Grafovi

Osnovni algoritmi sa grafovima

U ovom poglavlju će biti predstavljene metode predstavljanja i pretraživanja grafova. Pretraživanja grafa podrazumeva sistematično kretanje vezama grafa, tako da se obiđu određeni čvorovi. Algoritmi za pretraživanje grafova mogu dati dosta odgovora u vezi sa strukturom grafa. Mnogi algoritmi sa grafovima zapravo počinju pretraživanjem ulaznog grafa kako bi se dobile informacije o njegovoj strukturi. Ostali algoritmi sa grafovima najčešće predstavljaju samo kombinovanje osnovnih algoritama za pretraživanje grafova. Samim tim, algoritmi za pretraživanje grafova čine srž ove oblasti.

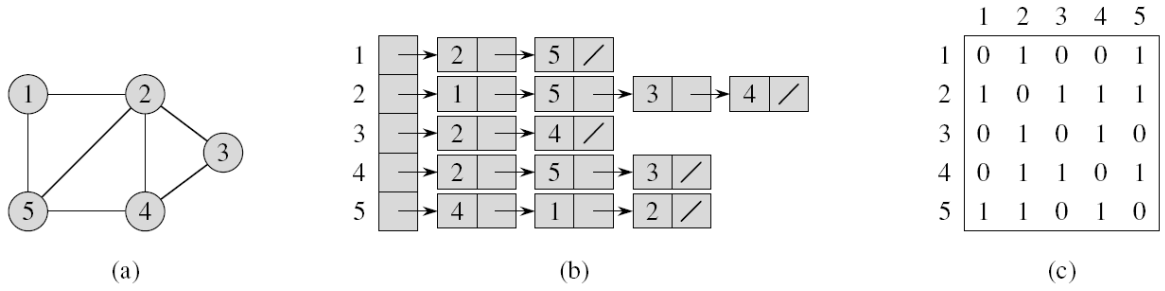
Predstavljanje grafova

Postoje dva standardna načina za predstavljanje grafa $G=(V,E)$:

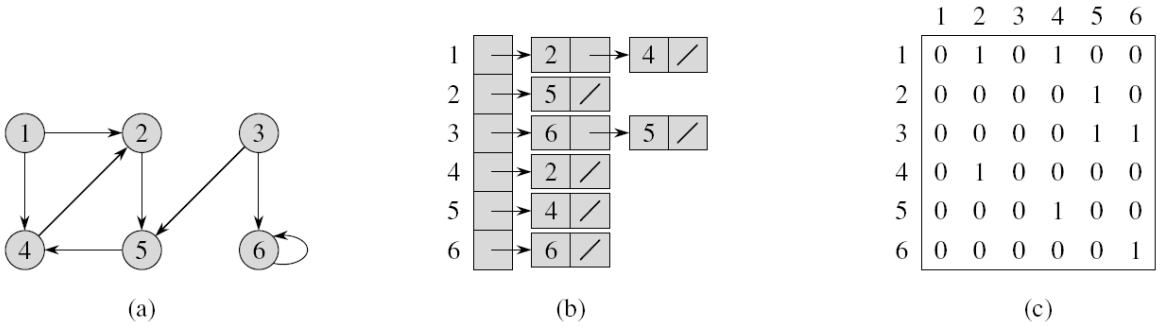
- Kolekcija lista povezanosti
- Matrica povezanosti

I jedan i drugi način su primenjivi i na usmerene i na neusmerene grafove. Predstavljanje grafa pomoću lista povezanosti omogućava kompaktno čuvanje **retkih** (razuđenih) grafova, kod kojih je broj veza $|E|$ mnogo manji od kvadrata broja čvorova $|V|^2$. Korišćenje matrice povezanosti se sa druge strane koristi u slučaju **gustih** grafova, gde je broj veza $|E|$ približan broju čvorova $|V|^2$ ili kada je potrebno brzo doći do informacije da li između neka dva čvora postoji veza.

Predstavljanje grafa $G=(V,E)$ pomoću **lista povezanosti** sastoji se od niza Adj, koji sadrži $|V|$ lista, po jednu za svaki čvor iz skupa V. Za svaki čvor $u \in V$, lista povezanosti $Adj[u]$ sadrži sve čvorove v takve da postoji veza $(u,v) \in E$. To zapravo znači da lista $Adj[u]$ sadrži sve čvorove grafa G koji su povezani sa čvorom u. Čvorovi u listi povezanosti su obično poređani proizvoljnim redosledom. Na Slici ###b je prikazano predstavljanje neusmerenog grafa sa Slike ###a pomoću lista povezanosti. Slično, na Slici ###b je dato predstavljanje usmerenog grafa sa Slike ###a.



Slika ###. Dva načina predstavljanja neusmerenog grafa: a) Neusmereni graf od 5 čvorova i 7 veza; b) Predstavljanje pomoću lista povezanosti; c) Predstavljanje pomoću matrice povezanosti.



Slika ###. Dva načina predstavljanja usmerenog grafa: a) Neusmereni graf od 6 čvorova i 8 veza; b) Predstavljanje pomoću lista povezanosti; c) Predstavljanje pomoću matrice povezanosti.

Ako je G usmeren graf, suma dužina svih lista povezanosti je $|E|$, s obzirom da je svaka veza (u,v) predstavljena jednim elementom u listi $Adj[u]$. Ukoliko je G neusmeren graf, suma dužina svih lista povezanosti je $2|E|$, pošto se za neusmerenu vezu (u,v) čvor u javlja u listi povezanosti čvora v i obrnuto. I za usmerene i za neusmerene grafove, prednost predstavljanja pomoću lista povezanosti je u tome što je potrebna količina memorije $\Theta(V+E)$. Liste povezanosti se lako mogu prilagoditi predstavljanju **težinskih grafova**, kod kojih svaka veza ima pridruženu težinu $w(u,v)$. Neka je, na primer, $G=(V,E)$ težinski graf sa težinskom funkcijom w. Težina $w(u,v)$ veze $(u,w)\in E$ se jednostavno smešta zajedno sa čvorom v u listu povezanosti čvora u. Predstavljanje grafa pomoću lista povezanosti je prilično robustno i lako može biti modifikovano tako da omogući predstavljanje različitih varijanti grafova.

Jedan od nedostataka predstavljanja pomoću lista povezanosti je nemogućnost da se brzo odredi da li u datom grafu postoji veza između čvorova u i v, već je potrebno pretražiti da li u listi $Adj[u]$ postoji čvor v. Ovaj nedostatak može biti prevaziđen predstavljanjem grafa pomoću matrice povezanosti, na račun povećanog korišćenja memorije.

Kod predstavljanja grafa $G(V,E)$ pomoću matrice povezanosti, pretpostavljamo da su čvorovi označeni brojevima 1, 2,..., $|V|$ na proizvoljan način. Onda graf G može predstaviti $|V|\times|V|$ matricom $A=(a_{ij})$, takvom da je

$$a_{ij}=\begin{cases}1 & \text{ako } (i,j)\in E\\ 0 & \text{ako } (i,j)\neq E\end{cases}$$

Na slikama ###c i ###c su prikazane matrice povezanosti neusmerenog i usmerenog grafa datih na slikama ###a i ###a, redom. Matrica povezanosti grafa zahteva $\Theta(V^2)$ memorije, nezavisno od broja veza u grafu.

Ako posmatramo matricu povezanosti sa Slike ###c, primetićemo simetriju oko glavne dijagonale matrice. Definišimo **transponovanu** matricu matrice $A=(a_{ij})$ kao $A^T=(a_{ij}^T)$, pri čemu je $a_{ij}^T=a_{ji}$. Pošto je u pitanju neusmereni graf, (u,v) i (v,u) predstavljaju jednu istu vezu, tako da je matrica povezanosti A neusmerenog grafa jednaka svojoj transponovanoj matrici, tj. $A=A^T$. U nekim aplikacijama je isplativo u memoriji čuvati samo gornju polovinu matrice, čime se količina potrebne memorije skoro prepolovi.

Kao i u slučaju predstavljanja pomoću lista povezanosti, matrica povezanosti se takođe može koristiti kod težinskih grafova. Na primer, ako je graf $G=(V,E)$ težinski, sa težinskom funkcijom w, težina $w(u,v)$ veze $(u,v)\in E$ se jednostavno može smestiti u red u i kolonu v matrice povezanosti. Ukoliko neka veza ne postoji, u matricu se na odgovarajuće mesto može smestiti NULL vrednost, a u velikom broju problema 0 ili ∞.

Iako je predstavljanje pomoću lista povezanosti efikasnije od predstavljanja matricom povezanosti, zbog jednostavnosti korišćenja matrice povezanosti ovaj način je pogodniji u slučaju relativno malih grafova. Štaviše, ukoliko graf nije težinski, postoji i dodatna prednost u korišćenju matrice povezanosti. Umesto da se koristi jedna reč za svaki element matrice, moguće je koristiti samo jedan bit po elementu.

Pretraga grafa po širini

Pretraga grafa po širini (*breadth-first search*) je jedan od najjednostavnijih algoritama za pretragu grafova i predstavlja osnovu za mnogo važne algoritme. Primov algoritam za određivanje minimalnog stabla razapinjanja i Dijsktrin algoritam za određivanje najkraćeg puta od jednog čvora koriste ideje slične pretrazi grafa po širini.

Za dati graf $G=(V,E)$ i određeni **izvorni** čvor s, pretraga grafa po širini sistematski ispituje veze grafa G da bi odredio sve čvorove dostupne iz čvora s. Ovaj algoritam računa rastojanje (najmanji broj veza) od čvora s do svakog dostupnog čvora. Algoritam takođe formira širinsko stablo sa korenom s, koje sadrži sve dostupne čvorove. Za bilo koji čvor v dostupan iz čvora s, put u širinskom stablu od čvora s do čvora v odgovara "najkraćem putu" od s do v u grafu G, tj. put koji sadrži najmanji broj veza. Ovaj algoritam je primenjiv i na usmerene i na neusmerene grafove.

Pretraga grafa po širini se tako naziva zato što ona pomera granicu između ispitanih i neispitanih čvorova podjednako duž čitave granice. To zapravo znači da algoritam ispituje sve čvorove na rastojanju k od čvora s pre nego što ispita bilo koji čvor na rastojanju $k+1$.

Da bi vodio evidenciju o napredovanju, pretraga grafa po širini boji sve čvorove belom, sivom ili crnom bojom. Svi čvorovi su na početku beli i kasnije mogu postati sivi, a zatim i crni. Za čvor kažemo da je **ispitan** onda kada je prvi put uzet u razmatranje prilikom pretrage i tada mu se dodeljuje boja koja nije bela. Dakle, i sivi i crni čvorovi predstavljaju ispitane čvorove, ali ih ovaj algoritam deli na sive i crne, da bi obezbedio pretragu po širini. Crni čvorovi predstavljaju već ispitane čvorove čiji su svi susedi takođe ispitani, što znači da su svi susedi crnog čvora ili sivi ili crni. Sivi čvorovi mogu imati bele susedne čvorove, što znači da oni predstavljaju granicu između ispitanih i neispitanih čvorova.

Pretraga po širini konstruiše širinsko stablo, koje inicijalno sadrži samo čvor s kao koren. Kad god se beli čvor v ispita u okviru pretraživanja suseda već ispitanog čvora u, čvor v i veza (u,v) se dodaju u stablo. Tada kažemo da je čvor u **prethodnik** ili **roditelj** čvora v u širinskom stablu. S obzirom da se svaki čvor ispituje samo jednom, on ima samo jednog roditelja. Relacije između predaka i potomaka u širinskom stablu se definišu u odnosu na koren s, kao što je i uobičajeno: ako je u na putu od korena s do čvora v, onda je u predak čvora v, a v je potomak čvora u.

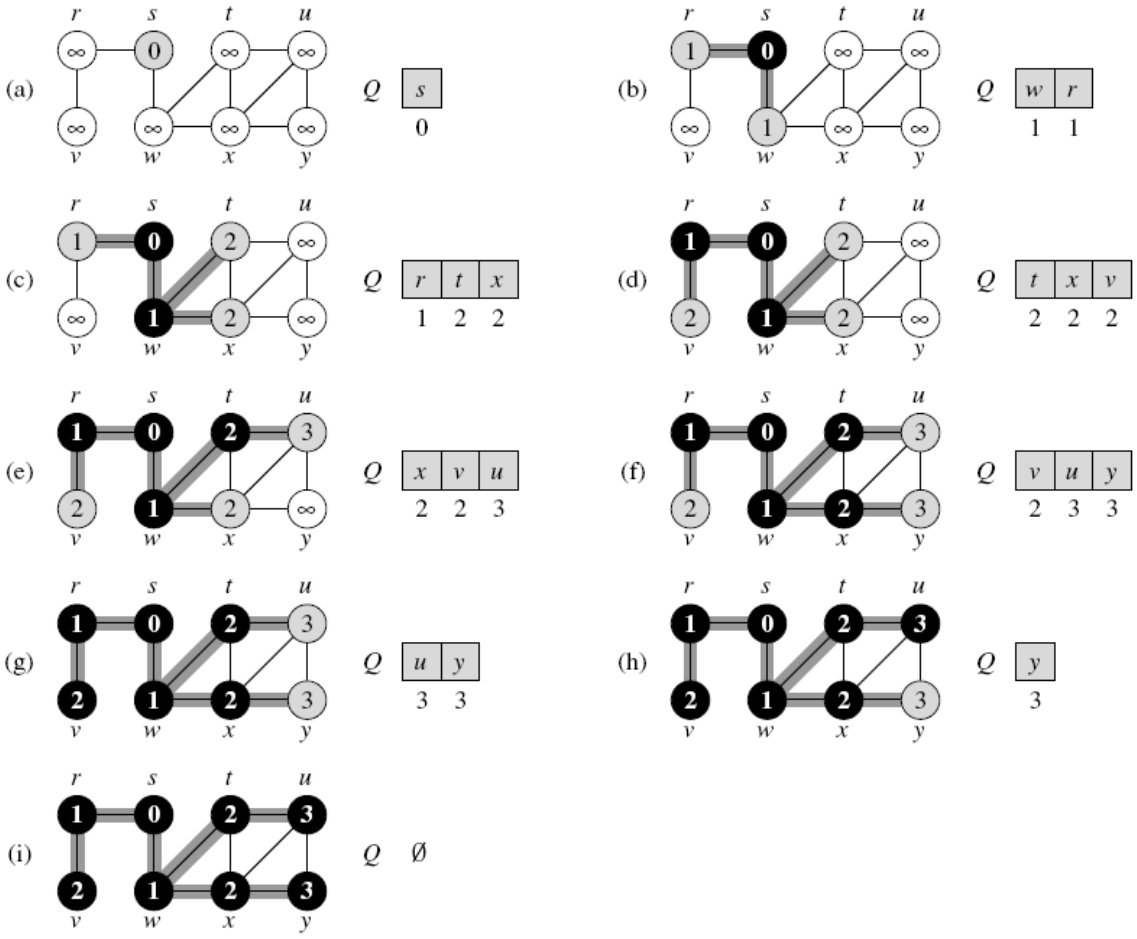
Prikazana procedura BFS vrši pretragu grafa po širini, pri čemu je pretpostavljeno da je ulazni graf $G=(V,E)$ predstavljen korišćenjem lista pripadnosti. Ova procedura koristi još nekoliko dodatnih struktura podataka za svaki čvor u grafu. Boja svih čvorova se čuvaju u nizu $color$, tako da se boja čvora u smešta u $color[u]$, a prethodnik čvora u se čuva u $\pi[u]$. Ukoliko čvor u nema prethodnika ($u=s$), ili još uvek nije ispitan, onda je $\pi[u]=NULL$. Rastojanje od čvora s do čvora v, koje se izračunava u okviru algoritma, čuva se u $d[u]$. Algoritam takođe koristi i FIFO listu Q za čuvanje sivih čvorova.

```
BFS( G , s )
   for each u ∈ V − {s}
      color[u] = WHITE
      d[u] = ∞
      π[u] = NULL
```

$color[s] = \text{GRAY}$
$d[s] = 0$
$\pi[s] = NULL$
$Q = NULL$
ENQUEUE(Q , s)

while $Q \neq 0$
  DEQUEUE(Q , u)
  for each $v \in Adj[u]$
    if $color[v] = \text{WHITE}$
      $color[v] = \text{GRAY}$
      $d[v] = d[u] + 1$
      $\pi[v] = u$
      ENQUEUE(Q , v)
  $color[u] = \text{BLACK}$

Na Slici ### je prikazan način funkcionisanja pretrage po širini na primeru neusmerenog grafa.



Slika ###. Izvršavanje BFS na neusmerenom grafu

Procedura BFS radi na sledeći način. Na početku se svi čvorovi osim izvornog čvora s boje u belu boju, $d[u]$ se za svaki čvor u postavlja na beskonačno, a prethodnik svakog čvora postavlja na NULL. Pošto algoritam kreće od čvora s, ovaj čvor se boji u sivo, pošto se smatra da je on ispitan. Dužina puta do čvora s je naravno $d[s]=0$, a pošto je u pitanju početni čvor, on nema prethodnika, pa je $\pi[s]=NULL$. U naredne dve linije se inicijalizuje lista Q i u nju se dodaje čvor s. **While** petlja iteriše sve dok ima sivih čvorova, koji zapravo predstavljaju čvorove koji su ispitani, ali čiji susedi još uvek nisu ispitani.

Unutar **while** petlje se sa liste skida prvi čvor u i pristupa se ispitivanju njegovih suseda v, koji još uvek nisu ispitani, tj. koji su bele boje. Svaki beli sused samim ovim postupkom postaje ispitan, tako da se on boji u sivo. Sada čvor u postaje prethodnik čvora v, a dužina puta do čvora v je jednaka dužini puta do njegovog prethodnika u uvećanoj za jedan. Na kraju se novoispitani čvor v dodaje na listu Q, kako bi kasnije bili ispitani njegovi susedi.

Kada se završi sa ispitivanjem svih suseda čvora u, ovaj čvor se boji u crno.

Širinsko stablo

Procedura BFS formira širinsko stablo na način prikazan na Slici ###. Stablo je predstavljeno vrednošću u nizu π za svaki čvor. Sledeća procedura štampa čvorove na najkraćem putu od čvora s do čvora v, pri čemu je pretpostavljeno da je procedura BFS već obavila izračunavanje stabla najkraćeg puta.

```
PRINT_PATH(G, s, v)
  if v = s
    print s
  else if π[v] = NULL
    print "Nema puta od s do v"
  else
    PRINT_PATH(G, s, π[v])
    print v
```

Ova procedura ima linearnu zavisnost vremena izvršavanja od broja čvorova na putu, pošto se svaki rekurzivni poziv odnosi na put koji je kraći za jedan čvor.

Pretraga grafa po dubini

Strategija prilikom **pretrage grafa po dubini** (*depth-first search*) podrazumeva traženje u dubinu grafa, kad god je to moguće. U pretrazi po dubini se pretražuju veze od poslednjeg ispitanog čvora v, sve dok ima neistraženih veza koje iz njega izlaze. Kada se ispitaju sve veze koje izlaze iz čvora v, pretraga se vraća jedan korak u nazad, kako bi se ispitale sve veze koje izlaze iz čvora iz koga se došlo do čvora v. Ovaj proces se nastavlja sve dok se ne ispitaju svi čvorovi dostupni iz izvornog čvora. Ukoliko preostane neki neispitan čvor, on se uzima za izvor i pretraga se ponavlja iz tog izvora. Čitav proces se ponavlja sve dok ima neispitanih čvorova.

Kao i kod pretrage po širini, kada se čvor v ispita prilikom pretrage susednih čvorova već ispitanog čvora u, pretraga po dubini beleži da je u prethodnik čvora v, odnosno da je $\pi[v]=u$.

Kao i kod pretrage po širini, čvorovi se tokom pretrage boje različitim bojama da bi se označio njihov status. Svi čvorovi su inicijalno beli, a zatim se boje u sivo kada su ispitani, odnosno u crno kada je njihova obrada u potpunosti završena, tj. kada je njihova lista suseda u potpunosti istražena. Ova tehnika garantuje da će svaki čvor završiti u samo jednom dubinskom stablu, tako da su sva stabla potpuno odvojena. Pored formiranja "dubinske šume", pretraga po dubini takođe vrši vremensko obeležavanje svih čvorova. Svaki čvor ima dva obeležja: $d[v]$ pamti kada je čvor v prvi put posećen (i obojen u sivo), a $f[v]$ beleži kada je završeno sa pretragom svih suseda čvora v (kada je obojen u crno). Ove oznake se koriste u mnogim algoritmima sa grafovima i u opštem slučaju su korisne pri tumačenju ponašanja pretrage po dubini.

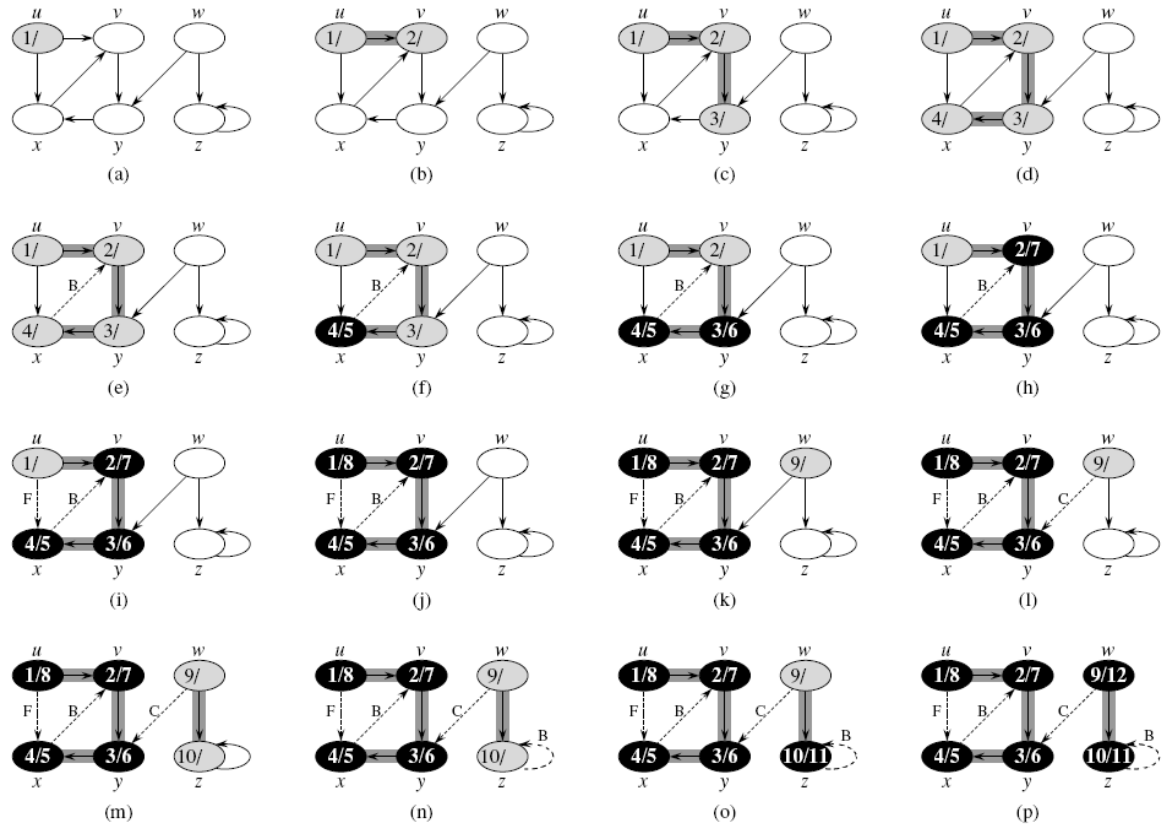
Procedura DFS pamti kada je ispitala čvor u u promenljivoj $d[u]$, a kada je završila sa čvorom u pamti u promenljivoj $f[u]$. Ove oznake su celobrojne vrednosti između 1 i $2|V|$, pošto se svaki od $|V|$ čvorova poseti samo jednom i napusti samo jednom. Za svaki čvor u važi da je $d[u] < f[u]$. Čvor u je beo pre trenutka $d[u]$, siv između trenutaka $d[u]$ i $f[u]$, a crn posle toga.

Sledeći pseudokod prikazuje osnovni algoritam pretrage po dubini. Ulazni graf G može biti usmeren ili neusmeren. Promenljiva $time$ je globalna promenljiva koja se koristi za vremensko obeležavanje čvorova.

```
DFS( G )
   for each u ∈ V
      color[u] = WHITE
      π[u] = NULL

   time = 0
   for each u ∈ V
      if color[u] = WHITE
         DFS_VISIT( u )


DFS_VISIT( u )
   color[u] = GRAY     // Cvor u je upravo ispitan
   time = time + 1
   d[u] = time

   for each v ∈ Adj[u]       // Ispitivanje veze (u,v)
      if color[v] = WHITE
         π[v] = u
         DFS_VISIT( v )
   color[u] = BLACK          // Cvor u je u potpunosti obradjen
   time = time + 1
   f[u] = time
```

Na Slici ### je prikazano izvršavanje procedure DFS na grafu sa Slike ###.

Procedura DFS funkcioniše na sledeći način. Na početku se svi čvorovi boje u belo, njihova π polja postavljaju na NULL. Globalno vreme se postavlja na nulu. Nakon toga se proveravaju svi čvorovi grafa i svaki beli čvor se posećuje korišćenjem procedure DFS_VISIT. Pri svakom pozivu DFS_VISIT(u), čvor u postaje koren stabla unutar dubinske šume. Nakon povratka u DFS, svaki čvor ima pridruženo **početno vreme** $d[u]$ i **završno vreme** $f[u]$.



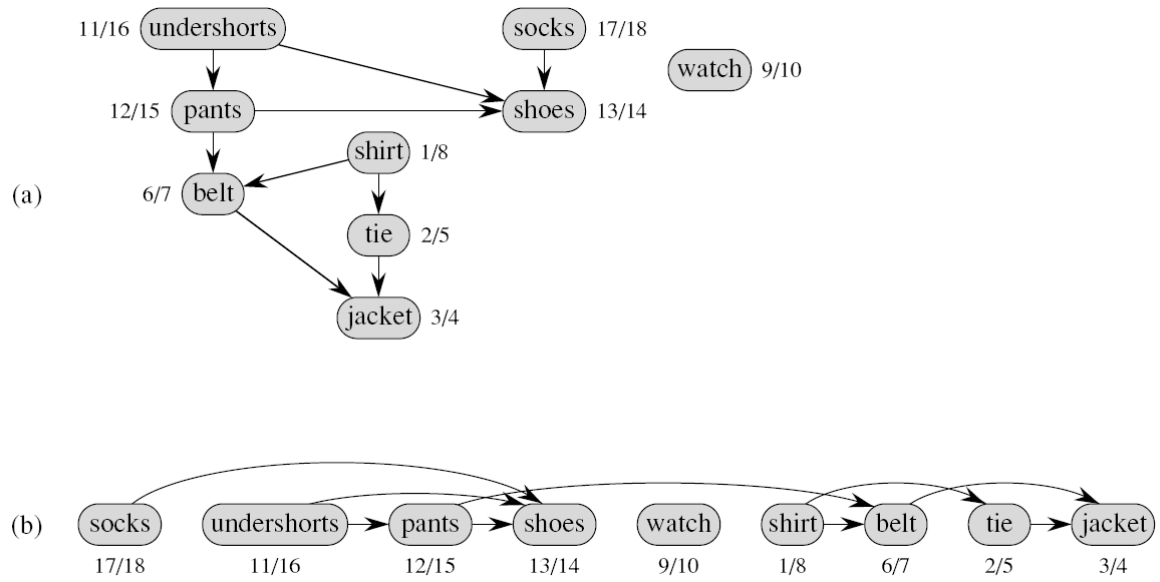
Slika ###. Izvršavanje pretrage po dubini na usmerenom grafu.

Pri svakom pozivu DFS_VISIT(u), čvor u je inicijalno beo. Čvor u se boji u sivo, globalno vreme se uvećava za jedan i ta vrednost se smešta u vreme posećivanja $d[u]$. U narednoj petlji se traže svi beli susedi čvora u i svaki od njih se rekurzivno posećuje. Konačno, kada su ispitani svi susedi čvora u, ovaj čvor se boji u crno i beleži se vreme njegovog napuštanja $f[u]$.

Topološko sortiranje

U ovom delu ćemo pokazati kako se pretraga po dubini može iskoristiti za topološko sortiranje usmerenog acikličnog grafa (često se na engleskom naziva DAG – *directed acyclic graph*). **Topološko sortiranje** usmerenog acikličnog grafa $G=(V,E)$ je linearno raspoređivanje svih njegovih čvorova, tako da ako graf G sadrži vezu (u,v), onda se u nizu čvor u javlja pre čvora v. Ukoliko graf nije usmeren, topološko sortiranje nije moguće. Topološko sortiranje grafa se može posmatrati kao raspoređivanje njegovih čvorova duž horizontalne linije, tako da sve veze budu usmerene sa leva na desno.

Usmereni aciklični grafovi se koriste u mnogim problemima, kako bi se prikazao redosled događaja. Na Slici ### je prikazan problem koji se javlja svako jutro kada se profesor Bistroum sprema za posao. Profesor mora da obuče određene delove odela pre drugih (na primer, mora da obuje čarape pre cipela). Sa druge strane, neki delovi mogu biti obučeni u proizvoljnom rasporedu (na primer, čarape i pantalone). Usmerena veza (u,v) unutar usmerenog acikličnog grafa sa Slike ###a ukazuje na to da se deo odela u mora obući pre dela v. Topološko sortiranje ovog usmerenog acikličnog grafa daje redosled oblačenja pojedinih delova odela. Na Slici ###b je prikazan topološki sortiran usmereni aciklični graf u vidu čvorova raspoređenih duž horizontalne linije tako da je svaka veza usmerena sa leva na desno.



Slika ###. a) Profesor Bistroum topološki sortira odeću kada se oblači. Svaka veza označava koji deo se mora obući pre nekog drugog. b) Topološki sortiran graf dat pod a).

Sledeći algoritam vrši topološko sortiranje grafa.

```
TOPOLOGICAL_SORT(G)
   pozvati DFS(G) da bi se izračunala završna vremena f[v] svakog čvora v
   po završetku obrade svakog čvora dodati ga na početak povezane liste
   return povezana lista čvorova
```

Slika ###b pokazuje da se topološki sortirani čvorovi pojavljuju u obrnutom rasporedu od njihovih završnih vremena.

Topološko sortiranje se može obaviti u vremenu $\Theta(V+E)$, imajući u vidu da pretraga po dubini zahteva $\Theta(V+E)$ vremena i da je za dodavanje svakog čvora na početak liste potrebno $O(1)$ vremena.

Najkraći put iz jednog polazišta

Motociklista želi da pronađe najkraći mogući put od Kragujevca do Novog Sada. Ako imamo mapu puteva u Srbiji na kojoj je označeno rastojanje između svaka dva susedna mesta, kako možemo odrediti najkraći put između ova dva grada.

Jedan od mogućih načina je pronađemo sve moguće puteve od Kragujevca do Novog Sada, a zatim da odaberemo najkraći od njih. Lako je, međutim, uočiti da čak i ako isključimo ciklične puteve, i dalje nam ostaju milioni mogućnosti, od kojih većina nije vredna razmatranja. Na primer, put od Kragujevca do Užica pa zatim do Novog Sada je očigledno loš izbor, zato što je Užice par stotina kilometara van puta.

U ovom i narednom poglavlju ćemo pokazati kako se ovakvi problemi mogu efikasno rešiti. Kod problema pronalaženja **najkraćeg puta**, dat je težinski, usmeren graf $G=(V,E)$ sa težinskom funkcijom w koja za svaku vezu definiše njenu težinu. **Težina** puta $p=(v_0,v_1,...,v_k)$, je suma težina veza koje ga čine:

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

Težinu najkraćeg puta od u do v definišemo kao

$$\delta(u,v)=\begin{cases}\min\{w(p): u \xrightarrow{p} v\} & \text{ako postoji put od } u \text{ do } v\\ \infty & \text{inače}\end{cases}$$

Najkraći put od čvora u do čvora v se onda definiše kao bilo koji put čija je težina $w(p)=\delta(u,v)$.

U primeru pronalaženja najkraćeg puta od Kragujevca do Novog Sada mapu puteva možemo predstaviti u obliku grafa: čvorovi predstavljaju mesta na mapi, veze predstavljaju deonice puta između mesta, a težine veza označavaju dužinu deonica puta.

Težina veza se može tumačiti ne samo kao dužina puta, već kao bilo koja druga mera. Ona se često koristi za predstavljanje vremena, cene, kazni, troškova ili bilo koje druge veličine koja se akumulira duž puta i koju želimo da minimizujemo.

Pretraga grafa po širini je algoritam koji pronalazi najkraći put kod netežinskih grafova, tj. grafova u kojima se može smatrati da svaka veza ima jediničnu težinu. Međutim, prilikom analiziranja najkraćih puteva kod težinskih grafova, veoma često se koriste koncepti pretrage po širini.

Varijante

U ovom poglavlju ćemo se fokusirati na problem **najkraćeg puta iz jednog polazišta**. Za dati graf $G=(V,E)$ želimo da pronađemo najkraći put iz jednog polaznog čvora $s\in V$ do svakog čvora $v\in V$. Mnogi drugi problemi se mogu rešiti korišćenjem problema jednog polazišta, kao što su na primer:

- **Problem najkraćeg puta do jednog odredišta**: Pronaći najkraći put do odredišnog čvora t iz svakog čvora v u grafu. Zamenom smerova svih veza u grafu, ovaj problem se svodi na pronalaženje najkraćeg puta iz jednog polazišta.
- **Problem najkraćeg puta između dva čvora**: Pronaći najkraći put između dva čvora u i v. Ukoliko rešimo problem najkraćeg puta iz jednog polazišta za čvor u, samim tim rešili smo i ovaj problem. Štaviše, ne postoji ni jedan poznati algoritam za ovaj problem koji bi brže radio od algoritma za pronalaženje najkraćeg puta iz jednog odredišta.
- **Problem najkraćeg puta između svaka dva čvora**: Pronaći najkraći put između čvorova u i v za svaki par čvorova u i v. Iako se ovaj problem može rešiti izvršavanjem algoritma za pronalaženje najkraćeg puta iz jednog polazišta za svaki od čvorova, postoje i efikasniji algoritmi za njegovo rešavanje.

Optimalna podstruktura najkraćeg puta

Algoritmi najkraćeg puta se obično zasnivaju na osobini da najkraći put između dva čvora sadrži u sebi druge najkraće puteve. Ova optimalna podstruktura je obeležje i dinamičkog programiranja i pohlepnih algoritama. Dijkstra algoritam, koji ćemo razmatrati je pohlepan algoritam, dok je Flojd-Woršelov algoritam za pronalaženje najkraćeg puta između svih parova čvorova algoritam dinamičkog programiranja. U nastavku ćemo preciznije definisati optimalnu podstrukturu najkraćeg puta.

Za dati težinski, usmereni graf $G=(V,E)$ sa težinskom funkcijom w, neka je $p=(v_1,v_v,...,v_k)$ najkraći put od čvora v_1 do čvora v_k i neka je za svako i i j takvo da važi $1\le i\le j\le k$, $p_{ij}=(v_i,v_{i+1},...,v_j)$ podput puta p od čvora v_i do čvora v_j. Onda je p_{ij} najkraći put od v_i do v_j.

Predstavljanje najkraćeg puta

Često želimo da odredimo ne samo težinu najkraćeg puta, već i čvorove duž najkraćeg puta. Za predstavljanje najkraćeg puta ćemo koristiti sličnu tehniku kao i u slučaju pretrage po širini. Ukoliko je dat graf $G=(V,E)$, za svaki čvor $v\in V$ čuvamo njegovog **prethodnika** u $\pi[v]$, koji može biti neki drugi čvor ili NULL. Algoritmi najkraćeg puta u ovom poglavlju zadaju vrednosti za π tako da se lanac prethodnika koji počinje u čvoru v proteže unazad duž najkraćeg puta od s do v. Tako za zadati čvor v

za koji je $\pi[v] \neq NULL$, procedura PRINT_PATH(G,s,v) iz prethodnih sekcija se može iskoristiti za štampanje najkraćeg puta od s do v.

Relaksacija

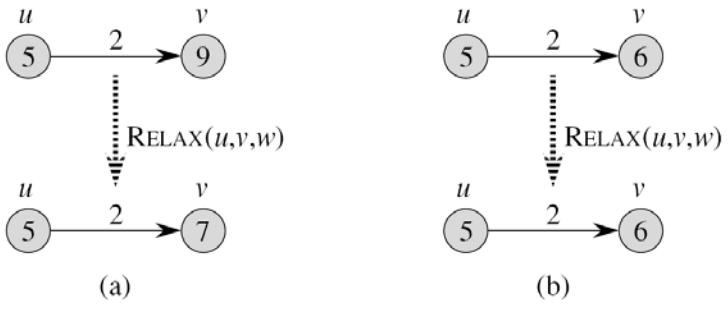
Algoritmi u ovom poglavlju koriste tehniku **relaksacije**. Za svaki čvor $v \in V$, čuvamo promenljivu $d[v]$, koja predstavlja gornju granicu težine najkraćeg puta od polazišta s do v. Promenljivu $d[v]$ nazivamo **procenom najkraćeg puta**. Procene najkraćeg puta i prethodnike inicijalizujemo sledećom $\Theta(V)$ procedurom.

```
INITIALIZE_SINGLE_SOURCE(G,s)
   for each v∈V[G]
      d[v]=∞
      π[v]=NULL
   d[s]=0
```

Proces relaksacije neke veze (u,v) se sastoji u ispitivanju da li je moguće poboljšati dosadašnji najkraći put do čvora v prolaskom kroz čvor u. Ukoliko jeste, vrši se ažuriranje promenljivih $d[v]$ i $\pi[v]$. Korak u relaksaciji može smanjiti vrednost procene najkraćeg puta $d[v]$ i promeniti prethodnika čvora v koji se čuva u $\pi[v]$. Sledeći kod vrši relaksaciju na vezi (u,v).

```
RELAX(u,v,w)
   if d[v]>d[u]+w(u,v)
      d[v]=d[u]+w(u,v)
      π[v]=u
```

Na Slici ### su prikazana dva primera relaksiranja veze: jedan u kome se smanjuje procena najkraćeg puta i drugi u kome nema promene procene.



Slika ###. Relaksacija veze

Svaki algoritam u ovom poglavlju poziva funkciju INITIALIZE_SINGLE_SOURCE i zatim ponavlja relaksiranje veza.

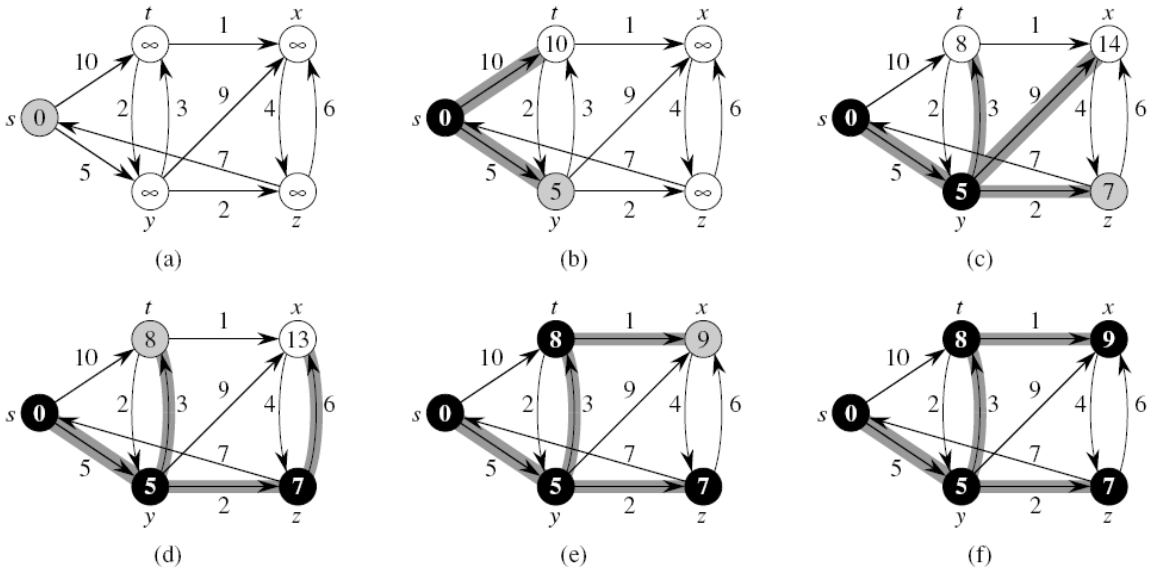
Dijkstra algoritam

Dijkstra algoritam rešava problem najkraćeg puta iz jednog polazišta na težinskom, usmerenom grafu $G=(V,E)$ u slučaju kada su sve veze nenegativne. Iz tog razloga u ovoj sekciji pretpostavljamo da je $w(u,w)\geq 0$ za svaku vezu $(u,v)\in E$.

Dijkstra algoritam čuva skup S čvorova čije su konačne težine najkraćih puteva od polazišta s već određene. Algoritam stalno iznova bira čvor $u\in V-S$ sa najmanjom procenom najkraćeg puta, dodaje ga u S i relaksira sve veze koje polaze iz čvora u. U sledećoj implementaciji koristimo listu Q sa prioritetom minimuma vrednosti d (sa liste se prvo skidaju članovi sa najmanjom vrednošću procene).

```
DIJKSTRA( G, w, s )
   INITIALIZE_SINGLE_SOURCE( G, s )
   S = ∅
   Q = V[G]
   while Q ≠ ∅
      u =EXTRACT_MIN( Q )
      S = S ∪ {u}
      for each v ∈ Adj[u]
         RELAX( u, v, w )
```

Dijkstra algoritam relaksira veze kao što je prikazano na Slici ###. U prvoj liniji se vrši standardna inicijalizacija nizova d i π, a linija koje sledi postavlja S na prazan skup. Algoritam održava invarijantu $Q=V-S$ na početku svake iteracije **while** petlje. Treća linija inicijalizuje listu Q tako da ona sadrži sve čvorove grafa. Svaki put kada se prođe kroz **while** petlju, čvor u se prebacuje iz liste Q u skup S čime se održava invarijanta (u prvom prolazu kroz petlju $u=s$). Samim tim, čvor u ima najmanju procenu najkraćeg puta od svih čvorova u skupu $V-S$. Zatim se relaksiraju sve veze (u,v) koje polaze iz čvora u i na taj način ažuriraju procenu $d[v]$ i prethodnika $\pi[v]$, ukoliko se kraći put do čvora v može postići preko čvora u. Primetimo da se posle treće linije čvorovi ne dodaju u listu Q, tako da se svaki čvor prebacuje iz Q u S samo jednom, pa samim tim **while** petlja iteriše tačno $|V|$ puta.



Slika ###. Izvršavanje Dijsktra algoritma.

Najkraći put između svih parova čvorova

Floyd-Warshall algoritam

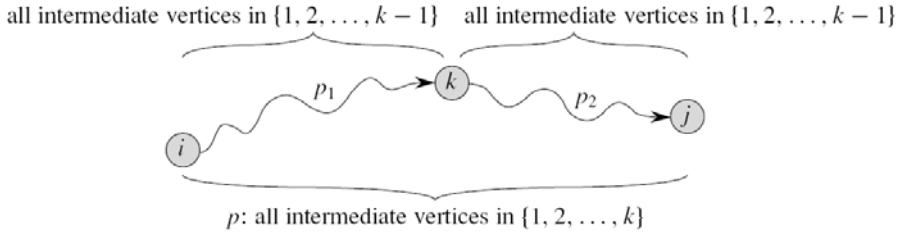
U ovom poglavlju ćemo iskoristiti princip dinamičkog-programiranja da bismo rešili problem najkraćih puteva između svih parova čvorova u usmerenom grafu $G=(V,E)$. Rezultujući algoritam, poznat kao Floyd-Warshall algoritam ima vreme izvršenja $\Theta(V^3)$. Graf može imati i veze negativne težine, ali ne sme imati negativne kružne puteve.

Struktura najkraćeg puta

U ovom algoritmu ćemo posmatrati međučvorove na najkraćem putu, pri čemu je međučvor prostog puta $p=(v_1,v_2,...,v_l)$ bilo koji čvor na ovom putu, osim čvorova v_1 i v_l, tj. bilo koji čvor iz skupa $\{v_2,v_3,...,v_{l-1}\}$.

Floyd-Warshall algoritam je zasnovan na sledećim zapažanjima. Ukoliko su čvorovi grafa G označeni sa $V=\{1,2,...,n\}$, posmatrajmo skup čvorova $\{1,2,...,k\}$ za neko k. Za bilo koji par čvorova $i,j\in V$, posmatrajmo sve puteve od i do j, čiji su svi međučvorovi iz skupa $\{1,2,...,k\}$, pri čemu je p najkraći put između njih (put p je prost). Floyd-Warshall algoritam koristi relaciju između puta p i najkraćih puteva od i do j sa svim međučvorovima iz skupa $\{1,2,...,k-1\}$. Relacija zavisi od toga da li je k međučvor na putu p.

- Ako k nije međučvor na putu p, onda su svi međučvorovi puta p iz skupa $\{1,2,...,k-1\}$. Odatle sledi da je najkraći put od čvora i do čvora j sa svim međučvorovima iz skupa $\{1,2,...,k-1\}$, takođe i najkraći put od i do j sa svim međučvorovima iz skupa $\{1,2,...,k\}$.
- Ako je k međučvor na putu p, onda put p delimo na $i \xrightarrow{p_1} k \xrightarrow{p_2} j$, kao što je prikazano na Slici ###. Put p_1 je najkraći put od i do k sa međučvorovima iz skupa $\{1,2,...,k\}$. Pošto čvor k nije međučvor puta p_1, zaključujemo da je p_1 najkraći put od i do k sa svim međučvorovima iz skupa $\{1,2,...,k-1\}$. Slično, p_2 je najkraći put od čvora k do čvora j sa svim međučvorovima iz skupa $\{1,2,...,k-1\}$.



Slika ###. Najkraći put između čvorova i i j.

Rekurzivno rešenje problema najkraćih puteva između svih parova čvorova

Na osnovu prethodnih razmatranja definišemo rekurzivnu formulaciju problema najkraćih puteva. Neka je $d_{ij}^{(k)}$ cena najkraćeg puta od čvora i do čvora j za koji su svi međučvorovi iz skupa $\{1,2,...,k\}$. Kada je $k=0$, put od čvora i do čvora j koji nema međučvorove numerisane brojevima većim od 0, zapravo uopšte nema međučvorove. Ovakav put ima najviše jednu vezu, pa je odatle $d_{ij}^{(k)}=w_{ij}$. Rekurzivna definicija koja sledi iz prethodne diskusije je data kao

$$d_{ij}^{(k)} = \begin{cases} w_{ij} & ,k = 0 \\ \min\left(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\right) & ,k \geq 1 \end{cases}$$

Pošto su za bilo koji put svi međučvorovi iz skupa $\{1,2,...,n\}$, matrica $D^{(n)} = \left(d_{ij}^{(n)}\right)$ daje konačan odgovor: $d_{ij}^{(n)} = \delta(i,j)$ za sve $i, j \in V$.

Izračunavanje cena najkraćih puteva po principu "od dna ka vrhu"

Da bi se izračunale vrednosti $d_{ij}^{(k)}$, može se iskoristiti sledeća procedura "od dna ka vrhu", bazirana na prethodnoj rekurzivnoj relaciji, pri čemu se k uvećava. Ulaz u ovaj algoritam je težinska matrica W dimenzija $n \times n$. Procedura vraća matricu $D(n)$ sa cenama najkraćih puteva.

```
FLOYD_WARSHALL(W)
    n = rows[W]
    D^(0) = W
    for k=1,n
       for i=1,n
          for j=1,n
             d_ij^(k) = min(d_ij^(k-1), d_ik^(k-1) + d_kj^(k-1))
    return D^(n)
```

Na Slici ### su prikazane matrice $D^{(k)}$ izračunate Floyd-Warshall algoritmom za graf prikazan na Slici ###. Vreme izvršenja Floyd-Warshall algoritma je određeno trima ugneždenim petljama, pa će biti $\Theta\left(n^3\right)$

Konstrukcija najkraćeg puta

Postoji više različitih metoda za konstruisanje najkraćih puteva u Floyd-Warshall algoritmu. Jedan od načina je izračunavanje matrice najkraćih puteva D, a zatim konstruisanje matrice prethodnika Π na osnovu matrice D. Ovaj metod se može implementirati sa kompleksnošću $O\left(n^3\right)$. Kada je matrica Π poznata, korišćenjem procedure PRINT_ALL_PAIRS_SHORTEST_PATH mogu se odštampati čvorovi na datom najkraćem putu.

Matricu prethodnika možemo izračunati "u hodu", prilikom izračunavanja matrice $D^{(k)}$. Tačnije, izračunavamo matrice $\Pi^{(0)}$, $\Pi^{(1)}$,..., $\Pi^{(n)}$, pri čemu je $\Pi = \Pi^{(n)}$, a $\pi_{ij}^{(k)}$ prethodnik čvora j na najkraćem putu od čvora i sa svim međučvorovima iz skupa $\{1,2,...,k\}$.

Možemo dati rekurzivnu formulaciju za $\pi_{ij}^{(k)}$. Kada je $k = 0$, najkraći put od i do j nema međučvorova. Odatle je

$$\pi_{ij}^{(0)} = \begin{cases} NULL & ,i = j \text{ ili } w_{ij} = \infty \\ i & ,i \neq j \ i \ w_{ij} < \infty \end{cases}$$

Za $k \geq 1$, ukoliko odredimo put $i \to k \to j$, gde je $k \neq j$, onda za prethodnika čvora j uzimamo čvor koji je bio prethodnik čvora j na najkraćem putu od k sa svim međučvorovima iz skupa $\{1,2,...,k-1\}$. U suprotnom, za prethodnika čvora j biramo čvor koji je bio prethodnik čvora j na najkraćem putu od i sa svim međučvorovima iz skupa $\{1,2,...,k-1\}$. Formalno

$$\pi_{ij}^{(k)} = \begin{cases} \pi_{ij}^{(k-1)} & , d_{ij}^{(k-1)} \le d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \\ \pi_{kj}^{(k-1)} & , d_{ij}^{(k-1)} > d_{ik}^{(k-1)} + d_{kj}^{(k-1)} \end{cases}$$

$$D^{(0)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(0)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & \text{NIL} & 4 & \text{NIL} & \text{NIL} \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(1)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(2)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(3)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(3)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(4)} = \begin{pmatrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi^{(4)} = \begin{pmatrix} \text{NIL} & 1 & 4 & 2 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(5)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi^{(5)} = \begin{pmatrix} \text{NIL} & 3 & 4 & 5 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

Slika ###. Sekvenca matrica $D^{(k)}$ i $\Pi^{(k)}$ izračunatih Floyd-Warshall algoritmom za graf sa Slike ###.

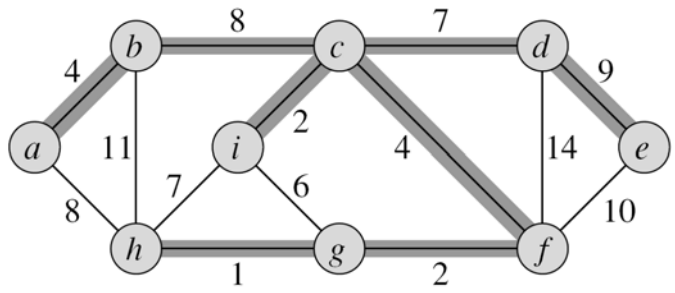
Minimalna stabla razapinjanja

Pri projektovanju elektronskih kola je često neophodno dovesti pinove (igličaste priključke) komponenti na isti električni potencijal, tako što se oni međusobno povežu žicama. Da bi se povezao skup od n pinova, možemo iskoristiti određeni raspored $n-1$ žica, od kojih bi svaka spajala dva pina. Od svih mogućih rasporeda, najpoželjniji je onaj pri kome je ukupna dužina potrebne žice minimalna.

Ovaj problem umrežavanja se može predstaviti korišćenjem povezanog, neusmerenog grafa $G=(V,E)$, gde je V skup pinova, a E skup mogućih veza između parova pinova, pri čemu za svaku vezu $(u,v)\in E$ postoji težina $w(u,v)$ kojom je definisana cena, tj. potrebna dužina žice za povezivanje pinova u i v. To praktično znači da treba da pronađemo aciklični podskup $T\subseteq E$, koji povezuje sve čvorove i čija je ukupna težina

$$w(T)=\sum_{(u,v)\in T} w(u,v)$$

minimalna. S obzirom da je T acikličan i povezuje sve čvorove, on može formirati stablo, koje nazivamo stablo razapinjanja, pošto predstavlja mrežu koja se razapinje preko celog grafa G. Problem određivanja stabla T nazivamo problem **minimalnog stabla razapinjanja**. Na Slici ### je prikazan primer povezanog grafa i njegovog minimalnog stabla razapinjanja.



Slika ###. Minimalno stablo razapinjanja povezanog grafa.

U ovom poglavlju ćemo istražiti dva algoritma za određivanje minimalnog stabla razapinjanja:

- Kruskalov algoritam i
- Primov algoritam

Svaki od njih se može jednostavno napraviti tako da se izvršava u vremenu $O(E\log_2 V)$, korišćenjem običnih binarnih hipova (*binary heap – binarna gomila*). Korišćenjem Fibonačijevih hipova, Primov algoritam se može ubrzati do $O(E+V\log_2 V)$, koji predstavlja poboljšanje ukoliko je $|V|$ mnogo manje od $|E|$.

Ova dva algoritma predstavljaju takozvane "pohlepne algoritme" (*greedy algorithms*). U svakom koraku algoritma, mora se napraviti jedan od nekoliko mogućih izbora. Pohlepna strategija podržava pravljenje onog izbora koji je u tom trenutku najbolji. Ovakva strategija u opštem slučaju ne garantuje da će biti pronađeno rešenje problema biti optimalno. Za minimalno stablo razapinjanja se, međutim, može dokazati da određena pohlepna strategija daje stablo razapinjanja sa minimalnom težinom.

Rast minimalnog stabla razapinjanja

Pretpostavimo da imamo povezani, neusmereni graf $G(V,E)$ sa težinskom funkcijom w i da želimo da pronađemo minimalno stablo razapinjanja za graf G. Dva algoritma koja ćemo razmatrati u ovom poglavlju koriste pohlepni pristup problemu, iako se načini na koji primenjuju ovaj pristup razlikuju.

Ova pohlepna strategija se izvodi praćenjem "generičkog" algoritma, koji uvećava minimalno stablo razapinjanja jednu po jednu vezu. Algoritam manipuliše skupom veza A, pri čemu održava sledeću invarijantu petlje:

Pre svake iteracije, A predstavlja podskup nekog minimalnog stabla razapinjanja.

U svakom koraku određujemo vezu (u,v) koja može biti dodata u skup A, a da se pri tome ne naruši ova invarijanta, u smislu da je $A \cup \{(u,v)\}$ takođe podskup minimalnog stabla razapinjanja. Ovakvu vezu nazivamo **bezbednom vezom** za skup A, zato što se ona može bezbedno dodati u skup A i da se pri tome očuva invarijanta.

```
GENERIC_MST( G, w )
   A = ∅
   while A ne formira stablo razapinjanja
      pronadji vezu (u,v) koja je bezbedna za A
      A = A ∪ {(u,v)}
   return A
```

Invarijantu petlje koristimo na sledeći način:

Inicijalizacija: Nakon prve linije skup A je prazan, tako da trivijalno zadovoljava invarijantu petlje.

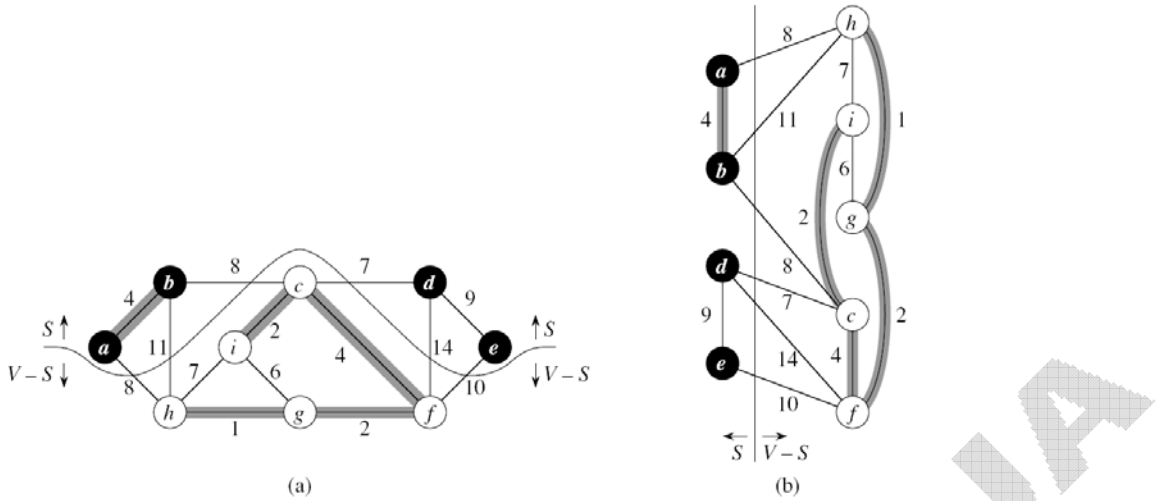
Održanje: Unutar **while** petlje invarijanta se održava tako što se dodaju samo bezbedne veze.

Okončanje: Sve veze dodate u skup A se nalaze u minimalnom stablu razapinjanja, tako da skup A koji procedura vraća mora biti minimalno stablo razapinjanja.

Posebno bitan deo je traženje bezbedne veze. Jedna bezbedna veza mora da postoji zato što invarijanta diktira da nakon pronalaženja veze postoji stablo razapinjanja T takvo da je $A \subseteq T$. Unutar **while** petlje, A mora biti odgovarajući podskup skupa T, pa samim tim mora postojati veza $(u,v) \in T$ takva da $(u,v) \notin A$ i da je (u,v) bezbedno za A.

U nastavku ove sekcije ćemo objasniti kako pronaći bezbedne veze. U narednim sekcijama ćemo opisati dva algoritma koja koriste ovo pravilo da bi efikasno pronašli bezbedne veze.

Da bismo lakše opisali ove algoritme, prvo ćemo uvesti neke definicije. **Rez** $(S, V-S)$ nekog neusmerenog grafa $G=(V,E)$ predstavlja podelu čvorova V. Slika ### ilustruje ovu definiciju. Za neku vezu $(u,v) \in E$ kažemo da preseca rez $(S, V-S)$ ukoliko jedna od njenih krajnjih tačaka pripada skupu S, a druga skupu $V-S$. Ako u skupu A ne postoji veza koja preseca rez, onda kažemo da rez **poštuje** skup A. Neka veza je **laka veza** koja preseca rez, ukoliko je njena težina najmanja od svih veza koje presecaju rez.



Slika ###. Dva načina prikazivanja reza $(S, V-S)$ grafa sa Slike ###.

Pravilo za prepoznavanje bezbednih ivica se može dati na sledeći način:

> Neka je $G=(V,E)$ povezan, neusmereni graf sa realnom težinskom funkcijom w definisanom na svim vezama iz skupa E. Neka je A podskup skupa E koji je uključen u neko minimalno stablo razapinjanja grafa G, neka je $(S,V-S)$ bilo koji rez grafa G koji poštuje A i neka je (u,v) laka veza koja preseca $(S,V-S)$. Onda je (u,v) bezbedna za A.

Dva algoritma za određivanje minimalnog stabla razapinjanja koja će biti obrađena su praktično razrada generičkog algoritma. Svaki od njih koristi specifično pravilo za određivanje bezbedne veze u GENERIC_MST algoritmu. U Kruskalovom algoritmu, skup A je šuma. Bezbedna veza koja se dodaje u A je uvek veza sa najmanjom težinom u grafu koja povezuje dve odvojene komponente. U Primovom algoritmu, skup A formira jedno stablo. Bezbedna veza koja se dodaje u A je uvek veza sa najmanjom težinom u grafu koja povezuje stablo sa čvorom koji nije u stablu.

Kruskalov algoritam

Kruskalov algoritam se direktno zasniva na algoritmu za određivanje minimalnog stabla razapinjanja. On pronalazi bezbednu vezu za dodavanje u rastuću šumu tako što, među svim vezama koje povezuju bilo koja dva stabla u šumi, pronalazi vezu (u,v) sa najmanjom težinom.

Implementacija Kruskalovog algoritma koristi disjunktne skupove podataka. Svaki skup sadrži neki reprezentativni element, koji predstavlja jedinstveni identifikator skupa. Da bismo mogli da vršimo osnovne operacije sa disjunktnim skupovima, uvešćemo sledeće funkcije:

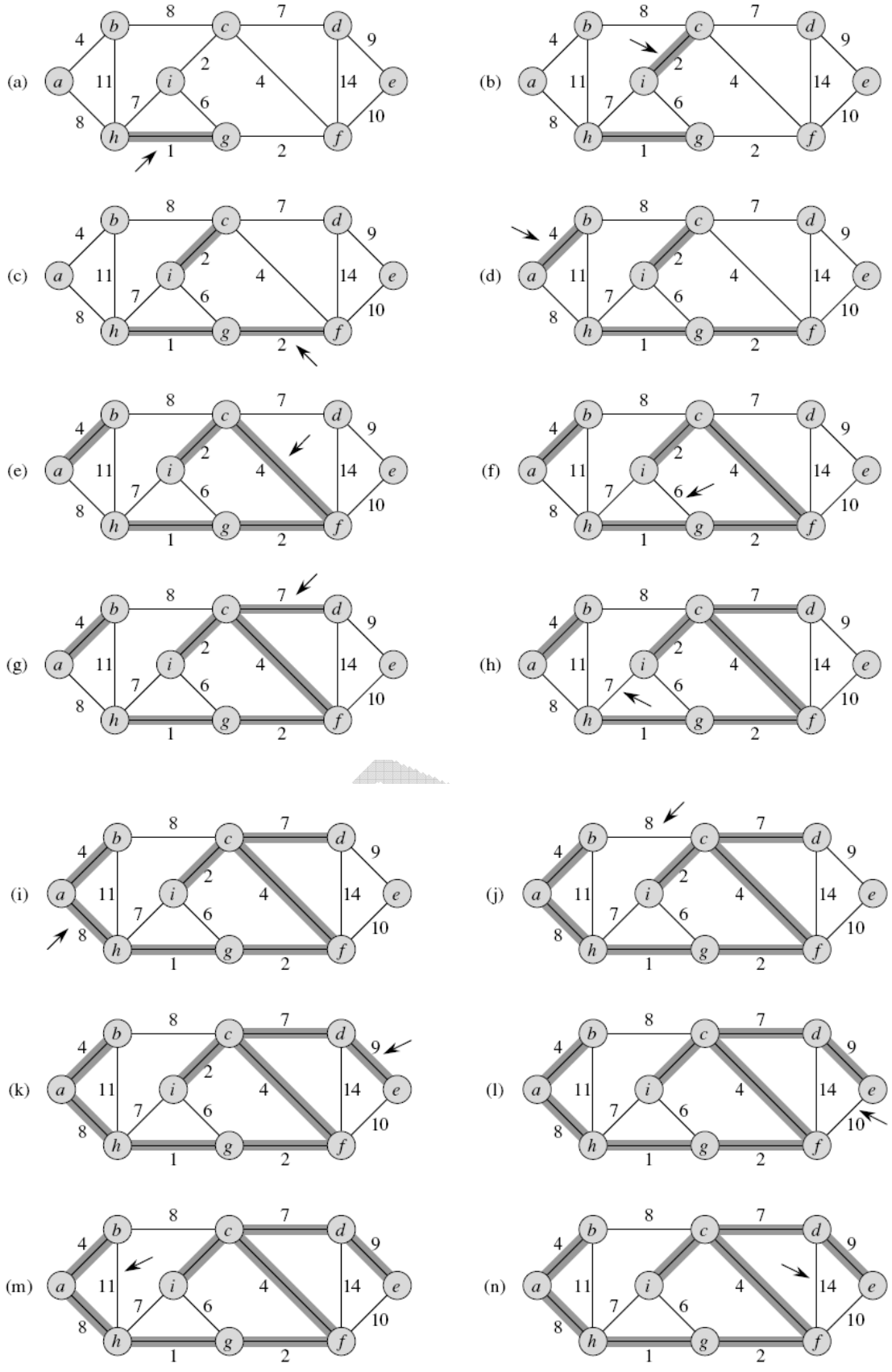
- MAKE_SET(x) pravi novi skup čiji je jedini (a samim tim i reprezentativni) član x. Pošto su skupovi disjunktni, neophodno je da x ne pripada ni jednom drugom skupu.
- UNION(x,y) sjedinjuje skupove koji sadrže x i y, recimo S_x i S_y, u novi skup koji predstavlja uniju ova dva skupa. Pretpostavka je da su ova dva skupa bila disjunktna pre izvršenja ove operacije. Reprezentativni element rezultujućeg skupa može biti bilo koji element skupa $S_x \cup S_y$.
- FIND_SET(x) vraća pokazivač na reprezentativni element skupa koji sadrži x.

U Kruskalovom algoritmu svaki skup sadrži čvorove u stablu trenutne šume. Korišćenjem funkcije FIND_SET lako možemo odrediti da li dva čvora u i v pripadaju istom stablu, jednostavnom proverom da li je FIND_SET(u) jednako FIND_SET(v). Spajanje stabala se vrši korišćenjem funkcije UNION.

```
MST_KRUSKAL( G , w )
   A = ∅
   for each v ∈ V[G]
      MAKE_SET( v )
   sortiranje veza po tezini u neopadajucem rasporedu
   for each (u,v) ∈ E , uzeto u neopadajucem redosledu
      if FIND_SET( u ) ≠ FIND_SET( v )
         A = A ∪ {(u,v)}
         UNION( u , v )
   return A
```

Kruskalov algoritam radi kao što je prikazano na Slici ###. Prve tri linije procedure MST_KRUSKAL inicijalizuju skup A kao prazan skup i prave $|V|$ stabala, tako da svako stablo sadrži jedan čvor. U sledećoj liniji se sve veze iz skupa E sortiraju po težini u neopadajućem poretku. Druga **for** petlja za svaku vezu (u,v) proverava da li njeni krajnji čvorovi u i v pripadaju istom stablu. Ako čvorovi pripadaju istom stablu, oni se ne mogu dodati šumi a da se pri tome ne formira ciklus, pa se ta veza odbacuje. Ukoliko ova dva čvora pripadaju različitim stablima, veza (u,v) se dodaje u skup A, a zatim se čvorovi ta dva stabla spajaju u jedan skup.

Vreme izvršavanja Kruskalovog algoritma za graf $G=(V,E)$ zavisi od implementacije struktura podataka o disjunktnim skupovima. Pretpostavićemo da je primenjena implementacija šume disjunktnih skupova sa najbržom mogućom asimptotskom kompleksnošću. Inicijalizacija skupa A uzima $O(1)$ vremena, a vreme potrebno za sortiranje veza je $O(E\log_2 E)$. Druga **for** petlja obavlja $O(E)$ FIND_SET i UNION operacija nad šumom disjunktnih skupova, što zajedno sa $|V|$ MAKE_SET operacija uzima $O((V+E)\alpha(V))$ vremena, pri čemu je α veoma sporo rastuća funkcija. S obzirom da je pretpostavka da je graf G povezan, imamo da je $|E|\geq|V|-1$, tako da operacije sa disjunktnim skupovima uzimaju $O(E\alpha(V))$ vremena. Štaviše, pošto je $\alpha(|V|)=O(\log_2 V)=O(\log_2 E)$, ukupno vreme izvršavanja Kruskalovog algoritma je $O(E\log_2 E)$. Ako uočimo da je $|E|<|V|^2$, onda imamo da je $\log_2|E|=O(\log_2 V)$, tako da je vreme izvršenja Kruskalovog algoritma ustvari $O(E\log_2 V)$.



Slika ###. Faze u izvršavanju Kruskalovog algoritma.

Primov algoritam

Kao i Kruskalov algoritam, Primov algoritam je specijalni slučaj generičkog algoritma za određivanje minimalnog stabla razapinjanja. Primov algoritam funkcioniše veoma slično Dijkstra algoritmu za pronalaženje najkraćeg puta u grafu. Primov algoritam ima osobinu da veze u skupu A uvek formiraju jedno stablo. Kao što je prikazano na Slici ###, stablo počinje od proizvoljnog čvora r i raste sve dok stablo ne pokrije sve čvorove u skupu V. U svakom koraku laka veza se dodaje u stablo A, tako da se stablo A povezuje sa izolovanim čvorom. Ova strategija je pohlepna s obzirom da se stablo u svakom koraku uvećava za jednu vezu koja što je manje moguće doprinosi težini stabla.

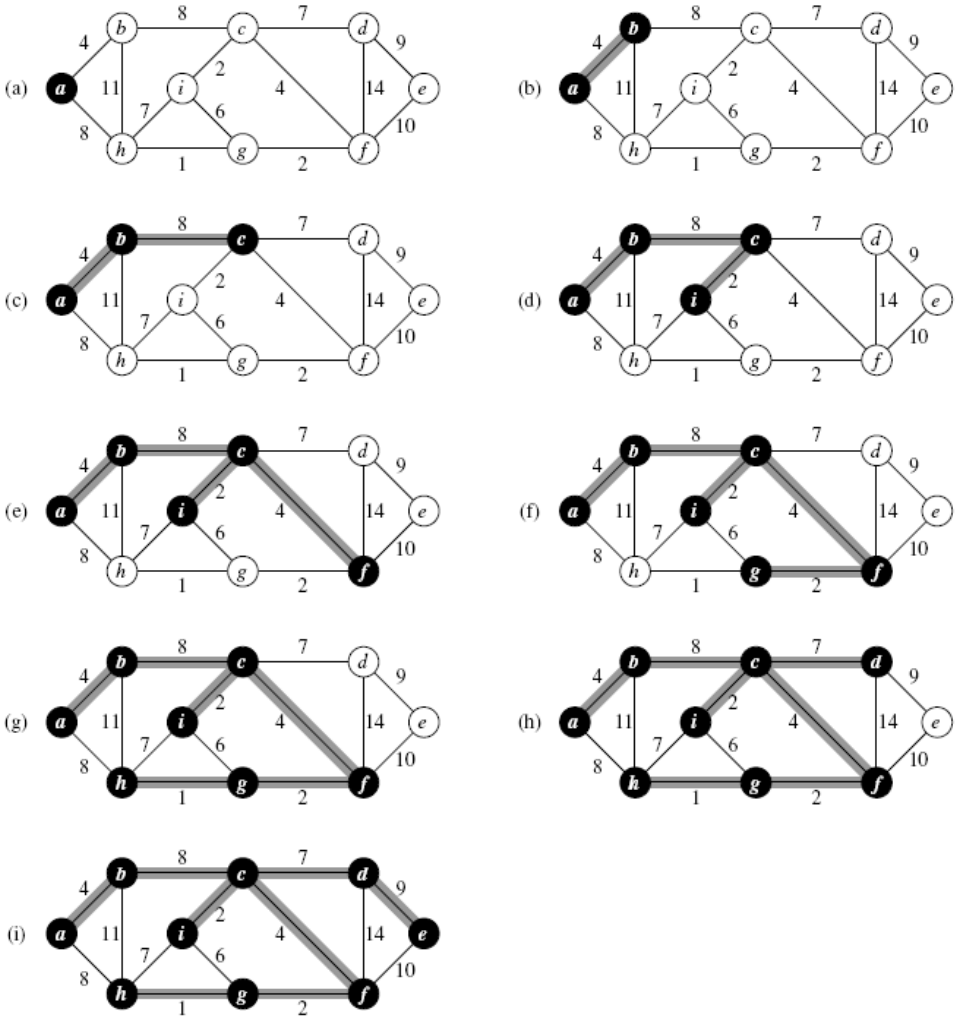
Ključna stvar u efikasnoj primeni Primovog algoritma je na jednostavan način izabrati vezu koja će biti dodata u stablo formirano od veza iz skupa A. U pseudokodu koji sledi, ulazi u algoritam su povezani graf G i koren minimalnog stabla razapinjanja r. Tokom izvršavanja algoritma svi čvorovi koji nisu u stablu nalaze se u redu Q sa prioritetom minimuma ključa key. Za svaki čvor v, $key[v]$ je minimalna težina bilo koje veze koja povezuje čvor v sa čvorom u stablu. Kada ne postoji takva veza $key[v]=\infty$. U promenljivoj $\pi[v]$ se čuva roditelj čvora v u stablu.

```
MST_PRIM(G,w,r)
   for each u∈V[G]
      key[u]=∞
      π[u]=NULL
   key[r]=0
   Q=V[G]
   while Q≠∅
      u=EXTRACT_MIN(Q)
      for each v∈Adj[u]
         if v∈Q and w(u,v)<key[v]
            π[v]=u
            key[v]=w(u,v)
```

Primov algoritam funkcioniše kao što je prikazano na Slici ###. Prvih pet linija pseudokoda postavljaju ključ svakog čvora na ∞ (osim za koren r, čiji je ključ postavljan na 0, tako da je on prvi čvor koji se obrađuje), postavlja roditelje svih čvorova na $NULL$ i inicijalizuje red Q tako da sadrži sve čvorove. Unutar **while** petlje se identifikuje čvor $u\in Q$ koji je na najmanjem rastojanju od stabla A. **For** petlja koja sledi ažurira promenljive key i π za svaki čvor v koji je povezan sa čvorom u, ali nije u stablu.

Performanse Primovog algoritma zavise od toga kako je implementiran red Q sa prioritetom minimuma. Ukoliko je Q implementiran kao binarni hip, prvih pet linija pseudokoda se izvršava u $O(V)$ vremenu. Telo **while** petlje se izvršava $|V|$ puta, a pošto operacija EXTRACT_MIN oduzima $O(\log_2 V)$ vremena, ukupno vreme potrebno za pozivanje EXTRACT_MIN je $O(V\log_2 V)$. Druga **for** petlja se izvršava ukupno $O(E)$ puta, pošto je suma dužina svih lista suseda $2|E|$. Unutar **for** petlje se testiranje pripadnosti redu Q može implementirati u konstantnom vremenu, tako što bi se za svaki čvor čuvao bit koji bi govorio da li on jeste ili nije u Q i koji bi se ažurirao kada se čvor ukloni iz reda. Poslednja dodela u algoritmu izaziva operaciju uklanjanja ključa iz reda, koja se može implementirati sa $O(\log_2 V)$. Iz svega ovoga sledi da je ukupno vreme izvršavanja Primovog algoritma $O(V\log_2 V + E\log_2 V)=O(E\log_2 V)$ što je jednako implementaciji Kruskalovog algoritma.

Vreme izvršavanja Primovog algoritma se može dodatno unaprediti korišćenjem Fibonačijevih hipova, pri čemu se dobija vreme $O(E + V \log_2 V)$.



Slika ###. Izvršavanje Primovog algoritma