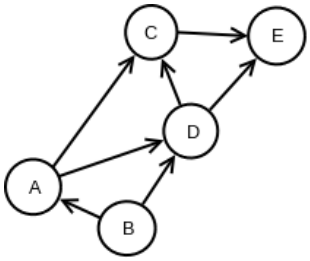
Topološko sortiranje

Pretpostavimo da je zadat skup poslova u vezi sa čijim redosledom izvršavanja postoje neka ograničenja. Neki poslovi zavise od drugih, odnosno ne mogu se započeti pre nego što se ti drugi poslovi završe. Sve zavisnosti su poznate, a cilj je napraviti takav redosled izvršavanja poslova koji zadovoljava sva zadata ograničenja; drugim rečima, traži se redosled izvršavanja za koji važi da svaki posao započinje tek kad su završeni svi poslovi od kojih on zavisi. Na primer, želimo da sagradimo kuću. Da bi to uspeli potrebno je da iskopamo temelj, da sazidamo zidove, da stavimo krov, da uvedemo struju i vodu. Pritom, naravno, nije moguće npr. sazidati zidove dok se ne stavi temelj, niti uvesti vodu dok se ne stavi krov na kuću. Potrebno je odrediti neki ispravan redosled izvršavanja ovih poslova.

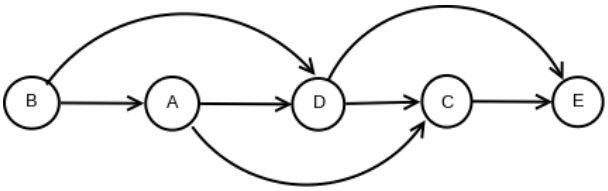
Dati problem ima primenu u raznim domenima: na primer, za određivanje redosleda u kom je potrebno izvršiti ponovno izračunavanje vrednosti formula u programima za tabelarna izračunavanja, utvrđivanje redosleda u kom treba izvršiti zadatke u mejkfajlu, unapređenje paralelizma instrukcija i slično. Želimo da osmislimo efikasan algoritam za formiranje takvog redosleda. Ovaj problem može se formulisati kao grafovski i naziva se *topološko sortiranje grafa* (eng. topological sort). Naime, zadatim poslovima i njihovim međuzavisnostima može se na prirodan način pridružiti usmereni graf $G = (V, E)$: svakom poslu pridružuje se čvor, a usmerena grana od čvora x do čvora y postoji ako se posao y ne može započeti pre završetka posla x. Zadatak je odrediti poredak čvorova, odnosno numeraciju čvorova brojevima od 1 do $n = |V|$ tako da za svaku granu grafa važi da je polazni čvor grane numerisan manjom vrednošću nego završni čvor te grane. Graf nad kojim razmatramo ovaj problem mora biti bez usmerenih ciklusa, jer se u protivnom neki poslovi nikada ne bi mogli započeti.



Slika 1: Usmereni aciklički graf u kojem postoji tačno jedno topološko uređenje čvorova.

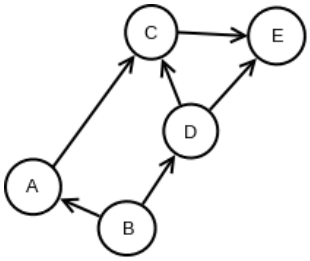
Problem: U zadatom usmerenom acikličkom grafu $G = (V, E)$ sa n čvorova numerisati čvorove brojevima od 1 do n, tako da ako je proizvoljan čvor v numerisan brojem k, onda su svi čvorovi do kojih postoji usmerena grana iz čvora v numerisani brojevima većim od k.

Na primer, u grafu prikazanom na slici 1 postoji samo jedno ispravno topološko uređenje čvorova i to je B, A, D, C, E. Čvor D, recimo, mora da bude numerisan većim brojem od čvorova B i A jer postoje grane do D iz čvorova A i B. Slično čvor D mora biti numerisan manjim brojem od čvorova C i E jer postoje grane od čvora D do čvorova C i E. Dakle, u ovom grafu redni broj čvora D mora biti 3. Graf sa slike 1 možemo predstaviti i pogodnije, tako da čvorovi budu poređani duž jedne prave uređeni u odnosu na topološki poredak. Onda su grane grafa uvek usmerene udesno (slika 2).



Slika 2: Usmereni aciklički graf kod koga su čvorovi poređani u redosledu topološkog uređenja.

U opštem slučaju može postojati veći broj ispravnih topoloških uređenja grafa. Ako razmotrimo graf sa slike 3, u njemu postoje dva ispravna topološka uređenja: B, A, D, C, E i B, D, A, C, E. Oni su prikazani na slici 4.



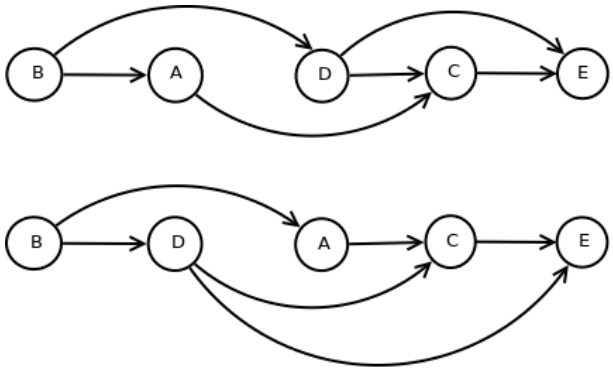
Slika 3: Usmereni aciklički graf u kojem postoje dva različita topološka uređenja čvorova.

Razmotrićemo dva različita algoritma za određivanje topološkog uređenja u acikličkom usmerenom grafu: Kanov algoritam i algoritam zasnovan na pretrazi grafa u dubinu.

Kanov algoritam

Prirodna je sledeća induktivna hipoteza: umemo da numerišemo na zahtevani način čvorove svih usmerenih acikličkih grafova sa manje od n čvorova.

Bazni slučaj je slučaj grafa koji sadrži tačno jedan čvor i on se trivijalno rešava. Kao i obično, posmatrajmo proizvoljni graf sa n čvorova, uklonimo jedan čvor iz njega, primenimo induktivnu hipotezu na preostale čvorove u grafu i pokušajmo



Slika 4: Dva moguća topološka uređenja grafa sa slike 3.

da proširimo numeraciju na polazni graf. Ono što je važno primetiti jeste da imamo slobodu izbora čvora koji uklanjamo. Treba izabrati čvor tako da što jednostavnije proširimo induktivnu hipotezu. Postavlja se pitanje koji čvor je najlakše numerisati? To je očigledno čvor (posao) koji ne zavisi od drugih poslova, odnosno čvor sa ulaznim stepenom nula; njemu se može dodeliti broj 1. Postavlja se pitanje da li u proizvoljnom usmerenom acikličkom grafu uvek postoji čvor sa ulaznim stepenom nula? Intuitivno se nameće potvrdan odgovor, jer se sa označavanjem negde mora započeti. Sledeća lema potvrđuje ovu činjenicu.

Lema: Usmereni aciklički graf uvek ima čvor ulaznog stepena nula.

Dokaz: Ako bi svi čvorovi grafa imali pozitivne ulazne stepene, mogli bismo da krenemo iz nekog čvora "unazad" prolazeći grane u suprotnom smeru. Međutim, broj čvorova u grafu je konačan, pa se u tom obilasku mora u nekom trenutku naići na neki čvor po drugi put, što znači da u grafu postoji usmereni ciklus. Ovo je suprotno pretpostavci da se radi o acikličkom grafu. Dakle u usmerenom acikličkom grafu uvek postoji čvor čiji je ulazni stepen nula.[1] □

Pretpostavimo da smo pronašli čvor čiji je ulazni stepen nula. Numerišimo ga sa 1, uklonimo sve grane koje vode iz njega, i numerišimo ostatak grafa (koji je takođe aciklički) brojevima od 2 do n: prema induktivnoj hipotezi oni se mogu numerisati brojevima od 1 do $n-1$, a zatim se svaki redni broj može povećati za jedan. Primetimo da je posle izbora čvora sa ulaznim stepenom nula, preostali problem sličan polaznom problemu. Napomenimo još u ovom trenutku da nije neophodno efektivno izbacivati grane iz grafa, jer je to operacija koja se ne izvršava efikasno ako je graf predstavljen listom povezanosti, već da je isti efekat moguće izvesti efikasnije, smanjivanjem za 1 ulaznog stepena čvora u koji data grana ulazi.

Dakle, problem se može rešiti sukcesivnim pronalaženjem čvorova sa ulaznim

[1] Analogno bi se pokazalo da u usmerenom acikličkom grafu uvek postoji čvor izlaznog stepena nula.

stepenom 0. Jedini problem pri realizaciji ovog algoritma je kako efikasno pronaći čvor sa ulaznim stepenom nula i kako popraviti ulazne stepene čvorova posle uklanjanja grana koje polaze iz datog čvora. Možemo alocirati niz `ulazniStepen` dimenzije jednake broju čvorova u grafu i inicijalizovati ga na vrednosti ulaznih stepena čvorova. Ulazne stepene čvorova u grafu možemo jednostavno odrediti prolaskom kroz skup svih grana u proizvoljnom redosledu i povećavanjem za jedan vrednosti `ulazniStepen[w]` svaki put kad se naiđe na neku granu (v, w). Ukoliko je graf zadat listom povezanosti, sve grane su navedene u listi povezanosti kojom je predstavljen i ovaj korak biće linearne složenosti po broju grana u grafu. Tokom izvršavanja algoritma bitno je održavati spisak čvorova čiji je ulazni stepen 0 jer ih je potrebno obraditi u nekom poretku – primetimo da takvih čvorova u svakom od koraka može biti puno. Dakle, potrebno je čvorove sa ulaznim stepenom nula čuvati u nekoj kolekciji u koju je efikasno umetati i iz koje je efikasno uklanjati elemente. U ove svrhe može se koristiti red (ili stek, što bi bilo jednako dobro). Prema prethodnoj lemi u polaznom acikličkom grafu postoji bar jedan čvor sa ulaznim stepenom nula. Neka je v jedan od takvih čvorova. Čvor v se kao prvi u redu lako pronalazi; on se uklanja iz reda i numeriše brojem 1. Zatim se za svaku granu (v, w) koja polazi iz čvora v vrednost `ulazniStepen[w]` smanjuje za jedan. Time se evidentira da zavisnosti koje potiču od trenutno numerisanog čvora više nisu od značaja: svakako će svi preostali čvorovi biti numerisani većim vrednostima od tekuće. Ako vrednost ulaznog stepena čvora w pri tome postane nula, čvor w upisuje se u red. Posle uklanjanja čvora v graf ostaje aciklički, pa u njemu prema prethodnoj lemi ponovo postoji čvor sa ulaznim stepenom nula. Algoritam završava sa radom kada red koji sadrži čvorove stepena nula postane prazan, jer su u tom trenutku svi čvorovi numerisani. Opisani algoritam zove se *Kanov algoritam*, po njegovom autoru Arturu Kanu.

U tabeli 1 ilustrovano je izvršavanje Kanovog algoritma na primeru grafa sa slike 3. Pretpostavićemo da je graf zadat listama povezanosti, tako da su za svaki čvor njegovi susedi poređani u leksikografski rastućem poretku. Za svaki od koraka algoritma prikazane su trenutne vrednosti ulaznih stepena čvorova grafa, sadržaj reda koji sadrži čvorove ulaznog stepena nula koji još uvek nisu numerisani i poslednji numerisani čvor u grafu. Primetimo da se u drugom koraku moglo desiti da se u red najpre doda čvor D, a zatim čvor A i u tom slučaju bilo bi dobijeno drugačije topološko uređenje: B, D, A, C, E.

$d(A)$	$d(B)$	$d(C)$	$d(D)$	$d(E)$	Red	Naredni numerisani čvor
1	0	2	1	2	B	
0		2	0	2	A, D	$B : 1$
		1		2	D	$A : 2$
		0		1	C	$D : 3$
				0	E	$C : 4$
						$E : 5$

Table 1: Primer izvršavanja Kanovog algoritma za graf sa slike 3.

Ako bi nakon završetka rada algoritma za neke čvorove važilo da nisu bili dodati u red, to bi značilo da postoji podskup skupa čvorova takav da u odgovarajućem indukovanom podgrafu svi čvorovi imaju ulazni stepen veći od nula, što znači da bi indukovani podgraf (a time i polazni graf) sadržao usmereni ciklus, suprotno pretpostavci da je graf aciklički.

Važno je napomenuti da u algoritmu ne moramo iz samog grafa izbacivati grane, odnosno menjati listu povezanosti kojom je graf zadat, već je jedino važno da za svaki čvor ažuriramo vrednost njegovog ulaznog stepena.

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

void topolosko_sortiranje(){

  int brojCvorova = listaSuseda.size();
  // niz koji cuva ulazne stepene cvorova
  vector<int> ulazniStepen(brojCvorova,0);
  // niz koji cuva redne brojeve cvorova u topoloskom uredjenju
  vector<int> topoloskoUredjenje;
  // broj posecenih cvorova
  int brojPosecenih = 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]]++;

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

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

  while(!cvoroviStepenaNula.empty()){
    // cvor sa pocetka reda numerisemo narednim brojem
    int cvor = cvoroviStepenaNula.front();
    cvoroviStepenaNula.pop();
    topoloskoUredjenje.push_back(cvor);

    brojPosecenih++;

    // za sve susede tog cvora azuriramo ulazne stepene
    for(int i = 0; i < listaSuseda[cvor].size(); i++){
      int sused = listaSuseda[cvor][i];
```

```
      ulazniStepen[sused]--;
      // ako je ulazni stepen suseda postao 0, dodajemo ga u red
      if (ulazniStepen[sused] == 0)
         cvoroviStepenaNula.push(sused);
    }
  }

  // ako smo numerisali sve cvorove u grafu
  if (brojPosecenih == brojCvorova){
     // stampamo dobijeno topolosko uredjenje
     cout << "Redosled cvorova u topoloskom uredjenju je:" << endl;
     for(int i = 0; i < brojCvorova; i++)
        cout << topoloskoUredjenje[i] << ": " << i+1 << endl;
  }
  else
     // zakljucujemo da graf sadrzi usmereni ciklus
     cout << "Graf nije aciklicki" << endl;
}

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

Vremenska složenost inicijalizacije niza `ulazniStepen` je $O(|V|+|E|)$, u slučaju kada je graf zadat listama povezanosti. U petlji `while` (kroz koju se prolazi $|V|$ puta) za pronalaženje čvora sa ulaznim stepenom nula potrebno je konstantno vreme (pristup redu). Svaka grana (v, w) razmatra se tačno jednom, u petlji kroz koju se prolazi nakon uklanjanja čvora v iz reda. Prema tome, ukupan broj promena vrednosti elemenata niza `ulazniStepen` u svim izvršavanjima spoljašnje `while` petlje jednak je broju grana u grafu. Vremenska složenost Kanovog algoritma je dakle $O(|V|+|E|)$, odnosno linearna je funkcija od veličine grafa.

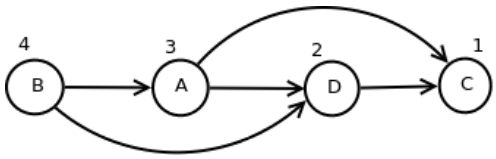
Algoritam zasnovan na DFS pretrazi

Kao što smo ranije zaključili u grafu $G = (V, E)$ važi:

- ako je grana $(u, v) \in E$ grana DFS drveta, direktna ili poprečna grana, za nju važi $u.Post > v.Post$,
- ako je grana $(u, v) \in E$ povratna grana u odnosu na DFS drvo, za nju važi $u.Post \leq v.Post$.

U usmerenom acikličkom grafu ne postoji ciklus, pa ne postoje povratne grane u odnosu na DFS drvo. Dakle, za svaku granu (u, v) grafa važi uslov $u.Post > v.Post$. Ako sa $n(x)$ označimo redni broj čvora x u topološkom poretku grafa G, za svaku granu (u, v) potrebno je da važi $n(u) < n(v)$. Odavde sledi da ako

čvorove grafa uredimo u opadajućem redosledu u odnosu na odlaznu numeraciju čvorova, dobićemo jedno topološko uređenje grafa. Ovo tvrđenje važi zato što će na ovaj način za proizvoljnu granu (u, v) acikličkog grafa važiti da je polazni čvor u numerisan manjom vrednošću od dolaznog čvora v, što je u skladu sa zahtevima problema koji rešavamo.



Slika 5: Usmereni aciklički graf: uz svaki čvor prikazana je vrednost njegove odlazne numeracije u odnosu na DFS pretragu pokrenutu iz čvora B.

Razmotrimo graf sa slike 5: on je usmeren i aciklički. Ako pokrenemo DFS pretragu iz čvora B redosled čvorova u kojima ćemo napuštati čvorove je C, D, A, B. Dakle, topološko uređenje grafa dobićemo obrtanjem ovog redosleda, odnosno redosled čvorova u topološkom poretku biće B, A, D, C. Primetimo da to odgovara i redosledu čvorova sleva nadesno u prikazu grafa kod koga su sve grane usmerene sleva udesno.

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

void dfs(int cvor, vector<bool> &posecen, vector<int> &odlazna){
  posecen[cvor] = true;

  // rekurzivno prolazimo kroz sve susede koje nismo obisli
  for (auto sused : listaSuseda[cvor]){
    if (!posecen[sused])
       dfs(sused,posecen,odlazna);
  }

  // u vektor odlazna dodajemo na kraj naredni cvor
  // koji napustamo pri DFS obilasku
  odlazna.push_back(cvor);
}

void topolosko_sortiranje(){

  int brojCvorova = listaSuseda.size();
  vector<bool> posecen(brojCvorova);
  // niz koji sadrzi redom cvorove prema redosledu napustanja
  vector<int> odlazna;

  for (int cvor = 0; cvor < brojCvorova; cvor++)
```

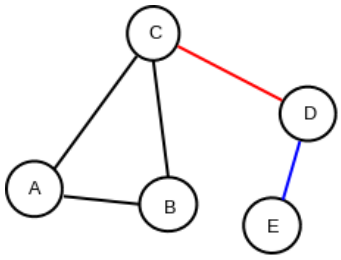
```
    if (!posecen[cvor])
       dfs(cvor,posecen,odlazna);

  // cvorove ispisujemo u opadajućem redosledu odlazne numeracije
  cout << "Redosled cvorova u topoloskom uredjenju je:" << endl;
  for(int i = brojCvorova - 1; i >= 0; i--)
    cout << odlazna[i] << ": " << brojCvorova - i << endl;

}

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

S obzirom na to da se prikazani algoritam svodi na DFS pretragu i određivanje odlazne numeracije čvorova, njegova vremenska složenost iznosi $O(|E| + |V|)$.

Mostovi i artikulacione tačke u neusmerenom grafu

U ovom poglavlju bavićemo se pitanjem koliko je dati neusmereni graf dobro povezan, odnosno interesovaće nas da li u njemu postoji slaba tačka, odnosno usko grlo, čijim bi otkazivanjem graf prestao da bude povezan. U raznim praktičnim primenama ovaj problem je veoma značajan i potrebno je omogućiti da se uska grla u grafu brzo detektuju.

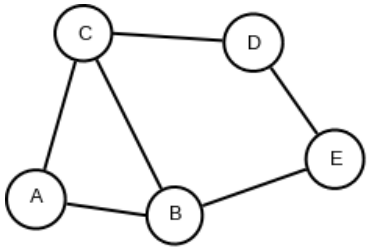
Granu neusmerenog grafa $G = (V, E)$ čijim se uklanjanjem iz grafa broj komponenti povezanosti grafa povećava nazivamo *most* (eng. bridge, cut edge). Specijalno, povezani graf nakon uklanjanja mosta prestaje da bude povezan. Na primer, ukoliko bismo u grafu prikazanom na slici 6 uklonili granu (C, D) ili granu (D, E) graf bi prestao da bude povezan, te ove dve grane, svaka za sebe, čine most u datom grafu. Postoje grafovi u kojima nema mostova (slika 7).



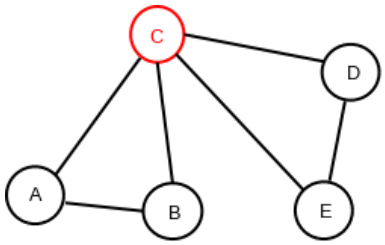
Slika 6: Primer grafa koji sadrži dva mosta: jedan je označen crvenom, a drugi plavom bojom.

Ukoliko u neusmerenom grafu $G = (V, E)$ postoji čvor $v \in V$ takav da se njegovim uklanjanjem iz grafa (zajedno sa granama koje su mu susedne) broj komponenti

povezanosti grafa povećava, onda takav čvor nazivamo *artikulacionom tačkom* (eng. articulation point, cut vertex). Specijalno, povezani graf nakon uklanjanja artikulacione tačke prestaje da bude povezan. Na primer, ukoliko bismo u grafu prikazanom na slici 8 uklonili čvor C graf bi prestao da bude povezan, te je čvor C jedna artikulaciona tačka ovog grafa. Štaviše, pokazuje se da je čvor C jedina artikulaciona tačka u ovom grafu.

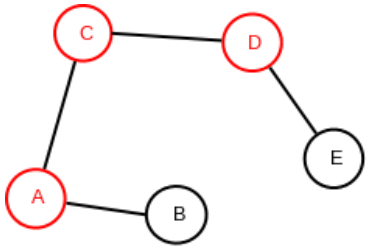


Slika 7: Primer grafa koji ne sadrži ni most ni artikulacionu tačku.



Slika 8: Graf koji sadrži jednu artikulacionu tačku: čvor C.

Graf može da ne sadrži artikulacione tačke (slika 7), a može i da sadrži veći broj artikulacionih tačaka (slika 9).



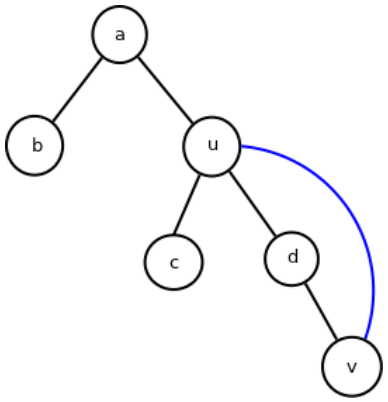
Slika 9: Graf koji sadrži veći broj artikulacionih tačaka: to su čvorovi A, C i D. Svaka grana ovog grafa je most.

Bez smanjenja opštosti može se pretpostaviti da je dati neusmereni graf povezan. Ako graf nije povezan, onda se artikulacione tačke mogu tražiti nezavisno u svakoj komponenti povezanosti grafa.

Direktan način da se u datom neusmerenom povezanom grafu $G = (V, E)$ pronađu svi mostovi bi podrazumevao da za svaku granu $e \in E$ grafa G proverimo da li je graf bez grane e povezan (npr. korišćenjem algoritma DFS). Složenost ovog algoritma bi iznosila $O(|E| \cdot (|V| + |E|))$[2]. Analogno, artikulacione tačke u datom neusmerenom povezanom grafu bismo mogli da odredimo tako što bismo za svaki čvor $v \in V$ grafa G proverili da li je graf bez čvora v povezan. Složenost ovog algoritma je $O(|V| \cdot (|V| + |E|))$. Postoje efikasniji algoritmi za određivanje artikulacionih tačaka i mostova u grafu. Mi ćemo u nastavku razmotriti algoritme koje su zajedno osmislili Tardžan i Hopkroft i koji su linearne vremenske složenosti. S obzirom na to da je algoritam za pronalaženje mostova donekle jednostavniji, krenućemo od njega.

Tardžanov algoritam za traženje mostova u grafu

Važi naredno tvrđenje: grana (u, v) je most u grafu G ako i samo ako ne pripada nijednom ciklusu u grafu G (jer nakon izbacivanja grane (u, v) ne postoji način kako doći od čvora u do čvora v). Specijalno, ako je graf drvo, onda je svaka grana u tom grafu most (slika 9).

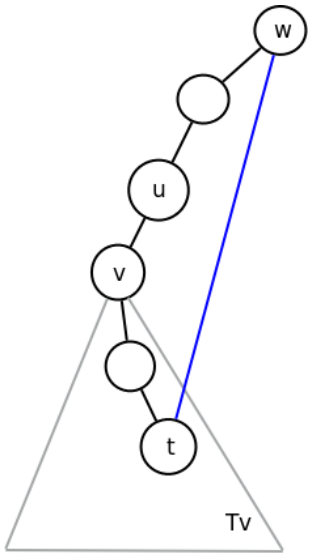


Slika 10: Grana (u, v) koja povezuje potomka i pretka u DFS drvetu ne može biti most.

Razmotrimo DFS drvo dobijeno DFS obilaskom datog grafa $G = (V, E)$. S obzirom na to da je polazni graf neusmeren, postoje dve vrste grana grafa u odnosu na DFS drvo: grane DFS drveta i grane koje povezuju potomka sa pretkom u odnosu na DFS drvo. Ako grana (u, v) povezuje potomka sa pretkom ona ne može biti most u grafu jer je deo ciklusa koji ta grana čini sa granama DFS drveta (slika 10). Dakle, mostovi mogu biti samo grane DFS drveta, te je dovoljno da algoritam razmatra samo njih kao kandidate. Algoritam se može dalje uprostiti. Neka je (u, v) grana DFS drveta. Pretpostavimo da je DFS

[2]Primetimo da je uklanjanje grane iz grafa u slučaju kada je graf zadat listama povezanosti neefikasno, ali pošto se nakon uklanjanja svake grane izvršava algoritam pretrage u dubine, ova operacija neće uticati na složenost kompletnog algoritma

pretraga najpre posetila čvor u pa čvor v, tj. da je čvor u roditelj čvora v u DFS drvetu grafa. Za granu (u, v) DFS drveta važi da je most ako njenim uklanjanjem graf postaje nepovezan, tj. poddrvo DFS drveta grafa G sa korenom u čvoru v ostaje nepovezano sa delom grafa "iznad" ove grane. Ovo važi u slučaju kad ne postoji način da se (nekom granom od potomka ka pretku) stigne iz poddrveta sa korenom u čvoru v do čvora u ili pretka čvora u. Kako se ovo može efikasno utvrditi?



Slika 11: Ilustracija definicije vrednosti $L(v)$.

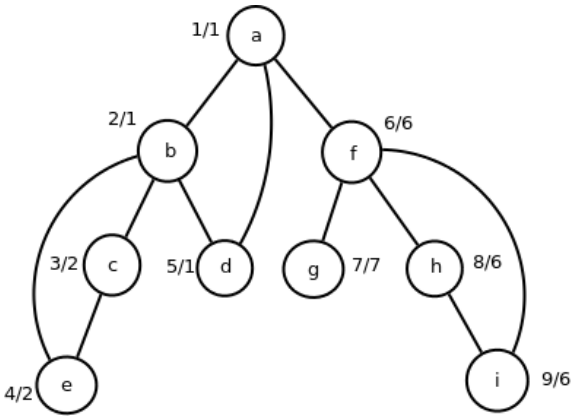
Potrebno je za svaki čvor $v \in V$ izračunati koliko se "visoko" možemo vratiti povratnim granama iz nekog čvora poddrveta sa korenom u čvoru v - to možemo kvantifikovati vrednošću dolazne DFS numeracije čvora do koga povratna grana vodi. Dakle, za svaki čvor $v \in V$ može se odrediti vrednost dolazne numeracije $v.Pre$ pri DFS obilasku – tu vrednost ćemo u nastavku kraće zvati rednim brojem čvora v. Neka je T_v poddrvo DFS drveta sa korenom u čvoru v (slika 11). Označimo sa $L(v)$ (eng. low link) manju od vrednosti rednog broja čvora v i najmanje među vrednostima rednih brojeva čvorova do kojih se može stići granom ka pretku čvora v iz proizvoljnog čvora poddrveta T_v (ne razmatrajući grane koje vode ka roditeljskom čvoru u DFS drvetu)[3]. Tada je:

$$L(v) = \min\{v.Pre, \min_{\substack{w \text{ je predak od } v \\ \text{postoji grana } (t,w),\ t \in T_v}} w.Pre\}.$$

Na slici 12 dat je primer grafa gde je uz svaki čvor v prikazana vrednost dolazne numeracije $v.Pre$ i vrednost $L(v)$. Na primer, važi da je $L(e) = 2$ jer iz čvora

[3]Moguće je da iz poddrveta DFS drveta sa korenom u čvoru v ne postoji nijedna grana ka pretku čvora v, pa onda vrednost $L(v)$ postavljamo na vrednost dolazne numeracije čvora v.

e granom (e, b) možemo stići do čvora b čija je dolazna numeracija jednaka 2. Slično, važi $L(b) = 1$ jer iz čvora d koji je potomak čvora b u DFS drvetu možemo stići granom (d, a) do čvora a čija je dolazna numeracija jednaka 1.



Slika 12: Primer grafa i odgovarajućeg DFS drveta: uz svaki čvor v prikazana je vrednost njegove dolazne numeracije i vrednost $L(v)$.

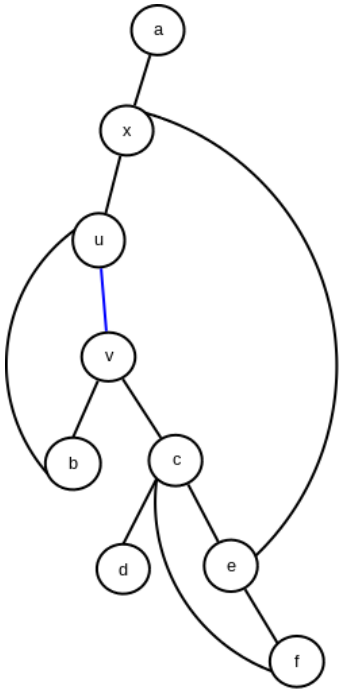
Poddrvo T_v DFS drveta sa korenom u čvoru v ostaće nakon izbacivanja grane (u, v) nepovezano sa delom grafa "iznad" ove grane ako i samo ako važi $L(v) > u.Pre$. Dakle, grana (u, v) je most u grafu G ako i samo ako važi $L(v) > u.Pre$.

Razmotrimo graf prikazan na slici 13. Grana (u, v) nije most u grafu jer se iz poddrveta T_v možemo vratiti u deo DFS drveta iznad ove grane. Preciznije, iz čvora b možemo se vratiti u čvor u, a iz čvora e u čvor x. Vrednost $L(v)$ biće jednaka rednom broju čvora x, a važi $x.Pre < u.Pre$ jer je x predak čvora u u DFS drvetu, te nije zadovoljen uslov $L(v) > u.Pre$. Čak i kad graf ne bi sadržao granu (x, e), nakon izbacivanja grane (u, v) mogli bismo se granom (b, u) iz poddrveta T_v vratiti do čvora u, a time i do proizvoljnog čvora iznad njega u DFS drvetu, te i u tom slučaju grana (u, v) ne bi bila most u grafu.

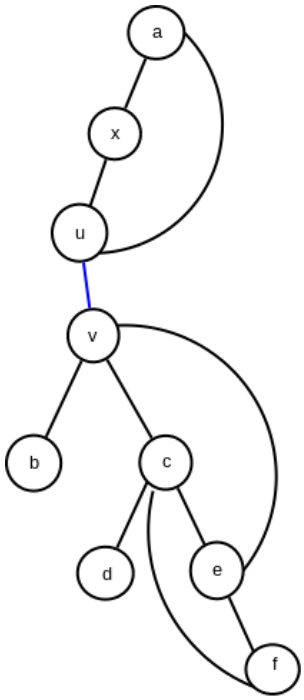
Razmotrimo sada graf sa slike 14. Nakon izbacivanja grane (u, v) iz grafa, iz poddrveta T_v možemo se vratiti najviše do čvora v, tj. važiće uslov $L(v) = v.Pre > u.Pre$ i grana (u, v) jeste most u ovom grafu.

Ostaje pitanje kako efikasno izračunati vrednosti $L(v)$ za sve čvorove v u grafu. Pokazuje se da se one mogu odrediti tokom DFS obilaska grafa, tako što se prilikom obrade grane (u, v) u toku poziva DFS algoritma za čvor u radi sledeće:

- ako grana (u, v) povezuje potomka i pretka u odnosu na DFS drvo i čvor v je predak čvora u, onda ako je $v.Pre < L(u)$, ažurira se vrednost $L(u)$ na $v.Pre$;
- ako je čvor v neoznačen, označava se, povezuje granom DFS drveta sa roditeljskim čvorom u i dobija vrednost $v.Pre$; tom vrednošću se inicijalizuje i vrednost $L(v)$; nakon rekurzivne obrade kompletnog poddrveta



Slika 13: Primer grafa u kome grana (u, v) nije most.



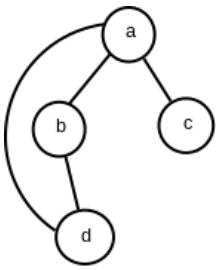
Slika 14: Primer grafa u kome je grana (u, v) most.

sa korenom u čvoru v proveravamo da li za roditeljski čvor u važi uslov $L(u) > L(v)$, i ako važi ažurira se vrednost $L(u)$ na $L(v)$.

```
vector<vector<int>> listaSuseda {{1, 2}, {0, 3, 4}, {0, 5, 8}, {1},
                                 {1, 6, 7}, {2,8}, {4}, {4}, {5,2}};
int vreme_dolazna = 1;
vector<bool> posecen;
vector<int> dolazna;
vector<int> low_link;
vector<int> roditelj;
// niz grana koje su mostovi u grafu
vector<pair<int,int>> most;

void dfs(int cvor){
  posecen[cvor] = true;
  dolazna[cvor] = low_link[cvor] = vreme_dolazna;
  vreme_dolazna++;

  // rekurzivno prolazimo kroz sve susede koje nismo obisli
  for (auto sused : listaSuseda[cvor]){
    if (!posecen[sused]){
      roditelj[sused] = cvor;
      // pokrecemo pretragu iz cvora 'sused'
      dfs(sused);

      // nakon obrade poddrveta sa korenom u cvoru 'sused'
      // vrednost L cvora 'sused' je finalna;
      // po potrebi azuriramo vrednost L za cvor 'cvor'
      if (low_link[sused] < low_link[cvor])
         low_link[cvor] = low_link[sused];

      // prilikom povratka granom proveravamo
      // da li je ispunjen uslov da nijedan cvor
      // u poddrvetu sa korenom u cvoru 'sused'
      // nije povezan sa nekim pretkom cvora 'cvor'
      if (low_link[sused] > dolazna[cvor])
         most.emplace_back(cvor,sused);
    }
    // ukoliko je sused vec posecen
    // i ako grana vodi ka nekom pravom pretku datog cvora
    // postavljamo vrednost L datog cvora na vrednost
    // dolazne numeracije suseda, ako je manja od tekuce vrednosti
    else if (sused != roditelj[cvor])
      if (dolazna[sused] < low_link[cvor])
         low_link[cvor] = dolazna[sused];
```

```
  }
}

void ispisi_mostove(int cvor){
  int brojCvorova = listaSuseda.size();
  posecen.resize(brojCvorova, false);
  dolazna.resize(brojCvorova);
  low_link.resize(brojCvorova);
  roditelj.resize(brojCvorova, -1);

  dfs(cvor);

  cout << "Mostovi u grafu su: ";
  for (int i = 0; i < most.size(); i++)
   cout << "(" << most[i].first << ", " << most[i].second << ") " ;
  cout << endl;
}

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

Primer: Razmotrimo kako bi na primeru jednostavnog neusmerenog grafa sa slike 15 teklo izvršavanje opisanog algoritma. Pretpostavimo da je graf zadat listama povezanosti tako da su susedi svakog čvora uređeni leksikografski rastuće i da DFS pretraga započinje iz čvora a.



Slika 15: Primer grafa koji sadrži jedan most: granu (a, c)

```
Pokrecemo DFS iz cvora a, postavljamo a.Pre <- 1 i L(a) <- 1
Razmatramo suseda b čvora a

Pokrecemo DFS iz cvora b, postavljamo b.Pre <- 2 i L(b) <- 2
Razmatramo suseda a čvora b
To je grana ka roditelju, koju dalje ne obradjujemo
Razmatramo suseda d čvora b
```

```
Pokrecemo DFS iz cvora d, postavljamo d.Pre <- 3 i L(d) <- 3
Razmatramo suseda a čvora d
Grana (d,a) je grana od potomka ka pretku,
pa postavljamo L(d) <- a.Pre, i dobijamo L(d) = 1
Razmatramo suseda b čvora d
To je grana ka roditelju, koju dalje ne obradjujemo

Vracamo se u cvor b
Posto vazi L(d) < L(b) postavljamo L(b) <- L(d),
pa vazi i L(b) = 1
S obzirom na to da je L(d) < b.Pre grana (b,d) nije most

Vracamo se u cvor a
Posto vazi L(b) = L(a) ne radimo nista
S obzirom na to da je L(b) = a.Pre grana (a,b) nije most

Razmatramo suseda d čvora a
Posto vazi L(d) = L(a) ne radimo nista

Pokrecemo DFS iz cvora c, postavljamo c.Pre <- 4 i L(c) <- 4
Razmatramo suseda a čvora c
To je grana ka roditelju, koju dalje ne obradjujemo

Vracamo se u cvor a
Posto vazi L(c) > L(a) ne radimo nista
S obzirom na to da je L(c) > a.Pre grana (a,c) jeste most
```

Algoritam za određivanje svih mostova u grafu se zasniva na DFS pretrazi, sa odgovarajućom dolaznom i odlaznom obradom koja je složenosti $O(1)$, pa je vremenska složenost ovog algoritma $O(|V| + |E|)$.

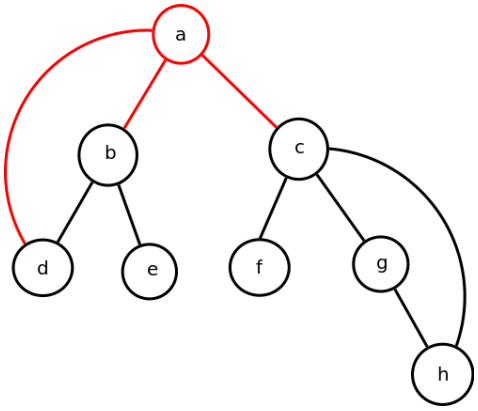
Tardžanov algoritam za traženje artikulacionih tačaka u grafu

Veoma nalik prethodnom algoritmu je i algoritam za traženje artikulacionih tačaka u grafu. Čvor u biće artikulaciona tačka grafa ako je ispunjen jedan od naredna dva uslova:

- u je koren DFS drveta i ima bar dva deteta;
- u nije koren DFS drveta i ima dete v u DFS drvetu takvo da nijedan čvor u poddrvetu T_v nije povezan sa nekim pretkom čvora u u DFS drvetu.

Ako je zadovoljen prvi uslov, s obzirom na to da u neusmerenim grafovima ne postoje poprečne grane, izbacivanje korena DFS drveta dovelo bi do "razbijanja" grafa na veći broj komponenti povezanosti (po jednu za svako dete korena DFS drveta). Drugi uslov odgovara situaciji kada nakon izbacivanja čvora u iz grafa više nije moguće doći iz proizvoljnog čvora poddrveta sa korenom u detetu v

čvora u do nekog pretka čvora u.



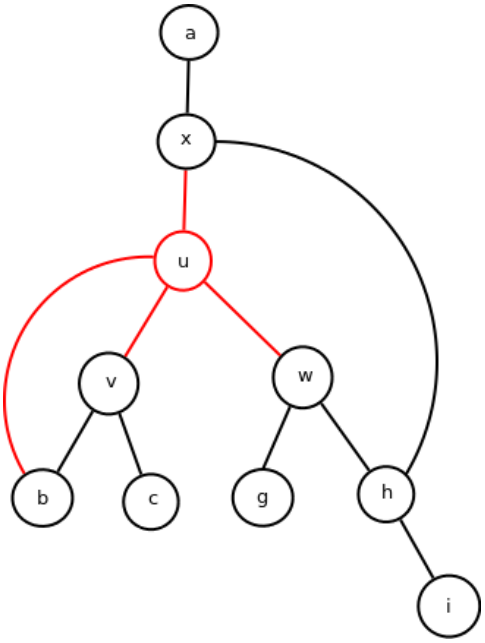
Slika 16: Primer grafa u kome je čvor a kao koren DFS drveta artikulaciona tačka.

Razmotrimo graf sa slike 16: čvor a kao koren DFS drveta ima dva deteta i nakon njegovog izbacivanja graf postaje nepovezan. Dakle, čvor a je artikulaciona tačka u grafu. Ovaj slučaj se može detektovati tako što prilikom DFS obilaska za svaki čvor vršimo proveru da li ima roditelja u DFS drvetu (jedino koren DFS drveta nema roditelja) i brojimo koliko čvor ima dece. Ako čvor nema roditelja i ima bar dva deteta, zaključujemo da je artikulaciona tačka.

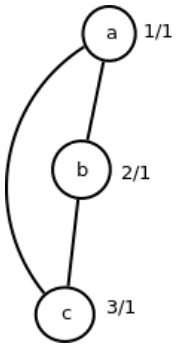
Razmotrimo primer grafa sa slike 17. Čvor u ima dva deteta u DFS drvetu: v i w. Nakon izbacivanja čvora u i njemu susednih grana iz grafa, iz poddrveta T_w možemo se vratiti u deo grafa iznad čvora u (jer je $L(w) = x.Pre$), međutim, iz poddrveta T_v možemo se vratiti najviše do čvora u (jer važi $L(v) = u.Pre$). S obzirom na to da čvor u ima dete v tako da nijedan čvor iz poddrveta T_v nije povezan sa nekim pretkom čvora u u DFS drvetu, čvor u jeste artikulaciona tačka u grafu.

Drugi uslov da bi čvor bio artikulaciona tačka sličan je uslovu za postojanje mosta: čvor u je artikulaciona tačka u grafu ako za nekog dete v čvora u važi uslov $L(v) \geq u.Pre$. Primetimo da za koren a DFS drveta grafa sa slike 18 i njegovo dete b važi uslov $L(b) = a.Pre$, a pritom čvor a nije artikulaciona tačka. Naime, kada je čvor koren DFS drveta ne postoji deo DFS drveta iznad njega. Dakle, slučaj čvora koji je koren DFS drveta mora se zasebno razmatrati.

```
vector<vector<int>> listaSuseda {{1, 2}, {0, 3, 4}, {0, 5, 8},
                      {1}, {1, 6, 7}, {2,8}, {4}, {4}, {5,2}};
int vreme_dolazna = 1;
vector<bool> posecen;
vector<int> dolazna;
vector<int> low_link;
vector<int> roditelj;
```



Slika 17: Primer grafa u kome je čvor u artikulaciona tačka.



Slika 18: Ilustracija grafa u kome za koren DFS drveta a i njegovo dete b važi uslov $L(b) \geq a.Pre$, a koren a DFS drveta nije artikulaciona tačka.

```
vector<bool> artikulacionaTacka;

void dfs(int cvor){
  posecen[cvor] = true;
  dolazna[cvor] = low_link[cvor] = vreme_dolazna;
  vreme_dolazna++;
  int broj_dece = 0;

  // rekurzivno prolazimo kroz sve susede koje nismo obisli
  for (auto sused : listaSuseda[cvor]){
    if (!posecen[sused]){
      // 'sused' postaje dete cvora 'cvor' u DFS drvetu
      broj_dece++;
      roditelj[sused] = cvor;
      // pokrecemo pretragu iz cvora 'sused'
      dfs(sused);

      // nakon obrade poddrveta sa korenom u susednom cvoru
      // po potrebi azuriramo vrednost L za cvor 'cvor'
      if (low_link[sused] < low_link[cvor])
         low_link[cvor] = low_link[sused];

      // proveravamo da li 'cvor' nije koren DFS drveta i
      // da li za cvor 'sused' vazi da nijedan cvor u njegovom poddrvetu
      // nije povezan sa nekim pretkom datog cvora
      if (roditelj[cvor] != -1
          && low_link[sused] >= dolazna[cvor])
         // ako je uslov ispunjen, zakljucujemo da je
         // cvor 'cvor' artikulaciona tacka
         artikulacionaTacka[cvor] = true;
    }
    else // posecen[sused]
    if (sused != roditelj[cvor])
      // po potrebi azuriramo vrednost L za cvor 'cvor'
      if (dolazna[sused] < low_link[cvor])
         low_link[cvor] = dolazna[sused];
  }
  // kada obradimo svu decu cvora 'cvor'
  // proveravamo da li je cvor 'cvor' koren DFS drveta
  // i da li ima vise od jednog deteta
  if (roditelj[cvor] == -1 && broj_dece > 1)
    // ako je uslov ispunjen, cvor je artikulaciona tacka
    artikulacionaTacka[cvor] = true;

}
```

```
void ispisi_artikulacione_tacke(int cvor){
  int brojCvorova = listaSuseda.size();
  posecen.resize(brojCvorova, false);
  dolazna.resize(brojCvorova);
  low_link.resize(brojCvorova);
  roditelj.resize(brojCvorova, -1);
  artikulacionaTacka.resize(brojCvorova, false);

  dfs(cvor);

  cout << "Artikulacione tacke u grafu su: ";
  for (int i=0; i<artikulacionaTacka.size(); i++){
    if (artikulacionaTacka[i])
      cout << i << " ";
 }
 cout << endl;
}

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

Komponente jake povezanosti grafa

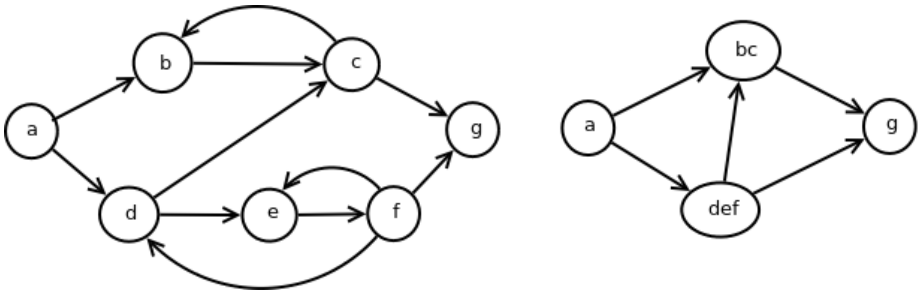
Za usmereni graf kažemo da je *jako povezan* ako je svaki čvor grafa dostižan iz svakog drugog čvora u grafu.

Na skupu čvorova usmerenog grafa $G = (V, E)$ može se definisati relacija $\sim$ *obostrane dostižnosti*: $u \sim v$ ako je čvor u dostižan iz čvora v i čvor v je dostižan iz čvora u. Pored toga, po definiciji za svaki čvor u važi $u \sim u$ (napomenimo da to ne znači da u grafu postoje petlje). Za relaciju obostrane dostižnosti važi da je:

- refleksivna – za svaki čvor $u \in V$ važi $u \sim u$,
- simetrična – za svaka dva čvora $u, v \in V$ važi $u \sim v$ akko $v \sim u$,
- tranzitivna – za svaka tri čvora $u, v, w \in V$ iz $u \sim v$ i $v \sim w$ sledi i $u \sim w$.

Relacija obostrane dostižnosti je stoga relacija ekvivalencije. Ona razlaže skup čvorova V u klase ekvivalencije koje nazivamo *komponentama jake povezanosti* grafa G (eng. strongly connected components). Na slici 19 (levo) prikazan je usmereni graf G koji ima četiri komponente jake povezanosti koje se sastoje redom od čvorova $\{a\}$, $\{b, c\}$, $\{d, e, f\}$ i $\{g\}$. Primetimo da dva čvora pripadaju istoj komponenti jake povezanosti ako i samo ako pripadaju nekom zajedničkom usmerenom ciklusu. Od grafa G može se formirati *komprimovani* graf G^C: to je usmereni aciklički graf koji se sastoji od komponenti jake povezanosti grafa G (slika 19, desno). Naime, svaki čvor u grafu G^C odgovara jednoj komponenti

jake povezanosti grafa G, a dva čvora u grafu G^C su povezana granom ako i samo ako u grafu G postoji bar jedna grana od nekog čvora prve komponente do nekog čvora druge komponente jake povezanosti. Jasno je da je graf G^C aciklički: ako bi u njemu postojao ciklus, to bi značilo da se sve komponente jake povezanosti koje pripadaju ciklusu mogu spojiti u jednu, veću komponentu jake povezanosti. U nastavku teksta ćemo komponente jake povezanosti zvati kraće samo komponente.

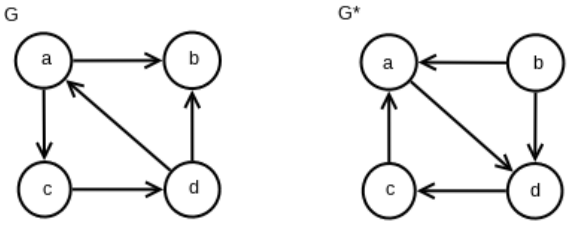


Slika 19: Graf G i odgovarajući komprimovani graf G^C čiji čvorovi odgovaraju komponentama jake povezanosti grafa G.

Jedan (direktan) način za određivanje komponenti u grafu $G = (V, E)$ sastojao bi se u tome da se za neki čvor $v_0 \in V$ odredi koji čvorovi pripadaju njegovoj komponenti, tako što bi se za sve preostale čvorove proveravalo da li su obostrano dostižni iz v_0 – to bi se moglo sprovesti pokretanjem DFS pretrage iz čvora v_0 (tako bi otkrili sve čvorove u dostižne iz čvora v_0) i iz svakog čvora u do kog se DFS pretragom stigne (tako bi proverili da li je iz čvora u dostižan čvor v_0). Nakon toga bi se sličan postupak ponavljao za neki čvor koji ne pripada komponenti kojoj pripada čvor v_0 (ako takav čvor postoji). Jasno je da bi ovo bilo veoma neefikasno i zahtevalo bi veliki broj pokretanja DFS algoritma.

Neka je G^* graf koji sadrži iste čvorove kao graf G, ali suprotno usmerene grane: ako grana (u, v) pripada grafu G, onda grana (v, u) pripada grafu G^*. Ovaj graf zvaćemo *transponovanim grafom* (eng. transpose graph) grafa G. Komponente bismo mogli da odredimo i na sledeći način: pokrenemo DFS obilazak iz čvora v_0 u grafu G i DFS obilazak iz čvora v_0 u grafu G^*. Prvi DFS obilazak pronalazi skup čvorova A koji su dostižni iz čvora v_0, a drugi DFS obilazak skup čvorova B iz kojih je dostižan čvor v_0. Komponenta jake povezanosti kojoj pripada čvor v_0 jednaka je $A \cap B$. Ovaj postupak bismo ponavljali za proizvoljni čvor koji ne pripada do sada određenim komponentama povezanosti, ukoliko takav čvor postoji. Razmotrimo primer grafa G i odgovarajućeg grafa G^* prikazanih na slici 20: DFS obilazak grafa G pokrenut iz čvora d obilazi skup čvorova $A = \{d, a, b, c\}$, dok DFS obilazak grafa G^* pokrenut iz čvora d obilazi skup čvorova $B = \{d, c, a\}$. S obzirom na to da je $A \cap B = \{d, a, c\}$, čvorovi d, a i c pripadaće istoj komponenti jake povezanosti. Primetimo da jedino čvor b ne pripada istoj komponenti povezanosti kao čvor d, a pošto DFS obilazak iz čvora

b u grafu G obilazi samo čvor b (jer je njegov izlazni stepen 0), to će čvor b činiti sam za sebe drugu (preostalu) komponentu jake povezanosti grafa G.



Slika 20: Graf G i njemu transponovani graf G^*.

Postoji nekoliko različitih algoritama linearne vremenske složenosti za određivanje komponenti jake povezanosti u usmerenom grafu, a najpoznatiji među njima su Tardžanov algoritam i Kosaradžuov algoritam. Oba algoritma su zasnovana na DFS obilasku grafa, samo se kod Tardžanovog algoritma sve radi u jednom prolazu kroz graf, dok se u Kosaradžuovom algoritmu dva puta poziva algoritam DFS obilaska. U nastavku ćemo razmotriti prvi od dva pomenuta algoritma.

Tardžanov algoritam

Prilikom DFS obilaska datog usmerenog grafa G implicitno se formira DFS drvo, odnosno DFS šuma. Bez narušavanja opštosti možemo pretpostaviti da je graf takav da postoji čvor iz kog se on može u potpunosti obići, odnosno da ima DFS drvo. Ako takav čvor ne postoji može se, na primer, dodati novi čvor v ulaznog stepena 0, i povezati granama sa svim čvorovima grafa G (slika 21); tada DFS pretraga pokrenuta iz čvora v sigurno obilazi sve čvorove grafa G. Prošireni graf sem komponenti grafa G ima samo još jednu dodatnu jednočlanu komponentu jake povezanosti $\{v\}$. Naime, nijedan drugi čvor ne može biti sa njim u komponenti jake povezanosti jer je ulazni stepen čvora v jednak 0.

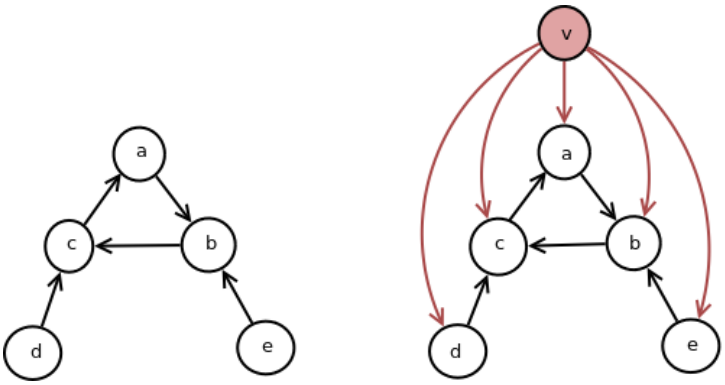
Nazovimo *baznim čvorom* b (eng. base vertex) komponente X onaj čvor te komponente koji ima najmanji redni broj pri dolaznoj DFS numeraciji:

$$b.Pre = \min_{v \in X} v.Pre$$

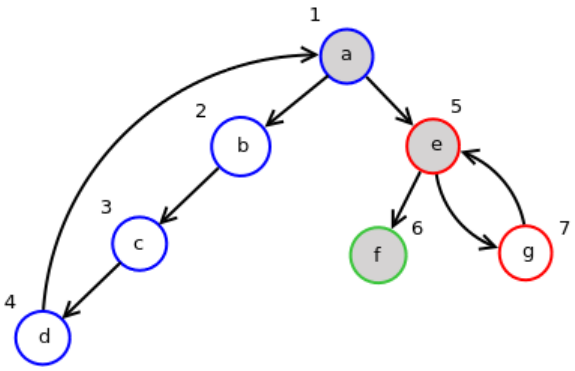
Razmotrimo graf sa slike 22: on sadrži tri komponente jake povezanosti: $\{a, b, c, d\}$, $\{e, g\}$ i $\{f\}$. Bazni čvor prve komponente biće čvor a, druge e, a treće f.

Lema 1: Neka je b bazni čvor komponente X. Tada za svako $v \in X$ važi da je v potomak čvora b u DFS drvetu i svi čvorovi na putu od b do v kroz grane DFS drveta pripadaju komponenti X.

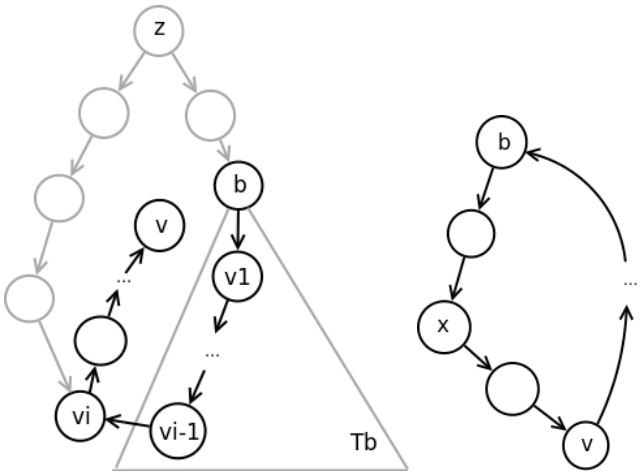
Dokaz: Pretpostavimo suprotno, odnosno da u komponenti X postoje čvorovi koji nisu u poddrvetu T_b DFS drveta sa korenom u b, i neka je v jedan od takvih čvorova. Neka je $v_0 = b, v_1, \ldots, v_k = v$ put od b do v kroz čvorove komponente



Slika 21: Graf nadograđujemo čvorom v kako bi postojao čvor iz koga se graf može u potpunosti obići.



Slika 22: Graf koji sadrži tri komponente jake povezanosti grafa. Uz svaki čvor data je njegova vrednost dolazne numeracije. Sivom bojom označeni su bazni čvorovi svake komponente.



Slika 23: Ilustracija uz dokaz prvog i drugog dela leme 1.

X. Neka je v_i prvi čvor na tom putu koji nije u poddrvetu T_b (dakle, čvor b je predak čvorova v_j za $j = 1, \ldots, i-1$). Tada je grana (v_{i-1}, v_i) poprečna u odnosu na DFS drvo. Naime, pošto v_i nije u poddrvetu T_b grana (v_{i-1}, v_i) nije grana DFS drveta, a ne može biti ni povratna jer bi onda čvor v_i bio predak čvora b i b ne bi bio bazni čvor te komponente. Sve poprečne grane u odnosu na DFS drvo su usmerene ulevo, pa i grana (v_{i-1}, v_i). Dakle, čvor v_i je levo od čvora v_{i-1}, pa važi $v_i.Pre < v_{i-1}.Pre$. Dodatno, oni imaju zajedničkog pretka z u DFS drvetu (slika 23 levo). Čvor z ne može biti jedan od čvorova $v_1, v_2, \ldots, v_{i-1}$ jer bi inače čvor v_i bio u poddrvetu sa korenom u čvoru b. Pošto je b predak čvora v_{i-1}, onda je z i zajednički predak čvorova v_i i b, te je čvor v_i levo i od čvora b. DFS obilazak iz čvora v_i počinje (i završava se) pre početka DFS obilaska iz čvora b, tj. važi $v_i.Pre < b.Pre$. Međutim, čvor v_i pripada komponenti u kojoj je bazni čvor b, jer je obostrano dostižan sa čvorom b: b je dostižan iz čvora v_i putem od v_i do $v_k = v$ i dalje putem od v_k do b koji postoji jer je v, po pretpostavci, u ovoj komponenti. Ovo je u suprotnosti sa pretpostavkom da je b bazni čvor ove komponente. Dakle, svi čvorovi komponente X se nalaze u poddrvetu T_b DFS drveta.

Dokažimo drugi deo leme. Neka je x proizvoljni čvor na putu od b do v (slika 23 desno). Postoji put od čvora b do čvora x kroz grane DFS drveta, a takođe i put od čvora x do čvora b tako što najpre pratimo put od x do v kroz grane DFS drveta, pa od v do b (ovaj put postoji jer čvorovi v i b pripadaju istoj komponenti). Stoga je x u istoj komponenti kao i čvor b. Zaključujemo da svi čvorovi na putu od čvora b do čvora v kroz grane DFS drveta pripadaju komponenti čiji je bazni čvor b. □

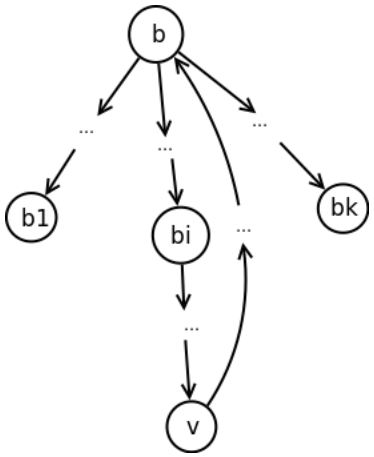
Primetimo da ovo važi za graf prikazan na slici 22. Naime, čvorovi koji pripadaju prvoj komponenti su $\{a, b, c, d\}$ i čvorovi b, c i d jesu potomci baznog čvora te

komponente – čvora a u DFS drvetu. Slično važi i za čvor g koji je potomak baznog čvora e svoje komponente.

Posebno, s obzirom da čvorovi a i d pripadaju istoj, prvoj komponenti jake povezanosti grafa, na osnovu drugog dela Leme 1 to važi i za čvorove b i c koji se nalaze na putu od čvora a do čvora d kroz grane DFS drveta.

Istaknimo još da su svi čvorovi $v \in X$ potomci baznog čvora b komponente X u DFS drvetu, ali da ne moraju svi potomci čvora b u DFS drvetu biti u komponenti X: čvorovi e, f i g grafa sa slike 22 su potomci čvora a, ali se ne nalaze sa njim u istoj komponenti.

Lema 2: Neka je b bazni čvor neke komponente i neka su $b_1, b_2, \ldots, b_k$ bazni čvorovi nekih drugih komponenti koji su potomci čvora b. Tada važi da se komponenta kojoj pripada čvor b sastoji od svih potomaka čvora b koji nisu potomci nijednog od čvorova $b_1, b_2, \ldots, b_k$.



Slika 24: Ilustracija uz dokaz leme 2.

Dokaz: Pretpostavimo suprotno, odnosno da postoji čvor v koji je u istoj komponenti kao i čvor b, a koji je potomak i čvora b i čvora b_i za neko i, $1 \leq i \leq k$ (slika 24). Pošto je b_i potomak čvora b, onda postoji put kroz grane DFS drveta od čvora b do čvora b_i. Slično, pošto je čvor v potomak čvora b_i postoji put od čvora b_i do čvora v, a na osnovu pretpostavke da su v i b u istoj komponenti, postoji i put od čvora v do čvora b. Nadovezivanjem ova dva puta dobija se put od čvora b_i do b. Odavde sledi da su čvorovi b i b_i u istoj komponenti što je u suprotnosti sa pretpostavkom leme jer su b i b_i bazni čvorovi različitih komponenti. Dakle, čvorovi koji su u istoj komponenti kao i čvor b ne mogu biti potomci i nekog drugog baznog čvora b_i, koji je potomak čvora b. □

U grafu prikazanom na slici 22 čvor a je bazni čvor. Čvorovi e i f su takođe bazni čvorovi i pritom su potomci baznog čvora a. Primetimo da su čvorovi koji pripadaju komponenti čiji je bazni čvor a oni koji su potomci od a, a nisu

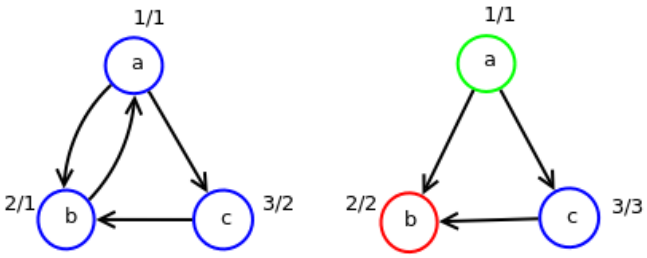
potomci ni čvora e ni čvora f: to su čvorovi b, c i d. Slično, čvorovi koji pripadaju komponenti sa baznim čvorom e su oni koji su potomci od e, a nisu potomci od f, što je samo čvor g.

Lema 1 i Lema 2 nam daju mogućnost da izračunamo koji su čvorovi zajedno sa nekim baznim čvorom u istoj komponenti jake povezanosti. Nedostaje još samo da formulišemo neki efektivni test kojim bismo mogli da ispitamo da li je neki čvor bazni čvor neke komponente.

S obzirom na to da je graf sa kojim radimo usmeren i može imati poprečne grane u odnosu na DFS drvo, vrednost low link $L(v)$ za čvorove v grafa definisaćemo nešto drugačije nego u slučaju neusmerenih grafova. Neka je X komponenta koja sadrži čvor v. Označimo sa A najmanji redni broj čvora komponente X do koga se iz čvora v može stići putem koji se sastoji od grana DFS drveta i koji se završava najviše jednom povratnom ili poprečnom granom. Vrednost $L(v)$ definisaćemo kao $\min\{v.Pre, A\}$, odnosno:

$$L(v) = \min\{v.Pre, \min_{\substack{(t,w) \text{ je poprečna ili povratna} \\ \text{postoji grana } (t,w),\ t \in T_v}} w.Pre\}.$$

Primetimo da poprečna grana (t, w) može povezivati dva čvora iz iste ili iz različite komponente (slika 25). Ukoliko povezuje čvorove iz iste komponente, onda se poprečna grana (t, w) razmatra prilikom definisanja vrednosti $L(v)$, a inače ne.

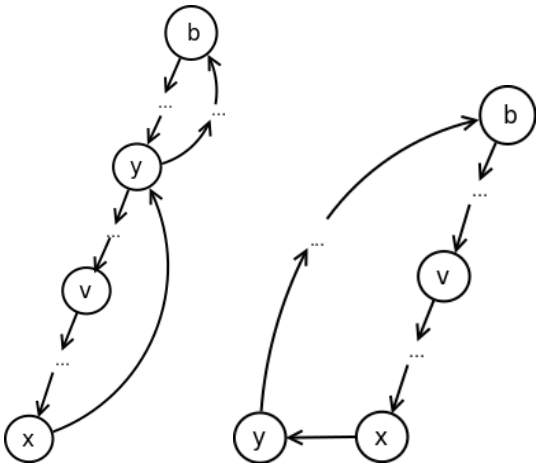


Slika 25: Ilustracija scenarija kada poprečna grana (c, b) vodi ka čvoru koji pripada istoj komponenti i kada poprečna grana vodi ka čvoru koji ne pripada istoj komponenti.

Lema 3: Čvor v je bazni čvor ako i samo ako važi $L(v) = v.Pre$.

Dokaz: Pokažimo prvi smer tvrđenja: ako je čvor v bazni onda važi $L(v) = v.Pre$. Pretpostavimo suprotno – da za bazni čvor v važi $L(v) < v.Pre$ i pokažimo da onda čvor v nije bazni čvor. Prema definiciji vrednosti L, postoji čvor w u istoj komponenti kao i v takav da je $L(v) = w.Pre$. Stoga je $w.Pre < v.Pre$ te čvor v ne može biti bazni čvor.

Dokažimo sada suprotni smer implikacije: ako za čvor v važi uslov $L(v) = v.Pre$, onda je čvor v bazni. Pretpostavimo suprotno: da važi uslov $L(v) = v.Pre$, a



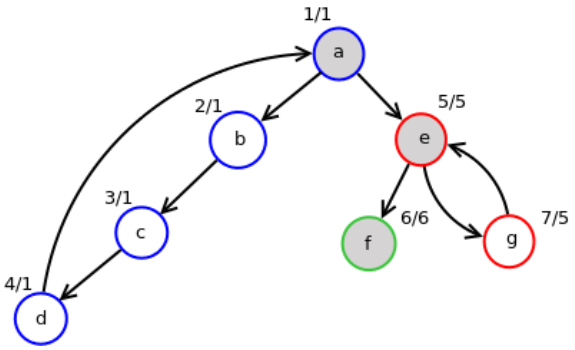
Slika 26: Ilustracija uz dokaz prvog i drugog slučaja u lemi 3.

da čvor v nije bazni čvor. Neka je b bazni čvor komponente koja sadrži čvor v. Prema Lemi 1 čvor b je predak čvora v. Pošto su b i v u istoj komponenti, postoji prosti put p od v do b. Neka je y prvi čvor na putu p koji nije u poddrvetu sa korenom u v i neka je x čvor koji mu prethodi na putu p:

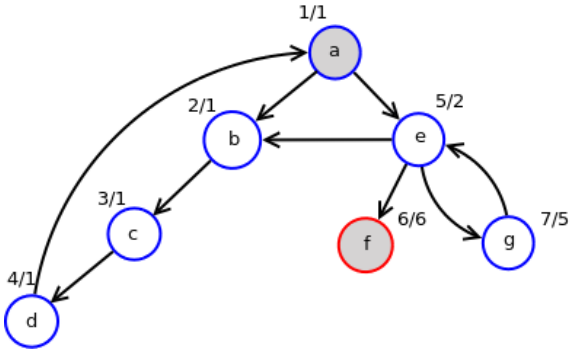
- ako je grana (x, y) povratna (slika 26, levo), onda je y predak čvora v i $L(v) \leq y.Pre < v.Pre$, suprotno pretpostavci (čvor y nije na putu od v do x, jer je put p po pretpostavci prost);
- ako je grana (x, y) poprečna (slika 26 desno), onda je ona usmerena ulevo i $y.Pre < x.Pre$. S obzirom na to da su poddrvo sa korenom u y i poddrvo sa korenom u v disjunktni i da postoji grana od potomka čvora v (odnosno x) do y, na osnovu svojstava poprečnih grana možemo zaključiti da važi $y.Pre < v.Pre$. Odavde sledi da je $L(v) \leq y.Pre$ jer je y u istoj komponenti jake povezanosti kao i v i dostižan je preko grana DFS drveta nakon koje sledi jedna povratna ili poprečna grana. Na osnovu ovoga i činjenice da je $y.Pre < v.Pre$, dobijamo da važi $L(v) < v.Pre$, suprotno pretpostavci. □

Na slici 27 prikazan je graf sa slike 22, samo je uz svaki čvor, pored vrednosti dolazne numeracije, prikazana i vrednost $L(v)$ čvora. Primetimo da su čvorovi za koje važi $v.Pre = L(v)$ baš oni čvorovi koji su bazni čvorovi svojih komponenti.

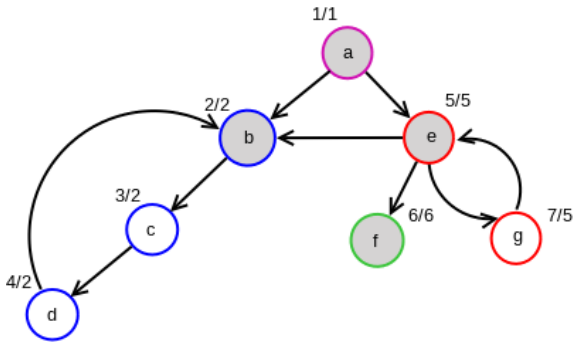
Razmotrimo graf sa slike 28. Grana (e, b) vodi iz čvora e ka čvoru b koji pripada istoj komponenti jake povezanosti, te je tokom pretrage u dubinu prilikom razmatranja grane (e, b) potrebno ažurirati vrednosti $L(e)$ na $b.Pre$. Međutim, u slučaju grafa sa slike 29 možemo uočiti da grana (e, b) povezuje dva čvora koja ne pripadaju istoj komponenti jake povezanosti, te se prilikom obrade grane (e, b) ne treba ažurirati vrednost $L(e)$ – ova vrednost ostaje jednaka $e.Pre$, te će čvor e biti bazni čvor svoje komponente. Zaista, iz čvora e nije moguće stići



Slika 27: Graf koji sadrži tri komponente jake povezanosti. Uz svaki čvor data je njegova vrednost dolazne numeracije i vrednost low link. Sivom bojom označeni su bazni čvorovi svake komponente.



Slika 28: Primer grafa kod koga poprečna grana (e, b) vodi ka čvoru koji pripada istoj komponenti jake povezanosti kao polazni čvor grane. Uz svaki čvor data je njegova vrednost dolazne numeracije i vrednost low link. Sivom bojom označeni su bazni čvorovi svake komponente.



Slika 29: Primer grafa kod koga poprečna grana (e, b) vodi ka čvoru koji ne pripada istoj komponenti jake povezanosti kao polazni čvor grane. Uz svaki čvor data je njegova vrednost dolazne numeracije i vrednost low link. Sivom bojom označeni su bazni čvorovi svake komponente.

do čvora a (pa ova dva čvora ne pripadaju istoj komponenti), a iz čvora b nije moguće doći do čvora e (pa ni ova dva čvora ne pripadaju istoj komponenti).

Razmotrimo najzad kako na osnovu prethodnih lema možemo formulisati algoritam za ispisivanje komponenti jake poveznosti grafa. U te svrhe prilagodićemo algoritam DFS pretrage. Želimo da u neku pogodnu strukturu podataka smeštamo čvorove u redosledu DFS obilaska, a da neposredno pre nego što napustimo neki čvor proverimo da li je on bazni i ako jeste ispišemo (i uklonimo) njega i sve njegove potomke koji se i dalje nalaze u toj strukturi podataka: to će biti čvorovi koji su nakon njega posećeni, pa i nakon njega dodati u tu strukturu. Dakle, elemente treba dodavati na kraj ove strukture podataka i uklanjati ih sa kraja ove strukture. Struktura podataka koju je pogodno koristiti u ove svrhe je stek. Naime, prilikom označavanja čvora v u toku DFS pretrage, čvor v se upisuje na namenski stek. Kada se završi poziv DFS algoritma iz čvora v, ako je v bazni čvor (a to utvrđujemo tako što se uverimo da važi $L(v) = v.Pre$) sa steka se uklanja čvor v i svi čvorovi iznad njega na steku (naravno, unazad: od elementa na vrhu steka pa sve do čvora v). Na taj način ako je čvor v bazni, izdvaja se komponenta koja sadrži čvor v i svi njeni čvorovi uklanjaju se sa steka. Redosled uklanjanja baznih čvorova nam garantuje da ćemo sa baznim čvorovima neke komponente ukloniti samo one potomke koji nisu potomci nekog drugog baznog čvora.

Dokaz korektnosti algoritma može se izvesti indukcijom po broju m komponenti jake povezanosti koje čine graf.

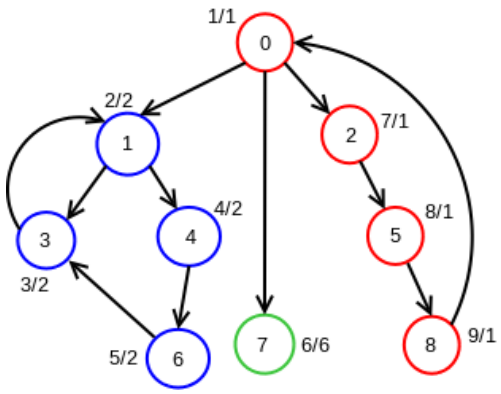
Za $m = 1$, po završetku DFS pretrage sa steka se skidaju svi čvorovi jedine komponente grafa.

Neka je tvrđenje tačno za grafove sa manje od m komponenti i neka je G graf sa m komponenti. Neka je b prvi bazni čvor na koji se nailazi pri DFS pretrazi (to

će biti čvor iz kog pokrećemo DFS pretragu), a neka su $b_1, b_2, \ldots, b_{m-1}$ bazni čvorovi ostalih komponenti. Oni moraju biti potomci čvora b. Prema induktivnoj hipotezi, posle završetka DFS pretrage iz čvora b_i, $1 \leq i \leq m-1$, izdvojene su sve komponente dostižne iz čvora b_i i svi njihovi čvorovi uklonjeni su sa steka. Na osnovu leme 2 važi da kada se završi DFS pretraga iz čvora b, na steku se nalaze tačno čvorovi iz komponente čiji je bazni čvor baš čvor b. Po završetku DFS pretrage se, dakle, sa steka izdvaja poslednja komponenta, kojoj pripada čvor b.

```
vector<vector<int>> listaSuseda {{1, 2}, {3, 4}, {5, 8}, {}, {6, 7},
            {8}, {1}, {}, {0}};
int vreme_dolazna = 1;

void dfs(int cvor, vector<int> &dolazna, vector<int> &low_link,
      stack<int> &redosledUObilasku, vector<int> &u_steku){
  dolazna[cvor] = low_link[cvor] = vreme_dolazna;
  vreme_dolazna++;
  redosledUObilasku.push(cvor);
  u_steku[cvor] = true;

  // rekurzivno prolazimo kroz sve susede koje nismo obisli
  for (auto sused : listaSuseda[cvor]){
    // ako cvor 'sused' do sada nismo posetili
    if (dolazna[sused] == -1){
       // pokrecemo DFS pretragu iz cvora 'sused'
       dfs(sused,dolazna,low_link,redosledUObilasku,u_steku);
       // ako je potrebno azuriramo vrednost L cvora 'cvor'
       if (low_link[sused] < low_link[cvor])
          low_link[cvor] = low_link[sused];
    }
    // ako je u pitanju povratna ili poprecna grana,
    // azuriramo vrednost L za cvor 'cvor'
    // samo ako se sused nalazi u steku
    // to znaci da sused pripada istoj komponenti povezanosti
    else if (u_steku[sused])
      if (dolazna[sused] < low_link[cvor])
         low_link[cvor] = dolazna[sused];
  }

  // ako je u pitanju bazni cvor komponente
  // stampamo sve cvorove te komponente
  if (dolazna[cvor] == low_link[cvor]){
    while(1){
      // ispisujemo element sa vrha steka i uklanjamo ga
      int cvor_komponente = redosledUObilasku.top();
      cout << cvor_komponente << " ";
```

```
      u_steku[cvor_komponente] = false;
      redosledUObilasku.pop();
      // ako smo stigli do baznog cvora prekidamo petlju
      if (cvor_komponente == cvor){
         cout << "\n";
         break;
      }
    }
  }
}

void ispisi_komponente(int cvor){
  int brojCvorova = listaSuseda.size();
  vector<int> dolazna(brojCvorova,-1);
  vector<int> low_link(brojCvorova);
  // stek na koji smestamo cvorove u redosledu DFS obilaska
  stack<int> redosledUObilasku;
  // vektor koji omogucava brzu proveru da li se cvor nalazi na steku
  vector<bool> u_steku(brojCvorova,false);

  cout << "Komponente jake povezanosti su: " << endl;
  dfs(cvor,dolazna,low_link,redosledUObilasku,u_steku);
}

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



Slika 30: Primer grafa koji ima tri komponente jake povezanosti: $\{1, 3, 4, 6\}$, $\{7\}$ i $\{0, 2, 5, 8\}$. Uz svaki čvor prikazan je redni broj u dolaznoj numeraciji i low link vrednost.

Primer: Razmotrimo izvršavanje ovog algoritma na primeru grafa prikazanog na slici 30.

```
stek: 0

stek: 0,1

stek: 0,1,3
zavrsava se rekurzivni poziv iz cvora 3 ali za njega je v.Pre = 3, a
L(v) = 2 pa on nije bazni cvor komponente

stek: 0,1,3,4

stek: 0,1,3,4,6
zavrsava se rekurzivni poziv iz cvora 6, ali za njega je v.Pre = 5, a
L(v) = 3 pa on nije bazni cvor komponente

zavrsava se rekurzivni poziv iz cvora 4, ali za njega je v.Pre = 4, a
L(v) = 2 pa on nije bazni cvor komponente

zavrsava se rekurzivni poziv iz cvora 1 i
s obzirom na to da za cvor 1 vazi uslov L(v) = v.Pre = 2
sa steka skidamo sve cvorove do cvora 1, dakle redom cvorove 6, 4, 3, 1
i oni predstavljaju zasebnu komponentu

stek: 0

stek: 0,7
zavrsava se rekurzivni poziv iz cvora 7 i
s obzirom na to da za cvor 7 vazi uslov L(v) = v.Pre = 6
sa steka skidamo samo cvor 7 i on predstavlja zasebnu komponentu

stek: 0

stek: 0,2

stek: 0,2,5

stek: 0,2,5,8
zavrsava se rekurzivni poziv iz cvora 8, ali za njega je v.Pre = 9, a
L(v) = 1 pa on nije bazni cvor komponente

zavrsava se rekurzivni poziv iz cvora 5, ali za njega je v.Pre = 8, a
L(v) = 1 pa on nije bazni cvor komponente

zavrsava se rekurzivni poziv iz cvora 2, ali za njega je v.Pre = 7, a
```

```
L(v) = 1 pa on nije bazni cvor komponente

zavrsava se rekurzivni poziv iz cvora 0 i
s obzirom na to da za cvor 0 vazi uslov L(v) = v.Pre = 1
sa steka skidamo sve cvorove do cvora 0, dakle redom cvorove 8, 5, 2, 0
i oni predstavljaju zasebnu (poslednju) komponentu grafa
```

Tardžanov algoritam za određivanje komponenti jake povezanosti oslanja se na DFS pretragu grafa, te je složenosti $O(|V| + |E|)$.