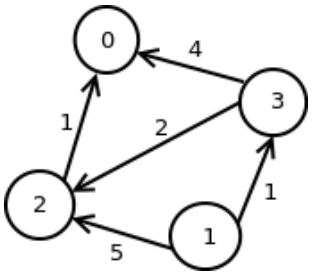
Grafovski algoritmi - čas 5

Najkraći putevi iz zadatog čvora

U ovom odeljku bavićemo se *težinskim grafovima*. Neka je $G = (V, E)$ usmereni graf sa nenegativnim *težinama* pridruženim granama. Ovde ćemo težine zvati dužinama, pošto ćemo pod terminom *dužina puta* razmatrati zbir dužina grana (a ne broj grana na tom putu). U C++-u ćemo težinski graf implementirati korišćenjem lista povezanosti kao `vector<vector<pair<int,int>>>`, gde uređeni parovi predstavljaju par indeksa čvora do koga vodi grana i njene težine. 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. Prema tome, algoritmi koje ćemo razmatrati se odnose i na neusmerene grafove.

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.

Na početku ćemo se baviti samo nalaženjem dužina najkraćih puteva (a ne i samih najkraćih puteva).

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).

Aciklički slučaj

Pretpostavimo najpre da je graf G aciklički. U tom slučaju problem je lakši i njegovo rešenje pomoći će nam da ga rešimo u opštem slučaju. Pokušaćemo 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 jednak k, onda se čvorovi sa rednim brojevima manjim od k ne moraju razmatrati: ne postoji način da se do njih dođe iz v. Pored toga, redosled dobijen topološkim sortiranjem je pogodan za primenu indukcije. Posmatrajmo poslednji čvor, odnosno čvor z sa rednim brojem n. Pretpostavimo (induktivna hipoteza) da znamo najkraće puteve od v do svih ostalih čvorova, sem do z. Označimo dužinu najkraćeg puta od v do w sa $w.SP$ (eng. shortest path). Da bismo odredili $z.SP$, dovoljno je da proverimo samo one čvorove w iz kojih postoji grana do 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 z, 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 $z.SP$.

Sada ćemo pokušati da usavršimo algoritam tako da se topološko sortiranje obavlja istovremeno sa nalaženjem 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 brojem manjim od rednog broja 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 do 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 v 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.

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 do w, pa je dužina najkraćeg puta 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, jer se do njih ne može doći iz z. Poboljšani algoritam prikazan je u nastavku.

```
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;

}

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 njegovog 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 numerisemo narednim brojem
    int cvor = cvoroviStepenaNula.front();
    cvoroviStepenaNula.pop();
    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;
}
```

Svaka grana se po jednom razmatra u toku inicijalizacije ulaznih stepenova, i po jednom u trenutku kad se njen polazni čvor uklanja iz liste. Pristup listi zahteva konstantno vreme. Svaki čvor se razmatra tačno jednom. Prema tome, vremenska složenost algoritma u najgorem slučaju je $O(|V| + |E|)$.

Dajkstrin algoritam

Kad graf nije aciklički, ne postoji topološki redosled, pa se razmatrani algoritmi ne mogu direktno primeniti. Međutim, osnovne ideje se mogu iskoristiti i u opštem slučaju, kada su dužine grana pozitivne. Jednostavnost razmotrenih algoritama 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.

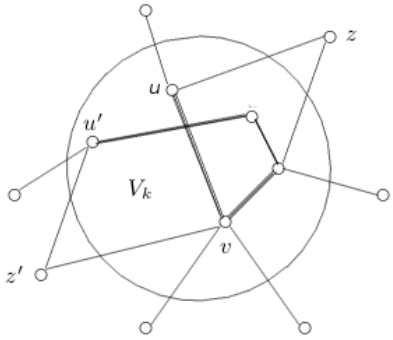
Ova osobina omogućuje nam da nađemo najkraći put od v do z, ne vodeći računa o čvorovima koji su posle z u topološkom redosledu. Može li se nekako definisati redosled čvorova proizvoljnog grafa (koji nije nužno aciklički) koji bi omogućio nešto slično?

Ideja je razmatrati čvorove grafa redom prema dužinama najkraćih puteva do njih od v. Te dužine se na početku, naravno, ne znaju; one se izračunavaju u toku izvršavanja algoritma. Najpre proveravamo sve grane koje izlaze iz 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 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 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) grana 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.



Slika 2: Nalaženje sledećeg najbližeg čvora zadatom čvoru v.

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

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

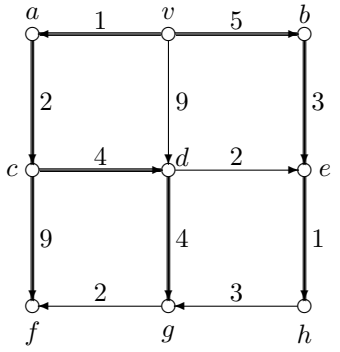
među svim čvorovima $w \notin V_k$. Iz već iznetih razloga, w je zaista $(k + 1)$-vi (sledeći) najbliži čvor čvoru v. Prema tome, njegovo dodavanje produžuje induktivnu hipotezu.

Algoritam je sada u potpunosti preciziran, ali mu se efikasnost može poboljšati. Osnovni korak algoritma je pronalaženje sledećeg najbližeg čvora. To se ostvaruje izračunavanjem 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: 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 V_k. Jedini način da se dobije novi najkraći put posle dodavanja 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 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.

Primer: Primer izvršavanja Dajkstrinog algoritma za nalaženje najkraćih puteva od čvora v u grafu dat je na slici 3. Prva vrsta odnosi se samo na puteve od jedne grane iz v. Bira se najkraći put, u ovom slučaju on vodi ka čvoru a. Druga vrsta pokazuje popravke dužina puteva uključujući sada sve puteve od jedne grane iz v ili a, i najkraći put sada vodi do c. U svakoj liniji bira se novi 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.

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 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

jedne jednake ∞, pa redosled elemenata u hipu nije bitan, sem što v mora biti na vrhu. Nalaženje čvora w 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 u u hipu. Prema tome, potrebno je na odgovarajući način popravljati hip. Problem sa ovim popravkama je u tome što hip kao struktura podataka ne omogućuje 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 i novom vrednošću rastojanja. 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 ranije bio obrađen. Ostaje bojazan da ovakva implementacija neće ugroziti 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 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.

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 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}}};

// 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;
}

// Dajkstrin algoritam za odredjivanje najkracih puteva
// do svih cvorova iz zadatog cvora
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);

  // 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;
    int duzinaPuta = najblizi.first;
    // postavljamo vrednost najkraceg puta do njega
    // kroz do sada posecene cvorove
    najkraciPut[cvor] = duzinaPuta;
    // ako cvor nije bio do sada posecen, postavljamo
    // informaciju da smo ga sada posetili
    if (!posecen[cvor]){
      posecen[cvor] = true;
```

```
      // stampamo i 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));
             }
          }
       }
    }
    // inace se radi o vec posecenom cvoru,
    // odnosno o nekoj prethodnoj (vecoj) vrednosti puta
    // do vec obradjenog cvora, te taj cvor ne treba ponovo brojati
    else
      i--;

  }
}

int main(){
  najkraciPuteviDajkstra();
  return 0;
}
```

Složenost: Popravka dužine puta zahteva $O(\log m)$ upoređivanja, gde je m veličina hipa. Ukupno može biti maksimalno $\max\{|V|, |E|\}$ brisanja iz hipa (u varijanti sa čuvanjem kopija). Popravke treba izvršiti najviše $|E|$ puta (jer svaka grana može da prouzrokuje najviše jednu popravku), pa je potrebno izvršiti najviše $O(|E| \log |V|)$ upoređivanja u hipu. Prema tome, vremenska

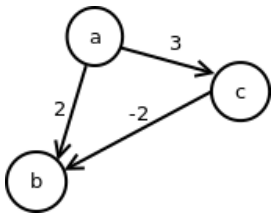
složenost algoritma je $O((|E| + |V|) \log |V|)$. Zapaža se da je algoritam sporiji nego algoritam koji isti problem rešava za acikličke grafove: u drugom slučaju sledeći čvor se uzima iz (proizvoljno uređene) liste, i nikakve popravke nisu potrebne.

Ukoliko bi se umesto binarnog hipa koristio Fibonačijev hip, amortizovana složenost operacije umetanja novog čvora bila bi $O(1)$ umesto $O(\log |E|)$ pa bi ukupna složenost Dajkstrinog algoritma bila $O(|E| + |V| \log |V|)$.

Ovakav tip algoritma se ponekad zove *pretraga sa prioritetom* — svakom čvoru dodeljuje se prioritet (u ovom slučaju trenutno najmanje poznato rastojanje od v), pa se čvorovi obilaze redosledom određenim 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.

Belman-Fordov algoritam

Ukoliko graf sadrži neku negativnu granu, Dajkstrin algoritam ne mora da vrati tačan rezultat. Na primer, ako bismo želeli da odredimo najkraće puteve iz čvora a do svih ostalih čvorova u grafu sa slike 4 i primenimo Dajkstrin algoritam on bi najpre odredio najkraći put do čvora b (kao čvora koji je najbliže čvoru a) i proglasio bi da je on dužine 2, a nakon toga bi odredio najkraći put do drugog najbližeg čvora c i proglasio bi da je dužina najkraćeg puta do njega 3 i putevi do ova dva čvora se ne bi dalje razmatrali. 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 4: Primer grafa za koji Dajkstrin algoritam ne računa dobro najkraća rastojanja.

Ukoliko graf ima negativne težine grana, ali ne sadrži ciklus negativne težine, najkraći putevi iz datog čvora se mogu odrediti Belman-Fordovim algoritmom.

Osnovni korak u Belman-Fordovom algoritmu je *relaksacija* grane (u, v), odnosno provera da li se vrednost $v.SP$ može zameniti manjom vrednošću $u.SP + duzina(u, v)$; ako je to tačno, popravlja se vrednost $v.SP$ i pamti da najkraći

put vodi kroz čvor u. Ideja je da se $|V| - 1$ put izvrši relaksacija svake od grana u grafu. Tvrdimo da će nakon toga sva rastojanja biti korektno određena.

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

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

  int brojCvorova = listaSuseda.size();
  // 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 njegovog prethodnika u
  // najkracem putu
  vector<int> roditelj(brojCvorova,-1);

  najkraciPut[0] = 0;

  // |V|-1 put prolazimo kroz skup svih grana
  for (int br=0; br<brojCvorova-1; br++)
    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 puta
            if (najkraciPut[i]+grana < najkraciPut[sused]){
                najkraciPut[sused] = najkraciPut[i]+grana;
                // postavljamo koji cvor je pretposlednji na tom putu
                roditelj[sused] = i;
            }
        }

    // ukoliko i dalje postoji put koji je moguce skratiti
    // vazi da graf sadrzi ciklus negativne tezine
    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 sadrzi ciklus negativne tezine" << 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;
}
```

Neka je n broj čvorova u grafu. Označimo sa $d_k(i)$ vrednost promenljive `najkraciPut[i]` nakon izvršenja k iteracija spoljašnje `for` petlje. Dokažimo dva trđenja u vezi ovog algoritma:

1. ukoliko postoji ciklus negativne dužine koji je dostižan iz početnog čvora v, onda za neku granu (u, w) važi $d_{n-1}(w) > d_{n-1}(u) + duzina(u, w)$
2. ako graf 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

Pokažimo najpre prvo tvrđenje. Pretpostavimo da G sadrži ciklus negativne dužine $c = (v_0, v_1, \ldots, v_k), v_k = v_0$, dostižan iz v. Tada je:

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

Pretpostavimo suprotno, da važi $d_{n-1}(v_i) \leq d_{n-1}(v_{i-1}) + duzina(v_{i-1}, v_i)$ za svako $i = 1, 2, \ldots, k$. Sabiranjem ovih nejednakosti za sve grane ciklusa dobija se:

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

Pošto je $v_0 = v_k$, zbirovi $\sum_{i=1}^{k} d_{n-1}(v_i)$ i $\sum_{i=1}^{k} d_{n-1}(v_{i-1})$ su jednaki, te važi:

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

suprotno pretpostavci da čvorovi $v_0, v_1, \ldots, v_k$ formiraju ciklus negativne dužine.

Pokažimo sada da ako graf ne sadrži ciklus negativne dužine, da će vrednosti $d_{n-1}(u)$ sadržati vrednosti 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. Ako ovo važi, onda će $d_{n-1}(u)$ biti minimalna dužina 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 ne sadrži ponovljene čvorove, te će se sastojati od maksimalno $n-1$ grana.

Bazni slučaj (za $k = 0$) je jednostavan, važi da je $d_k(u) = 0$ ako je $u = v$, a inače je $d_k(u) = \infty$. Stoga tvrđenje važi jer put dužine 0 postoji samo od nekog čvora do njega samog i dužine je 0.

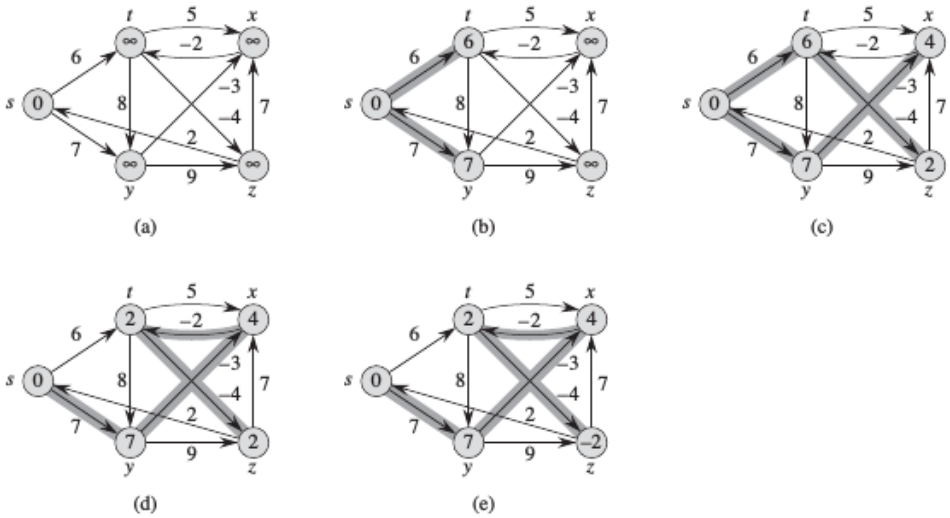
Induktivna hipoteza: Pretpostavimo da za sve čvorove u važi da je $d_{k-1}(u)$ minimalna dužina puta od v do u koji se sastoji od najviše $k-1$ grana.

Ako je $u \neq v$, neka je P najkraći put od v do u sa najviše k grana i neka je w čvor neposredno ispred u na putu P. Označimo sa Q put od v do w. 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 $d_{k-1}(w)$.

U k-toj iteraciji spoljašnje petlje postavljamo da je $d_k(u) = \min\{d_{k-1}(u), d_{k-1}(w) + duzina(w, u)\}$. Znamo da važi $d_{k-1}(w) + duzina(w, u) = duzina(Q) + duzina(w, u) = duzina(P)$, pa stoga važi da je $d_k(u) \leq duzina(P)$. Takođe, s obzirom na to da je $d_{k-1}(u)$ dužina minimalnog prostog puta od v do u koji se sastoji od najviše $k-1$ grana, $d_{k-1}(u)$ mora biti barem jednako veliko kao $duzina(P)$, s obzirom na to da P radi nad većim brojem grana.

Stoga važi $d_k(u) = duzina(P)$, odnosno $d_k(u)$ je minimalna dužina puta od čvora v do čvora u koji koristi najviše k grana.

Složenost algoritma je $O(|V||E|)$, zbog dve umetnute `for` petlje.



Slika 5: Primer izvršavanja Belman-Fordovog algoritma. Vrednosti d_k prikazane su unutar čvorova, a šrafirane grane vode ka prethodnicima čvorova na (trenutno) najkraćim putevima: ako je grana (u, v) šrafirana, onda se do čvora v najkraćim putem dolazi preko čvora u. U primeru u svakom prolazu se grane relaksiraju u sledećem redosledu: $(t, x), (t, y), (t, z), (x, t), (y, x), (y, z), (z, x), (z, s), (s, t), (s, y)$. (a) Situacija pre prvog prolaska kroz grane. (b)-(e) Situacija nakon svakog narednog prolaska kroz grane.

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 cena mreže bude minimalna. Sistem računara može biti predstavljen grafom čiji čvorovi odgovaraju računarima, a grane – potencijalnim vezama između računara, sa odgovarajućom (pozitivnom) cenom. Problem je pronaći povezani podgraf (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 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. 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.

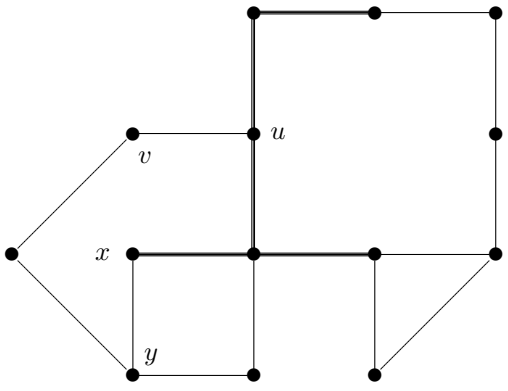
Bazni slučaj je trivijalan. Ako je zadat problem MCST sa m grana, kako se on može svesti na problem sa manje od m grana? Tvrdimo da grana najmanje cene mora biti uključena u minimalno povezujuće drvo. Ako ona ne bi bila uključena, onda bi njeno dodavanje minimalnom povezujućem drvetu zatvorilo neki ciklus; 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, preostali problem više 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 je označimo, 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 povezjuće 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 6; 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 pripada MCST, onda ona ne pripada ni tom putu od u do v. Međutim, pošto u pripada, a v ne pripada T, na tom putu mora da postoji bar 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 6: Nalaženje sledeće grane minimalnog povezujućeg drveta.

Opisani algoritam poznat je pod nazivom *Primov algoritam* i sličan je Dajkstri-

nom 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 na taj način 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.

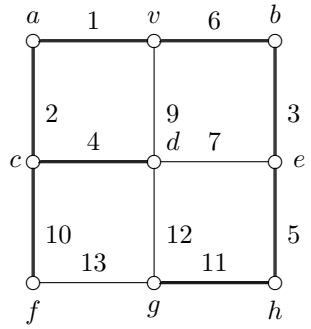
```
// uredjen par vrednosti rastojanja do cvora i indeksa cvora
// vazan je redosled komponenti zbog operacije poredjenja po rastojanju
typedef pair<int,int> rastojanjeDoCvora;

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 najkraceg grane koja vodi do njega
  vector<int> najkraciPut(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 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;
  int duzinaPuta = najblizi.first;
  // postavljamo vrednost najkraceg puta do njega
  // kroz do sada posecene cvorove
  najkraciPut[cvor] = duzinaPuta;
  // ako cvor nije bio do sada posecen, postavljamo
  // informaciju da smo ga sada posetili
  if (!posecen[cvor]){
    posecen[cvor] = true;
  // 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 (duzinaGrane < najkraciPut[sused]){
          najkraciPut[sused] = 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));
        }
      }
    }
  }
  // inace se radi o nekoj prethodnoj (vecoj) vrednosti
  // puta do vec obradjenog cvora, taj cvor ne treba ponovo brojati
  else
    i--;
}
cout << "Minimalno povezujuce drvo se sastoji od grana: " << endl;
for(int i=1; i<brojCvorova; i++)
  cout << "(" << roditelj[i] << "," << i << ") tezine "
```

```
        << najkraciPut[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 7. 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 7: Primer izvršavanja Primovog algoritma za nalaženje MCST.

Algoritam za konstrukciju minimalnog povezujućeg drveta je takođe primer pohlepnog metoda. Pretpostavimo da radimo sa skupom elemenata kojima su pridružene cene, i da je cilj pronaći skup elemenata se maksimalnom (ili minimalnom) cenom koji zadovoljava neka ograničenja. U problemu MCST elementi su grane grafa, a ograničenje je da grane odgovaraju povezujućem drvetu. Suština pohlepnog metoda je u tome da se u svakom koraku uzimaju elementi sa najvećom 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.

Kruskalov algoritam

Drugi efikasan algoritam za određivanje minimalnog povezujućeg drveta grafa $G = (V, E)$ je takođe pohlepan, ali do minimalnog povezujućeg drveta ne dolazi dodavanjem novih grana na trenutno drvo, nego na trenutnu šumu. Dodavanjem svake nove grane smanjuje se broj drveta u šumi, tako da se na kraju dolazi do

samo jednog drveta. Pri tome se grane za dodavanje razmatraju redosledom prema rastućim cenama; ako grana koja je na redu povezuje dva čvora u različitim drvetima trenutne šume, onda se ta grana uključuje u šumu, čime se dva drveta spajaju u jedno. U protivnom, ako grana povezuje dva čvora u istom drvetu trenutne šume, grana se preskače. Ovaj algoritam poznat je pod nazivom *Kruskalov algoritam*.

Dokažimo da ćemo na kraju ovog algoritma dobiti minimalno povezujuće drvo datog grafa. Jednostavnosti radi pretpostavimo da su sve cene grana različite. Lako se pokazuje da će algoritam vratiti povezujuće drvo datog grafa, te ćemo samo pokazati da je ono i minimalno. 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 nam grane sortirane u rastućem redosledu svojih cena. S obzirom da je $K \neq T$ 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 da Kruskalov algoritam razmatra grane u ovom redosledu i ne dodaje samo one koje zatvaraju neki ciklus, mora važiti da grana e pripada K, a ne pripada T. U minimalnom povezujućem drvetu T mora postojati jedinstveni put P od čvora a do čvora b. Na tom putu mora da postoji bar jedna grana e' čija je cena veća od cene grane e (ako to ne bi važilo sve ostale grane bi bile uključene Kruskalovim algoritmom u drvo K, kao i e te bi suprotno pretpostavci K sadržalo ciklus). 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.

Da se ustanovi da li su krajevi u, v trenutne grane (u, v) u istom ili različitim drvetima trenutne šume, pogodno je koristiti strukturu podataka za disjunktne skupove: trenutna drveta šume su disjunktni podskupovi skupa čvorova. Operacije `podskup(u)` i `podskup(v)` pronalaze predstavnike u', v' dva podskupa (korene drveta kojim su oni predstavljeni), pa su 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')`.

```
// 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) {
  for (int i = 0; i < n; i++) {
    roditelj[i] = i;
    rang[i] = 0;
  }
```

```
}

// funkcija koja izracunava kom skupu 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 skupa
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 minimalnoPovezujuceDrvoKruskal(){

  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> najkraciPut(brojCvorova,numeric_limits<int>::max());
  // niz koji za svaki cvor cuva roditelja u minimalnom povezujucem drvetu
  vector<int> roditelj(brojCvorova);
  // niz koji za svaki cvor cuva njegov rang
  vector<int> rang(brojCvorova);

  // inicijalizujemo union-find strukturu
  inicijalizuj(roditelj, rang, brojCvorova);

  // inicijalizujemo skup svih grana
```

```
  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 neopadajucem poretku 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 MCST
    // i pravimo uniju skupa cvorova koji pripadaju tim drvetima
    if (skup_u != skup_v){
      roditelj[v] = u;
      najkraciPut[v] = duzina;
      unija(skup_u, skup_v, roditelj, rang);
      brojGrana++;
    }
  }

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

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

```
}
```

Funkcija za inicijalizaciju 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 algoritma je $O(|V|) + O(|E|) + O(|E|\log|E|) + O(|E|\log|V|) = O(|E|\log|V|)$ (koristimo činjenicu da je $O(\log|E|) = O(\log|V|)$ zbog $|E| \leq |V|^2$).

grana	dužina	uključena?	šuma
			$v, a, b, c, d, e, f, g, h$
av	1	da	$\underline{a}v, b, \underline{c}, d, e, f, g, h$
ac	2	da	$acv, \underline{b}, c, d, \underline{e}, f, g, h$
be	3	da	$a\underline{c}v, be, \underline{d}, e, f, g, h$
cd	4	da	$acdv, b\underline{e}, f, g, \underline{h}$
eh	5	da	$acd\underline{v}, \underline{b}eh, f, g$
bv	6	da	$abc\underline{de}hv, f, g$
de	7	ne	$abc\underline{d}eh\underline{v}, f, g$
dv	8	ne	$ab\underline{c}dehv, \underline{f}, g$
cf	9	da	$abcdef\underline{h}v, \underline{g}$
gh	10	da	$abc\underline{d}ef\underline{g}hv$
dg	11	ne	$abcde\underline{fg}hv$
fg	12	ne	$abcdefghv$

Table 1: Primer izvršavanja Kruskalovog algoritma za graf sa slike 7

Primer izvršavanja Kruskalovog algoritma na grafu sa slike 7 prikazan je u tabeli 1. S obzirom na to da su dužine svih grana različite, rezultat je isto minimalno povezujuće drvo prikazano na slici 7.