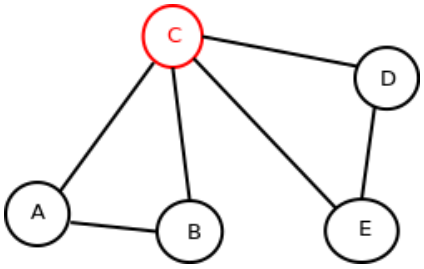
Grafovski algoritmi - čas 4

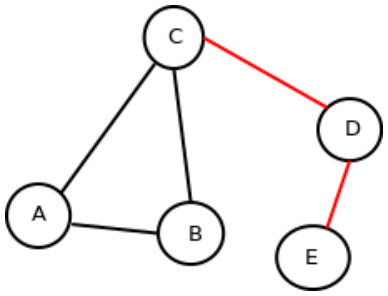
Artikulacione tačke i mostovi

Ukoliko u neusmerenom povezanom grafu $G = (V, E)$ postoji čvor $v \in V$ takav da njegovim uklanjanjem graf prestaje da bude povezan, onda takav čvor nazivamo *artikulacionom tačkom* (eng. articulation point, cut vertex). Na primer, ukoliko bismo u grafu sa slike 1 uklonili čvor C (zajedno sa svim granama koje su mu susedne) preostali graf ne bi ostao povezan, te je čvor C artikulaciona tačka ovog grafa.



Slika 1: Primer grafa koji sadrži artikulacionu tačku C.

Ako u povezanom grafu postoji grana čijim uklanjanjem iz grafa on prestaje da bude povezan, ovakvu granu nazivamo *most* (eng. bridge, cut edge). Na primer, ukoliko bismo u grafu na slici 2 uklonili granu CD ili granu DE graf bi prestao da bude povezan, te ove dve grane, svaka za sebe, čine most.



Slika 2: Primer grafa koji sadrži most.

Direktan način da u datom grafu pronađemo artikulacionu tačku podrazumeva da po jedan čvor uklanjamo iz grafa i da proveravamo da li je dobijeni graf nepovezan (npr. korišćenjem DFS algoritma). Složenost ovog algoritma je $O(|V| \cdot (|V| + |E|))$.

Analogno, mostove u datom grafu bismo mogli da odredimo uklanjanjem po jedne grane iz grafa i proverom da li graf ostaje povezan. Složenost ovog algoritma je $O(|E| \cdot (|V| + |E|))$. Postoje i efikasniji algoritmi za određivanje artikulacionih tačaka i mostova u grafu. Mi ćemo u nastavku razmotriti algoritme koje su 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. Važi sledeće tvrđenje: svaka grana (u, v) grafa koja ne pripada nekom ciklusu je most (jer nakon izbacivanja grane (u, v) ne postoji način kako doći od čvora u do čvora v). Specijalno, ako je graf šuma, 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 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 se granama DFS drveta može stići od čvora u do čvora v. Za granu (u, v) DFS drveta važi da je most ako njenim uklanjanjem graf postaje nepovezan, tj poddrvo sa korenom u v ostaje nepovezano sa delom grafa "iznad" ove grane. To će važiti ako ne postoji način da se (nekom granom od potomka ka pretku) stigne do čvora u ili pretka čvora u iz podgrafa sa korenom v. Kako ovo utvrditi? Za svaki čvor v potrebno je odrediti vrednost pri dolaznoj numeraciji $v.Pre$ i najmanju među vrednostima dolazne numeracije čvorova do kojih se može stići iz proizvoljnog čvora poddrveta sa korenom v – označimo tu vrednost sa $v.minPre$ (eng. low link). Važi: $v.minPre = \min\{v.Pre, u.Pre\}$, gde je u predak čvora v u DFS drvetu i postoji grana koja povezuje nekog potomka čvora v sa čvorom u. Poddrvo sa korenom v ostaće nepovezano sa delom grafa "iznad" ove grane ako bude važilo $v.minPre \geq u.Pre$. Dakle, uslov koji grana (u, v) treba da zadovoljava da bi bila most je da važi $v.minPre \geq u.Pre$.

Tokom DFS pretrage datog neusmerog grafa, prolasku svake grane (u, v) pridružujemo neku akciju:

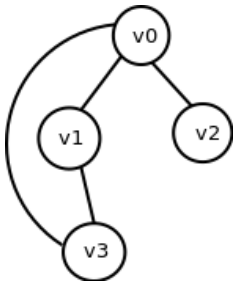
- ako je u pitanju grana koja povezuje potomka u sa pretkom v, onda ako je vrednost $v.Pre$ manja od tekuće vrednosti $u.minPre$, vrednost $u.minPre$ postavljamo na $v.Pre$
- ako je u pitanju grana DFS drveta, onda čvoru v postavljamo vrednost $v.Pre$, vrednost $v.minPre$ inicijalizujemo na $v.Pre$, a nakon rekurzivne obrade kompletnog poddrveta sa korenom u čvoru v ako je vrednost $v.minPre$ manja od vrednosti $u.minPre$ ažuriramo vrednost $u.minPre$

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

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

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

      // 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 grana vodi ka nekom pretku datog cvora
    // postavljamo vrednost minPre 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 3 kako bi išlo izvršavanje ovog algoritma.



Slika 3: Primer grafa koji sadrži jedan most: (v_0, v_2)

```
Pokrecemo DFS iz cvora v0, postavljamo v0.Pre=1 i v0.minPre=1
Razmatramo suseda v1 čvora v0

Pokrecemo DFS iz cvora v1, postavljamo v1.Pre=2 i v1.minPre=2
Razmatramo suseda v3 čvora v1

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

Vracamo se u cvor v1
Posto vazi v3.minPre < v1.minPre postavljamo v1.minPre=v3.minPre=1
S obzirom da je v3.minPre < 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 v1.minPre < v0.minPre postavljamo v0.minPre=v1.minPre=1
S obzirom da je v1.minPre = v0.Pre grana (v0,v1) nije most

Razmatramo suseda v3 čvora v0
v3.minPre=v0.minPre pa ne radimo nista
```

```
Pokrecemo DFS iz cvora v2, postavljamo v2.Pre=4 i v2.minPre=4
Razmatramo suseda v0 čvora v2
To je grana ka roditelju, koju ne razmatramo

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

Na sličan način možemo zaključiti da će čvor u biti artikulaciona tačka grafa ako je ispunjen jedan od naredna dva uslova:

1. u je koren DFS drveta i ima bar dva deteta
2. u nije koren DFS drveta i ima dete v u DFS drvetu takvo da nijedan čvor u poddrvetu sa korenom 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 (po jedna za svako dete korena DFS drveta). Drugi uslov označava situaciju kada nakon izbacivanja nekog čvora iz grafa više nije moguće doći iz proizvoljnog čvora poddrveta sa korenom u nekom od deteta tog čvora do proizvoljnog čvora "iznad" njega u DFS drvetu.

Prvi od uslova se može detektovati tako što za svaki čvor proverimo da li ima roditelja prilikom DFS obilaska (jedino koren nema roditelja) i brojimo koliko ima dece. Drugi uslov je ekvivalentan uslovu kod određivanja mostova, tj. biće zadovoljen ako važi $v.minPre \geq u.Pre$.

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

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

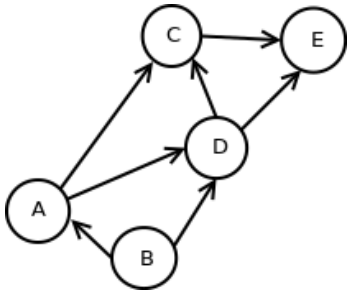
```
      roditelj[sused] = cvor;
      dfs(sused);
      if (min_pretka[sused] < min_pretka[cvor])
         min_pretka[cvor] = min_pretka[sused];

      // 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)
         artikulacioneTacke[cvor] = true;

      // 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])
         artikulacioneTacke[cvor] = true;
    }
    else // posecen[sused]
    if (sused != roditelj[cvor])
      if (dolazna[sused] < min_pretka[cvor])
         min_pretka[cvor] = dolazna[sused];
  }
}

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;
```

```
}
```

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 raspored za koji važi da svaki posao započinje tek kad budu završeni svi poslovi od kojih on zavisi. Potrebno je konstruisati efikasni algoritam za formiranje takvog rasporeda. Ovaj problem se zove *topološko sortiranje*. Zadatim poslovima i njihovim međuzavisnostima može se na prirodan način pridružiti 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 aciklički (bez usmerenih ciklusa), jer se u protivnom neki poslovi nikada ne bi mogli započeti.

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 sa k, onda svi čvorovi do kojih postoji usmereni put iz v imaju broj veći od k.



Slika 4: Aciklički usmereni graf u kojem postoji tačno jedno topološko uređenje čvorova B, A, D, C, E.

Na primer, u grafu prikazanom na slici 4 postoji samo jedno ispravno topološko uređenje čvorova B, A, D, C, E. U opštem slučaju, može postojati veći broj ispravnih topoloških uređenja čvorova.

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 jednog čvora, odnosno posla, je trivijalan. Kao i obično, posmatrajmo proizvoljni graf sa n čvorova, uklonimo jedan čvor, primenimo induktivnu hipotezu i pokušajmo da proširimo numeraciju na polazni graf. Imamo slobodu izbora n-tog čvora. Trebalo bi ga izabrati tako da ostatak posla bude što jednostavniji. Potrebno je numerisati čvorove. 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 se uvek može pronaći 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 ciklus. Ovo je međutim suprotno pretpostavci da se radi o acikličkom grafu. Dakle u usmerenom acikličkom grafu uvek postoji čvor sa ulaznim stepenom nula. Slično bi se pokazalo da postoji i čvor sa izlaznim stepenom nula.

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 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, ostatak posla jednostavan.

Jedini problem pri realizaciji ovog algoritma je kako pronaći čvor sa ulaznim stepenom nula i kako popraviti ulazne stepene čvorova posle uklanjanja grane. Možemo alocirati niz `ulazniStepen` dimenzije jednake broju čvorova u grafu i inicijalizovati ga na vrednosti ulaznih stepena čvorova. Ulazne stepene možemo jednostavno odrediti prolaskom kroz skup svih grana proizvoljnim redosledom (sve grane su navedene u listi povezanosti) i povećavanjem za jedan vrednosti `ulazniStepen[w]` svaki put kad se naiđe na 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 v sa ulaznim stepenom nula. Čvor v se kao prvi u redu lako pronalazi; on se uklanja iz reda. Zatim se za svaku granu (v, w) koja izlazi iz v vrednost `ulazniStepen[w]` smanjuje za jedan. Ako brojač pri tome dobije vrednost nula, čvor w stavlja se u red. Posle uklanjanja čvora v graf ostaje aciklički, pa u njemu prema 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. Interesantno je da ne moramo iz grafa izbacivati grane, već je samo važno da ažuriramo za svaki čvor njegov ulazni stepen.

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

```
void topolosko_sortiranje(){

  int brojCvorova = listaSuseda.size();
  // niz koji za svaki cvor cuva njegov ulazni stepen
  vector<int> ulazniStepen(brojCvorova,0);
  // niz koji za svaki cvor cuva njegov redni broj 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]]++;

  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]--;
      // ukoliko je stepen nekog od suseda pao na 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;
}
```

Složenost izračunavanja početnih vrednosti elementa niza `ulazniStepen` je $O(|V|+|E|)$. Za nalaženje čvora sa ulaznim stepenom nula potrebno je konstantno vreme (pristup redu). Svaka grana (v, w) razmatra se tačno jednom, u trenutku kad se v uklanja iz reda. Prema tome, 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 ulaza.

Algoritam zasnovan na DFS pretrazi

Kao što smo ranije zaključili ako je (u, v) grana DFS drveta, direktna ili poprečna grana, za nju važi $u.Post > v.Post$, dok samo za povratne grane važi $u.Post \leq v.Post$. Stoga ako za granu (u, v) važi $u.Post > v.Post$ onda ona ne može biti povratna grana.

U usmerenom acikličkom grafu ne postoji ciklus, te ne može postojati ni povratna grana u odnosu na DFS drvo. Stoga, ako uredimo čvorove grafa u opadajućem redosledu u odnosu na odlaznu numeraciju čvorova, dobićemo topološko uređenje grafa. Ovo važi zato što za proizvoljnu granu grafa (u, v), s obzirom na to da nije povratna, važi $u.Post > v.Post$, pa s obzirom na to da smo čvorove numerisali opadajuće prema odlaznoj numeraciji, čvor u biće numerisan manjom vrednošću od čvora v.

```
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);

  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, vremenska složenost i ovog algoritma je $O(|E| + |V|)$.

Komponente jake povezanosti grafa

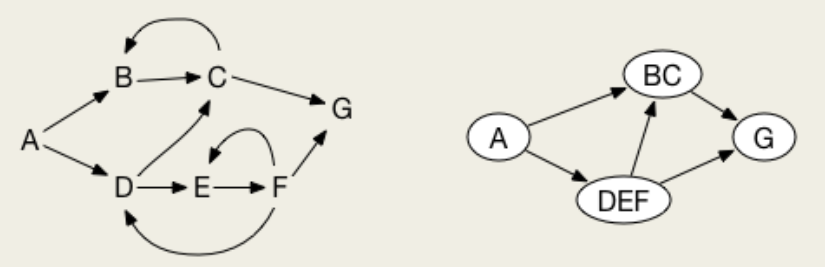
Za usmereni graf kažemo da je *jako povezan* ako je svaki čvor grafa dostižan iz svakog drugog čvora u grafu.

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 dostižan iz čvora u. Za ovu relaciju važi da je:

- refleksivna – za svaki čvor $u \in V$ je $u \sim u$,
- 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 $u \sim w$.

Stoga je ona relacija ekvivalencije. Ona razlaže skup čvorova V u klase ekvivalencije koje nazivamo *komponente jake povezanosti* grafa G (eng. strongly connected components). Na slici 5 prikazan je usmereni graf koji ima četiri komponente jake povezanosti koje se sastoje redom od čvorova $\{A\}, \{B, C\}, \{D, E, F\}, \{G\}$. Primetimo da svi čvorovi nekog ciklusa pripadaju istoj komponenti jake povezanosti. Graf G se može "kompresovati" i razmatrati kao usmereni aciklički graf koji se sastoji od svojih komponenti jake povezanosti (slika 5, desno): svaki čvor u ovom grafu odgovara jednoj komponenti jake

povezanosti, a dva čvora su povezana granom samo ako i samo ako u polaznom grafu postoji grana između nekog čvora jedne komponente i nekog čvora druge komponente. Jasno je da ovaj graf mora biti aciklički jer ako bi u njemu postojao ciklus to bi značilo da se sve komponente koje pripadaju ciklusu mogu spojiti u jednu veću komponentu povezanosti.



Slika 5: Graf G i usmereni aciklički graf koji se sastoji od komponenti jake povezanosti grafa G.

Direktan način da bi se odredile komponente jake povezanosti sastojao bi se u tome da se za prvi čvor v_0 odredi koji čvorovi pripadaju njegovoj jakoj komponenti povezanosti, tako što bi se za sve preostale čvorove proveravalo da li su obostrano dostižni iz v_0 – to bi moglo da se uradi DFS pretragom iz čvora v_0 i iz svakog novog čvora čija se pripadnost komponenti ispituje. Nakon toga bi se sličan proces ponavljao za naredni čvor koji ne pripada jakoj komponenti kojoj pripada čvor v_0. Jasno je da bi ovo bilo veoma neefikasno.

Postoji nekoliko različitih algoritama linearne vremenske složenosti za određivanje komponenti jake povezanosti 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 ova dva algoritma.

Tardžanov algoritam

Prilikom DFS obilaska datog usmerenog grafa G implicitno se formira DFS drvo, odnosno 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. Nazovimo *baznim čvorom* neke jake komponente onaj čvor te komponente koji ima najmanju vrednost dolazne numeracije.

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

Dokaz: Dokažimo najpre prvo tvrđenje. Neka je v proizvoljni čvor iz X različit

od b. Važi da je (a) v potomak čvora b, (b) b potomak čvora v ili (c) nijedno od ova dva. Slučaj (b) nije moguć jer ako bi čvor b bio potomak čvora v onda bi on imao veću vrednost dolazne numeracije od v što je u suprotnosti sa pretpostavkom leme.

Pretpostavimo da važi (c). Put od čvora b do čvora v mora da postoji jer ova dva čvora pripadaju istoj jakoj komponenti. Razmotrimo jedan takav put p i neka je r najbliži zajednički predak svih čvorova na tom putu. Čvor r mora pripadati tom putu p, a čvorovi b i v moraju biti potomci različite dece čvora r. Označimo sa T_b poddrvo sa korenom u čvoru b, a sa T_v poddrvo sa korenom u čvoru v. S obzirom na to da je vrednost dolazne numeracije čvora b manja od vrednosti čvora v i s obzirom na to da su T_v i T_b disjunktna drveta, ne može postojati grana ni od jednog čvora iz T_b ka nekom čvoru iz T_v. Stoga je jedini put od b do v kroz čvor r. Međutim, s obzirom na to da je r predak čvora b on ima manju vrednost dolazne numeracije od čvora b a pripada istoj jakoj komponenti jer postoji put od b do r, a i od r do b kroz grane drveta. Ovo je u suprotonosti sa pretpostavkom da je b bazni čvor te komponente, pa slučaj (c) nije moguć. Stoga važi da je čvor v potomak čvora b.

Dokaz drugog dela leme je jednostavan. Neka je x čvor na putu od v do b. Postoji put od b do x kroz grane DFS drveta, a takođe i put od x do b tako što prvo idemo od x do v, pa od v do b. Stoga je x u istoj jakoj komponenti kao i čvor b.

Lema: 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 je jaka komponenta kojoj pripada čvor b skup svih potomaka čvora b koji nisu potomci nijednog drugog čvora $b_1, b_2, \ldots, b_k$.

Pretpostavimo suprotno, odnosno da postoji čvor v koji je u istoj jakoj komponenti kao i b i koji je potomak i čvora b i čvora b_i za neko i, $1 \leq i \leq k$. Mora da postoji put od v do b, a takođe i put od čvora b preko čvora b_i do čvora v (koji se sastoji od grana DFS drveta). Odavde sledi da su b i b_i u istoj jakoj komponenti što je u suprotnosti sa pretpostavkom.

Lema: Čvor v je bazni čvor akko važi $v.Pre = v.minPre$.

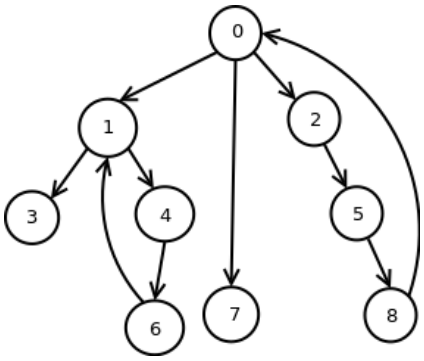
Da bismo izdvojili čvorove koji pripadaju poddrvetu sa korenom u datom baznom čvoru, možemo iskoristiti stek na koji ćemo stavljati čvor prilikom prve posete tokom DFS obilaska grafa. Kada tokom obilaska naiđemo na čvor koji se već nalazi na steku, znamo da će jednoj komponenti povezanosti pripadati svi čvorovi koji se nalaze na steku počev od tog čvora. Poprečne grane neće biti razmatrane jer kada stignemo do čvora koji je već posećen, vršimo obradu samo ako se on nalazi na steku (što neće biti slučaj sa krajnjim čvorom poprečne grane).

```
vector<vector<int>> listaSuseda {{1, 2}, {3, 4}, {5, 8}, {}, {6, 7},
            {8}, {1}, {}, {0}};
int vreme = 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;
  vreme++;
  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 minPre 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 koren 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<int> 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 6.



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

```
stek: 0

stek: 0,1

stek: 0,1,3
zavrsava se rekurzivni poziv iz cvora 3 i
s obzirom na to da za cvor 3 vazi uslov v.Pre=v.minPre=3
sa steka skidamo samo cvor 3 i on predstavlja zasebnu komponentu

stek: 0,1

stek: 0,1,4

stek: 0,1,4,6
zavrsava se rekurzivni poziv iz cvora 6, ali za njega je v.Pre=5, a
v.minPre=2 pa on nije bazni cvor komponente

zavrsava se rekurzivni poziv iz cvora 4, ali za njega je v.Pre=4, a
v.minPre=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 v.Pre=v.minPre=2
sa steka skidamo sve cvorove do cvora 1, dakle cvorove 6,4,1
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 v.Pre=v.minPre=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
v.minPre=1 pa on nije bazni cvor komponente

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

zavrsava se rekurzivni poziv iz cvora 2, ali za njega je v.Pre=7, a
v.minPre=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 v.Pre=v.minPre=1
sa steka skidamo sve cvorove do cvora 0, dakle cvorove 8,5,2,0
i oni predstavljaju zasebnu komponentu
```

Ovaj algoritam je zasnovan na DFS obilasku grafa i složenosti je $O(|V| + |E|)$.