje podrazumijeva niti jednog neusporedivog muškarca i ženu, odnosno da ne vole jedni druge (svoje sadašnje supružnike).

Slika 101.: Prikaz primjera profila sklonosti

Albert	Diane	Emily	Fergie
Bradley	Emily	Diane	Fergie
Charles	Diane	Emily	Fergie

Izvor: Dickerson, n_d, str. 5

Slika 102.: Prikaz primjera profila sklonosti

Diane	Bradley	Albert	Charles
Emily	Albert	Bradley	Charles
Fergie	Albert	Bradley	Charles

Izvor: Dickerson, n_d, str. 5

Slika 103.: Prikaz primjera prvog podudaranja

Albert	Diane	Emily	Fergie
Bradley	Emily	Diane	Fergie
Charles	Diane	Emily	Fergie

Izvor: Dickerson, n_d, str. 6

Slika 104.: Prikaz primjera prvog podudaranja

Diane	Bradley	Albert	Charles
Emily	Albert	Bradley	Charles
Fergie	Albert	Bradley	Charles

Izvor: Dickerson, n_d, str. 6

Postavlja se pitanje: „Je li prethodni primjer zapravo primjer stabilnog podudaranja“? Odgovor na to pitanje je ne, iz razloga jer Albert i Emily tvore par za blokiranje. Slika 105. prikazuje prethodnu tvrdnju:

Albert	Diane	Emily	Fergie
Bradley	Emily	Diane	Fergie
Charles	Diane	Emily	Fergie

Izvor: Dickerson, n_d, str. 7

Slika 106. prikazuje prethodnu tvrdnju:

Diane	Bradley	Albert	Charles
Emily	Albert	Bradley	Charles
Fergie	Albert	Bradley	Charles

Izvor: Dickerson, n_d, str. 7

Slika 107.: Prikaz primjera drugog podudaranja

Albert	Diane	Emily	Fergie
Bradley	Emily	Diane	Fergie
Charles	Diane	Emily	Fergie

Izvor: Dickerson, n_d, str. 8

Slika 108.: Prikaz primjera drugog podudaranja

Diane	Bradley	Albert	Charles
Emily	Albert	Bradley	Charles
Fergie	Albert	Bradley	Charles

Izvor: Dickerson, n_d, str. 8

Prethodno prikazani primjer je primjer stabilnog podudaranja (Fergie i Charles su nesretni te bespomoćni).

Postavljaju se neka pitanja glede problema braka:

1. Da li uvijek postoji stabilno rješenje problema braka?
2. Može li se učinkovito izračunati takvo rješenje?
3. Može li se izračunati najbolje stabilno rješenje koje je ujedno i efikasno?

Gale- Shapley (1962.): (Dickerson, n_d, str. 11).

1. Svatko je bez premca (nepodudaran).
2. Dok je neki muškarac *m* bez premca:
    a) *w:*= najomiljenija žena koja još nije predložena
    b) Ako je *w* također bez premca:
    c) *W* i *m* su uključeni
    d) Inače, ako *w* preferira *m* prema trenutnom *m:*
    e) *W* i *m* su uključeni, *m* je bez premca.

f) Inače, *w* odbija *m*.

3. Vraćaju se odgovarajući parovi.

Zahtjev koji se ovdje postavlja je da Gale Shapley završava u polinomnom vremenu (najviše n^2 iteracije vanjske petlje). Dokaz toga je da svaka iteracija ima jednog čovjeka koji predlaže nekome kome se do sada nikada prije nije predlagalo. Isti sadrži *n* muškaraca te *n* žena -> *n* n* mogućih događaja (*n* (n- 1) +1* iteracija).

Sljedeći zahtjev je da Gale Shapley rezultira savršenim podudaranjem. Dokaz (suprotnosti) je da se pretpostavlja da je isti neusporediv na završetku. Ovdje postoji *n* muškaraca te *n* žena -> *w* i također nema premca. Stoga, jednom kada se žena podudara, ona nikada ne bude bez premca (ona samo mijenja partnere). Iz tog razloga nitko nije predložio *w*. Za razliku od toga, *m* je predložen svima (prema definiciji Gale Shapleya): ><.

Naredni zahtjev je da Gale Shapley rezultira stabilnim podudaranjem (tj. nema parova za blokiranje). Dokaz suprotnosti (1):

1. Valja pretpostaviti da *m* i *w* čine par za blokiranje (slučaj r. 1: *m* nikada nije predložen za *w*).
2. Gale Shapley: muškarci predlažu redoslijed preferencija.
3. *m* preferira trenutnog partnera *w' > w*.
4. -> *m* i *w* se ne blokiraju.

Sljedeći zahtjev se krije u tome da Gale Shapley rezultira stabilnim podudaranjem (nema parova za blokiranje). Dokaz suprotnosti (2):

Slučaj (2): *m* je predložen za *w:*

1. *w* je odbacio *m* u nekom trenutku.
2. Gale Shapley: žene odbijaju samo bolje partnere.
3. *w* preferira trenutnog partnera *m' > m.*
4. -> *m* i *w* se ne blokiraju

Slučaj (1) i (2) imaju prostor *(><).*

Naime, ovdje se javlja pojam čovjekove optimalnosti/ pesimalnosti:

1. Neka je *S* skup stabilnih podudaranja.
2. *m* je važeći partner *w* ako postoji neki stabilni *S* u *S* gdje su upareni.

3. Podudaranje je optimalno ako čovjek dobije svog najboljeg partnera.
4. Je li to savršeno podudaranje? Je li to stabilno podudaranje?
5. Podudaranje je pesimalno ako svaki čovjek dobije svog najgoreg partnera.

Sljedeći zahtjev je Gale Shapley s čovjekom koji isti predlaže (rezultira optimalnim podudaranjem). Slijedi dokaz suprotnosti (1):

1. Muškarci predlažu kako bi barem jedan čovjek trebao biti odbijen od strane valjanog partnera.
2. Neka su *m* i *w* prvi takvi koji se odbacuju u *S* (to se događa zato što je *w* odabrao neki *m' >m).*
3. Neka *S'* bude stabilno podudaranje s *m*, *w* uparenim (kod *S'* postoji definicija valjanosti).

Naredni zahtjev je Gale Shapley s čovjekom koji isti predlaže (rezultira optimalnim podudaranjem). Dokaz suprotnosti (2) je:

1. Neka bude partner *m'* sa *S'.*
2. *m'* nije odbačen od strane valjane žene u *S* prije nego što *m* bude odbijen od strane *w* (prema pretpostavci).
3. *-> m* preferira *w do w'.*
4. Valja znati da se preferira *m'* iznad *m*, svog partnera u *S'.*
5. *-> m'* i *w* čine blok za blokiranje u *S' (><).*

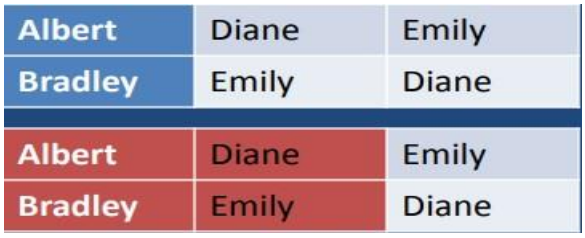
Sljedeći zahtjev je Gale Shapley s čovjekom koji predlaže, a rezultira s žensko-pesimalnim podudaranjem. Dokaz suprotnosti: (Dickerson, n_d, str. 21).

1. *m* i *w* odgovaraju *S*, *m* nije najgora vrijednost.
2. ->postoji stabilna *S'* s *w*, koji je uparen s *m' < m.*
3. Pretpostavlja se da *w* bude partner (*m* u *S*).
4. *m* preferira *w* s *w* (po čovjekovoj optimalnosti).
5. *->m* i *w* formiraju par za blokiranje u *S' (><).*

Također, ovdje se postavlja i pitanje poticaja, a ono glasi: „Mogu li obje strane imati koristi od pogrešnog izvješćivanja?“. Odgovor na isto pitanje je da sudionici mogu označiti moguće podudarnosti kao neprihvatljive (oblik skraćivanja popisa prioriteta). Tako se smatra da bilo koji algoritam daje optimalno podudaranje žena (muškaraca). Prethodno definirana tvrdnja se smatra dominantnom strategijom (Roth, 1982.).

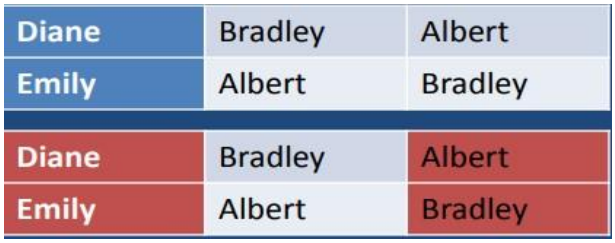
U Gale Shapleyu gdje se sami muškarci predlažu, žene mogu imati koristi od pogrešnog izvješćivanja o sklonostima.

Slika 109.: Prikaz iskrenog izvješćivanja



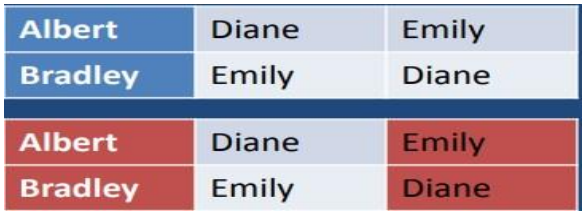
Izvor: Dickerson, n_d, str. 23

Slika 110.: Prikaz iskrenog izvješćivanja



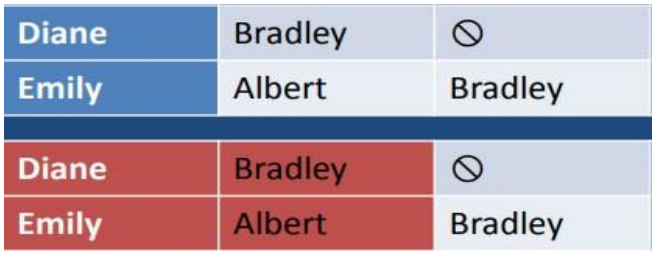
Izvor: Dickerson, n_d, str. 23

Slika 111.: Prikaz strategijskog izvješćivanja



Izvor: Dickerson, n_d, str. 23

Slika 112.: Prikaz strategijskog izvješćivanja



Izvor: Dickerson, n_d, str. 23

Ovdje se definira sljedeći zahtjev gdje ne postoji mehanizam podudaranja koji je:

1. Dokaz strategije kod obje strane.
2. Uvijek rezultira stabilnim ishodom s obzirom na otkrivene preferencije.

Kod produženja stabilnog braka postoji jedno- na- mnogo podudaranja gdje spada sljedeće: (Dickerson, n_d, str. 26).

1. Problem bolnica/ stanovnika (inače problem fakulteta/ studenata, odnosno problem priznavanja).
2. Stroga rangiranja prednosti glede svih strana.
3. Jedna strana (primjerice, bolnica) može prihvatiti *q> 1* stanovnika.
4. Uveden je 1962. godine (Gale i Shapley).

Odgođeno prihvaćanje:

1. Stanovnici su bez premca (prazne liste čekanja).
2. Svi stanovnici imaju pravo na prvi izbor.
3. Svaka bolnica se smješta na prvo mjesto na listi čekanja.
4. Odbijeni stanovnici primjenjuju se na drugi izbor.
5. Bolnice ažuriraju liste čekanja.(…).
6. Isto se ponavlja dok svi stanovnici ne budu na popisu ili dok se ne prijave u sve bolnice.

Već oko 20 godina većina ljudi smatra da su bolnice/ stanovnici ≠ braka vrlo slični problemi. Stoga, Roth (1985.) pokazuje da ne postoji stabilan algoritam podudaranja i da je ispitivanje istine dominantna strategija za bolnice.

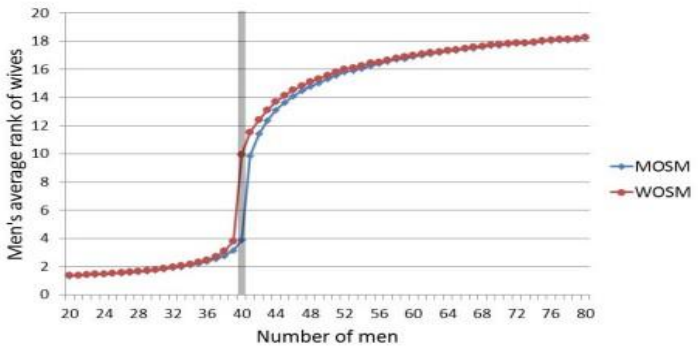
Podudaranje u praksi:

1. 1940.- ih je postojalo decentralizirano rezidentno bolničko podudaranje.
2. 1950-e uvodi se odgođeni algoritam prihvaćanja za bolnicu.
3. 1970.- ih parovi sve više ne koriste odgođeni algoritam za bolnicu.
4. 1998. podudaranje s nekoliko ograničenja (primjerice, stabilno podudaranje više ne postoji).

Također, postoji i neravnoteža (Ashlagi, 2013.). Konkretno, ovdje se postavlja pitanje u slučaju da postoji *n* muškaraca i da taj *n* bude različit od *n* žena? Kako to isto može utjecati na same sudionike? Koja je tada veličina jezgre? Ovdje se smatra da je dobro

biti na kratkoj strani tržišta. Naime, vremenska složenost je *log (n),* dok se rang dobiva na slučajan način.

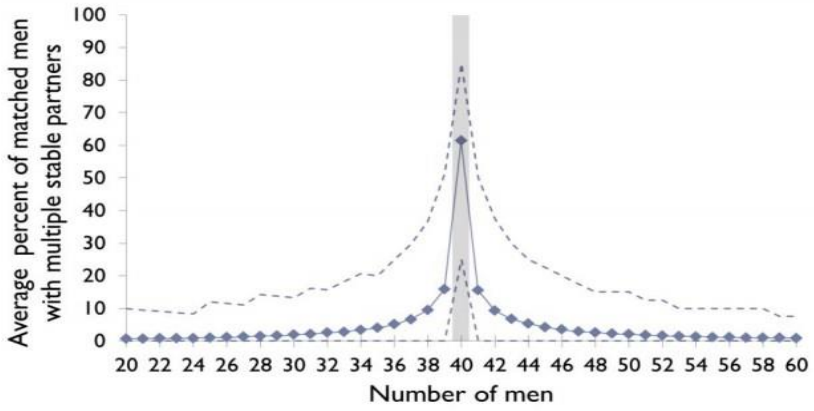
Slika 113.: Prikaz prethodne tvrdnje



Izvor: Dickerson, n_d, str. 30

Ista neravnoteža kaže da ne postoji puno stabilnog podudaranja *s* čak i kod malih neravnoteža na tržištima.

Slika 114.: Prikaz prethodne tvrdnje



Izvor: Dickerson, n_d, str. 31

Nadalje, postoji teorem o ruralnoj bolnici (Roth, 1986.) (Dickerson, n_d, str. 32):

1. Ovdje se obuhvaća skup stanara i bolnica koje nemaju premca koji je isti za sva stabilna podudaranja.
2. Pretpostavlja se da su *n* muškarci, dok su *n +1* žene (jedna je žena bez premca u svim stabilnim susretima).

3. Spuštanje istog stabilnog podudaranja.
4. Poduzimaju se stabilne podudarnosti s *n* žena (treba ostati stabilan ako se dodaje *w* u slučaju da nema muškaraca koji preferiraju *w* za njihovo trenutno podudaranje).
5. Prosječan rang muškaraca kod podudaranja je nizak.

Online dolazak prema Khulleru i sur. 1993.:

1. Kod slučajnih preferencija, muškarci stižu s vremenom, te kada se jednom podudaraju nitko se ne može prebaciti.
2. Kod algoritma se uspoređuje *m* s najvišim slobodnim *w*.
3. U prosjeku je vremenska složenost *O(n log(n))* kod nestabilnih parova.
4. Nijedan nedeterministički ili slučajni algoritam ne može imati bolju vremensku složenost od Ω (n^2) kod nestabilnih parova.
5. Nije bolji s randomizacijom.

Nepotpune formulacije prema Manlove et al. 2002.:

1. Prije su bile kompletne te stroge postavke (jednostavno izračunavanje, puno lijepih svojstava).
2. Nepotpune postavke (mogu postojati stabilna podudaranja različitih veličina).
3. Pronalaženje maksimalnog ili minimalnog kardinalnog stabilnog podudaranja (određivanje jesu li <*m, w*> stabilni, pronalaženje „egalitarnog“ podudaranja, i sl.).

11.2 Problem zadatka

Problem pronalaženja odgovarajućeg podudaranja postaje složeniji u ponderiranom grafu. U takvom grafu vlada zainteresiranost za pronalaženje podudaranja s maksimalnom ukupnom težinom. Problem se naziva problem zadatka. Problem zadatka za cjelokupni bipartitni graf, s dva skupa vrhova iste veličine, naziva se optimalnim zadatkom. (Drozdek, 2001., str. 424).

$O(|V|)^3$ algoritam je posljedica Kuhna i Munkera. Za bipartitni graf *G = (V, E), V = U U W* definira se funkcija oznake *f: U U W -> R* tako da je oznaka *f (v)* broj dodijeljen svakoj točki *v* tako da za sve vrhove *v, u, f (u) + f (v)≥ težina (rub* (u_v)). Stvori se skup

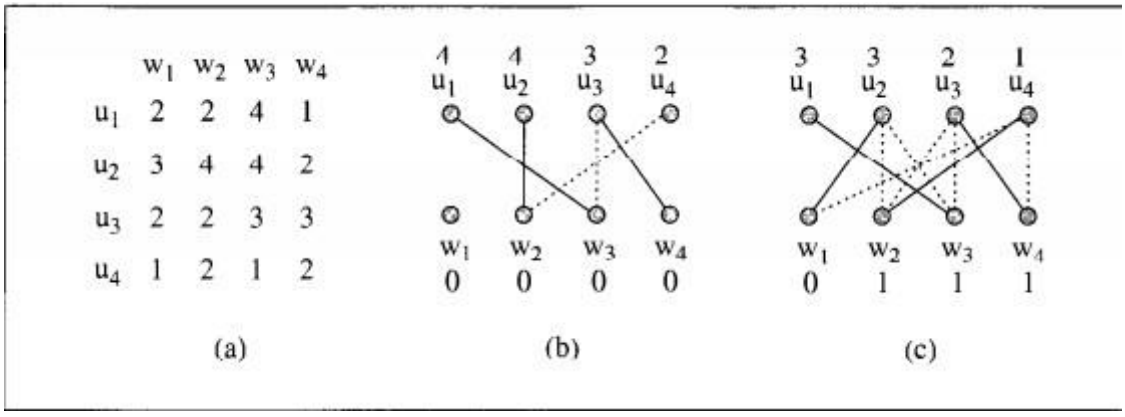
H = { težina (u_v) ∈ E: f (u) + f (v) = težina (rub (u_v))}, a zatim podgraf jednakosti G_f = *(V, H)*. Kuhn-Munkersov algoritam temelji se na teoremu koji kaže da za funkciju obilježavanja *f* i podgraf jednakosti G_f graf *G* sadrži savršeno podudaranje, pa je onda to podudaranje optimalno.

Slika 115.: Prikaz algoritma

```
optimalAssignment()
    G_f = equality subgraph for some vertex labeling f;
    M = matching in G_f;
    S = {some unmatched vertex u}; // beginning of an augmenting path P;
    T = null;
    while M is not a perfect matching
        Γ(S) = {w: ∃u∈ S: edge(uw)∈G_f}; // vertices adjacent in G_f to the vertices in S;
        if Γ(S) == T
            d = min{(f(u) + f(w) - weight(edge(uw)): u∈S, w∉T};
            for each vertex v
                if v ∈ S
                    f(v) = f(v) - d;
                else if v ∈ T
                    f(v) = f(v) + d;
            construct new equality subgraph G_f and new matching M;
        else // if T ⊂ Γ(S)
            w = a vertex from Γ(S) - T;
            if w is unmatched // the end of the augmenting path P;
                P = augmenting path just found;
                M = M ⊕ P;
                S = {some unmatched vertex u};
                T = null;
            else S = S ∪ {neighbor of w in M};
                T = T ∪ {w};
```

Izvor: Drozdek, 2001., str. 424 i 425

Slika 116. prikazuje potpuni bipartitni graf $G = (\{u_1, ..., u_4\} U \{w_1, ..., w_4\}, E)$ koji ima težine definirane matricom na slici *a* ili primjer primjene algoritma funkcije *optimalAssignment():*



Izvor: Drozdek, 2001., str. 425

Za početno obilježavanje odabire se funkcija *f* takva da je *f (u) = max (težina (rub (uw)))*, tj. maksimalna težina je u matrici težine u retku za vrh *u, i f (w) = 0*, tako da je za graf *G* početno obilježavanje kao na slici *b*. Odabrala se podudarnost kao u slici *b* te se postavio skup *S* na $\{u_4\}$ i *T* na nulu.

U prvoj iteraciji *while* petlje, $\tau(S) = \{w_2, w_4\}$, jer su i w_2 i w_4 susjedi u_4, što je jedini element *S*, jer je T podskup $\tau(S)$, to jest, $\emptyset \subset \{w_2, w_4\}$, izvršena je klauzula vanjskog *else*, pri čemu je $w = w_2$ (jednostavno se izabere prvi element ako $\tau(S)$ nije u *T*), i zato što *w2* nije usklađen, klauzula *internal else* je izvršena, u kojem se proširi *S* na $\{u_2, u_4\}$, jer je u_2 usklađen i susjedan s w_2, i proširuje *T* na $\{w_2\}$.

Sva iteracija prikazana je u sljedećoj tablici (slika 117.):

Iteration	Γ(S)	S	w	T
0	∅	$\{u_4\}$		∅
1	$\{w_2, w_4\}$	$\{u_2, u_4\}$	w_2	$\{w_2\}$
2	$\{w_2, w_3, w_4\}$	$\{u_1, u_2, u_4\}$	w_3	$\{w_2, w_3\}$
3	$\{w_2, w_3, w_4\}$	$\{u_1, u_2, u_3, u_4\}$	w_4	$\{w_2, w_3, w_4\}$
4	$\{w_2, w_3, w_4\}$			

Izvor: Drozdek, 2001., str. 425

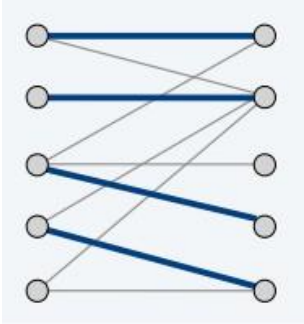
U četvrtoj iteraciji stanje vanjskog *if* izraza postaje istinito jer su skupovi *T* i $\tau(S)$ sada jednaki, dakle udaljenost $d = min \{(f (u) + f (w) težina (rub (uw)) u \in S, w \notin T\}$, jer je w_1

jedini vrh koji nije u $T = \{w_2, w_3, w_4\}$, $d = min \{(f(u) + f(w_1) - težina (rub (uw_1))): u \in S = \{u_1, u_2, u_3, u_4\}\} = min \{(4 + 0-2), (4 + 0-2), (3 + 0-4), (2 + 0-1)\} = 1$. S udaljenosti, oznake vrha u grafu *G* se ažuriraju kako bi postale oznake na slici *c*. Oznake svih četiriju točaka u *S* smanjene su s $d = 1$, a sve tri točke u *T* su povećane pored toga, kreira se podgraf jednakosti koji uključuje sve rubove, kao na slici *c*, a zatim se pronalazi podudarnost koja uključuje rubove nacrtane krutim crtama. To je savršeno podudaranje, i stoga, optimalni zadatak je taj koji zaključuje izvršenje algoritma.

11.3 Usklađivanje u nebibarnim grafovima

Kod ove vrste grafova postoji sljedeći problem. S obzirom na bipartitni graf *G=(L, R, E),* treba pronaći podudaranje maksimalne kardinalnosti. Preciznije, podskup rubova $M \subset E$ je podudarnost ako se svaki čvor pojavljuje na točno jednom rubu u *M.*

Slika 118.: Prikaz podudaranja



Izvor: Tardos i sur., 2005., str. 29

Kod podudaranja (usklađivanja) u nebibarnim grafovima je definirano na općim grafovima gdje postoji: (Dickerson, n_d, str. 36)

1. Skup rubova, svaki vrh je uključen najviše jednom (nema više „muškaraca“ ili „žena“).
2. Stabilni problem cimera je stabilan brak koji se može generalizirati na bilo koji graf.
3. Svaka točka rangira sve na *n- 1* (ostale točke). Primjerice, varijacije s/ bez skraćenja.
4. Isti je pojam skalabilnosti.

Slika 119. prikazuje podudaranje koje nije stabilno. Konkretno, svatko tko je uparen s *Draculom (i)* preferira neki drugi *v (ii)* koji je preferiran od strane *v:*

Alana	Brian	Cynthia	Dracula
Brian	Cynthia	Alana	Dracula
Cynthia	Alana	Brian	Dracula
Dracula	(Anyone)	(Anyone)	(Anyone)

Izvor: Dickerson, n_d, str. 37

Naime, postavlja se pitanje: „Može li se izgraditi algoritam koji će pronaći stabilno podudaranje ili koji će pak, sadržavati izvješća koja ustvari ne postoje u polinomnom vremenu“? Odgovor na to pitanje je da, po Irvingu 1985., iz razloga jer se gradi isti na Gale- Shapley idejama te radu McVitiea i Wilsona 1971.

Prva faza Iringovog algoritma: (Dickerson, n_d, str. 39).

1. Pokretanje algoritma odgođenog prihvatnog tipa.
2. Ako je najmanje jedna osoba bez premca: nepostojanje.
3. Inače se stvara smanjeni skup postavki.
4. Prijedlog iz *b->a* skraćuje sve *x* nakon *b.*
5. Uklanjanje željenih postavki iz *x-a.*
6. Napomena: *a* se nalazi na vrhu popisa *b.*
7. Ako je bilo koji popis prazan: nepostojanje.
8. Inače postoji „stabilna tablica“ i nastavlja se do druge faze.

Algoritam stabilnih tablica:

1. *A* je prvi na *b* popisu, a *b* zadnji na *a*.
2. *A* se ne nalazi na popisu *b* (*b* nije na popisu, preferira se zadnji element na popisu za *b*).
3. Nijedan smanjeni popis nije prazan. Ovdje se nameću dvije napomene. Prva je ta da je stabilna tablica s duljinom svih (popisa prvog) stabilno podudaranje, dok je druga da se bilo koja stabilna sutablica stabilne tablice može dobiti eliminacijom rotacije.

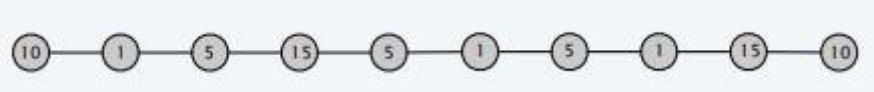
Druga faza Irvingovog algoritma:

1. Stabilna tablica ima duljinu prvog popisa (povratno podudaranje).
2. Identificiranje rotacije. *($(a_0, b_0), (a_1, b_1), \ldots, (a_{k-1}, b_{k-1})$)* tako da je b_i prvi na reduciranoj listi a_i, dok je b_{i+1} drugi na reduciranoj listi a_i *(i +1 je mod k)* b_0.
3. Uklanjanje (a_0 odbija b_0), predlaže se b_1 (tko prihvaća), i sl.
4. Ako bilo koji popis postane prazan: nepostojanje podudaranja.
5. Ako sutablica udovoljava duljini prvog popisa: vraća se podudaranje.

Naime, kod Iringovog algoritma postoji zahtjev glede problema stabilnih cimera koji završava u polinomnom vremenu (vremenska složenost $O(n^2)$). To zahtjeva određena razmatranja strukture podataka (naivni algoritam ima trajanje oko $O(n^3)$).

Mjesto natjecateljskog objekta kao ulaz se uzima graf koji sadrži težinu na svakom čvoru. Ono što se događa ovdje je da se dva natjecatelja izmjenjuju glede odabira čvorova. Točnije, ovdje nije dopušteno odabrati čvor u slučaju da je odabran bilo koji od njegovih susjeda. Konkretno, cilj je da se odabere podskup maksimalne težine čvorova.

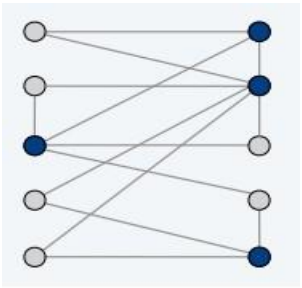
Slika 120.: Prikaz slučaja kada drugi igrač može jamčiti 20, ali ne i 25



Izvor: Tardos i sur., 2005., str. 31

Ovdje postoji još pojam nezavisnog seta gdje se problem ogleda u grafu *G=(V, E)* kamo se treba pronaći nezavisni skup maksimalne kardinalnosti. Naime, ovdje je podskup A podskup *S* od *V* koji je neovisan za svaki *(u, v) e E* (gdje *u* nije u *S* ili *v* nije u *S*, ili oba slučaja).

Slika 121.: Prikaz neovisnog skupa



Izvor: Tardos i sur., 2005., str. 30

Postoji šest reprezentativnih problema, a to su: (Tardos i sur., 2005., str. 32).

1. Varijacije na temu samostalnog skupa.
2. Intervalni raspored: *O(nlogn)* glede pohlepnog algoritma.
3. Raspoređivanje ponderiranih intervala: *O(nlogn)* glede algoritma dinamičkog programiranja.
4. Dvostruko podudaranje: $O(n^k)$ kod algoritma koji se temelji na maksimalnom protoku.
5. Nezavisni skup: *NP*- kompletan.
6. *N*- konkurentna lokacija objekta: *PSPACE*- kompletan.

Zaključak

Tema ovog diplomskog rada je bila apstraktni tip podataka graf s naglaskom na same grafove, ali i na izradu istih. Preciznije, stavljen je naglasak na implementacije grafova, tj. na pretraživanje grafova i traženje najkraćih puteva pritom koristeći različite algoritme. Međutim, ovdje su opisani samo neki važniji algoritmi koji se koriste kod grafova.

Kao što je već spomenuto, graf se sastoji od lukova (bridova) te od čvorova (vrhova). Naime u matematici, rub je definiran polaznom i krajnjom točkom, pa se kaže da isti "pokazuje" ili "ide" od *x* do *y*. Čvorovi mogu biti dio strukture grafa ili mogu biti vanjski entiteti predstavljeni pomoću cjelobrojnih indeksa ili referenci.

Postoje i dvije glavne vrste grafova, a to su usmjereni i nesumjereni graf. Kod usmjerenog grafa rubovi imaju smjer, tj. rubovi ovdje prelaze iz jedne verzije u drugu, dok kod neusmjerenog grafa rubovi nemaju smjer.

Važno je zaključiti da postoje tri glavne strukture za reprezentaciju grafova koje se koriste u praksi. Prva se naziva lista susjedstva i implementira se kao niz s jednom vezanom listom za svaki izvorni čvor, a sadrži destinacijske čvorove rubova koji izlaze iz svakog čvora. Druga je dvodimenzionalna Booleova matrica susjedstva, koja pokazuje postoji li rub između svakog pojedinog para čvorova. Liste susjedstava su bolji izbor za rijetke grafove, dok su za guste grafove bolji izbor matrice susjedstva. Za grafove koji pokazuju neku pravilnost u položaju rubova, simbolički su grafovi također dobar izbor. Treća struktura, koja se koristi za prikaz grafova, je matrica incidencije. Ona je matrica koja predstavlja graf tako da se uz pomoć te matrice može nacrtati graf. Kao i u svakoj matrici, postoje i redovi i stupci matrica incidencije. Redovi matrice predstavljaju broj čvorova, dok stupci matrice predstavljaju broj rubova u danom grafu. Ako postoji *"n"* broj redaka u danoj matrici incidencije, to znači da u grafu postoji *"n"* broj čvorova. Slično tome, ako postoji *'m'* broj stupaca u datoj matrici incidencije, to znači da u tom grafu postoji *'m'* broj rubova.

Naime, postoje dvije vrste pretraživanja grafova, a to su pretraživanje u širinu i u dubinu. Pretragom po širini se polazi od jednog čvora gdje se prvo obiđu svi njegovi direktni susjedi koji se još nisu posjetili. Potom se nastavlja dalje od tih susjeda obilazak na isti način. Kao i kod pretrage po dubini, da bi osigurali da se svaki čvor posjeti točno jedanput, uvodi se logički niz *'m'.* Čvorove koje su se posjetili čuvaju se

u nizu samog reda, iz kojeg se redom uzima jedan po jedan čvor da bi se nastavio obilazak.

U matematičkom polju teorije grafova razgranato stablo povezanog, neusmjerenog grafa je stablo koje se sastoji od svih vrhova i nekih (ili možda čak i svih) rubova grafa. Neformalno, razgranato stablo predstavlja odabir rubova od koje se formira stablo koje obuhvaća svaki vrh. To znači da svaki vrh postoji u stablu, ali bez petlji. S druge strane, svaki most mora pripadati stablu. Razgranato stablo povezanog grafa se također može definirati kao maksimalan skup rubova koji ne sadrži petlje, ili kao minimalan skup rubova koji sadrže sve vrhove. U određenim poljima teorije grafova, često je korisno pronaći minimalno razgranato stablo opterećenog grafa. Drugi problemi optimizacije razgranatog stabla su također proučavani, uključujući i maksimalno razgranato stablo, minimalno stablo koje obuhvaća najmanje *k* vrhova, minimalno razgranato stablo s najviše *k* grana po vrhu, ograničeno razgranato stablo, razgranato stablo s najvećim brojem listova (usko povezano s najmanjim dominirajućim skupom) i razgranato stablo s najmanje listova (usko povezano s problemom Hamiltonovog puta). Zaključno, kod minimalnog razgranatog stabla se ističu dva algoritma koja se pretežito koriste, a to su Primov te Kruskalov algoritam. Detaljnije, Primov algoritam gradi minimalno razgranato stablo na sljedeći način: povezuje dva čvora čiji rub trenutno ima najmanju težinu, postupak ponavlja sve dok ne budu povezani svi čvorovi. Prilikom povezivanja čvorova pazi se da se ne napravi ciklus. Za razliku od Primovog algoritma, Kruskalov algoritam gradi minimalno razgranato stablo na sljedeći način: kreće od prvog čvora u grafu, provjerava sve težine rubova koje izlaze iz tog čvora, uzima rub s najmanjom težinom i spaja polazni čvor s tim susjedom. Postupak se ponavlja sve dok svi čvorovi ne budu povezani i pri tom se pazi da nema ciklusa.

Kod algoritma s jednim izvorom je opisan Bellman- Ford algoritam. To je algoritam koji izračunava najkraće puteve od jednog izvora (vrha) na sve ostale vrhove u ponderirani digraf. Sporiji je od Dijkstrinog algoritma za isti problem, ali svestraniji, jer može rukovati grafovima u kojima su neke od rubnih težina negativni brojevi. Negativne težine na rubovima nalaze se u različitim primjenama grafova, otuda je i korisnost ovog algoritma. Tako ako graf sadrži "negativni ciklus" (tj. ciklus čiji rubovi zbrajaju negativnu vrijednost), koji je dostupan od izvora, tada ne postoji najjeftiniji put. Pod tim se misli na bilo koji put koji ima točku negativnog ciklusa te koji može biti jeftiniji još jednom

šetnjom negativnog ciklusa. U takvom slučaju, algoritam Bellman-Ford može otkriti i prijaviti negativan ciklus.

Druga vrsta algoritma, koja je obrađena, je Dijkstrin algoritam, koji služi za nalaženje najkraćeg puta u usmjerenom grafu s nenegativnim vrijednostima rubova. Primjerice, ako čvorovi predstavljaju gradove, a vrijednosti rubove kao udaljenosti između onih gradova koji su direktno povezani, Dijkstrin algoritam nalazi najkraći put između ta dva grada.

Kod najkraćih puteva svih parova pripada Floyd- Warshallov algoritam te Johnsonov algoritam. Tako je Floyd- Warshall algoritam učinkovit algoritam za nalaženje svih parova najkraćeg puta na grafu. Posebice je zajamčeno da će se naći najkraći put između svakog para vrha u grafu. Graf stoga može imati rubove negativne težine, ali ne može imati cikluse negativne težine (tada je najkraći put nedefiniran). Ovaj se algoritam također može koristiti za otkrivanje prisutnosti negativnih ciklusa. Konkretno, to je graf ako je na kraju algoritma udaljenost od vrha sama po sebi negativna.

Johnson algoritam je način da se pronađu najkraći putevi između svih parova vrhova u usmjerenom grafu. Također, omogućuje da neke težine rubova budu negativni brojevi, ali isto tako ne mogu postojati ciklusi negativne težine. Točnije, on djeluje korištenjem algoritma Bellman-Ford za izračunavanje transformacije ulaznog grafa koji uklanja sve negativne težine. Međutim, on koristi Dijkstrin algoritam na transformiranom grafu. Slična se tehnika preusmjeravanja koristi i u algoritmu Suurballea za pronalaženje dva odvojena puta minimalne ukupne duljine između istih dvaju vrhova u grafu s negativnim rubnim težinama.

U ovom diplomskom radu je pisano o grafovima koji predstavljaju prirodan i dobar način za modeliranje nelinearnih relacija. Operacije s grafovima su složenije nego s drugim strukturama. Stoga, ne postoji zasebna struktura zvana graf. Valja naglasiti i to da su grafovi najsličniji strukturi podataka stablo. Same implementacije grafova u programskim jezicima Python te C++-u sadrže određene operacije koje imaju određenu složenost. Tako operacije *InsertVertex, InsertEdge, DeleteEdge, Adjacent* i *IsEmpty* imaju konstantnu složenost (O(1)), dok *DeleteVertex* ima kvadratnu složenost ($O(n^2)$). Ukratko, u ovom radu je stavljen naglasak i na lakšem razumijevanju i razlikovanju samih algoritama pretraživanja grafova po samom prikazu i nalazak najkraćeg puta kod pojedinog grafa.

Literatura

1) Abdelnabi Khaled Abdelaziz, O. (2019a), *Minimum Spanning Tree*, dostupno na: https://www.hackerearth.com/practice/algorithms/graphs/minimum-spanning-tree/tutorial/, pristup: 16.4.2019.
2) Abdelnabi Khaled Abdelaziz, O. (2019b), *Shortest Path Algorithms*, dostupno na: https://www.hackerearth.com/practice/algorithms/graphs/shortest-path-algorithms/tutorial/, pristup: 3.5.2019.
3) Algolist, (n_d), *Introduction to graphs*, Algorithms and Data Structures with implementations in Java and C++, dostupno na: http://www.algolist.net/Data_structures/Graph, pristup: 2.5.2019.
4) Anonymous_1, (2014.), *Graph1*, dostupno na: http://www.mrgeek.me/wp-content/uploads/2014/04/graph1_example.png, pristup: 10.4.2019.
5) Anonymous_2, (n_d), *Eulerian graphs and semi eulerian graphs*, dostupno na: http://mathonline.wikidot.com/eulerian-graphs-and-semi-eulerian-graphs, pristup: 9.4.2019.
6) Anonymous_3, (n_d), *Hamiltonian graphs and semi hamiltonian graphs*, dostupno na: http://mathonline.wikidot.com/hamiltonian-graphs-and-semi-hamiltonian-graphs, pristup: 9.4.2019.
7) Anonymous_4, (n_d), dostupno na: https://i.stack.imgur.com/VW9yr.png, pristup: 15.4.2019.
8) Anonymous_5, (2019.), *NP Hard and NP-Complete Classes*, dostupno na: https://www.tutorialspoint.com/design_and_analysis_of_algorithms/design_and_analysis_of_algorithms_np_hard_complete_classes.htm, pristup: 16.4.2019.
9) Anonymous_6, (2019.), *Spanning tree*, dostupno na: https://www.tutorialspoint.com/data_structures_algorithms/spanning_tree.htm, pristup: 16.4.2019.
10) Anonymous_7, (n_d), *Minimalno povezujuće stablo*, dostupno na: poincare.matf.bg.ac.rs/~jelenagr/AIDA/mcst.pdf, pristup: 3.5.2019.
11) Anonymous_8, (n_d), *Prim's Algorithm*, dostupno na: https://www.programiz.com/dsa/prim-algorithm, pristup: 17.4.2019.
12) Anonymous_9, (2014.), *The Minimum Spanning Tree Algorithm*, dostupno na: https://www-m9.ma.tum.de/graph-algorithms/mst-prim/index_en.html, pristup: 3.5.2019.

13) Anonymous_10, (2019.), *What Is the Best Shortest Path Algorithm?*, dostupno na: https://www.myrouteonline.com/blog/what-is-the-best-shortest-path-algorithm, pristup: 2.6.2019.
14) Anonymous_11, (n_d), *The Single Source Shortest Path algorithm*, dostupno na: https://neo4j.com/docs/graph-algorithms/current/algorithms/single-source-shortest-path/, pristup: 29.5.2019.
15) Anonymous_12, (n_d), *Bellman Ford's Algorithm*, dostupno na: https://www.programiz.com/dsa/bellman-ford-algorithm, pristup: 4.5.2019.
16) Anonymous_13, (2014.), *The classic among shortest path algorithms*, dostupno na: https://www-m9.ma.tum.de/graph-algorithms/spp-dijkstra/index_en.html, pristup: 17.4.2019.
17) Anonymous_14, (n_d), *All-Pairs Shortest Path*, dostupno na: http://mathworld.wolfram.com/All-PairsShortestPath.html, pristup: 4.5.2019.
18) Anonymous_15, (n_d), *All Pairs Shortest Path (APSP) Problem*, dostupno na: https://www.cs.rochester.edu/u/nelson/courses/csc_173/graphs/apsp.html, pristup: 4.5.2019.
19) Anonymous_16, (n_d), *The All Pairs Shortest Path algorithm*, dostupno na: https://neo4j.com/docs/graph-algorithms/current/algorithms/all-pairs-shortest-path/, pristup: 3.5.2019.
20) Anonymous_17, (n_d), *All-Pairs Shortest Paths*, dostupno na: http://jeffe.cs.illinois.edu/teaching/algorithms/book/09-apsp.pdf, pristup: 4.5.2019.
21) Anonymous_18, (2015.), *Shortest Paths between all Pairs of Nodes*, dostupno na: https://www-m9.ma.tum.de/graph-algorithms/spp-floyd-warshall/index_en.html, pristup: 11.6.2019.
22) Anonymous_19, (2019.), *Johnson's Algorithm*, dostupno na: https://brilliant.org/wiki/johnsons-algorithm/, pristup: 5.5.2019.
23) Anonymous_20, (2018.), *Graph – Detect Cycle in a Directed Graph*, dostupno na: https://algorithms.tutorialhorizon.com/graph-detect-cycle-in-a-directed-graph/, pristup: 6.5.2019.
24) Anonymous_21, (2018.), *Graph – Detect Cycle in Undirected Graph using DFS*, dostupno na: https://algorithms.tutorialhorizon.com/graph-detect-cycle-in-undirected-graph-using-dfs/, pristup: 6.5.2019.

25) Anonymous_22, (n_d), *Graph Algorithms III: Union-Find*, dostupno na: https://www.cs.cmu.edu/~avrim/451f11/lectures/lect1020.pdf, pristup: 6.5.2019.
26) Anonymous_23, (n_d), *Graph Union*, dostupno na: http://mathworld.wolfram.com/GraphUnion.html, pristup: 6.5.2019.
27) Anonymous_24, (n_d), *Acyclic*, dostupno na: https://www.techopedia.com/definition/26/acyclic, pristup: 11.5.2019.
28) Anonymous_25, (2016.), *Acyclic Graph & Directed Acyclic Graph: Definition, Examples*, dostupno na: https://www.statisticshowto.datasciencecentral.com/directed-acyclic-graph/, pristup: 11.5.2019.
29) Anonymous_26, (n_d), *Acyclic Graph*, dostupno na: http://mathworld.wolfram.com/AcyclicGraph.html, pristup: 11.5.2019.
30) Anonymous_27, (2019.), *Flow Network* ,dostupno na: https://brilliant.org/wiki/flow-network/, pristup: 11.5.2019.
31) Anonymous_28, (n_d), *The relabel-to-front algorithm*, dostupno na: http://www.euroinformatica.ro/documentation/programming/!!!Algorithms_CORMEN!!!/DDU0165.html, pristup: 23.5.2019.
32) Anonymous_29, (n_d), *Graph Theory – Matchings*, dostupno na: https://www.tutorialspoint.com/graph_theory/graph_theory_matchings.htm, pristup: 17.5.2019.
33) Anonymous_30, (2014.), *BFS algorithm in Python*, dostupno na: https://stackoverflow.com/questions/23477921/bfs-algorithm-in-python, pristup: 20.7.2019.
34) Anonymous_31, (n_d), *breadth-first-search*, dostupno na: https://www.techiedelight.com/breadth-first-search/, pristup: 3.8.2019.
35) Anonymous_32, (2013.), *Depth-first search (DFS) – Python implementation using stack*, dostupno na: https://ownagezone.wordpress.com/2013/02/22/depth-first-search-dfs-python-implementation-using-stack/, pristup: 22.7.2019.
36) Anonymous_33, (n_d), *depth-first-search*, dostupno na: https://www.techiedelight.com/depth-first-search/, pristup: 4.8.2019.
37) Anonymous_34, (n_d), *Prim's Spanning Tree Algorithm*, dostupno na: https://bradfieldcs.com/algos/graphs/prims-spanning-tree-algorithm/, pristup: 22.7.2019.

38) Anonymous_35, (2010.), *Disjoint Sets*, dostupno na: https://programmingpraxis.com/2010/04/02/disjoint-sets/, pristup: 22.7.2019.
39) Anonymous_36, (2010.), *Minimum Spanning Tree: Kruskal's Algorithm*, dostupno na: https://programmingpraxis.com/2010/04/06/minimum-spanning-tree-kruskals-algorithm/, pristup: 25.7.2019.
40) Anonymous_37, (n_d), *Minimum Spanning Tree*, dostupno na: https://www.hackerearth.com/practice/algorithms/graphs/minimum-spanning-tree/tutorial/, pristup: 3.8.2019.
41) Anonymous_38, (n_d), *Python Program to Implement Bellman-Ford Algorithm*, dostupno na: https://www.sanfoundry.com/python-program-implement-bellman-ford-algorithm/, pristup: 25.7.2019.
42) Anonymous_39, (2015.), *Removing graph nodes of a specific type and with exactly two neighbors*, dostupno na: https://codereview.stackexchange.com/questions/81880/removing-graph-nodes-of-a-specific-type-and-with-exactly-two-neighbors, pristup: 25.7.2019.
43) Anonymous_40, (n_d), *Single-source-shortest-paths-bellman-ford-algorithm*, dostupno na: https://www.techiedelight.com/single-source-shortest-paths-bellman-ford-algorithm/, pristup: 4.8.2019.
44) Anonymous_41, (n_d), *Python Program to Implement Dijkstra's Shortest Path Algorithm*, dostupno na: https://www.sanfoundry.com/python-program-implement-dijkstras-shortest-path-algorithm/, pristup: 25.7.2019.
45) Anonymous_42, (n_d), *Single-source-shortest-paths-dijkstras-algorithm*, dostupno na: https://www.techiedelight.com/single-source-shortest-paths-dijkstras-algorithm/, pristup: 4.8.2019.
46) Anonymous_43, (n_d), *Python Program to Implement Floyd-Warshall Algorithm*, dostupno na: https://www.sanfoundry.com/python-program-implement-floyd-warshall-algorithm/, pristup: 28.7.2019.
47) Anonymous_44, (n_d), *Flloyd Warshall Algorithm – Dynamic Programming Solutions*, dostupno na: https://www.sanfoundry.com/dynamic-programming-solutions-flloyd-warshall-algorithm/, pristup: 4.8.2019.
48) Bačlija, A. (n_d), *Algoritam Bellman- Ford*, Prirodno- matematički fakultet, Podgorica, dostupno na: https://dokumen.tips/documents/algoritam-bellman-ford.html, pristup: 4.5.2019.

49) Bujanović, T. (2016.), *Binarna stabla. Minimalna razapinjuća stabla*, Sveučilište u Zagrebu, PMF, dostupno na: https://web.math.pmf.unizg.hr/nastava/matsoft/DobreDZ/2015-16/LaTeX/TomislavBujanovic.pdf, pristup: 16.4.2019.
50) Burfield, C. (2013.), *All-Pairs Shortest Paths with Matrix Multiplication*, dostupno na: http://www-math.mit.edu/~rothvoss/18.304.1PM/Presentations/2-Chandler-slideslect2.pdf, pristup: 4.5.2019.
51) Chumbley, A., Moore, K., Hor, T., Khim, J. i Ross, E. (2019.), *Ford-Fulkerson Algorithm*, dostupno na: https://brilliant.org/wiki/ford-fulkerson-algorithm/, pristup: 17.4.2019.
52) Clifford A. Shaffer, (2013.), *Data Structures and Algorithm Analysis*, Edition 3.2 (C++ Version), Department of Computer Science Virginia Tech Blacksburg, VA 24061
53) Cormen, H. Thomas, Leiserson, E. Charles, Rivest, L. Ronald, Stein Clifford, (2001.), *Introduction to Algorithms*, Second Edition, Cambridge , Massachusetts London, England
54) Dale, N. (2003.), *C++ Data structures*, Third Edition, University of Texas, Austin
55) Dickerson, John, P. (n_d), *Stable Matching*, Carnegie Mellon University, dostupno na: http://www.cs.cmu.edu/~arielpro/15896s16/slides/896s16-16.pdf, pristup: 20.5.2019.
56) Drozdek, A. (2001.), *Data structures and algorithms in c++,* Second Edition, Brooks/Cole
57) Geeksforgeeks_1, (n_d), *Graph Data Structure And Algorithms*, A computer science portal for geeks, dostupno na: https://www.geeksforgeeks.org/graph-data-structure-and-algorithms/, pristup: 2.5.2019.
58) Geeksforgeeks_2, (n_d), *Graph and its representations*, A computer science portal for geeks, dostupno na: https://www.geeksforgeeks.org/graph-and-its-representations/, pristup: 10.4.2019.
59) Geeksforgeeks_3, (n_d), *Biconnected graph*, A computer science portal for geeks, dostupno na: https://www.geeksforgeeks.org/biconnectivity-in-a-graph/, pristup: 10.4.2019.
60) Geeksforgeeks_4, (n_d), *Mathematics | Euler and Hamiltonian Paths*, A computer science portal for geeks, dostupno na:

https://www.geeksforgeeks.org/mathematics-euler-hamiltonian-paths/, pristup: 9.4.2019.

61) Geeksforgeeks_5, (n_d), *Strongly Connected Components*, A computer science portal for geeks, dostupno na: https://www.geeksforgeeks.org/strongly-connected-components/, pristup: 15.4.2019.

62) Geeksforgeeks_6, (n_d), *Tarjan's Algorithm to find Strongly Connected Components*, A computer science portal for geeks, dostupno na: https://www.geeksforgeeks.org/tarjan-algorithm-find-strongly-connected-components/, pristup: 15.4.2019.

63) Geeksforgeeks_7, (n_d), *Bellman–Ford Algorithm*, A computer science portal for geeks, dostupno na: https://www.geeksforgeeks.org/bellman-ford-algorithm-dp-23/, pristup: 4.5.2019.

64) Geeksforgeeks_8, (n_d), *Detect Cycle in a Directed Graph*, A computer science portal for geeks, dostupno na: https://www.geeksforgeeks.org/detect-cycle-in-a-graph/, pristup: 11.5.2019.

65) Geeksforgeeks_9, (n_d), *Detect cycle in an undirected graph*, A computer science portal for geeks, dostupno na: https://www.geeksforgeeks.org/detect-cycle-undirected-graph/, pristup:11.5.2019.

66) Geeksforgeeks_10, (n_d), *Disjoint Set (Or Union-Find)* | *Set 1 (Detect Cycle in an Undirected Graph)*, A computer science portal for geeks, dostupno na: https://www.geeksforgeeks.org/union-find/, pristup: 11.5.2019.

67) Geeksforgeeks_11, (n_d), *Max Flow Problem Introduction*, A computer science portal for geeks, dostupno na: https://www.geeksforgeeks.org/max-flow-problem-introduction/, pristup: 11.5.2019.

68) Geeksforgeeks_12, (n_d), *Ford-Fulkerson Algorithm for Maximum Flow Problem*, A computer science portal for geeks, dostupno na: https://www.geeksforgeeks.org/ford-fulkerson-algorithm-for-maximum-flow-problem/, pristup: 17.4.2019.

69) Geeksforgeeks_13, (n_d), *Maximum Bipartite Matching*, A computer science portal for geeks, dostupno na: https://www.geeksforgeeks.org/maximum-bipartite-matching/, pristup: 11.5.2019.

70) Geeksforgeeks_14, (n_d), *Push Relabel Algorithm* | *Set 1 (Introduction and Illustration)*, A computer science portal for geeks, dostupno na:

https://www.geeksforgeeks.org/push-relabel-algorithm-set-1-introduction-and-illustration/, pristup: 11.5.2019.

71) Geeksforgeeks_15, (n_d), *Relabel-to-front Algorithm*, A computer science portal for geeks, dostupno na: https://www.geeksforgeeks.org/relabel-to-front-algorithm/, pristup: 12.5.2019.

72) Geeksforgeeks_16, (n_d), *Mathematics | Matching (graph theory)*, A computer science portal for geeks, dostupno na: https://www.geeksforgeeks.org/mathematics-matching-graph-theory/, pristup: 17.5.2019.

73) Geeksforgeeks_17, (n_d), *Prim's algorithm using priority_queue in STL*, A computer science portal for geeks, dostupno na: https://www.geeksforgeeks.org/prims-algorithm-using-priority_queue-stl/, pristup: 4.8.2019.

74) Jaimini, V. (n_d), Biconnected Components, dostupno na: https://www.hackerearth.com/practice/algorithms/graphs/biconnected-components/tutorial/, pristup: 2.5.2019.

75) Jurašić, A. (n_d), *Obilasci grafova*, Uniri, dostupno na: http://www.math.uniri.hr/~ajurasic/D-OSMO.pdf, pristup: 2.7.2019.

76) Marinović, M. (n_d), *Mreže računala- Teorija grafova*, Sveučilište u Zagrebu, Fakultet Elektrotehnike i računarstva, dostupno na: http://www.zemris.fer.hr/predmeti/mr/arhiva/2002-2003/seminari/finished/pdf/grafovi.pdf

77) Moore, K. i Landman, N. (2019.), *Matching (Graph Theory)*, dostupno na: https://brilliant.org/wiki/matching/, pristup: 17.5.2019.

78) Nikšić, F. (2004.), *Algoritmi u teoriji grafova*, Zagreb, dostupno na: http://web.studenti.math.pmf.unizg.hr/~fniksic/files/maturalni.pdf

79) Samual, S. (2018.), *Biconnected Graph*, dostupno na: https://www.tutorialspoint.com/Biconnected-Graph, pristup: 2.5.2019.

80) Sedgewick, R. i Wayne, K. (n_d), *4.3 Minimum spanning tree*, Fourth Edition, Princeton, dostupno na: https://algs4.cs.princeton.edu/lectures/43MinimumSpanningTrees.pdf, pristup: 16.4.2019.

81) Singhal, A. (2018.), *Euler Graph | Euler Trail | Euler Circuit*, Gate Vidyalay, dostupno na: https://www.gatevidyalay.com/euler-graph-euler-trail-euler-circuit/, pristup: 9.4.2019.
82) Šego, V. (2009.), *P=NP?*, dostupno na: https://bib.irb.hr/datoteka/465090.mfl-p_np.proc.pdf, pristup: 16.4.2019.
83) Tardos, E. i Kleinberg, J. (2005.), *Algorithm design- Representative problems*, Princeton, Pearson-Addison Wesley, dostupno na: https://www.cs.princeton.edu/courses/archive/spring13/cos423/lectures/01StableMatching-2x2.pdf, pristup: 23.5.2019.
84) Weiss, M., A. (2014.), *Data Structures and Algorithm Analysis in C++*, Fourth Edition, Florida International University
85) Wolfman, S., (2000.), *Lecture 24: From Dijkstra to Prim*, Washington, dostupno na: https://courses.cs.washington.edu/courses/cse373/01sp/Lect24_2up.pdf, pristup: 3.5.2019.

Popis slika