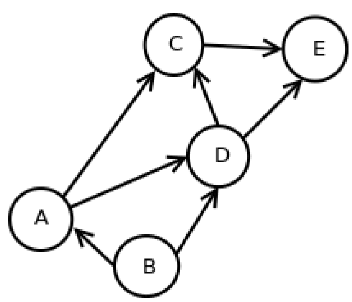
Grafovski algoritmi - čas 4

Topološko sortiranje

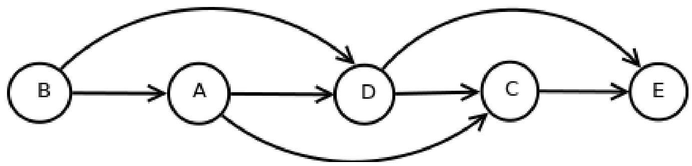
Pretpostavimo da je zadat skup poslova u vezi sa čijim redosledom izvršavanja postoje neka ograničenja. Neki poslovi zavise od drugih, odnosno ne mogu se započeti pre nego što se ti drugi poslovi završe. Sve zavisnosti su poznate, a cilj je napraviti takav redosled izvršavanja poslova koji zadovoljava sva zadata ograničenja; drugim rečima, traži se takav redosled izvršavanja za koji važi da svaki posao započinje tek kad su završeni svi poslovi od kojih on zavisi. Ovaj problem je važan u raznim domenima, kao što je određivanje redosleda u kom je potrebno izvršiti ponovno izračunavanje vrednosti formula u programima za tabelarna izračunavanja, redosled u kom treba izvršiti zadatke u mejkfajlu, unapređenje paralelizma instrukcija i slično. Zadatak koji želimo da rešimo jeste konstruisati efikasan algoritam za formiranje takvog redosleda. Ovaj problem može se formulisati kao grafovski i naziva se *topološko sortiranje grafa*. Zadatim poslovima i njihovim međuzavisnostima može se na prirodan način pridružiti usmereni graf. Svakom poslu pridružuje se čvor, a usmerena grana od posla x do posla y postoji ako se posao y ne može započeti pre završetka posla x. Jasno je da graf 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.

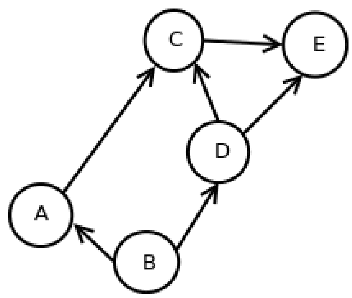
Problem: U zadatom usmerenom acikličkom grafu $G = (V, E)$ sa n čvorova numerisati čvorove brojevima od 1 do n, tako da ako je proizvoljan čvor v numerisan brojem k, onda su svi čvorovi do kojih postoji usmerena grana iz čvora v numerisani brojevima većim od k.

Na primer, u grafu prikazanom na slici 1 postoji samo jedno ispravno topološko uređenje čvorova i to je B, A, D, C, E. Graf sa slike 1 možemo predstaviti i drugačije, tako da čvorovi budu poređani u topološkom poretku, a onda su grane uvek usmerene od jednog čvora ka čvoru koji je desno od njega.



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

U opštem slučaju, može postojati veći broj ispravnih topoloških uređenja čvorova. Ako razmotrimo graf sa slike 3, u njemu postoje dva ispravna topološka uređenja: B, A, D, C, E i B, D, A, C, E.



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

Razmotrićemo dva različita algoritma za rešavanje ovog problema.

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, odnosno posao, 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 da proširimo numeraciju na polazni graf. Ono što je važno primetiti jeste da imamo slobodu izbora čvora koji uklanjamo. Trebalo bi ga izabrati 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. 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 sa ulaznim stepenom 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 sa ulaznim stepenom nula.[1] $\square$

Pretpostavimo da smo pronašli čvor sa ulaznim stepenom 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). Vidi se da je posle izbora čvora sa ulaznim stepenom nula, preostali problem sličan polaznom problemu.

Dakle, problem se može rešiti stalnim pronalaženjem čvorova sa ulaznim stepenom 0. Jedini problem pri realizaciji ovog algoritma je kako 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 proizvoljnim redosledom (sve grane su navedene u listi povezanosti kojom je predstavljen graf) i povećavanjem za jedan vrednosti `ulazniStepen[w]` svaki put kad se naiđe na neku granu (v, w). Čvorovi sa ulaznim stepenom nula stavljaju se u red (ili stek, što bi bilo jednako dobro). Prema prethodnoj lemi u 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 izlazi iz čvora v vrednost `ulazniStepen[w]` smanjuje za jedan. Ako neka od tih vrednosti 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 kad red koji sadrži čvorove stepena nula postane prazan, jer su u tom trenutku svi čvorovi numerisani. Opisani algoritam zove se *Kanov algoritam*.

U tabeli 1 ilustrovano je izvršavanje Kanovog algoritma na primeru grafa sa slike 3 zadatog listom povezanosti kod koje su za svaki čvor susedi poređani leksikografski. 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. U drugom koraku se moglo desiti da se u red doda čvor D, a zatim čvor A i u tom slučaju bilo bi dobijeno uređenje: B, D, A, C, E.

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

[1] Analogno bi se pokazalo da u usmerenom acikličkom grafu uvek postoji čvor sa izlaznim stepenom nula.

$d(A)$	$d(B)$	$d(C)$	$d(D)$	$d(E)$	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.

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 njegov ulazni stepen.

```
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);
    }
  }

  if (brojPosecenih == brojCvorova){
     cout << "Redosled cvorova u topoloskom uredjenju je:" << endl;
     for(int i = 0; i < brojCvorova; i++)
        cout << topoloskoUredjenje[i] << ": " << i+1 << endl;
  }
  else
     cout << "Graf nije aciklicki" << endl;
}

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

Vremenska složenost izračunavanja početnih vrednosti elementa niza `ulazniStepen` je $O(|V| + |E|)$. U petlji `while` (kroz koju se prolazi $|V|$ puta) za nalaženje 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 posle uklanjanja čvora v iz reda. Prema tome, ukupan broj promena vrednosti elemenata niza `ulazniStepen` jednak je broju grana u grafu. Vremenska složenost 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 važi:

- ako je grana (u, v) grana DFS drveta, direktna ili poprečna grana, za nju važi $u.Post > v.Post$,
- ako je grana (u, v) povratna grana, 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 $u.Post > v.Post$. Stoga, ako uredimo čvorove grafa u opadajućem redosledu u odnosu na odlaznu numeraciju čvorova, dobićemo 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.

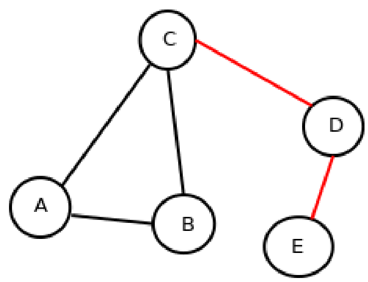
```
vector<vector<int>> listaSuseda {{1, 2}, {3, 4}, {5}, {}, {6, 7},
            {8}, {}, {}, {}};

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 ova implementacija svodi na DFS pretragu i određivanje odlazne numeracije čvorova, vremenska složenost ovog algoritma je $O(|E| + |V|)$.

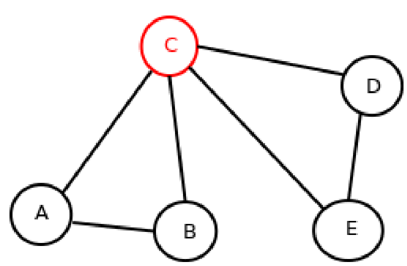
Mostovi i artikulacione tačke u neusmerenom grafu

Ako u neusmerenom grafu $G = (V, E)$ postoji grana čijim se uklanjanjem iz grafa broj komponenti povezanosti grafa povećava, ovakvu granu nazivamo *most* (eng. bridge, cut edge). Specijalno, povezani graf nakon uklanjanja mosta prestaje da bude povezan. Na primer, ukoliko bismo u grafu na slici 4 uklonili granu CD ili granu DE graf bi prestao da bude povezan, te ove dve grane, svaka za sebe, čine most u datom grafu. Postoje grafovi u kojima nema mostova.



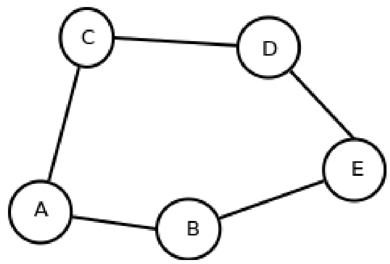
Slika 4: Primer grafa koji sadrži most.

Ukoliko u neusmerenom grafu $G = (V, E)$ postoji čvor $v \in V$ takav da se njegovim uklanjanjem (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 sa slike 5 uklonili čvor C preostali graf bi prestao da bude povezan, te je čvor C jedna artikulaciona tačka ovog grafa.

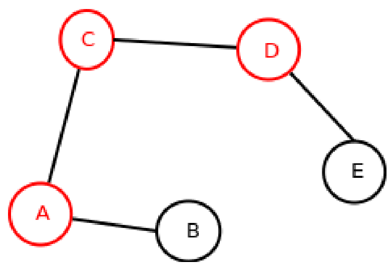


Slika 5: Primer grafa koji sadrži jednu artikulacionu tačku.

Graf može da ne sadrži artikulacione tačke (videti primer na slici 6), a može i da sadrži veći broj artikulacionih tačaka (videti primer na slici 7).



Slika 6: Primer grafa koji ne sadrži artikulacionu tačku.



Slika 7: Primer grafa koji sadrži veći broj artikulacionih tačaka.

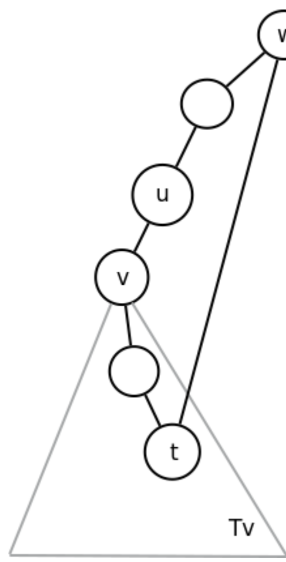
Bez smanjenja opštosti može se pretpostaviti da je graf povezan. Ako graf nije povezan, onda se artikulacione tačke mogu tražiti nezavisno u svakoj komponenti povezanosti.

Direktan način da u datom neusmerenom povezanom grafu pronađemo mostove podrazumeva da za svaku granu $e \in E$ proverimo da li je graf bez grane e povezan (npr. korišćenjem algoritma DFS). Složenost ovog algoritma je $O(|E| \cdot (|V| + |E|))$. Analogno, artikulacione tačke u datom neusmerenom povezanom grafu bismo mogli da odredimo tako što bismo za svaki čvor $v \in V$ proverili da li je graf bez čvora v povezan. Složenost ovog algoritma je $V(|E| \cdot (|V| + |E|))$. 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.

Razmotrimo DFS drvo dobijeno DFS obilaskom datog grafa. 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 ona čini sa granama DFS drveta. 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 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 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 utvrditi?



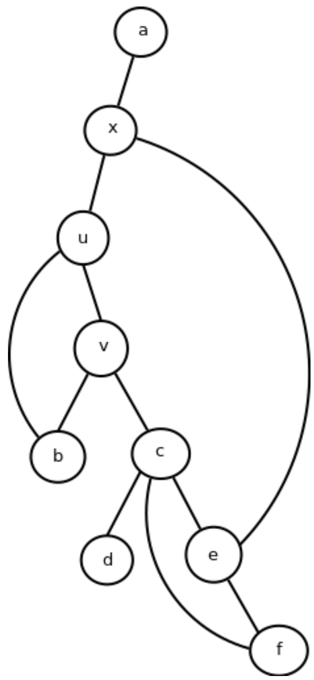
Slika 8: Ilustracija definicije vrednosti $L(v)$.

Za svaki čvor v u grafu može se odrediti redni broj u okviru 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. 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 potomku (ne roditelju) iz proizvoljnog čvora poddrveta T_v. Tada je:

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

Poddrvo T_v sa korenom 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 ako i samo ako važi $L(v) > u.Pre$.



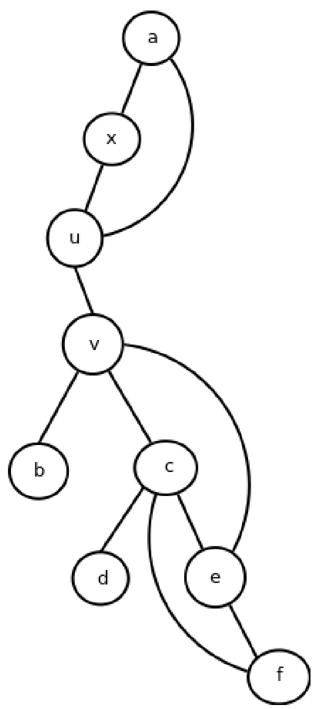
Slika 9: Primer grafa u kome grana (u, v) nije most.

Razmotrimo primer grafa sa slike 9. 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 $L(v)$ jednaka je rednom broju čvora x, te nije zadovoljen uslov $L(v) > u.Pre$ (važi $x.Pre < u.Pre$ jer je x predak čvora u u DFS drvetu). 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.

Razmotrimo sada primer grafa sa slike 10. 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.

Vrednosti $L(v)$ za sve čvorove grafa mogu se odrediti u toku DFS pretrage tako što se prilikom obrade grane (u, v) u toku poziva DFS iz čvora u radi sledeće:

- ako je čvor v predak čvora u, onda ako je $v.Pre < L(u)$, ažurira se vrednost $L(u) \leftarrow v.Pre$;
- ako je čvor v neoznačen, označava se, povezuje granom DFS drveta sa ocem u i dobija vrednost $v.Pre$; tom vrednošću se inicijalizuje vrednost $L(v)$; nakon rekurzivne obrade kompletnog poddrveta sa korenom u čvoru v ako je $L(v) < L(u)$, ažurira se vrednost $L(u) \leftarrow L(v)$.

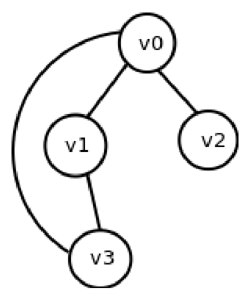


Slika 10: Primer grafa u kome je grana (u, v) most.

```
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> min_pretka;
vector<int> roditelj;
vector<pair<int,int>> most;

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

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

      // nakon obrade poddrveta sa korenom u susednom cvoru
      // azuriramo vrednost L za cvor ako je potrebno
      if (min_pretka[sused] < min_pretka[cvor])
         min_pretka[cvor] = min_pretka[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 (min_pretka[sused] > dolazna[cvor])
         most.push_back(make_pair(cvor,sused));
    }
    // ukoliko je sused vec posecen i ako grana vodi ka nekom 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] < min_pretka[cvor])
         min_pretka[cvor] = dolazna[sused];
  }
}

void ispisi_mostove(int cvor){
  int brojCvorova = listaSuseda.size();
  posecen.resize(brojCvorova, false);
  dolazna.resize(brojCvorova);
  min_pretka.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;
}
```

Pogledajmo na primeru jednostavnog neusmerenog grafa sa slike 11 kako bi išlo izvršavanje ovog algoritma. Pretpostavimo da se DFS pretraga započinje iz čvora v_0.



Slika 11: Primer grafa koji sadrži jedan most: granu (v_0, v_2)

```
Pokrecemo DFS iz cvora v0, postavljamo v0.Pre <- 1 i L(v0) <- 1
Razmatramo suseda v1 čvora v0

Pokrecemo DFS iz cvora v1, postavljamo v1.Pre <- 2 i L(v1) <- 2
Razmatramo suseda v3 čvora v1

Pokrecemo DFS iz cvora v3, postavljamo v3.Pre <- 3 i L(v3) <- 3
Razmatramo suseda v0 čvora v3
Grana (v3,v0) je grana od potomka ka pretku pa postavljamo L(v3) <- v0.Pre,
pa vazi L(v3) = 1
Razmatramo suseda v1 čvora v3
To je grana ka roditelju, koju ne razmatramo

Vracamo se u cvor v1
Posto vazi L(v3) < L(v1) postavljamo L(v1) <- L(v3) pa vazi L(v1) = 1
S obzirom na to da je L(v3) < v1.Pre grana (v1,v3) nije most
```

```
Razmatramo suseda v0 čvora v1
To je grana ka roditelju, koju ne razmatramo

Vracamo se u cvor v0
Posto vazi L(v1) = L(v0) ne radimo nista
S obzirom na to da je L(v1) = v0.Pre grana (v0,v1) nije most

Razmatramo suseda v3 čvora v0
Posto vazi L(v3) = L(v0) ne radimo nista

Pokrecemo DFS iz cvora v2, postavljamo v2.Pre <- 4 i L(v2) <- 4
Razmatramo suseda v0 čvora v2
To je grana ka roditelju, koju ne razmatramo

Vracamo se u cvor v0
Posto vazi L(v2) > L(v0) ne radimo nista
S obzirom na to da je L(v2) > v0.Pre grana (v0,v2) jeste most
```

Algoritam 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

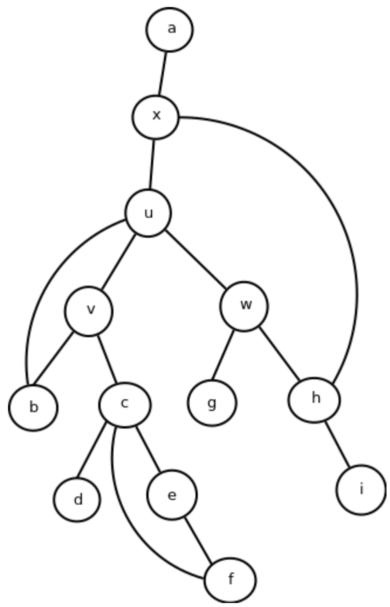
Veoma nalik prethodnom algoritmu je i algoritam za traženje artikulacionih tačaka u grafu. Čvor u biće artikulaciona tačka grafa ako je ispunjen jedan od naredna dva uslova:

- 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 jedna 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 sinu v čvora u do nekog pretka čvora u.

Prvi od navedenih uslova se može detektovati tako što prilikom DFS obilaska za svaki čvor vršimo proveru da li ima roditelja (jedino koren DFS drveta nema roditelja) i brojimo koliko čvor ima dece. Drugi uslov je sličan uslovu za postojanje mosta, tj. čvor u će biti artikulaciona tačka u grafu ako za nekog sina v čvora u važi uslov $L(v) \geq u.Pre$.

Razmotrimo primer grafa sa slike 12. 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



Slika 12: Primer grafa u kome je čvor u artikulaciona tačka.

možemo se vratiti u deo grafa iznad čvora u (jer je $L(w) = x.Pre$), međutim, iz poddrveta T_v sa korenom u čvoru vw možemo se najviše vratiti do čvora u (jer važi $L(v) = u.Pre$). S obzirom na to da čvor v ima dete v za koje važi $L(v) = u.Pre$, čvor u jeste artikulaciona tačka u grafu.

```
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> min_pretka;
vector<int> roditelj;
vector<bool> artikulacionaTacka;

void dfs(int cvor){
  posecen[cvor] = true;
  dolazna[cvor] = min_pretka[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]){
      broj_dece++;
```

```
      roditelj[sused] = cvor;
      dfs(sused);

      // nakon obrade poddrveta sa korenom u susednom cvoru
      // azuriramo vrednost L za cvor ako je potrebno
      if (min_pretka[sused] < min_pretka[cvor])
         min_pretka[cvor] = min_pretka[sused];

      // proveravamo da li je ispunjen drugi uslov:
      // cvor nije koren DFS drveta i postoji dete tog cvora
      // takvo da nijedan cvor u njegovom poddrvetu
      // nije povezan sa nekim pretkom datog cvora
      if (roditelj[cvor] != -1
          && min_pretka[sused] >= dolazna[cvor])
         artikulacionaTacka[cvor] = true;
    }
    else // posecen[sused]
    if (sused != roditelj[cvor])
      if (dolazna[sused] < min_pretka[cvor])
         min_pretka[cvor] = dolazna[sused];
  }
  // proveravamo da li je ispunjen prvi uslov:
  // cvor je koren DFS drveta i ima vise od jednog deteta
  if (roditelj[cvor] == -1 && broj_dece > 1)
    artikulacionaTacka[cvor] = true;

}

void ispisi_artikulacione_tacke(int cvor){
  int brojCvorova = listaSuseda.size();
  posecen.resize(brojCvorova, false);
  dolazna.resize(brojCvorova);
  min_pretka.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 $u \sim u$. Za ovu relaciju važi da je:

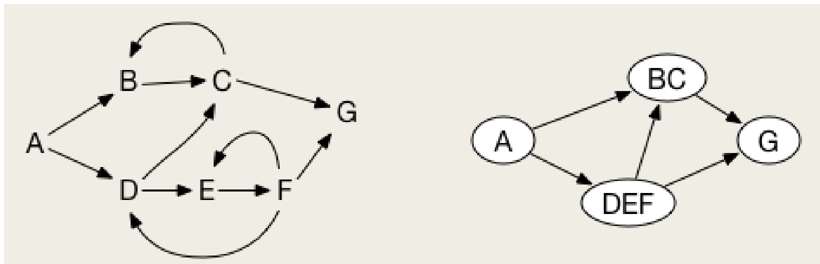
- refleksivna – za svaki čvor $u \in V$ važi $u \sim u$,
- simetrična – za svaka dva čvora $u, v \in V$ važi $u \sim v$ akko $v \sim u$,
- tranzitivna – za svaka tri čvora $u, v, w \in V$ iz $u \sim v$ i $v \sim w$ sledi i $u \sim w$.

Relacija $\sim$ 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 13 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 povezanosti ako i samo ako pripadaju nekom 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 13, desno): svaki čvor u grafu G^C odgovara jednoj komponenti jake povezanosti grafa G, a dva čvora u G^C su povezana granom ako i samo ako u grafu G postoji bar jedna grana između nekog čvora jedne komponente i 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 koje pripadaju ciklusu mogu spojiti u jednu, veću komponentu jake povezanosti.

U nastavku komponente jake povezanosti zvaćemo kraće samo komponente.

Jedan (direktan) način za određivanje komponenti u grafu sastojao bi se u tome da se za prvi čvor v_0 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 DFS pretragom iz čvora v_0 i iz svakog novog čvora čija se pripadnost komponenti ispituje. Nakon toga bi se sličan proces 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.

Neka je G^* graf koji sadrži iste čvorove kao i graf G, ali suprotno usmerene grane (ako grana (u, v) pripada grafu G, onda grana (v, u) pripada grafu 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



Slika 13: Graf G i odgovarajući komprimovani graf G^C čiji čvorovi odgovaraju komponentama jake povezanosti grafa G.

obilazak pronalazi skup čvorova A koji su dostižni iz čvora v_0, a drugi skup čvorova B iz kojih je dostižan čvor v_0. Komponenta povezanosti kojoj pripada čvor v_0 jednaka je $A \cap B$. Ovaj postupak bismo ponavljali za čvor koji ne pripada do sada određenim komponentama povezanosti, ukoliko takav čvor postoji.

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

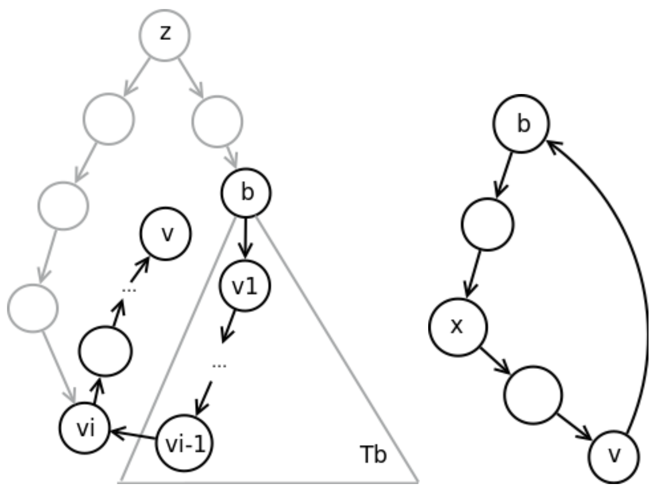
Tardžanov algoritam

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_0 sa ulaznim stepenom 0, i povezati granama sa svim čvorovima grafa G; tada DFS pretraga pokrenuta iz čvora v_0 označava sve čvorove grafa G. Prošireni graf sem komponenti grafa G ima samo još jednu dodatnu komponentu $\{v_0\}$.

Nazovimo *baznim čvorom* neke komponente onaj čvor te komponente koji ima najmanji redni broj pri dolaznoj DFS numeraciji.

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 pripadaju komponenti X.

Dokaz: Pretpostavimo da u komponenti X postoje čvorovi koji nisu u poddrvetu 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 u komponenti X. Neka je v_i, $1 \leq i \leq k$, prvi čvor na tom putu koji nije u poddrvetu čvora 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 (ne može biti povratna jer bi onda čvor v_i bio predak čvora b i b ne bi bio bazni čvor te komponente). Sve poprečcne 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}, te oni imaju zajedničkog pretka z u DFS drvetu. 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 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 čvora b.

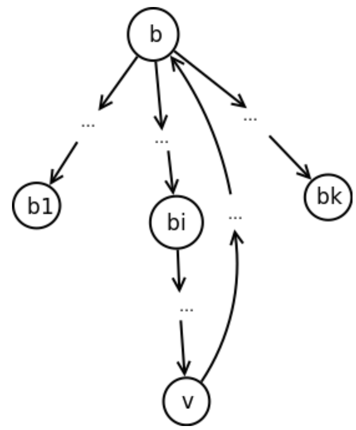


Slika 14: Ilustracija uz dokaz prvog i drugog dela leme 1.

Dokažimo drugi deo leme. Neka je x proizvoljni čvor na putu od b do v. Postoji put od čvora b do čvora x kroz grane DFS drveta, a takođe i put od čvora x do čvora b tako što prvo idemo od x do v, 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 pripadaju komponenti čiji je bazni čvor b. $\square$

Lema 2: Neka je b bazni čvor i neka su $b_1, b_2, \ldots, b_k$ bazni čvorovi 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 $b_1, b_2, \ldots, b_k$.

Dokaz: Pretpostavimo suprotno, odnosno da postoji čvor v koji je u istoj komponenti kao i b i koji je potomak i čvora b i čvora b_i za neko i, $1 \leq i \leq k$. Pošto je b_i potomak čvora b, onda postoji put kroz grane DFS drveta od b do b_i. Pošto je čvor v potomak čvora b_i postoji put od b_i do v, a na osnovu pretpostavke da su v i b u istoj komponenti, postoji i put od v do b, tako da



Slika 15: Ilustracija uz dokaz leme 2.

postoji put od čvora b_i do b. Odavde sledi da su b i b_i u istoj komponenti što je u suprotnosti sa pretpostavkom leme. □

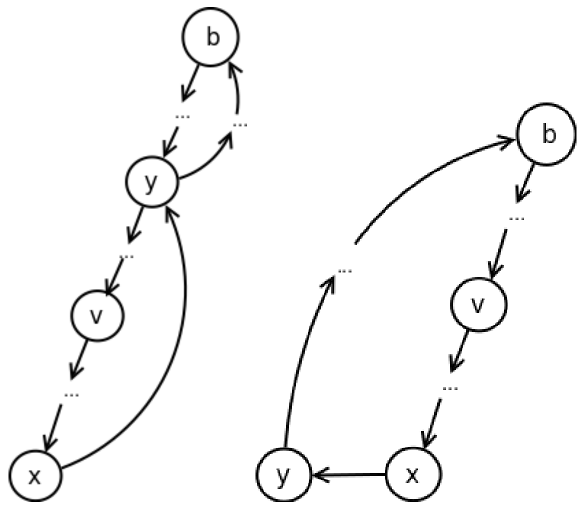
S obzirom na to da je graf sa kojim radimo usmeren i može imati poprečne grane u odnosu na DFS drvo, vrednost $L(v)$ za čvorove v grafa definisaćemo na malo drugačiji način nego u slučaju neusmerenih grafova. Neka je X komponenta koja sadrži čvor v. Označimo sa A najmanji redni broj čvora u komponenti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

Lema 3: Čvor v je bazni čvor akko 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 da važi uslov $L(v) = v.Pre$, a 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, 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 put);
- ako je grana (x, y) poprečna, onda je grana usmerena ulevo i $y.Pre < x.Pre$. S obzirom na to da su poddrvo sa korenom u y i poddrvo sa korenom u v



Slika 16: Ilustracija uz dokaz prvog i drugog slučaja u lemi 3.

disjunktni i da postoji grana od potomka čvora v (odnosno x) do y, možemo zaključiti na osnovu svojstava poprečnih grana da važi $y.Pre < v.Pre$. Dakle, važi 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 $L(v) < v.Pre$, što je kontradikcija. $\square$

Da bi se u okviru Tardžanovog algoritma izdvajale komponente jake povezanosti, modifikuje se DFS pretraga. Prilikom označavanja čvora v u toku pretrage, čvor v se upisuje na stek. Kada se završi poziv DFS 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. Na taj način ako je čvor v bazni, izdvaja se komponenta koja sadrži čvor v i svi njeni čvorovi uklanjaju se sa steka.

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

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

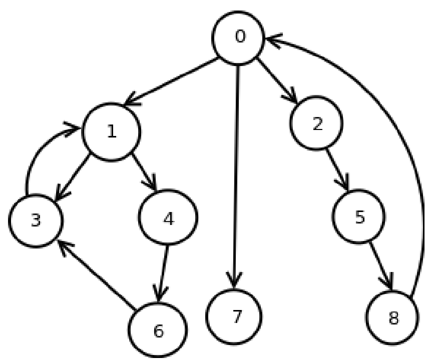
Neka je tvrđenje tačno za grafove sa manje od m komponenti. Neka je G graf sa m komponenti.

Neka je b prvi bazni čvor na koji se nailazi pri DFS pretrazi, 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 iz *b* 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> &min_pretka,
      stack<int> &redosledUObilasku, vector<int> &u_steku){
  dolazna[cvor] = min_pretka[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]){
    if (dolazna[sused] == -1){
       dfs(sused,dolazna,min_pretka,redosledUObilasku,u_steku);

       if (min_pretka[sused] < min_pretka[cvor])
          min_pretka[cvor] = min_pretka[sused];
    }
    // azuriramo vrednost L za cvor
    // samo ako se sused nalazi u steku
    else if (u_steku[sused])
      if (dolazna[sused] < min_pretka[cvor])
         min_pretka[cvor] = dolazna[sused];
  }

  // ako je u pitanju bazni cvor komponente
  // stampamo sve cvorove te komponente
  if (dolazna[cvor] == min_pretka[cvor]){
    while(1){
      int cvor_komponente = redosledUObilasku.top();
      cout << cvor_komponente << " ";
      u_steku[cvor_komponente] = false;
      redosledUObilasku.pop();
      if (cvor_komponente == cvor){
         cout << "\n";
         break;
      }
    }
  }
}

void ispisi_komponente(int cvor){
```

```
  int brojCvorova = listaSuseda.size();
  vector<int> dolazna(brojCvorova,-1);
  vector<int> min_pretka(brojCvorova);
  stack<int> redosledUObilasku;
  vector<bool> u_steku(brojCvorova,false);

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

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

Razmotrimo izvršavanje algoritma na primeru grafa prikazanog na slici 17.



Slika 17: Primer grafa čije su komponente jake povezanosti $\{1,3,4,6\}$, $\{7\}$, $\{0,2,5,8\}$.

```
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) = 2 pa on nije bazni cvor komponente
```

```
zavrsava se rekurzivni poziv iz cvora 4, ali za njega je v.Pre = 5, 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 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 cvorove 8,5,2,0
i oni predstavljaju zasebnu komponentu
```

Tardžanov algoritam za određivanje komponenti jake povezanosti zasnovan je na DFS obilasku grafa i složenosti je $O(|V| + |E|)$.