Grafovski algoritmi - čas 5

U problemima koji slede bavićemo se *težinskim grafovima*: grafovima u kojima je svakoj grani pridružena njena *težina*. Težine ćemo često zvati i dužinama i pod terminom *dužina puta* razmatraćemo zbir dužina grana na tom putu (a ne broj grana na tom putu).

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 koristimo listu povezanosti i ako su dužine grana celobrojne, možemo upotrebiti strukturu podataka oblika:

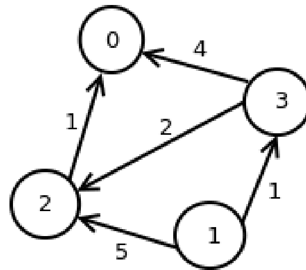
```
vector<vector<pair<int,int>>> listaSuseda(n);
```

Novu granu u graf dodajemo na sledeći način:

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

Na primer, 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}}};
```



Slika 1: Usmereni težinski graf.

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 odnose se i na usmerene i na neusmerene grafove.

Najkraći putevi iz zadatog čvora

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

Postoji mnogo situacija u kojima se pojavljuje ovaj problem. 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 je trivijalan. Neka je $|V| = n$. Možemo da iskoristimo topološko sortiranje grafa iz prethodnog odeljka. Ako je redni broj čvora v od koga treba odrediti najkraće puteve jednak k, onda se čvorovi sa rednim brojevima manjim od k ne moraju razmatrati: ne postoji način da se do njih dođe iz čvora v. Pored toga, redosled 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). Da bismo odredili vrednost $z.SP$, dovoljno je da proverimo samo one čvorove w iz kojih postoji grana do čvora z. Pošto se najkraći putevi do ostalih čvorova već znaju, $z.SP$ jednako je minimumu zbira $w.SP + duzina(w, z)$, po svim čvorovima w iz kojih vodi grana do z. Da li je time problem rešen? Pitanje je da li dodavanje čvora z može da skrati 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, nalaženje najkraćih puteva bez njega, i vraćanje z nazad 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 v do prvih $n - 1$ čvorova.

Kad je dat aciklički graf sa n čvorova (topološki uređenih), uklanjamo n-ti čvor, indukcijom rešavamo smanjeni problem, nalazimo najmanju među vrednostima $w.SP + duzina(w, z)$ za sve čvorove w takve da $(w, z) \in E$ i nju proglašavamo za $z.SP$. Iz ovog razmatranja direktno sledi odgovarajući rekurzivni algoritam.

Sada ćemo pokušati da usavršimo algoritam tako da se Kanov algoritam za topološko sortiranje obavlja istovremeno sa pronalaženjem najkraćih puteva.

Drugim rečima, cilj je objediniti dva prolaza (za topološko sortiranje i 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 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; 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 ovom slučaju u 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 počevši od v. Ovaj redosled oslobađa nas potrebe da redne brojeve unapred znamo, pa ćemo biti u stanju da izvršavamo istovremeno oba algoritma. Dakle, možemo razmotriti narednu induktivnu hipotezu.

Induktivna hipoteza: Ako se zna topološki redosled čvorova, umemo da izračunamo dužine najkraćih puteva 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 $w.SP + duzina(w, z)$. 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 z. Da ne bismo za svaki čvor određivali iz kojih čvorova postoji grana ka njemu (što nije efikasna operacija u reprezentaciji grafa listama povezanosti) možemo pamtiti dužine 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 svaki čvor x proveravati da li je vrednost $z.SP + duzina(z, x)$ manja od $x.SP$ i ako jeste ažurirati vrednost.

Za svaki čvor pamtimo njegovog prethodnika (roditelja) na najkraćem putu od čvora v. Na taj način možemo jednostavno da rekonstruišemo same najkraće puteve.

```
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 izdvojenog cvora do datog cvora
// kroz grane drveta najkracih puteva
void odstampajPutDoCvora(int cvor, vector<int> roditelj){

  if (roditelj[cvor] == -1)
    return;
  odstampajPutDoCvora(roditelj[cvor],roditelj);
  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 najkraceg puta do njega
  vector<int> najkraciPut(brojCvorova,numeric_limits<int>::max());
  // niz koji za svaki cvor cuva prethodnika u najkracem putu
  vector<int> roditelj(brojCvorova,-1);

  najkraciPut[0] = 0;

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

  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 topolskom 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 kraci put do nekog cvora preko upravo razmatranog cvora
      // vrsimo 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 preko koji je pretposlednji 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. Jednostavnost algoritma za određivanje najkraćih puteva iz zadatog čvora u acikličkom grafu posledica je sledeće osobine topološkog redosleda: ako je z čvor sa rednim brojem k, onda:

1. ne postoje putevi od z do čvorova sa rednim brojevima manjim od k, i
2. ne postoje putevi od čvorova sa rednim brojevima većim od k do z.

Ova osobina 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; one 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 v do x je grana (v, x). Dužine svih drugih puteva do x su veće ili jednake od dužine ove grane. Čvor x je pritom najbliži od svih čvorova čvoru v. Prema tome, znamo najkraći put do x, 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 do 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: prva je (v, x), a druga je grana iz čvora x. Neka je sa $duzina(u, w)$ označena dužina grane (u, w). Biramo najmanji od izraza $duzina(v, y)$ $(y \neq x)$ ili $duzina(v, x) + duzina(x, z)$ $(z \neq v)$. Još jednom zaključujemo da se drugi putevi ne moraju razmatrati, jer je ovo najkraći put za odlazak iz v (izuzev do x). Može se formulisati sledeća induktivna hipoteza.

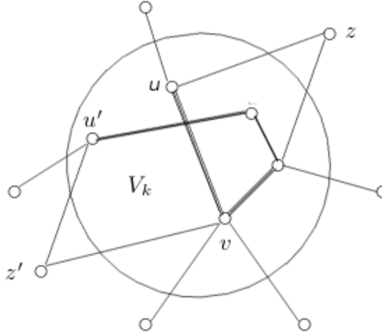
Induktivna hipoteza: Za zadati graf i njegov čvor v, umemo da pronađemo k čvorova najbližih čvoru v, kao i dužine najkraćih puteva do njih.

Zapazimo da je indukcija po 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 dobije 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 w. 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 (prema induktivnoj hipotezi već poznat) i grane (u, z). Dovoljno je uporediti sve takve puteve i izabrati najkraći među njima, videti ilustraciju na slici 2.

Algoritam određen ovom induktivnom hipotezom izvršava se na sledeći način. U svakoj iteraciji dodaje se novi čvor u skup V_k. To je čvor w za koji je najmanja dužina

$$\min \{u.SP + duzina(u, w) | \, u \in V_k\} \tag{1}$$



Slika 2: Nalaženje sledećeg najbližeg čvora zadatom čvoru v.

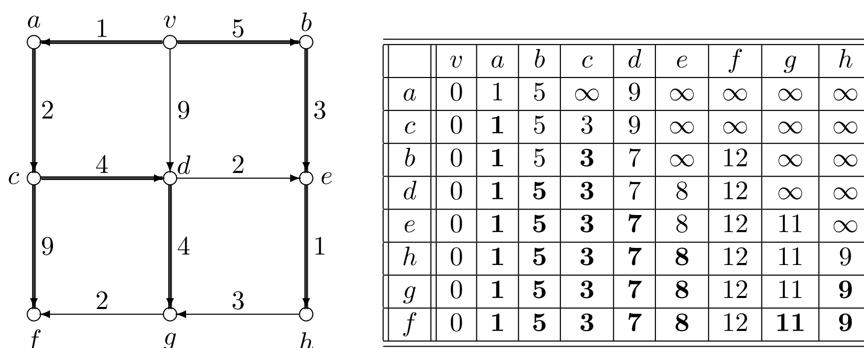
među svim čvorovima $w \notin V_k$. Iz već iznetih razloga, w je zaista $(k+1)$-vi (sledeći) najbliži čvor čvoru v. Prema tome, njegovo dodavanje produžuje induktivnu hipotezu.

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 (1). Međutim, nije neophodno u svakom koraku proveravati sve vrednosti $u.SP + duzina(u, w)$. Većina tih vrednosti ne menja se pri dodavanju novog čvora u V_k: mogu se promeniti samo one vrednosti koje odgovaraju putevima kroz novododati čvor. Mi možemo da pamtimo dužine poznatih najkraćih puteva do svih čvorova van V_k, i da im popravljamo vrednosti samo pri proširivanju skupa V_k. Jedini način da se dobije novi najkraći put nakon dodavanja čvora w u V_k je da taj put prolazi kroz w. Prema tome, treba proveriti sve grane od w ka čvorovima van V_k. Za svaku takvu granu (w, z) upoređujemo dužinu $w.SP + duzina(w, z)$ sa vrednošću $z.SP$, i po potrebi popravljamo $z.SP$. Svaka iteracija obuhvata nalaženje čvora sa najmanjom vrednošću SP, i popravku vrednosti SP za neke od preostalih čvorova. Ovaj algoritam poznat je kao *Dajkstrin algoritam*. On pripada grupi pohlepnih algoritama jer se u svakom koraku bira lokalno optimalno rešenje – najbliži čvor i nakon odabira 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 puteve pronalazili jedan po jedan. Svaki novi put je određen jednom granom, koja produžuje prethodno poznati najkraći put do novog čvora. Sve te grane formiraju drvo sa korenom v. Ovo drvo zove se *drvo najkraćih puteva*, i važno je za rešavanje mnogih problema sa putevima. Ako su dužine svih grana jednake, onda je drvo najkraćih puteva u stvari BFS drvo sa korenom u čvoru v. U primeru na slici 3 podebljane su grane koje pripadaju drvetu najkraćih puteva sa korenom u čvoru v.

Primer: Ilustracija izvršavanja Dajkstrinog algoritma za nalaženje najkraćih

puteva od čvora v prikazana je na slici 3. 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 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.



	v	a	b	c	d	e	f	g	h
a	0	1	5	∞	9	∞	∞	∞	∞
c	0	**1**	5	3	9	∞	∞	∞	∞
b	0	**1**	5	**3**	7	∞	12	∞	∞
d	0	**1**	**5**	**3**	7	8	12	∞	∞
e	0	**1**	**5**	**3**	**7**	8	12	11	∞
h	0	**1**	**5**	**3**	**7**	**8**	12	11	9
g	0	**1**	**5**	**3**	**7**	**8**	12	11	**9**
f	0	**1**	**5**	**3**	**7**	**8**	12	**11**	**9**

Slika 3: Primer izvršavanja Dajkstrinog algoritma.

U Dajkstrinom algoritmu potrebno je da pronalazimo najmanju vrednost u skupu dužina puteva i da često popravljamo dužine puteva. Dobra struktura podataka za nalaženje minimalnih elemenata i za popravke dužina elemenata je red sa prioritetom, odnosno min-hip. Pošto je potrebno da pronađemo čvor sa najmanjom dužinom puta do njega, sve čvorove van skupa V_k čuvamo u hipu, sa ključevima jednakim dužinama trenutno najkraćih puteva od v do njih. Na početku su sve dužine puteva sem one do čvora v jednake ∞, pa redosled elemenata u hipu nije bitan, sem što 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 za svaku granu (w, u) proverava se da li korišćenje te grane skraćuje put do čvora u. Međutim, kad se promeni dužina puta do nekog čvora u, može se promeniti 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 smanjiti i eventualno 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. Međutim, problem sa ovim popravkama je u tome što hip kao struktura podataka ne podržava efikasno pronalaženje zadatog elementa. Iz tog razloga umesto da se vrednost rastojanja do nekog čvora u hipu zameni novom (manjom) vrednošću, vršiće se umetanje novog čvora sa istom oznakom čvora i

novom 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 zadatog čvora 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;

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
// do svih cvorova iz cvora sa indeksom 0
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 najkraceg puta do njega
  vector<int> najkraciPut(brojCvorova,numeric_limits<int>::max());
  // niz koji za svaki cvor cuva roditelja 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 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, 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 najkraceg puta do njega
  // kroz do sada posecene cvorove
  najkraciPut[cvor] = najblizi.first;
  // stampamo informaciju o najkracem putu do tog cvora
  // na ovaj nacin 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 kraci od prethodnog
        // najkraceg puta, azuriramo vrednost najkraceg puta i
        // roditeljskog cvora preko koga se dolazi do tog cvora
        if (najkraciPut[cvor] + duzinaGrane < najkraciPut[sused]){
           najkraciPut[sused] = najkraciPut[cvor] + duzinaGrane;
           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(najkraciPut[sused], sused));
              }
          }
      }
  }
}

int main(){
  najkraciPuteviDajkstra();
  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 algoritam sporiji nego algoritam koji isti problem rešava za acikličke grafove.[1]

Ovakav tip algoritma se ponekad zove *pretraga sa prioritetom* — svakom čvoru dodeljuje se prioritet (u ovom slučaju trenutno najmanje poznato rastojanje od čvora v), pa se čvorovi obilaze redosledom koji je određen prioritetom. Kad se završi razmatranje čvora, proveravaju se sve njemu susedne grane. Ta provera može da dovede do promene nekih prioriteta. Način izvođenja tih promena je detalj po kome se jedna pretraga sa prioritetom razlikuje od druge. Pretraga sa prioritetom složenija je od obične pretrage. Ona je korisna kod problema sa težinskim grafovima.

Minimalno povezujuće drvo

Razmotrimo sistem računara koje treba povezati optičkim kablovima. Potrebno je obezbediti 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 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. Nije teško videti da taj podgraf mora da bude drvo. Ako bi podgraf imao ciklus, onda bi se iz ciklusa mogla ukloniti jedna grana — time se dobija podgraf koji je i dalje povezan, a ima manju

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

cenu, jer su cene grana pozitivne. Traženi podgraf zove se *minimalno povezujuće (razapinjuće) drvo* (MCST, skraćenica od eng. minimum–cost spanning tree) i ima mnogo primena. Slično se može rešiti 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 se koristi i kod konstrukcije približnih algoritama, na primer, za rešavanje problema trgovačkog putnika.

Naš cilj je konstrukcija efikasnog algoritma za nalaženje minimalnog povezujućeg drveta. Zbog jednostavnosti, pretpostavimo da su cene grana različite. Ova pretpostavka ima za posledicu da je minimalno povezujuće drvo jedinstveno, što olakšava rešavanje problema. Bez ove pretpostavke algoritam ostaje nepromenjen, izuzev što se, prilikom nailaska na grane jednake cene, proizvoljno bira jedna od njih.

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

Primov algoritam

U kontekstu ovog problema težine grana težinskog grafa G su u stvari njihove cene. Prirodno je koristiti sledeću induktivnu hipotezu.

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

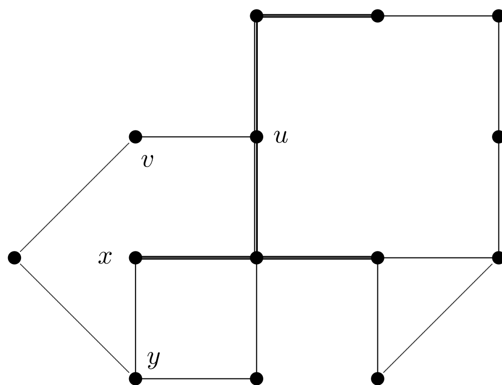
Bazni slučaj je trivijalan. Ako je zadat problem MCST sa m grana, kako se on može svesti na problem sa manje od m grana? Tvrdimo da grana najmanje cene mora biti uključena u minimalno povezujuće drvo. Ako ona ne bi bila uključena, onda bi njeno dodavanje minimalnom povezujućem drvetu zatvorilo neki ciklus;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

Problem je u tome što posle uklanjanja grane, dobijeni problem nije ekvivalentan polaznom. Prvo, izbor jedne grane ograničava mogućnosti izbora drugih grana. Drugo, posle uklanjanja grane graf ne mora da ostane povezan.

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 drvo, i da tu činjenicu, njen izbor, koristimo dalje u algoritmu. Algoritam se izvršava tako što se jedna po jedna grana bira i dodaje u minimalno povezujuće drvo. Prema tome, indukcija je ne prema veličini grafa, nego prema broju *izabranih grana* u zadatom (fiksiranom) grafu.

Induktivna hipoteza: Za zadati povezan graf $G = (V, E)$ umemo da pronađemo 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 MCST. Zbog toga u MCST 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 MCST. Označimo tu granu sa (u, v) (videti sliku 4; grane drveta T su podebljane). Pošto je MCST povezujuće drvo, ono sadrži tačno jedan put od u do v (između svaka dva čvora u drvetu postoji tačno jedan put). Ako grana (u, v) ne bi pripadala MCST, 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 dodamo (u, v) drvetu MCST, a izbacimo (x, y), dobijamo povezujuće drvo manje cene, što je kontradikcija.



Slika 4: Nalaženje sledeće grane minimalnog povezujućeg drveta.

Opisani algoritam poznat je pod nazivom *Primov algoritam* i sličan je Dajkstrinom algoritmu za nalaženje najkraćih puteva od zadatog čvora. Prva izabrana grana je grana sa najmanjom cenom. T se definiše kao drvo sa samo tom jednom granom. U svakoj iteraciji pronalazi se grana koja povezuje T sa nekim čvorom van T, a ima najmanju cenu. U algoritmu za nalaženje najkraćih puteva od zadatog čvora tražili smo najkraći put do čvora van T. Prema tome, jedina razlika između MCST algoritma i algoritma za nalaženje najkraćih puteva je

u tome što se minimum traži ne po dužini puta, nego 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 ∞ ako takva grana ne postoji. U svakoj iteraciji mi biramo granu najmanje cene i povezujemo odgovarajući čvor w sa drvetom T. Zatim proveravamo sve grane susedne čvoru w. Ako je cena neke takve grane (w, z) (za $z \notin T$) manja od cene trenutno najjeftinije poznate grane do z, onda popravljamo 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 {{{1,3}, {2,2}}, {{3,1}, {4,2}},
            {{5,3}}, {}, {{6,1}, {7,3}}, {{8,4}}, {}, {}, {}};

// 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 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 grana do svih cvorova
  priority_queue<rastojanjeDoCvora,vector<rastojanjeDoCvora>,
    greater<rastojanjeDoCvora>> rastojanja;

  // ubacujemo polazni cvor u hip
  // i postavljamo duzinu grane do njega na 0
  rastojanja.push(make_pair(0,0));
  najkracaGrana[0] = 0;

  // duzine grana do 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
  // 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 = 1; i < brojCvorova; i++)
    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, $O((|E| + |V|) \log |V|)$.

Primer: Algoritam za konstrukciju MCST ilustrovaćemo primerom na slici 5. Čvor u prvoj koloni tabele je onaj koji je dodat u odgovarajućem koraku. Prvi dodati čvor je v, i u prvoj vrsti navedene su sve grane iz 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 su grane grafa koje pripadaju MCST podebljane.

	v	a	b	c	d	e	f	g	h
v	–	$v(1)$	$v(6)$	∞	$v(9)$	∞	∞	∞	∞
a	–	–	$v(6)$	$a(2)$	$v(9)$	∞	∞	∞	∞
c	–	–	$v(6)$	–	$c(4)$	∞	$c(10)$	∞	∞
d	–	–	$v(6)$	–	–	$d(7)$	$c(10)$	$d(12)$	∞
b	–	–	–	–	–	$b(3)$	$c(10)$	$d(12)$	∞
e	–	–	–	–	–	–	$c(10)$	$d(12)$	$e(5)$
h	–	–	–	–	–	–	$c(10)$	$h(11)$	–
f	–	–	–	–	–	–	–	$h(11)$	–
g	–	–	–	–	–	–	–	–	–

Slika 5: Primer izvršavanja Primovog algoritma za nalaženje MCST.

Algoritam za konstrukciju minimalnog povezujućeg drveta je takođe primer pohlepnog algoritma, jer se u svakom koraku uzimaju elementi sa najmanjom cenom. U algoritmu MCST uveli smo neka dodatna ograničenja pri izboru grana: razmatrane su samo grane povezane sa tekućim drvetom. Zbog toga algoritam nije čisti primer pohlepnog algoritma.