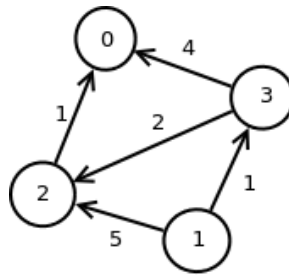


Težinski grafovi

U problemima koji slede bavićemo se *težinskim grafovima* (eng. weighted graph), odnosno grafovima u kojima je svakoj grani pridružena njena *težina* (eng. weight). Umesto težina grane često ćemo koristiti i termin dužina grane i pod terminom *dužina puta* razmatraćemo zbir dužina grana na tom putu (a ne broj grana na tom putu). Na slici 1 prikazan je jedan usmereni težinski graf. U ovom grafu dužina puta (1, 2, 0) iznosi $5 + 1 = 6$, dok dužina puta (1, 3, 2, 0) iznosi $1 + 2 + 1 = 4$.



Slika 1: Usmereni težinski graf.

U programskom jeziku C++ težinski graf možemo predstaviti ili matricom povezanosti u kojoj se umesto logičkih vrednosti čuvaju težine grana (uz neku specijalnu numeričku vrednost koja označava da čvorovi nisu povezani) ili listama povezanosti gde se u svakom elementu liste povezanosti čuva indeks krajnjeg čvora grane i težina grane. Ako težinski graf predstavljamo listama povezanosti i ako su dužine grana celobrojne, graf možemo deklarirati na sledeći način:

```
vector<vector<pair<int,int>>> listaSuseda(n);
```

Novu granu (*cvorOd*, *cvorDo*) težine *t* dodajemo u graf na sledeći način:

```
listaSuseda[cvorOd].emplace_back(cvorDo, t);
```

Podsetimo se da se primenom funkcije `emplace_back` najpre konstruiše uređeni par čvora i njegove težine, a zatim taj uređeni par dodaje na kraj vektora.

Usmereni težinski graf sa slike 1 zadajemo listama povezanosti na sledeći način:

```
vector<vector<pair<int,int>>> listaSuseda  
    {{}, {{2,5}, {3,1}}, {{0,1}}, {{0,4}, {2,2}}};
```

Ako je graf neusmeren, možemo ga smatrati usmerenim, pri čemu svakoj njegovoj neusmerenoj grani odgovaraju dve usmerene grane iste dužine, u oba smera. Algoritmi koje ćemo razmatrati nad težinskim grafovima odnose se i na usmerene i na neusmerene grafove.

Najkraći putevi iz zadanog čvora

Problem: Za dati usmereni graf $G = (V, E)$ i jedan njegov čvor v pronaći najkraće puteve od čvora v do svih ostalih čvorova u grafu G .

Na primer, zadatak može biti da se u grafu sa slike 1 odrede najkraći putevi od čvora 1 do svih ostalih čvorova u grafu. Najkraći put do čvora 3 bi predstavljala direktna grana (1, 3) dužine 1, dok bi najkraći put do čvora 2 bio put (1, 3, 2) ukupne dužine $1 + 2 = 3$, a ne direktna grana (1, 2) dužine 5.

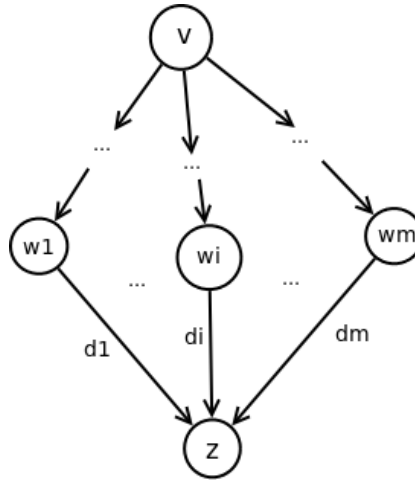
Postoji mnogo situacija u kojima se javlja potreba da se izračunaju najkraći putevi kroz graf. Na primer, graf može odgovarati auto-karti: čvorovi su gradovi, a dužine grana su dužine direktnih puteva između gradova (ili vreme potrebno da se taj put pređe, ili izgradi, itd, zavisno od problema). Zadatak je pronaći najkraći put (u smislu dužine ili proteklog vremena) od jednog grada do drugog.

Aciklički slučaj

Pretpostavimo najpre da je graf G aciklički. U tom slučaju problem je lakši i njegovo rešenje pomoći će nam da problem rešimo i u opštem slučaju.

Pokušajmo da problem rešimo indukcijom po broju čvorova. Bazni slučaj odgovara grafu sa jednim čvorom i on ima trivijalno rešenje: najkraći put od tog čvora do njega samog je jednak 0. Neka je $|V| = n$. Pošto je graf aciklički, možemo da iskoristimo topološko sortiranje grafa. Najpre, ako je redni broj čvora v od koga treba odrediti najkraće puteve u topološkom sortiranju jednak k , onda se čvorovi sa rednim brojevima manjim od k ne moraju ni razmatrati jer ne postoji način da se iz čvora v do njih dođe. Dodatno, redosled čvorova dobijen topološkim sortiranjem je pogodan za primenu indukcije. Posmatrajmo poslednji čvor u topološkom redosledu čvorova, odnosno čvor z sa rednim brojem n . Pretpostavimo (induktivna hipoteza) da znamo najkraće puteve od čvora v do svih ostalih čvorova, sem do z . Označimo dužinu najkraćeg puta od čvora v do čvora w sa $w.SP$ (eng. shortest path), a dužinu grane (x, y) sa $d(x, y)$. Da bismo odredili vrednost $z.SP$, dovoljno je da proverimo samo one čvorove w iz kojih postoji grana do čvora z (slika 2). Pošto se najkraći putevi do ostalih čvorova već znaju, vrednost $z.SP$ biće jednaka minimumu zbira $w.SP + d(w, z)$, po svim čvorovima w iz kojih vodi grana do čvora z . Da li je time problem rešen? Važno pitanje je da li dodavanje čvora z u skup čvorova do kojih smo odredili najkraće puteve može da skрати najkraći put do nekog drugog čvora. Međutim, pošto je z poslednji čvor u topološkom redosledu, ni jedan drugi čvor nije dostižan iz z , pa se dužine ostalih najkraćih puteva ne menjaju. Dakle, uklanjanje čvora z iz grafa, pronalaženje najkraćih puteva u preostalom grafu i vraćanje čvora z nazad u graf su osnovni delovi algoritma. Drugim rečima, sledeća induktivna hipoteza rešava problem.

Induktivna hipoteza: Ako se zna topološki redosled čvorova, umemo da izračunamo dužine najkraćih puteva od čvora v do prvih $n - 1$ čvorova u ovom redosledu.



Slika 2: Računanje najkraćeg puta od čvora v do poslednjeg čvora z u topološkom poretku.

Razmotrimo sada dati aciklički graf koji ima n čvorova (topološki uređenih). Iz njega uklanjamo čvor z koji je poslednji u topološkom redosledu i indukcijom rešavamo smanjeni problem. Nakon toga nalazimo najmanju među vrednostima $w_i.SP + d(w, z)$ za sve čvorove w_i takve da $(w_i, z) \in E$ i nju proglašavamo za $z.SP$:

$$z.SP = \min_{i:(w_i,z) \in E} \{w_i.SP + d(w_i, z)\}.$$

Iz ovog razmatranja direktno sledi odgovarajući rekurzivni algoritam.

Pokušajmo sada da unapredimo algoritam za traženje najkraćih puteva u acikličkom grafu, tako da se Kanov algoritam za topološko sortiranje obavlja istovremeno sa pronalaženjem najkraćih puteva. Drugim rečima, cilj je objediniti dva prolaza kroz graf (jedan za topološko sortiranje i drugi za nalaženje najkraćih puteva) u jedan.

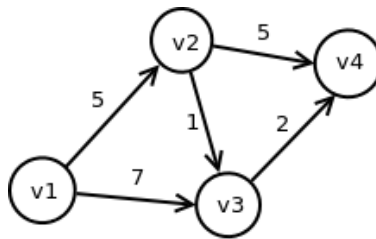
Razmotrimo način na koji se algoritam rekurzivno izvršava (posle nalaženja topološkog redosleda). Pretpostavimo, zbog jednostavnosti, da je redni broj čvora v od koga treba izračunati najkraće puteve u topološkom redosledu 1 (čvorovi sa rednim brojevima manjim od rednog broja čvora v ionako nisu dostižni iz v). Prvi korak je poziv rekurzivne procedure. Procedura zatim poziva rekurzivno samu sebe, sve dok se ne dođe do čvora v . U tom trenutku se dužina najkraćeg puta od čvora v do čvora v postavlja na 0, i rekurzija počinje da se "razmotava". Zatim se razmatra čvor u sa rednim brojem 2 u topološkom poretku; dužina najkraćeg puta do njega izjednačuje se sa dužinom grane (v, u) , ako ona postoji; u protivnom, ne postoji put od v do u . Sledeći korak je provera čvora x sa rednim brojem 3. U čvor x ulaze najviše dve grane – od čvorova v i/ili u , pa se upoređuju dužine odgovarajućih puteva. Umesto ovakvog izvršavanja rekurzije

unazad, pokušaćemo da iste korake izvršimo preko niza čvorova sa rastućim rednim brojevima u topološkom redosledu.

Indukcija se primenjuje prema rastućim rednim brojevima u topološkom redosledu počevši od v . Ovaj redosled oslobađa nas potrebe da redne brojeve unapred znamo, pa ćemo biti u stanju da izvršavamo istovremeno i algoritam za određivanje topološko uređenja i algoritam za određivanje najkraćih puteva od čvora v . Dakle, možemo razmotriti narednu induktivnu hipotezu.

Induktivna hipoteza: Ako znamo prvih m čvorova u topološkom redosledu čvorova, umemo da izračunamo dužine najkraćih puteva od čvora v do čvorova sa rednim brojevima od 1 do m .

Razmotrimo čvor sa rednim brojem $m + 1$, koji ćemo označiti sa z . Da bismo pronašli najkraći put do z , moramo da proverimo sve grane koje vode u z . Topološki redosled garantuje da sve takve grane polaze iz čvorova sa manjim rednim brojevima. Prema induktivnoj hipotezi ti čvorovi su već razmatrani, pa se dužine najkraćih puteva do njih znaju. Za svaku granu (w, z) znamo dužinu $w.SP$ najkraćeg puta od v do w , pa je dužina najkraćeg puta od v do z preko w jednaka $w.SP + d(w, z)$. Pored toga, kao i ranije, ne moramo da vodimo računa o eventualnim promenama najkraćih puteva ka čvorovima sa manjim rednim brojevima od rednog broja čvora z , jer se do njih ne može doći iz čvora z . Da ne bismo za svaki čvor određivali iz kojih čvorova postoji grana ka njemu jer to nije efikasna operacija u reprezentaciji grafa listama povezanosti, možemo pamtiti dužine trenutno poznatih najkraćih puteva do svih čvorova sa većim rednim brojem od tekućeg čvora. Prilikom razmatranja čvora z , kao čvora sa rednim brojem $m + 1$, potrebno je jedino razmotriti grane (z, x) koje polaze iz njega i za čvor x proveriti da li je vrednost $z.SP + d(z, x)$ manja od $x.SP$ i ako jeste ažurirati vrednost $x.SP$. Dakle umesto da grane grafa obrađujemo “unazad”, odnosno u trenutku kada obrađujemo njihove krajnje čvorove, mi ih obrađujemo “unapred”, u trenutku kada obrađujemo njihove polazne čvorove.



Slika 3: Računanje najkraćeg puta u acikličkom grafu: čvorovi su označeni redom koji odgovara njihovoj poziciji u topološkom poretku.

Primer: Razmotrimo izvršavanje algoritma na primeru acikličkog grafa sa slike 3. Redosled čvorova u topološkom sortiranju grafa je v_1, v_2, v_3, v_4 . Neka je zadatak odrediti najkraće puteve iz čvora v_1 . Na početku postavljamo vrednost najkraćeg puta do čvora v_1 na 0 (i to proglašavamo konačnom vrednošću najkraćeg puta

do čvora v_1), a vrednosti najkraćih puteva do svih ostalih čvorova na $+\infty$. Razmatramo grane koje polaze iz čvora v_1 : to su (v_1, v_2) i (v_1, v_3) i ažuriramo najkraće puteve do čvora v_2 i do čvora v_3 na $v_2.SP = \min\{+\infty, 0 + 5\} = 5$, a $v_3.SP = \min\{+\infty, 0 + 7 = 7\}$. Drugi čvor u topološkom redosledu je čvor v_2 : tekuću vrednost najkraćeg puta do njega proglašavamo za konačnu vrednost najkraćeg puta, odnosno proglašavamo $v_2.SP = 5$. Zatim razmatramo grane koje izlaze iz čvora v_2 : to su grane (v_2, v_3) i (v_2, v_4) i ažuriramo najkraće puteve do čvorova v_3 i v_4 : $v_3.SP = \min\{7, 5 + 1\} = 6$ i $v_4.SP = \min\{+\infty, 5 + 5\} = 10$. Treći po redu čvor u topološkom poretku je čvor v_3 te tekuću vrednost najkraćeg puta do njega proglašavamo za konačnu. Razmatramo jedinu granu (v_3, v_4) koja polazi iz čvora v_3 i vršimo ažuriranje vrednosti najkraćeg puta do čvora v_4 : $v_4.SP = \min\{10, 6 + 2\} = 8$. U ovom trenutku proglašavamo kao konačnu i dužinu najkraćeg puta do čvora v_4 čime se izvršavanje algoritma završava.

Do sada smo diskutovali izračunavanje dužina najkraćih puteva, a ne i računanje samih najkraćih puteva. Da bismo mogli da rekonstruišemo najkraće puteve do svakog čvora, za svaki čvor pamtimo njegovog prethodnika (roditelja) na najkraćem putu od čvora v . Jedino čvor v neće imati roditelja na najkraćem putu. Na taj način možemo jednostavno da rekonstruišemo same najkraće puteve.

Razmotrimo implementaciju algoritma: pretpostavićemo da je potrebno odrediti najkraće puteve od čvora 0 do svih ostalih čvorova.

```
vector<vector<pair<int,int>>> listaSuseda {{{1,3}, {2,2}}, {{3,1},
    {4,2}}, {{5,3}}, {}, {{6,1}, {7,3}}, {{8,4}}, {}, {}, {}};

// funkcija koja stampa put od polaznog cvora v do datog cvora
// kroz grane drveta najkracih puteva
void odstampajPutDoCvora(int cvor, vector<int> roditelj){
    // ako smo stigli unazad do polaznog cvora, završavamo
    if (roditelj[cvor] == -1)
        return;
    // inace, stampamo deo puta od roditeljskog cvora do kraja
    odstampajPutDoCvora(roditelj[cvor], roditelj);
    // na kraju stampamo tekuci cvor
    cout << " " << cvor;
}

// funkcija koja stampa najkraci put do datog cvora i njegovu duzinu
void odstampajNajkraciPut(int cvor, vector<int> roditelj,
    vector<int> najkraciPut){
    cout << "Najkraci put do cvora " << cvor << " je: 0";
    odstampajPutDoCvora(cvor, roditelj);
    cout << " i duzine je " << najkraciPut[cvor] << endl;
}

// funkcija koja racuna najkrace puteve od cvora 0 u aciklickom grafu
```

```

void aciklicki_najkraci_putevi(){

    int brojCvorova = listaSuseda.size();
    // niz koji za svaki cvor cuva njegov ulazni stepen
    vector<int> ulazniStepen(brojCvorova,0);
    // niz koji za svaki cvor cuva duzinu
    // trenutno poznatog najkraceg puta do njega
    vector<int> najkraciPut(brojCvorova,numeric_limits<int>::max());
    // niz koji za svaki cvor cuva roditelja u najkracem putu
    vector<int> roditelj(brojCvorova,-1);

    // najkraci put od cvora 0 do njega samog postavljamo na 0
    najkraciPut[0] = 0;

    // inicijalizujemo niz ulaznih stepena cvorova
    // potreban za izvršavanje Kanovog algoritma
    for (int i = 0; i < listaSuseda.size(); i++)
        for (int j = 0; j < listaSuseda[i].size(); j++)
            ulazniStepen[listaSuseda[i][j].first]++;

    // red koji cuva cvorove ulaznog stepena nula
    queue<int> cvoroviStepenaNula;

    // cvorove koji su ulaznog stepena 0 dodajemo u red
    for (int i = 0; i < brojCvorova; i++)
        if (ulazniStepen[i] == 0)
            cvoroviStepenaNula.push(i);

    while(!cvoroviStepenaNula.empty()){

        // cvor sa pocetka reda je naredni u topoloskom redosledu
        int cvor = cvoroviStepenaNula.front();
        cvoroviStepenaNula.pop();
        // do njega je odredjen najkraci put i stampamo ga
        odstampajNajkraciPut(cvor,roditelj,najkraciPut);

        for (int i = 0; i < listaSuseda[cvor].size(); i++){
            // ukoliko je do nekog cvora kraci put preko upravo razmatranog cvora
            // ursimo azuriranje najkraceg rastojanja do tog cvora
            int sused = listaSuseda[cvor][i].first;
            int grana = listaSuseda[cvor][i].second;
            if (najkraciPut[cvor] + grana < najkraciPut[sused]){
                najkraciPut[sused] = najkraciPut[cvor] + grana;
                // azuriramo koji je roditeljski cvor na najkracem putu
                roditelj[sused] = cvor;
            }
        }
    }
}

```

```

        ulazniStepen[sused]--;
        // ukoliko je stepen nekog od suseda pao na 0, dodajemo ga u red
        if (ulazniStepen[sused] == 0)
            cvoroviStepenaNula.push(sused);
    }
}
}

int main(){
    aciklicki_najkraci_putevi();
    return 0;
}

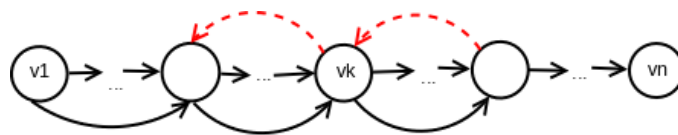
```

U algoritmu se svaka grana razmatra po jednom u toku inicijalizacije ulaznih stepena čvorova, i po jednom u trenutku kad se njen polazni čvor uklanja iz reda. Pristup redu zahteva konstantno vreme. Svaki čvor se razmatra tačno jednom. Prema tome, vremenska složenost algoritma za određivanje najkraćih puteva u acikličkom grafu je u najgorem slučaju $O(|V| + |E|)$.

Dajkstrin algoritam

Kad graf nije aciklički, ne postoji topološki redosled, pa se razmatrani algoritam ne može direktno primeniti. Međutim, osnovne ideje se mogu iskoristiti i u opštem slučaju, kada su dužine grana pozitivne. Naime, jednostavnost algoritma za određivanje najkraćih puteva iz zadanog čvora u acikličkom grafu posledica je sledeće osobine topološkog redosleda: ako je z čvor sa rednim brojem k u topološkom poretku, onda:

- ne postoje putevi od čvora z do čvorova sa rednim brojevima manjim od k , i
- ne postoje putevi od čvorova sa rednim brojevima većim od k do čvora z (slika 4).

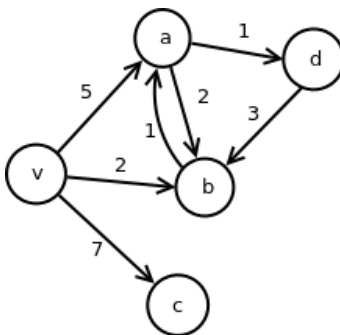


Slika 4: Kada čvorove poredamo slevo nadesno u redosledu topološkog poretka, u grafu mogu postojati samo grane sleva nadesno, koje vode od čvorova numerisanih manjih brojem ka čvorovima označenim većim brojem (označene crnom bojom), dok grane koje vode zdesna ulevo, od čvorova numerisanih većom vrednošću ka čvorovima označenim manjom vrednošću, ne mogu postojati (označene crvenom bojom).

Ova osobina topološkog redosleda omogućuje nam da nađemo najkraći put od

čvora v do čvora z , ne vodeći računa o čvorovima koji su posle z u topološkom redosledu. Može li se nekako definisati redosled čvorova proizvoljnog grafa (koji nije nužno aciklički) koji bi omogućio nešto slično?

Ideja je razmatrati čvorove grafa redom prema dužinama najkraćih puteva do njih od čvora v . Te dužine se na početku, naravno, ne znaju, već se izračunavaju u toku izvršavanja algoritma. Najpre proveravamo sve grane koje izlaze iz čvora v . Neka je (v, x) najkraća među njima. Pošto su, po pretpostavci, sve dužine grana pozitivne, najkraći put od čvora v do čvora x je grana (v, x) . Dužine svih drugih puteva od čvora v do čvora x su veće ili jednake od dužine ove grane. Čvor x je, dodatno, najbliži od svih čvorova čvoru v . Prema tome, znamo najkraći put do čvora najbližeg čvoru v , i to može da posluži kao baza indukcije. Pokušajmo da napravimo sledeći korak. Kako možemo da pronađemo najkraći put do nekog drugog čvora? Biramo čvor koji je drugi najbliži čvoru v (x je prvi najbliži). Jedini putevi koje treba uzeti u obzir su druge grane iz čvora v ili putevi koji se sastoje od dve grane, tako da je prva grana (v, x) , a druga je neka grana koja polazi iz čvora x . Neka je sa $d(u, w)$ označena dužina grane (u, w) . Biramo najmanju od vrednosti izraza $d(v, y)$ ($y \neq x$) ili $d(v, x) + d(x, z)$ ($z \neq v$). Još jednom zaključujemo da se drugi putevi ne moraju razmatrati, jer je ovo najkraći put za odlazak iz v (izuzev do x).



Slika 5: Usmereni težinski graf koji sadrži cikle.

Primer: Razmotrimo graf sa slike 5 i problem traženja najkraćih puteva od čvora v . Prvi najbliži čvor čvoru v je jedan od suseda čvora v i to onaj do kog vodi najkraća grana, a to je čvor b . Drugi najbliži čvor čvoru v je ili neki drugi sused čvora v ili neki čvor do kog vodi grana iz čvora b (kao prvog najbližeg čvora čvoru v) – to će biti čvor a do kog vodi put (v, b, a) dužine $2 + 1 = 3$. Treći najbliži čvor čvoru v biće čvor d do kog vodi put (v, b, a, d) dužine $2 + 1 + 1 = 4$, a četvrti najbliži čvor biće čvor c do kog vodi direktna grana iz čvora v .

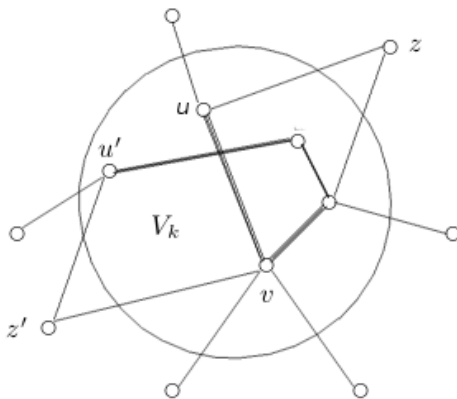
U opštem slučaju može se formulisati sledeća induktivna hipoteza.

Induktivna hipoteza: Za zadati graf $G = (V, E)$ i njegov čvor v , umemo da pronađemo k čvorova najbližih čvoru v , kao i dužine najkraćih puteva do njih.

Zapazimo da se indukcija primenjuje broju čvorova do kojih su dužine najkraćih

puteva već izračunate, a ne po veličini grafa. Pored toga, pretpostavlja se da su to čvorovi najbliži čvoru v , i da umemo da ih pronađemo. Mi umemo da pronađemo prvi najbliži čvor, pa je baza (slučaj $k = 1$) rešena. Kad k dostigne vrednost $|V| - 1$, rešen je kompletan problem.

Označimo sa V_k skup koji se sastoji od k najbližih čvorova čvoru v , uključujući i v . Problem je pronaći čvor w koji je najbliži čvoru v među čvorovima van V_k , i pronaći najkraći put od v do w . Najkraći put od v do w može da sadrži samo čvorove iz V_k . On ne može da sadrži neki čvor y van V_k , jer bi u tom slučaju čvor y bio bliži čvoru v od čvora w (pošto su težine svih grana pozitivne). Prema tome, da bismo pronašli čvor w , dovoljno je da proverimo grane koje spajaju čvorove iz V_k sa čvorovima koji nisu u V_k ; sve druge grane se za sada mogu ignorisati. Neka je (u, z) proizvoljna grana grafa G takva da je $u \in V_k$ i $z \notin V_k$. Takva grana određuje put od v do z koji se sastoji od najkraćeg puta od v do u (koji je prema induktivnoj hipotezi već poznat) i grane (u, z) . Dovoljno je uporediti dužine svih takvih puteva i izabrati najkraći među njima (slika 6).



Slika 6: Nalaženje sledećeg najbližeg čvora zatom čvoru v .

Algoritam određen ovom induktivnom hipotezom izvršava se na sledeći način. U svakoj iteraciji u skup V_k se dodaje novi čvor. To je čvor $w \notin V_k$ za koji je najmanja vrednost izraza:

$$\min \{u.SP + d(u, w) \mid u \in V_k\} \quad (1)$$

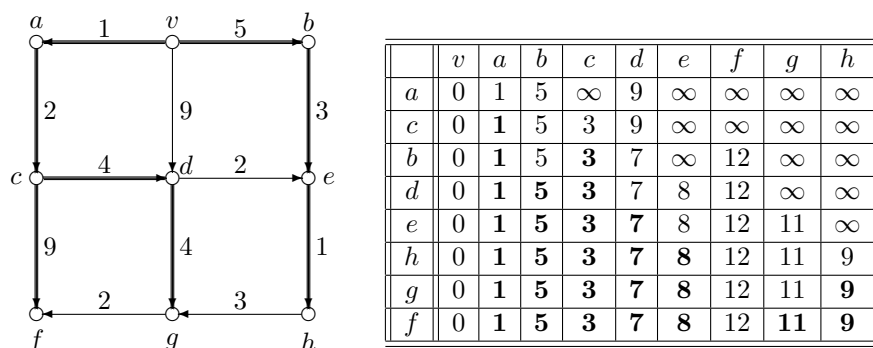
među svim čvorovima $w \notin V_k$. Iz već iznetih razloga, w je zaista $(k + 1)$ -vi (sledeći) najbliži čvor čvoru v . Prema tome, njegovo dodavanje skupu V_k proširuje induktivnu hipotezu.

Algoritam je sada u potpunosti preciziran, ali mu se efikasnost može poboljšati. Osnovni korak algoritma je pronalaženje sledećeg najbližeg čvora. To se ostvaruje izračunavanjem dužine najkraćeg puta prema formuli (1). Međutim, nije neophodno u svakom koraku proveravati sve vrednosti $u.SP + d(u, w)$. Većina

tih vrednosti se ne menja pri dodavanju novog čvora u skup V_k : mogu se promeniti samo one vrednosti koje odgovaraju putevima kroz novododati čvor. Mi možemo da pamtimo dužine trenutno poznatih najkraćih puteva do svih čvorova van V_k , i da im pri proširivanju skupa V_k popravljamo vrednosti. Jedini način da se dobije novi najkraći put nakon dodavanja čvora w u V_k je da taj put prolazi kroz novododati čvor w . Prema tome, treba proveriti sve grane od čvora w ka čvorovima van V_k . Za svaku takvu granu (w, z) upoređujemo dužinu $w.SP + d(w, z)$ sa trenutnom vrednošću $z.SP$, i po potrebi popravljamo vrednost $z.SP$. Primetimo da smo na ovaj način određivali i najkraće puteve u acikličkom grafu. Svaka iteracija algoritma obuhvata nalaženje čvora sa najmanjom vrednošću SP , i popravku vrednosti SP za neke od preostalih čvorova. Ovaj algoritam dobio je naziv *Dajkstrin algoritam* po Edzgaru Dajkstri koji ga je osmislio 1956. godine. On pripada grupi pohlepnih algoritama jer se u svakom koraku bira lokalno optimalno rešenje – najbliži čvor i nakon obrade trenutno najbližeg čvora rastojanje do njega se više nikada ne razmatra.

Najkraće puteve od čvora v do svih ostalih čvorova našli smo tako što smo pronalazili jedan po jedan najkraći put. Svaki novi put je određen jednom granom, koja produžuje prethodno poznati najkraći put do novog čvora. Sve te grane formiraju drvo sa korenom u čvoru v . Ovo drvo naziva se *drvo najkraćih puteva* (eng. shortest-path tree), i važno je za rešavanje mnogih problema sa putevima. Ako bi dužine svih grana bile međusobno jednake, onda bi drvo najkraćih puteva u stvari bilo BFS drvo sa korenom u čvoru v . U primeru na slici 7 podebljane su grane koje pripadaju drvetu najkraćih puteva sa korenom u čvoru v . Primetimo da, recimo, najkraći put do čvora h dobijamo tako što granom (e, h) produžimo prethodno pronađeni najkraći put do čvora e : put (v, b, e) .

Primer: Ilustracija izvršavanja Dajkstrinog algoritma za nalaženje najkraćih puteva od čvora v prikazana je na slici 7. Prva vrsta tabele odnosi se samo na puteve koji se sastoje od jedne grane koja polazi iz čvora v . Bira se najkraći od tih puteva (grana) i u ovom slučaju on vodi ka čvoru a . Druga vrsta tabele pokazuje popravke dužina najkraćih puteva uključujući sada sve puteve koji se sastoje od jedne grane koja polazi iz čvora v ili od dve grane preko čvora a , i najkraći put u ovom slučaju vodi do čvora c . U svakoj vrsti tabele bira se novi, naredni najbliži čvor, i prikazuju se dužine trenutnih najkraćih puteva od v do svih čvorova. Podebljana su rastojanja za koja se pouzdano zna da su najkraća. Na osnovu tabele moguće je rekonstruisati i same najkraće puteve. Na primer, ako želimo da saznamo koji je to najkraći put od čvora v do čvora h dužine 9, razmatramo kolonu koja odgovara čvoru h i tražimo poslednju promenu vrednosti u toj koloni – ona odgovara vrsti označenoj čvorom e i to znači da je čvor e roditeljski čvor čvora h . Zatim, na osnovu tabele određujemo roditeljski čvor čvora e , gledajući kolonu koja odgovara čvoru e i utvrđivanjem na kom čvoru se desila poslednja promena vrednosti u ovoj tabeli – to je čvor b te je on roditeljski čvor čvora e , dok je roditeljski čvor čvora b polazni čvor v . Konačno, najkraći put rekonstruišemo čitajući dobijeni niz čvorova unazad i kao najkraći put od čvora v do čvora h dobijamo put v, b, e, h .



Slika 7: Primer izvršavanja Dajkstrinog algoritma.

U Dajkstrinom algoritmu potrebno je da $O(|V|)$ puta pronalazimo najmanju vrednost u skupu dužina puteva i da često popravljamo dužine puteva. Struktura podataka pogodna za efikasno nalaženje minimalnih elemenata i za popravke dužina elemenata je red sa prioritonom, odnosno kao njegova realizacija min-hip. Pošto je potrebno da pronađemo čvor do koga je dužina puta najmanja, sve čvorove van skupa V_k čuvamo u hipu, sa ključevima jednakim dužinama trenutno najkraćih puteva od čvora v do njih. Na početku su sve dužine puteva osim one do čvora v jednake $+\infty$, pa redosled elemenata u hipu nije bitan, sem što čvor v mora biti na vrhu. Nalaženje čvora w koji je među čvorovima van V_k najbliži čvoru v je jednostavno: on se uzima sa vrha hipa. Posle toga se za svaku granu (w, u) proverava da li najkraći put do čvora w produžen granom (w, u) skraćuje trenutno poznati najkraći put do čvora u . Međutim, kad se promeni dužina puta do nekog čvora u , može se promeniti i položaj čvora u u hipu. Prema tome, potrebno je na odgovarajući način popravljati hip. S obzirom da se putevi do čvora mogu samo skratiti, to znači da se vrednost ključa u hipu može samo smanjiti i eventualno pri tom postati manja od vrednosti ključa svog roditelja (pošto je prethodna vrednost elementa bila manja od vrednosti njegove dece u hipu, to će važiti i za smanjenu vrednost ključa). Operacija smanjivanja vrednosti ključa u min-hipu nije direktno podržana u standardnoj biblioteci, ali se odgovarajuća popravka hipa može izvesti razmenom vrednosti elementa i njegovog roditelja, sve dok uslov hipa ne bude zadovoljen. Ova operacija se dakle može izvesti u složenosti $O(\log n)$, gde je n broj elemenata u hipu. Međutim, problem sa ovim popravkama je u tome što hip kao struktura podataka ne podržava efikasno pronalaženje zadatog elementa. Naime, pretraga elementa u hipu izvršava se u linearnoj vremenskoj složenosti u odnosu na broj elemenata u hipu. Iz tog razloga umesto da se vrednost rastojanja do nekog čvora u hipu zameni novom (manjom) vrednošću, vrši se umetanje novog čvora sa istom oznakom čvora i novom (manjom) vrednošću rastojanja (ova tehnika se naziva *tehnikom lenjog brisanja*). S obzirom na to da je nova vrednost rastojanja manja od stare, novi čvor će sigurno biti skinut iz hipa pre starog, te je jedino potrebno prilikom uzimanja elementa sa vrha hipa proveriti da li taj čvor nije već ranije

bio obrađen.

Ostaje bojazan da ovakva implementacija može da ugrozi vreme izvršavanja operacija. Ukoliko bismo našli način da efikasno pronalazimo elemente u zadatom hipu, u hipu bismo čuvali u svakom trenutku $O(|V|)$ elemenata. Međutim, u implementaciji koja čuva kopije čvorova prilikom obrade svake (usmerene) grane može se dodati maksimalno jedan novi element u hip. Dakle ukupan broj čvorova biće sigurno manji ili jednak od $O(|E|)$, te će svaka od operacija umetanja elementa u hip i brisanja minimalnog elementa iz hipa biti složenosti $O(\log |E|)$. S obzirom na to da važi da je $|E| \leq |V|^2$, i stoga $\log |E| \leq 2 \cdot \log |V|$, složenosti $O(\log |E|)$ i $O(\log |V|)$ se asimptotski ne razlikuju, te će operacije nad hipom koji čuva kopije čvorova biti iste asimptotske složenosti kao i u slučaju kada nema kopija.

Dajkstrin algoritam za nalaženje najkraćih puteva od čvora sa oznakom 0 prikazan je u nastavku.

```
// uredjen par vrednosti rastojanja do cvora i indeksa cvora;
// vazan je redosled komponenti u uredjenom paru
// zbog operacije poredjenja po rastojanju
typedef pair<int,int> rastojanjeDoCvora;

// tezijski graf predstavljen listom povezanosti
vector<vector<pair<int,int>>> listaSuseda {{{1,3}, {2,1}, {3,2}},
                                         {{3,4}, {4,3}}, {{5,3}}, {}, {{6,1}, {7,3}},
                                         {{0,2}}, {}, {{1,1}}};

// Dajkstrin algoritam za odredjivanje najkracih puteva
// iz cvora sa indeksom 0 do svih cvorova u grafu
void najkraciPuteviDajkstra(){

    int brojCvorova = listaSuseda.size();
    // niz koji cuva informaciju o tome da li je cvor posecen
    vector<bool> posecen(brojCvorova,false);
    // niz koji za svaki cvor cuva duzinu trenutno poznatog
    // najkraceg puta do njega
    vector<int> najkraciPut(brojCvorova,numeric_limits<int>::max());
    // niz koji za svaki cvor cuva roditeljski cvor u drvetu najkracih puteva
    vector<int> roditelj(brojCvorova,-1);

    // min-hip u koji smestamo rastojanja do svih cvorova
    priority_queue<rastojanjeDoCvora,vector<rastojanjeDoCvora>,
                  greater<rastojanjeDoCvora>> rastojanja;

    // ubacujemo polazni cvor u hip
    // i postavljamo rastojanje do njega na 0
    rastojanja.push(make_pair(0,0));
```

```

najkraciPut[0] = 0;

// rastojanja do svih ostalih cvorova postavljamo na
// maksimalnu mogucu vrednost i ubacujemo ih u hip
for(int cvor = 1; cvor < brojCvorova; cvor++){
    rastojanja.push(make_pair(numeric_limits<int>::max(),cvor));

// odredjujemo narednih (brojCvorova-1) cvorova i
// rastojanja do njih
for(int i = 0; i < brojCvorova; i++){

    // izdvajamo naredni najblizi cvor
    rastojanjeDoCvora najblizi = rastojanja.top();
    rastojanja.pop();
    int cvor = najblizi.second;

// ako je taj cvor vec posecen, ne treba ga ponovo brojati
// i preskacemo njegovu obradu
if (posecen[cvor]){
    i--;
    continue;
}
// inace pmatimo da smo ga sada posetili
posecen[cvor] = true;

// postavljamo vrednost najkraceg puta do njega
// kroz do sada posecene cvorove
najkraciPut[cvor] = najblizi.first;
// stampamo informaciju o najkracem putu do tog cvora;
// najkraci putevi se ispisuju u redosledu njihovog "otkrivanja"
odstampajNajkraciPut(cvor,roditelj,najkraciPut);

// za sve susede tekuceg cvora
for (int j = 0; j < listaSuseda[cvor].size(); j++){
    // ako do sada nisu bili poseceni
    if (!posecen[listaSuseda[cvor][j].first]){
        int sused = listaSuseda[cvor][j].first;
        int duzinaGrane = listaSuseda[cvor][j].second;
        // ukoliko je put kroz tekuci cvor do suseda kraci od poznatog
        // najkraceg puta, azuriramo vrednost najkraceg puta do suseda
        // i vrednost njegovog roditeljskog cvora
        if (najkraciPut[cvor] + duzinaGrane < najkraciPut[sused]){
            najkraciPut[sused] = najkraciPut[cvor] + duzinaGrane;
            roditelj[sused] = cvor;
            // ubacujemo element u hip; ako je
            // postojala prethodna vrednost, ne brisemo je:

```

```

        // nova vrednost ce se naci u hipu iznad stare
        rastojanja.push(make_pair(najkraciPut[sused], sused));
    }
}
}
}
}

int main(){
    najkraciPuteviDijkstra();
    return 0;
}

```

Složenost: Kao što smo već pomenuli, operacije umetanja i brisanja iz hipa biće složenosti $O(\log |V|)$. Možemo imati najviše $O(|V| + |E|)$ umetanja u hip (u varijanti sa čuvanjem kopija čvorova) i najviše $O(|V| + |E|)$ brisanja iz hipa. Prema tome, vremenska složenost algoritma je $O((|V| + |E|) \log |V|)$. Zapaža se da je Dajkstrin algoritam sporiji nego algoritam koji određuje najkraće puteve od zadanog čvora u acikličkom grafu.¹

Tip pretrage koji se javlja u Dajkstrinom algoritmu se naziva i *pretraga sa prioritetom* — svakom čvoru dodeljuje se prioritet (u ovom slučaju trenutno najmanje poznato rastojanje od polaznog čvora), pa se čvorovi obilaze redosledom koji je određen prioritetom. Kad se završi razmatranje tekućeg čvora, razmatraju se sve njemu susedne grane i pritom se mogu promeniti prioriteti nekih drugih čvorova. Način izvođenja tih promena je detalj po kome se jedna pretraga sa prioritetom razlikuje od druge. Pretraga sa prioritetom je karakteristična za težinske grafove i složenija je od obične pretrage.

Belman-Fordov algoritam

Razmotrimo sada problem određivanja najkraćih puteva iz zadanog čvora u opštem slučaju, kada težine grana mogu biti i negativne. Prirodno je zapitati se da li negativne težine grana imaju ikakvog smisla. Naime, u kontekstu rastojanja na mapama nemaju, međutim pokazuje se da težinski grafovi čije težine mogu biti i pozitivne i negativne nisu neuobičajeni u modelovanju nekih problema iz realnog sveta. Razmotrimo problem merenja razlike u nadmorskoj visini između mesta u kom se nalazimo i mesta u koje putujemo. Kretanje uzbrdo možemo modelovati granom pozitivne težine, a kretanje nizbrdo granom negativne težine.

Drugi problem koji možemo modelovati na ovaj način je problem održavanja salda na računima: izvršavanje različitih transakcija može dovesti do zarade – koju ćemo modelovati granom pozitivne težine ili do gubitka – koji ćemo modelovati granom negativne težine. Možemo recimo razmotriti problem konverzije iz jedne novčane valute u drugu: pri svakoj konverziji možemo nešto zaraditi ili izgubiti.

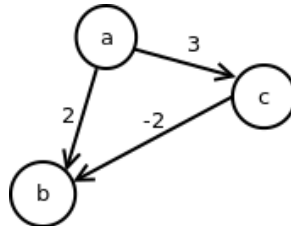
¹Ukoliko bi se umesto binarnog hipa koristio Fibonačijev hip, vremenska složenost Dajkstrinog algoritma bila bi jednaka $O(|E| + |V| \log |V|)$.

Na primer, ako dolare konvertujemo u funte, zatim ih konvertujemo u eure, a onda u japanske jene, može se desiti da završimo sa više novca u odnosu na ono sa čime smo krenuli.

Razmotrimo, takođe, problem putovanja iz jednog grada u drugi, pri čemu je moguće na nekom od delova puta pokupiti nekog putnika i zaraditi određenu količinu novca na tom delu puta. Cilj nam je da dođemo iz jednog grada u drugi na najjeftiniji način. Ovaj problem možemo modelovati u vidu problema traženja najkraćeg puta u grafu čije težine grana mogu biti pozitivne i odgovaraju troškovima putovanja između ta dva grada (ako na tom delu puta putujemo sami) ili negativne (ako na nekom delu puta pokupimo jednog ili više putnika i od njih zaradimo više novca nego što su troškovi tog dela puta).

Ukoliko graf sadrži neku granu negativne dužine, Dajkstrin algoritam ne garantuje da ćemo do svakog čvora odrediti zaista najkraće puteve.

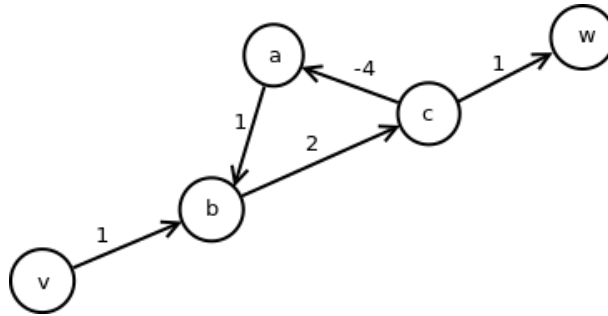
Na primer, ako bismo u grafu sa slike 8 želeli da odredimo najkraće puteve iz čvora a do svih ostalih čvorova i u te svrhe primenimo Dajkstrin algoritam, on bi najpre odredio najkraći put do čvora b (kao čvora koji je od svih čvorova do kojih postoji grana iz a najbliži čvoru a) i proglasio bi da je on dužine 2, a nakon toga bi odredio najkraći put do čvora c kao drugog najbližeg čvora čvoru a i proglasio bi da je dužina najkraćeg puta do njega 3 čime bi se izvršavanje algoritma završilo. Međutim, jasno je da najkraći put od čvora a do čvora b vodi preko čvora c i dužine je 1, ali Dajkstrin algoritam ne bi razmatrao ovaj put.



Slika 8: Primer grafa koji sadrži granu negativne dužine i za koji Dajkstrin algoritam ne računa dobro najkraća rastojanja.

Ukoliko graf ima negativne dužine grana, ali ne sadrži ciklus negativne dužine, najkraći putevi iz datog čvora v mogu se odrediti tzv. *Belman-Fordovim algoritmom*. Ričard Belman i Lester Ford su prvi objavili ovaj algoritam 1958. i 1959. godine, međutim, smatra se da je do njega ipak prvi došao Alfonso Simbel 1955. godine. Uslov da graf ne sadrži ciklus negativne dužine je prirodan jer inače najkraći putevi ne bi mogli biti određeni. Razmotrimo primer grafa sa slike 9: najkraći put od čvora v do čvora w ne bi se mogao odrediti jer bi svaki prolazak kroz ciklus (b, c, a, b) smanjivao dužinu puta za 1.

Osnovni korak u Belman-Fordovom algoritmu je isti kao i u Dajkstrinom algoritmu i u algoritmu za određivanje najkraćih puteva u acikličkim grafovima, a to je tzv. *relaksacija grane* (u, w) , odnosno provera da li je vrednost $w.SP$ trenutno



Slika 9: Primer grafa koji sadrži ciklus negativne dužine.

najkraćeg utvrđenog rastojanja do čvora w veća od vrednosti $u.SP + d(u, w)$; ako je to tačno, ažurira se vrednost $w.SP$ i pamti da najkraći put do čvora w vodi kroz čvor u .

Ideja Belman-Fordovog algoritma je da se ukupno $|V| - 1$ put izvrši relaksacija svih grana u grafu u proizvoljnom redosledu grana. Tvrdimo da će nakon toga biti korektno izračunati najkraći putevi od čvora v do svih čvorova u grafu. Za čvorove koji su nedostižni iz početnog čvora, dužina najkraćeg puta ostaće jednaka $+\infty$.

```

// funkcija koja koriscenjem Belman-Fordovog algoritma
// racuna najkrace puteve do svih cvorova
void najkraciPuteviBelmanFord(){

    int brojCvorova = listaSuseda.size();
    // niz koji za svaki cvor cuva duzinu trenutno poznatog
    // najkraceg puta do njega
    vector<int> najkraciPut(brojCvorova, numeric_limits<int>::max());
    // niz koji za svaki cvor cuva njegovog prethodnika u
    // najkracem putu
    vector<int> roditelj(brojCvorova, -1);

    najkraciPut[0] = 0;

    // |V|-1 put prolazimo kroz skup svih grana
    for (int k = 0; k < brojCvorova - 1; k++)
        for (int i = 0; i < listaSuseda.size(); i++)
            for (int j = 0; j < listaSuseda[i].size(); j++){

                int sused = listaSuseda[i][j].first;
                int grana = listaSuseda[i][j].second;
                // ukoliko je potrebno vrsimo relaksaciju grane
                if (najkraciPut[i] + grana < najkraciPut[sused]){

```



```

        najkraciPut[sused] = najkraciPut[i] + grana;
        // i postavljamo koji cvor mu prethodi na najkracem putu
        roditelj[sused] = i;
    }
}

// ukoliko i dalje postoji put koji je moguće skratiti
// onda graf sadrži ciklus negativne dužine
for (int i = 0; i < listaSuseda.size(); i++)
    for (int j = 0; j < listaSuseda[i].size(); j++){

        int sused = listaSuseda[i][j].first;
        int grana = listaSuseda[i][j].second;
        if (najkraciPut[i] + grana < najkraciPut[sused]){
            cout << "Graf sadrži ciklus negativne dužine" << endl;
            return;
        }
    }

// stampamo najkrace puteve do svih cvorova u grafu
for (int i = 0; i < brojCvorova; i++)
    odstampajNajkraciPut(i, roditelj, najkraciPut);
}

int main(){
    najkraciPuteviBelmanFord();
    return 0;
}

```

Dokažimo korektnost Belman-Fordovog algoritma. Neka je n broj čvorova u grafu. Označimo sa $d_k(i)$ vrednost promenljive `najkraciPut[i]` nakon izvršavanja k iteracija spoljašnje `for` petlje. Dokažimo korektnost ovog algoritma.

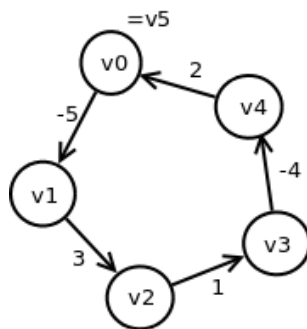
Tvrđenje: Ukoliko u grafu G postoji ciklus negativne dužine koji je dostižan iz početnog čvora v , onda nakon izvršavanja Belman-Fordovog algoritma postoji grana $(u, w) \in E$ koja se može relaksirati, tj. za koju važi $d_{n-1}(w) > d_{n-1}(u) + d(u, w)$; ako graf G ne sadrži ciklus negativne dužine, onda za svaki čvor $u \in V$ važi da je $d_{n-1}(u)$ jednako najkraćem rastojanju od čvora v do čvora u .

Dokaz: Pokažimo najpre prvo tvrđenje. Pretpostavimo da G sadrži ciklus negativne dužine $c = (v_0, v_1, \dots, v_k), v_k = v_0$, dostižan iz čvora v (slika 10).

Tada je:

$$\sum_{i=1}^k d(v_{i-1}, v_i) < 0 \quad (2)$$

Pretpostavimo suprotno, da za svaku granu $(u, w) \in E$ važi $d_{n-1}(w) \leq d_{n-1}(u) +$



Slika 10: Ciklus negativne dužine.

$d(u, w)$. Onda i za svaku granu ciklusa $(v_{i-1}, v_i) \in E, i = 1, \dots, k$ važi:

$$d_{n-1}(v_i) \leq d_{n-1}(v_{i-1}) + d(v_{i-1}, v_i).$$

Sabiranjem ovih nejednakosti za sve grane ciklusa dobija se:

$$\sum_{i=1}^k d_{n-1}(v_i) \leq \sum_{i=1}^k d_{n-1}(v_{i-1}) + \sum_{i=1}^k d(v_{i-1}, v_i)$$

Pošto je $v_0 = v_k$, važi:

$$\sum_{i=1}^k d_{n-1}(v_i) = \sum_{i=1}^k d_{n-1}(v_{i-1})$$

te dalje važi:

$$\sum_{i=1}^k d(v_{i-1}, v_i) \geq 0$$

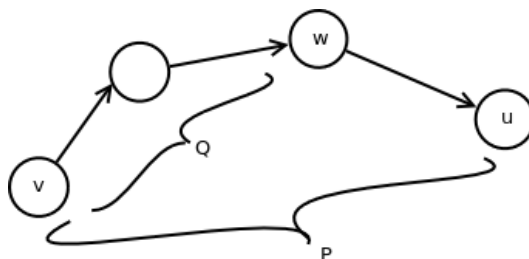
suprotno pretpostavci (2).

Pokažimo sada drugo tvrđenje: da ako graf ne sadrži ciklus negativne dužine, onda za svako $u \in V$ vrednost $d_{n-1}(u)$ sadrži dužinu najkraćeg puta od čvora v do čvora u . Pokazaćemo indukcijom po k da $d_k(u)$ sadrži minimalnu dužinu puta od v do u koji se sastoji od najviše k grana: dakle, da $d_0(u)$ sadrži dužinu najkraćeg puta od v do u od maksimalno 0 grana, $d_1(u)$ dužinu najkraćeg puta od v do u od maksimalno 1 grane, $d_2(u)$ dužinu najkraćeg puta od maksimalno 2 grane i sl. Ako bi ovo važilo, onda bi $d_{n-1}(u)$ sadržalo dužinu najkraćeg puta od čvora v do čvora u koji se sastoji od maksimalno $n - 1$ grana. Ovo je sigurno i dužina najkraćeg puta između ova dva čvora jer iz činjenice da graf ne

sadrži ciklus negativne dužine sledi da najkraći put između dva čvora ne sadrži ponovljene čvorove, te se sastoji od maksimalno $n - 1$ grana.

Bazni slučaj tvrđenja (za $k = 0$) je jednostavan: prema samom algoritmu važi da je $d_0(u) = 0$ ako je $u = v$, a inače je $d_0(u) = +\infty$. Stoga tvrđenje važi jer put dužine 0 postoji samo od čvora v do njega samog i dužine je 0.

Induktivna hipoteza: Pretpostavimo da za sve čvorove u važi da $d_{k-1}(u)$ sadrži minimalnu dužinu puta od čvora v do čvora u koji se sastoji od najviše $k - 1$ grana.



Slika 11: Ilustracija uz dokaz korektnosti Belman-Fordovog algoritma.

Neka je P najkraći put od čvora v do čvora u sa najviše k grana i neka je w čvor neposredno ispred u na putu P (slika 11). Označimo deo puta P od čvora v do čvora w sa Q . Tada se put Q sastoji od najviše $k - 1$ grana i mora biti najkraći mogući put od v do w koji se sastoji od maksimalno $k - 1$ grana (inače bismo u najkraćem putu P deo Q zamenili kraćim putem od v do w). Prema induktivnoj hipotezi dužina puta Q jednaka je $d_{k-1}(w)$.

Razmotrimo izvršavanje poslednje, k -te iteracije spoljašnje `for` petlje: u njoj se razmatraju sve grane grafa, pa i grana (w, u) . Proverava se da li je zbir dužine grane (w, u) i trenutno najkraćeg puta do čvora w manji od trenutno najkraćeg puta do čvora u i ako jeste, vrši se ažuriranje. Dakle, važi jednakost:

$$d_k(u) = \min\{d_{k-1}(u), d_{k-1}(w) + d(w, u)\}.$$

Znamo da važi

$$d_{k-1}(w) + d(w, u) = d(Q) + d(w, u) = d(P)$$

pa stoga važi da je:

$$d_k(u) = \min\{d_{k-1}(u), d(P)\} \quad (3)$$

Takođe, s obzirom na to da je $d_{k-1}(u)$ dužina najkraćeg prostog puta od v do u koji se sastoji od najviše $k - 1$ grana, važi:

$$d_{k-1}(u) \geq d(P) \quad (4)$$

jer P razmatra veći broj grana (ima veći broj opcija na raspolaganju).

Iz nejednakosti (3) i (4) sledi

$$d_k(u) = d(P),$$

odnosno $d_k(u)$ je minimalna dužina puta od čvora v do čvora u koji koristi najviše k grana. \square

Primetimo da se relaksacija grana ne mora vršiti istim redosledom u svakoj od iteracija, jer će svaki redosled obilaska grana očuvati invarijantu: da je nakon k iteracija vrednost $d_k(u)$ jednaka minimalnoj dužini puta od v do u koji se sastoji od maksimalno k grana.

Složenost Belman-Fordovog algoritma iznosi $O(|V| \cdot |E|)$, jer spoljašnjom for petljom prolazimo $O(|V|)$ puta, a u svakoj iteraciji ove petlje prolazimo skupom svih grana u grafu.

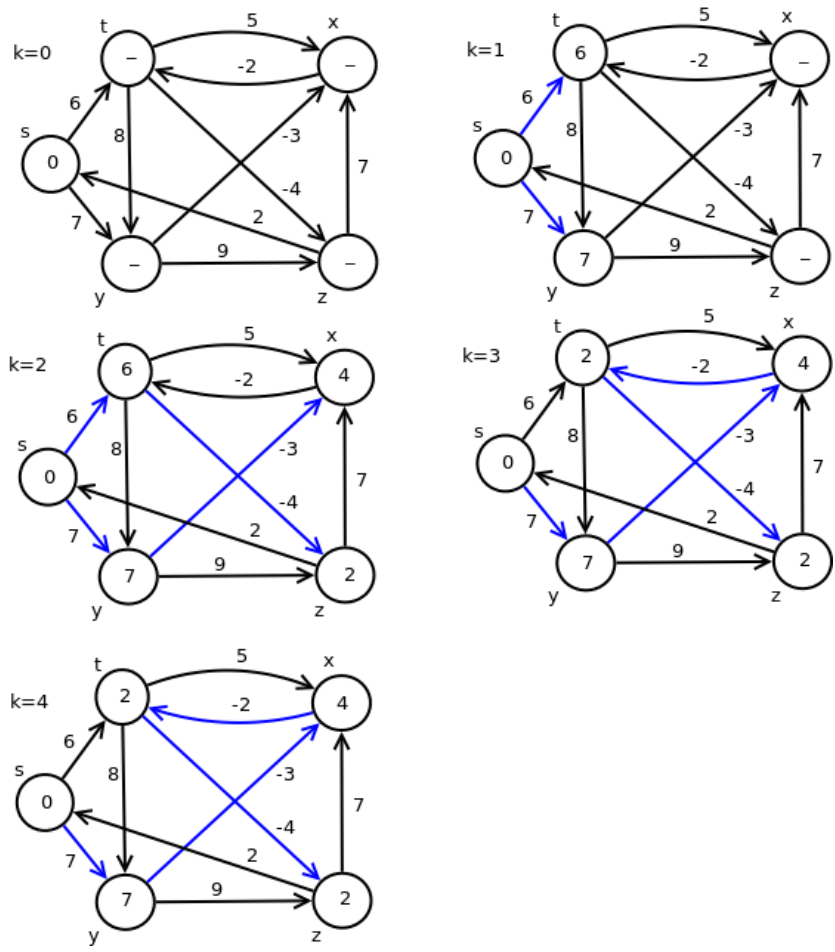
Na slici 12 ilustrovano je izvršavanje Belman-Fordovog algoritma.

Minimalno povezujuće drvo

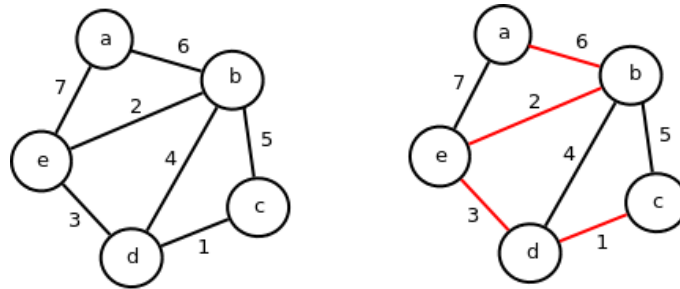
Razmotrimo sistem računara koje treba povezati optičkim kablovima tako da postoji veza između svaka dva računara. Poznati su troškovi postavljanja kabla između svaka dva računara. Cilj je projektovati mrežu optičkih kablova tako da ukupna cena mreže bude minimalna. Sistem računara može biti predstavljen neusmerenim grafom čiji čvorovi odgovaraju računarima, a grane – potencijalnim vezama između računara, sa odgovarajućim (pozitivnim) cenama. Onda se problem svodi na pronalaženje povezanog podgrafa (sa granama koje odgovaraju postavljenim optičkim kablovima) koji sadrži sve čvorove, takav da mu ukupna suma cena grana bude minimalna (slika 13). Nije teško videti da taj podgraf mora da bude drvo. Naime, ako bi podgraf imao ciklus, onda bi se iz ciklusa mogla ukloniti jedna proizvoljna grana čime bi se dobio podgraf koji je i dalje povezan, a ima manju cenu, jer su cene grana pozitivne. Traženi podgraf zove se *minimalno povezujuće (razapinjuće) drvo* (MCST, skraćeno od eng. minimum-cost spanning tree) i ima mnogo primena. Sličan prethodnom problemu je i problem povezivanja N gradova autoputevima, tako da postoji put između svaka dva grada, a da ukupna cena izgradnje autoputa bude minimalna moguća. Minimalno povezujuće drvo ima primenu i u konstrukciji približnih algoritama, na primer, za rešavanje problema trgovačkog putnika.

Dakle, problem kojim se bavimo je konstrukcija efikasnog algoritma za nalaženje minimalnog povezujućeg drveta u datom neusmerenom težinskom grafu sa pozitivnim težinama grana. Težine grana ćemo u kontekstu ovog grafa zvati i njihovim cenama. Postoji više različitih algoritama za određivanje minimalnog povezujućeg drveta datog grafa: mi ćemo razmotriti dva – Primov i Kruskalov algoritam. Ukoliko su cene svih grana u grafu međusobno različite, minimalno povezujuće drvo grafa biće jedinstveno, što olakšava rešavanje problema.

Problem: Za zadati neusmereni povezani težinski graf $G = (V, E)$ čije su grane pozitivne težine konstruisati povezujuće drvo T minimalne cene.



Slika 12: Primer izvršavanja Belman-Fordovog algoritma. Graf ima 5 čvorova, te se algoritam sastoji od 4 iteracija, za $k = 1, 2, 3, 4$. Prva slika odgovara inicijalizaciji, a naredne slike pojedinačnim prolazima kroz sve grane. Vrednosti d_k prikazane su unutar čvorova, a grane plave boje vode od prethodnika čvorova na (trenutno) najkraćim putevima: ako je grana (u, v) plave boje, onda se do čvora v najkraćim putem dolazi preko čvora u . U svakom prolazu grane se relaksiraju u narednom redosledu: (t, x) , (t, y) , (t, z) , (x, t) , (y, x) , (y, z) , (z, x) , (z, s) , (s, t) , (s, y) .



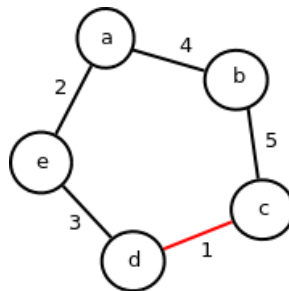
Slika 13: Neusmereni povezan težinski graf i njegovo minimalno povezujuće drvo.

Primov algoritam

Pošto je drvo određeno skupom grana grafa prirodno je koristiti indukciju po skupu grana. Nameće se naredna induktivna hipoteza.

Induktivna hipoteza: Umemo da konstruišemo minimalno povezujuće drvo za povezan graf sa manje od m grana.

Bazni slučaj je trivijalan. Ako je zadat problem konstrukcije minimalnog povezujućeg drveta sa m grana, kako se on može svesti na problem sa manje od m grana? Tvrdimo da grana najmanje cene mora biti uključena u minimalno povezujuće drvo. Ako ona ne bi bila uključena, onda bi njeno dodavanje minimalnom povezujućem drvetu zatvorilo neki ciklus; uklanjanjem proizvoljne druge grane iz tog ciklusa ponovo se dobija drvo, ali manje cene — što je u suprotnosti sa pretpostavkom o minimalnosti konstruisanog povezujućeg drveta. Dakle, mi znamo jednu granu koja mora da pripada minimalnom povezujućem drvetu. Da li možemo da je uklonimo iz grafa i da primenimo induktivnu hipotezu na ostatak grafa, koji sada ima manje od m grana? Da li je ovo regularna primena indukcije?

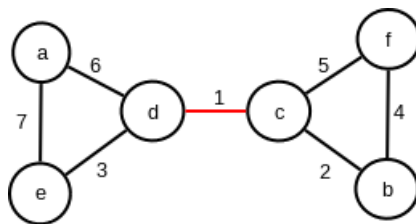


Slika 14: Graf koji je u obliku ciklusa.

Prvi problem u ovakvoj primeni indukcije je u tome što posle uklanjanja grane, dobijeni problem nije ekvivalentan polaznom. Prvo, izbor jedne grane ograničava

mogućnosti izbora drugih grana. Razmotrimo primer grafa sa slike 14. Grana (c, d) je grana minimalne cene u grafu, i ako nju uklonimo i problem svedemo na traženje minimalnog povezujućeg drveta u grafu G bez grane (c, d) dobili bismo da njega čine sve ostale grane grafa: grane (d, e) , (e, a) , (a, b) i (b, c) . Dodavanjem grane (c, d) u ovo minimalno povezujuće drvo dobijamo ciklus, a znamo da ciklus ne može biti deo minimalnog povezujućeg drveta.

Drugi problem sa primenom ovakve induktivne hipoteze je taj što nakon uklanjanja grane graf ne mora da ostane povezan. Razmotrimo graf sa slike 15: nakon uklanjanja grane (c, d) minimalne težine, graf se raspada na dve komponente povezanosti.



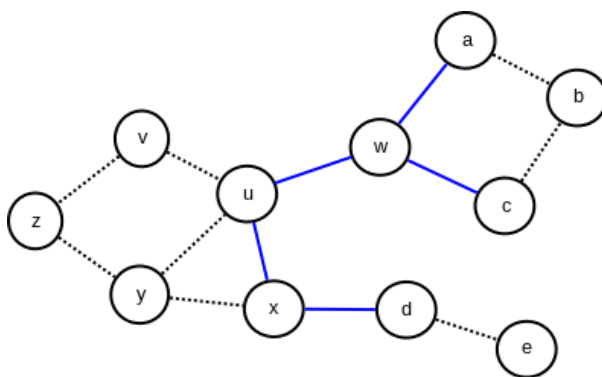
Slika 15: Graf koji se nakon uklanjanja grane minimalne težine raspada na dve komponente povezanosti.

Rešenje nastalog problema je u preciziranju induktivne hipoteze. Mi znamo kako da izaberemo prvu granu, ali ne možemo da je uklonimo i prosto zaboravimo na nju, jer ostali izbori zavise od nje. Dakle, umesto da granu uklonimo, treba da naznačimo da je ona uključena u minimalno povezujuće drvo, i da tu činjenicu, njen izbor, koristimo dalje u algoritmu. Na taj način se jedna po jedna grana bira i dodaje u minimalno povezujuće drvo. Prema tome, indukcija se ovde ne vrši po veličini grafa, već prema broju *izabranih grana* u zadatom (fiksiranom) grafu.

Induktivna hipoteza: Za zadati povezan graf $G = (V, E)$ umemo da pronađemo povezan podgraf – drvo T sa k grana ($k < |V| - 1$), tako da je drvo T podgraf minimalnog povezujućeg drveta grafa G .

Bazni slučaj za ovu hipotezu smo već razmotrili — on se odnosi na izbor prve grane. Pretpostavimo da smo pronašli drvo T koje zadovoljava induktivnu hipotezu i da je potrebno da T proširimo narednom granom. Kako da pronađemo novu granu za koju ćemo biti sigurni da pripada minimalnom povezujućem drvetu? Primenićemo sličan pristup kao i pri izboru prve grane. Za T se već zna da je deo konačnog minimalnog povezujućeg drveta. Zbog toga u minimalnom povezujućem drvetu mora da postoji bar jedna grana koja povezuje neki čvor iz T sa nekim čvorom u ostatku grafa. Pokušaćemo da pronađemo takvu granu. Neka je E_k skup svih grana koje povezuju T sa čvorovima van T . Tvrdimo da grana sa najmanjom cenom iz E_k pripada minimalnom povezujućem drvetu. Označimo tu granu sa (u, v) (videti sliku ??; grane drveta T su podebljane).

Pošto je drvo koje konstruišemo povezujuće, ono sadrži tačno jedan put od u do v jer između svaka dva čvora u drvetu postoji tačno jedan put. Ako grana (u, v) ne bi pripadala minimalnom povezujućem drvetu, onda ona ne bi pripadala ni putu od u do v . Međutim, pošto u pripada, a v ne pripada T , na tom putu mora da postoji bar još jedna grana (x, y) takva da $x \in T$ i $y \notin T$. Cena ove grane veća je od cene (u, v) , jer je cena (u, v) najmanja među cenama grana koje povezuju T sa ostatkom grafa. Sada možemo da primenimo slično zaključivanje kao pri izboru prve grane. Ako granu (u, v) dodamo minimalnom povezujućem drvetu, a iz njega izbacimo granu (x, y) , dobijamo povezujuće drvo manje cene, što je kontradikcija.



Slika 16: Nalaženje sledeće grane minimalnog povezujućeg drveta.

Napomenimo da ako u grafu postoje grane iste težine, prilikom nailaska na grane jednake cene, proizvoljno se bira jedna od njih.

Opisani algoritam poznat je pod nazivom *Primov algoritam* i sličan je Dajkstrinom algoritmu za nalaženje najkraćih puteva od zadatog čvora. Interesantno je ovaj algoritam prvi osmislio Vojteh Jarnik 1930. godine, a da su tek kasnije nezavisno do njega došli Robert Prim 1957. godine i Edzger Dajkstra 1959. godine.

Primovim algoritmom minimalno povezujuće drvo T se konstruiše proširivanjem trenutno konstruisanog drveta T jednom po jednom granom. Drvo T se inicijalizuje na drvo koje sadrži samo granu minimalne cene u grafu. U svakoj iteraciji pronalazi se grana koja povezuje T sa nekim čvorom van T , a ima najmanju cenu. Jedina razlika između Primovog algoritma za konstrukciju minimalnog povezujućeg drveta i Dajkstrinog algoritma za nalaženje najkraćih puteva je u tome što se minimum ne traži po dužini puta, već po ceni grane. Ostatak algoritma prenosi se praktično bez promene. Za svaki čvor w van T pamtimo cenu grane minimalne cene do w od nekog čvora iz T , odnosno $+\infty$ ako takva grana ne postoji. U svakoj iteraciji biramo granu najmanje cene i povezujeemo odgovarajući čvor w sa drvetom T . Zatim proveravamo sve grane susedne čvoru w koje vode do čvora van T . Ako je cena neke takve grane (w, z) (za $z \notin T$)

manja od cene trenutno najjeftinije poznate grane do z , onda ažuriramo cenu čvora z i granu koja kroz drvo vodi do njega.

```
// uredjen par vrednosti rastojanja do cvora i indeksa cvora
// vazan je redosled komponenti zbog operacije poredjenja po rastojanju
typedef pair<int,int> rastojanjeDoCvora;

vector<vector<pair<int,int>>> listaSuseda {{{1,7}, {2,6}},
    {{0,7}, {2,2}, {3,3}}, {{0,6}, {1,2}, {3,4}, {4,5}},
    {{1,3}, {2,4}, {4,1}}, {{2,5}, {3,1}}};

// Primov algoritam za odredjivanje minimalnog povezujuceg drveta
void minimalnoPovezujuceDrvoPrim(){

    int brojCvorova = listaSuseda.size();
    // niz koji cuva informaciju o tome da li je cvor posecen
    vector<bool> posecen(brojCvorova,false);
    // niz koji za svaki cvor cuva duzinu trenutno poznate najkrace grane
    // koja vodi do njega
    vector<int> najkracaGrana(brojCvorova,numeric_limits<int>::max());
    // niz koji za svaki cvor cuva roditelja u minimalnom povezujucem drvetu
    vector<int> roditelj(brojCvorova,-1);

    // hip u koji smestamo duzine najkracih grana do svih cvorova
    priority_queue<rastojanjeDoCvora,vector<rastojanjeDoCvora>,
        greater<rastojanjeDoCvora>> rastojanja;

    // odredjujemo granu minimalne tezine
    int min = numeric_limits<int>::max();
    int min_beg = -1;
    int min_end = -1;
    for (int i = 0; i < listaSuseda.size(); i++)
        for (int j = 0; j < listaSuseda[i].size(); j++)
            if (listaSuseda[i][j].second < min){
                min = listaSuseda[i][j].second;
                min_beg = i;
                min_end = listaSuseda[i][j].first;
            }

    // ubacujemo krajnji cvor grane minimalne tezine u hip
    // i postavljamo duzinu grane do njega na 0
    rastojanja.push(make_pair(0,min_beg));
    najkracaGrana[min_beg] = 0;
    roditelj[min_end] = min_beg;

    // duzine grana do ostalih cvorova postavljamo na
```

```

// maksimalnu mogucu vrednost i ubacujemo ih u hip
for(int cvor = 0; cvor < brojCvorova; cvor++){
    if (cvor != min_beg){
        rastojanja.push(make_pair(numeric_limits<int>::max(), cvor));

// odredjujemo narednih (brojCvorova-1) cvorova i
// duzine najkracih grana do njih
for(int i = 0; i < brojCvorova; i++){

    // izdvajamo naredni cvor sa najmanjom duzinom grane do njega
    rastojanjeDoCvora najblizi = rastojanja.top();
    rastojanja.pop();
    int cvor = najblizi.second;

    // ako je taj cvor vec posecen, onda ga treba preskociti
    // i ne treba ga ponovo brojati
    if (posecen[cvor]){
        i--;
        continue;
    }
    // ako cvor nije bio do sada posecen, postavljamo
    // informaciju da smo ga sada posetili
    posecen[cvor] = true;

    // postavljamo vrednost najkrace grane do njega
    // iz do sada posecenih cvorova
    najkracaGrana[cvor] = najblizi.first;

    // za sve susede tekuceg cvora
    for (int j = 0; j < listaSuseda[cvor].size(); j++){
        // ako do sada nisu bili poseceni
        if (!posecen[listaSuseda[cvor][j].first]){
            int sused = listaSuseda[cvor][j].first;
            int duzinaGraneSuseda = listaSuseda[cvor][j].second;
            // ukoliko je grana iz tekuceg cvora kraca od
            // prethodno najkrace grane do tog cvora,
            // azuriramo vrednost najkrace grane i roditeljskog cvora
            // preko koga se dolazi do tog cvora
            if (duzinaGraneSuseda < najkracaGrana[sused]){
                najkracaGrana[sused] = duzinaGraneSuseda;
                roditelj[sused] = cvor;
                // ubacujemo element u hip, ukoliko je
                // postojala prethodna vrednost, ne brisemo je;
                // nova vrednost ce se naci u hipu iznad stare
                rastojanja.push(make_pair(najkracaGrana[sused], sused));
            }
        }
    }
}

```

```

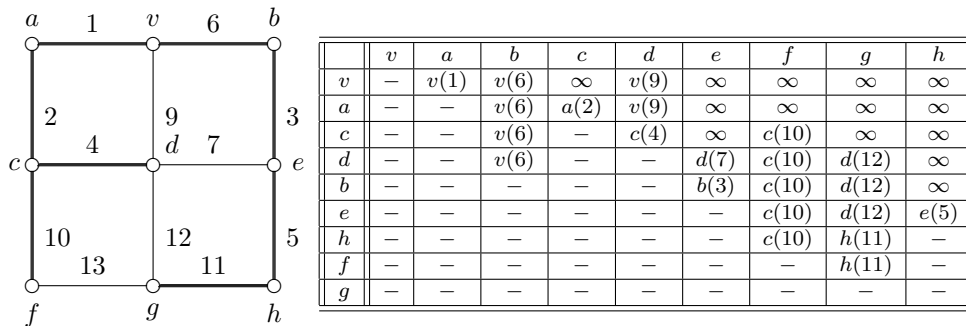
    }
  }
}
cout << "Minimalno povezujuce drvo se sastoji od grana: " << endl;
for(int i = 0; i < brojCvorova; i++)
  if (i! = min_beg)
    cout << "(" << roditelj[i] << ", " << i << ") cene "
      << najkracaGrana[i] << endl;
}

int main(){
  minimalnoPovezujuceDrvoPrim();
  return 0;
}

```

Složenost Primovog algoritma identična je složenosti Dajkstrinog algoritma za nalaženje najkraćih rastojanja od zadatog čvora i iznosi $O((|E| + |V|) \log |V|)$.

Primer: Primov algoritam za konstrukciju minimalnog povezujućeg drveta ilustriramo primerom na slici 17. Čvor u prvoj koloni tabele je onaj koji je dodat u odgovarajućem koraku. Prvi dodati čvor je v , i u prvoj vrsti navedeni su čvorovi do kojih postoje grane iz čvora v sa svojim cenama. U svakoj vrsti bira se grana sa najmanjom cenom. Spisak trenutno najboljih grana i njihovih cena popravlja se u svakom koraku (prikazani su samo krajevi grana). Na slici grafa su grane koje pripadaju minimalnom povezujućem drvetu podebljane.



Slika 17: Primer izvršavanja Primovog algoritma za nalaženje minimalnog povezujućeg drveta.

Primov algoritam je donekle primer pohlepnog algoritma, jer se u svakom koraku biraju grane sa najmanjom cenom. U algoritmu smo uveli neka dodatna ograničenja pri izboru grana: razmatrane su samo grane povezane sa tekućim drvetom. Zbog toga algoritam nije čisti primer pohlepnog algoritma.

Kruskalov algoritam

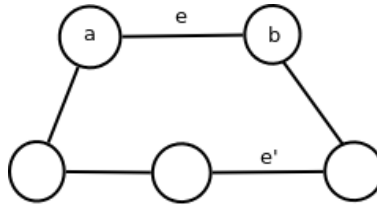
Drugi efikasan algoritam za određivanje minimalnog povezujućeg drveta datog neusmerenog težinskog grafa $G = (V, E)$ je takođe pohlepan, ali do minimalnog povezujućeg drveta ne dolazi dodavanjem novih grana na trenutno drvo, nego na trenutnu šumu. Inicijalno svaki čvor grafa predstavlja zasebno drvo i odgovarajuća šuma sadrži $|V|$ drveta i nijednu granu. Zatim se postepeno spajaju po dva drveta ove šume, dodavanjem grana. Dodavanjem svake nove grane, broj drveta u šumi se smanjuje za 1, te će se nakon dodavanja $|V| - 1$ grane dobiti tačno jedno drvo. Pri tome se grane koje se dodaju razmatraju u neopadajućem redosledu cena; ako grana koja je sledeća na redu povezuje dva čvora u različitim drvetima trenutne šume, onda se ta grana uključuje u šumu, čime se ta dva drveta spajaju u jedno. U protivnom, ako grana povezuje dva čvora iz istog drveta trenutne šume, grana se preskače. Ovaj algoritam je prvi objavio Džozef Kruskal 1956. godine i poznat je pod nazivom *Kruskalov algoritam*.

Dokažimo da će ovaj algoritam uvek vratiti minimalno povezujuće drvo datog grafa. Lako se pokazuje da algoritam vraća povezujuće drvo datog grafa, te ćemo samo pokazati da je ono i minimalno. Jednostavnosti radi pretpostavimo da su sve grane u grafu različite cene. Pretpostavimo suprotno: da je Kruskalov algoritam vratio drvo K koje nije minimalno povezujuće drvo datog grafa. Označimo sa T minimalno povezujuće drvo datog grafa. Neka su grane oba drveta K i T sortirane u rastućem redosledu svojih cena. Iz $K \neq T$ sledi da postoji barem jedna grana u kojoj se ova dva drveta razlikuju. Razmotrimo najraniju granu $e = (a, b)$ u rastućem redosledu grana prema cenama u kojoj se K i T razlikuju (tj. grana e pripada jednom, a ne pripada drugom drvetu). S obzirom na to da Kruskalov algoritam razmatra grane u rastućem redosledu cena i ne dodaje samo one koje zatvaraju neki ciklus, mora važiti da grana e pripada drvetu K koje je vratio Kruskalov algoritam, a ne pripada minimalnom povezujućem drvetu T (tabela 1). U minimalnom povezujućem drvetu T mora postojati jedinstveni put P od čvora a do čvora b . Na tom putu postoji bar jedna grana čija je cena veća od cene grane e . Naime, ako to ne bi važilo sve ostale grane bi kao grane manje cene bile uključene Kruskalovim algoritmom u drvo K , uz granu e , te bi suprotno pretpostavci drvo K sadržalo ciklus. Označimo sa e' granu puta P čija je cena veća od cene grane e (slika 18). Ako iz drveta T izbacimo granu e' , a dodamo granu e dobijamo povezujuće drvo manje cene što je suprotno pretpostavci da je T minimalno povezujuće drvo datog grafa.

grane u rastućem redosledu cena	e_1	e_2	e_3	...	$e = (a, b)$...
grana pripada drvetu K ?	√	√	√	√	√	...
grana pripada drvetu T ?	√	√	√	√	×	...

Tabela 1: Ilustracija uz dokaz korektnosti Kruskalovog algoritma.

Za utvrđivanje da li su krajnji čvorovi u i v tekuće grane (u, v) u istom ili u različitim drvetima trenutne šume, može se iskoristiti union-find struktura podataka za disjunktne podskupove: trenutna drveta šume su disjunktne pod-



Slika 18: Ilustracija uz dokaz korektnosti Kruskalovog algoritma.

skupovi skupa čvorova grafa. Operacije `podskup(u)` i `podskup(v)` pronalaze predstavnike u', v' dva podskupa (korene drveća kojim su oni predstavljeni), pa su čvorovi u i v u istom podskupu akko je $u' = v'$. Ako je $u' \neq v'$, onda se ta dva podskupa zamenjuju svojom unijom, tj. primenjuje se operacija `unija(u', v')`, a dva poddrveća trenutne šume se granom spajaju u jedno.

```
// uredjen par vrednosti koji predstavlja granu
typedef pair<int,int> grana;

vector<vector<pair<int,int>>> listaSuseda {{{1,3}, {2,2}}, {{3,1}, {4,2}},
    {{5,3}}, {}, {{6,1}, {7,3}}, {{8,4}}, {}, {}, {}};

// funkcija za inicijalizaciju union-find strukture
void inicijalizuj(vector<int> &roditelj, vector<int> &rang, int n) {
    // svaki cvor je podskup za sebe
    for (int i = 0; i < n; i++) {
        roditelj[i] = i;
        rang[i] = 0;
    }
}

// funkcija koja izracunava kom podskupu pripada neki element
int predstavnik(int x, vector<int> &roditelj) {
    int koren = x;
    while (koren != roditelj[koren])
        koren = roditelj[koren];
    while (x != koren) {
        int tmp = roditelj[x];
        roditelj[x] = koren;
        x = tmp;
    }
    return koren;
}

// funkcija koja pravi uniju dva podskupa
void unija(int x, int y, vector<int> &roditelj, vector<int> &rang) {
```

```

int fx = predstavnik(x, roditelj);
int fy = predstavnik(y, roditelj);
if (rang[fx] < rang[fy])
    roditelj[fx] = fy;
else if (rang[fy] < rang[fx])
    roditelj[fy] = fx;
else {
    roditelj[fx] = fy;
    rang[fy]++;
}
}

// Kruskalov algoritam za odredjivanje minimalnog povezujuceg drveta
void minimalnoPovezujeDrvoKruskal(){

    int brojCvorova = listaSuseda.size();
    // niz koji cuva informaciju o tome da li je cvor posecen
    vector<bool> posecen(brojCvorova, false);
    // niz koji za svaki cvor cuva duzinu najkrace grane koja vodi do njega
    vector<int> najkracaGrana(brojCvorova, numeric_limits<int>::max());
    // niz koji za svaki cvor cuva roditelja u minimalnom povezujucem drvetu
    // tj. drugi cvor grane koja pripada minimalnom povezujucem drvetu
    vector<int> roditelj(brojCvorova);
    // niz koji za svaki cvor cuva njegov rang
    vector<int> rang(brojCvorova);

    // inicijalizujemo union-find strukturu
    // svaki cvor grafa predstavlja podskup za sebe
    inicijalizuj(roditelj, rang, brojCvorova);

    // kreiramo jedinstveni niz svih grana u grafu
    vector<pair<int, grana>> grane;
    for (int i = 0; i < brojCvorova; i++)
        for (int j = 0; j < listaSuseda[i].size(); j++){
            int sused = listaSuseda[i][j].first;
            int duzinaGrane = listaSuseda[i][j].second;
            grane.push_back({duzinaGrane, {i, sused}});
        }

    // sortiramo skup grana u neopadajućem redosledu cena
    sort(grane.begin(), grane.end());
    int brojGrana = 0;

    // prolazimo redom kroz skup grana
    for(auto it = grane.begin(); it != grane.end(); it++){

```

```

// ako smo u skup grana dodali |V|-1 grana
// ne treba prolaziti kroz preostale grane
if (brojGrana == brojCvorova - 1) break;

int u = it->second.first;
int v = it->second.second;
int duzina = it->first;

int skup_u = predstavnik(u, roditelj);
int skup_v = predstavnik(v, roditelj);

// ako tekuca grana povezuje dva cvora koja pripadaju
// razlicitim drvetima onda tu granu dodajemo u minimalno povezujuce drvo
// i pravimo uniju skupova cvorova koji pripadaju tim drvetima
if (skup_u != skup_v){
    unija(skup_u, skup_v, roditelj, rang);
    // ako je u roditelj od v u skupu, onda postavljamo granu do v
    if (u == roditelj[v])
        najkracaGrana[v] = duzina;
    // inace postavljamo granu do u
    else
        najkracaGrana[u] = duzina;
    brojGrana++;
}
}

cout << "Minimalno povezujuce drvo se sastoji od grana: " << endl;
for(int i = 1; i < brojCvorova; i++)
    cout << "(" << roditelj[i] << "," << i << ") cene "
        << najkracaGrana[i] << endl;
}

int main(){
    minimalnoPovezujuceDrvoKruskal();
    return 0;
}

```

Funkcija za inicijalizaciju union-find strukture za disjunktne skupove je složenosti $O(|V|)$, dodavanje grana u skup grana je složenosti $O(|E|)$, njihovo sortiranje je prosečne složenosti $O(|E| \log |E|)$, a nakon toga se glavna petlja izvršava $|E|$ puta, dok su operacije koje se pozivaju u petlji (predstavnik i unija) složenosti $O(\log |V|)$. Dakle, ukupna složenost Kruskalovog algoritma iznosi $O(|V|) + O(|E|) + O(|E| \log |E|) + O(|E| \log |V|) = O(|E| \log |V|)$ (koristimo činjenicu da je $O(\log |E|) = O(\log |V|)$ zbog $|E| \leq |V|^2$).

Primer izvršavanja Kruskalovog algoritma na grafu sa slike 17 prikazan je u tabeli 2. Prolazimo redom kroz skup grana u rastućem redosledu cena i završavamo

naredna grana	dužina grane	uključena u MCST?	trenutna suma
			$v, a, b, c, d, e, f, g, h$
(a, v)	1	da	$\underline{a}v, b, c, d, e, f, g, h$
(a, c)	2	da	$acv, \underline{b}, c, d, e, f, g, h$
(b, e)	3	da	$acv, be, \underline{d}, e, f, g, h$
(c, d)	4	da	$acd\underline{v}, be, f, g, \underline{h}$
(e, h)	5	da	$acd\underline{v}, \underline{be}h, f, g$
(b, v)	6	da	$abcde\underline{h}v, f, g$
(d, e)	7	ne	$abcde\underline{h}v, f, g$
(d, v)	8	ne	$abcde\underline{h}v, \underline{f}, g$
(c, f)	9	da	$abcde\underline{f}h\underline{v}, g$
(g, h)	10	da	$abcde\underline{f}gh\underline{v}$

Tabela 2: Primer izvršavanja Kruskalovog algoritma za graf sa slike 17.

obradu kada u skup dodamo $|V| - 1$ (u ovom slučaju 8) grana. S obzirom na to da su dužine svih grana različite, kao rezultat dobijamo minimalno povezujuće drvo prikazano na slici 17.