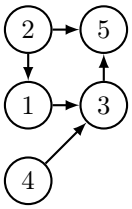
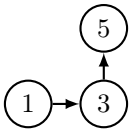
Topološko sortiranje

Razmotrimo sledeći primer. Na ispitu je zadato 5 zadataka. Poznato je da rešenja nekih zadataka zavise od rešenja drugih. Da bi student uradio treći zadatak, potrebno je da iskoristi rešenja iz prvog i četvrtog zadatka. Da bi uradio peti potrebno je da iskoristi rešenja iz trećeg i drugog zadatka. Da bi uradio prvi zadatak potrebno je da iskoristi rešenje iz drugog zadatka. Odrediti jedan redosled kojim student može da radi zadatke kako bi ih sve uradio.

Jasno je da ove zavisnosti možemo prikazati grafom:

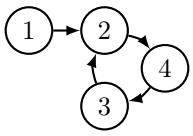


Jedno rešenje bismo mogli odrediti narednim postupkom. Tražimo koji zadatak možemo odmah da uradimo. U grafu su oni predstavljeni čvorovima čiji je ulazni stepen 0 i u ovom konkretnom primeru su to zadaci 2 i 4. Nakon što uradimo te zadatke graf izgleda ovako:



Ponavljamo postupak - radimo sve zadatke koji više ne zavise ni od jednog drugog zadatka tj. biramo sve čvorove čiji je ulazni stepen sada 0. Ovde je to zadatak 1. Postupak se dalje nastavlja na sličan način i nakon toga radimo zadatak 3 i konačno zadatak 5. Rešenje koje smo dobili ovim postupkom je $2-4-1-3-5$. U ovom primeru postoje još dva moguća rešenja i to su $2-1-4-3-5$ i $4-2-1-3-5$. Jedan takav niz naziva se topološko uređenje čvorova datog grafa.

Topološko uređenje ne mora uvek postojati. U ovom primeru postojanje ciklusa u grafu predstavlja postojanje zadataka koji zavise jedan od drugog. U takvom slučaju nije moguće pronaći redosled zadataka takav da zadatak koji radimo ne zavisi od rešenja zadataka onih zadataka koje još nismo uradili. Naredni primer ilustruje takvu situaciju.

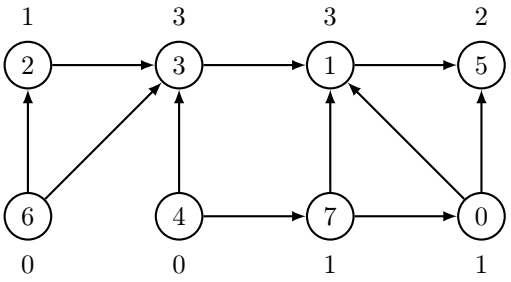


Nakon urađenog zadatka 1 preostaju zadaci 2, 3 i 4 pri čemu nijedan nije moguće uraditi.

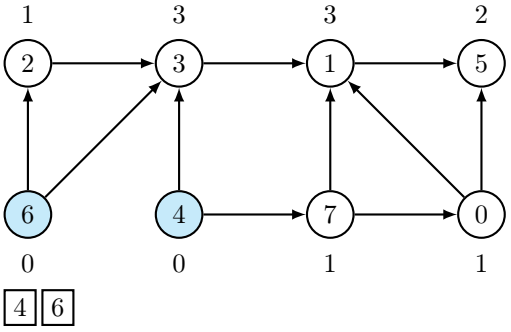
Kanov algoritam

Jedan od načina za određivanje topološkog sortiranja grafa je Kanov algoritam. Ideja ovog algoritma je upravo ona koju smo malopre prikazali na primeru. Redom ćemo u niz dodavati one čvorove čiji je ulazni stepen 0, pri čemu ćemo ih "izbacivati" iz grafa zajedno sa njima incidentnim granama. U samoj implementaciji zapravo nema potrebe da uklanjamo čvor iz grafa, već je dovoljno da to izbacivanje simuliramo tako što ćemo umanjiti ulazni stepen svakog njegovog suseda za jedan.

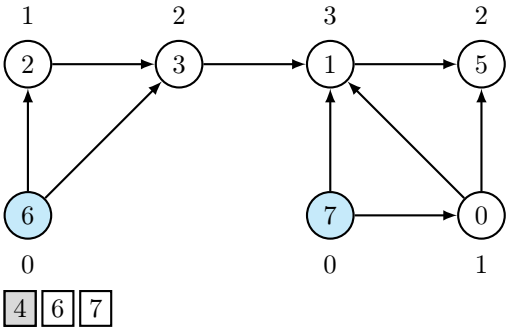
Prikažimo postupak na sledećem primeru.



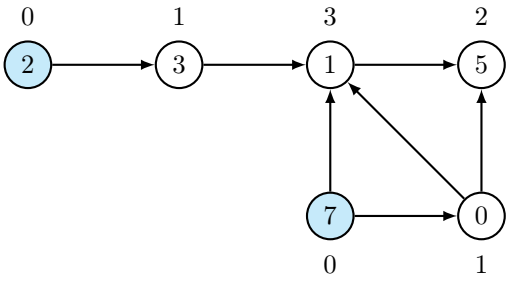
Određujemo ulazni stepen svakog čvora i pronalazimo čvorove sa ulaznim stepenom 0. Njih dodajemo u niz.



Redom obrađujemo čvorove u nizu. Prvi čvor koji obrađujemo je 4. Umanjujemo ulazne stepene njegovih suseda (čvorovi 3 i 7) za jedan.

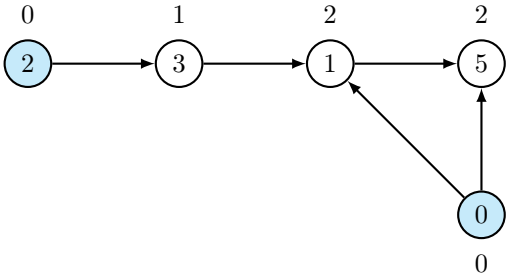


Nakon toga je ulazni stepen čvora 7 jednak 0 pa dodajemo i njega na kraj niza. Obrađujemo čvor 6. Umanjujemo ulazne stepene njegovih suseda (čvorovi 2 i 3) za jedan.



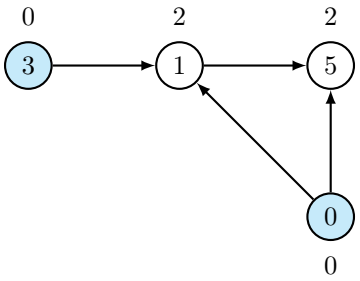
Nakon toga je ulazni stepen čvora 2 jednak 0 pa ga dodajemo na kraj niza. Obrađujemo čvor 7. Umanjujemo ulazne stepene njegovih suseda (čvorovi 0 i 1) za jedan.

4	6	7	2



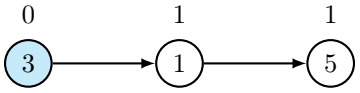
Nakon toga je ulazni stepen čvora 0 jednak 0 pa ga dodajemo na kraj niza. Obrađujemo čvor 2. Umanjujemo ulazne stepene njegovih suseda (čvor 3) za jedan.

4	6	7	2	0



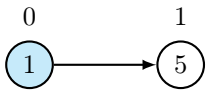
Nakon toga je ulazni stepen čvora 3 jednak 0 pa ga dodajemo na kraj niza. Obrađujemo čvor 0. Umanjujemo ulazne stepene njegovih suseda (čvorovi 1 i 5) za jedan.

4	6	7	2	0	3



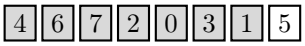
Obrađujemo čvor 3. Umanjujemo ulazne stepene njegovih suseda (čvor 1) za jedan.

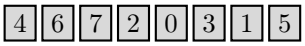
4	6	7	2	0	3



Nakon toga je ulazni stepen čvora 1 jednak 0 pa ga dodajemo na kraj niza. Obrađujemo čvor 1. Umanjujemo ulazne stepene njegovih suseda (čvor 5) za jedan.

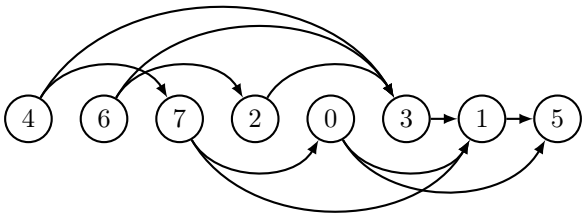
Nakon toga je ulazni stepen čvora 5 jednak 0 pa ga dodajemo na kraj niza. Obrađujemo čvor 5. Umanjujemo ulazne stepene njegovih suseda (nema takvih) za jedan.





Kako smo stigli do kraja niza i obradili sve čvorove, algoritam je završen i u nizu nam se nalazi jedno topološko uređenje.

Kada rasporedimo čvorove po dobijenom uređenju graf izgleda ovako:



Primećujemo da su sve grane grafa usmerene udesno. U slučaju da graf sadrži ciklus ne postoji redosled čvorova sa takvim svojstvom - bar jedna grana tog ciklusa bi morala biti usmerena ulevo. U tom slučaju algoritam neće obraditi sve čvorove, pa nam to daje način da prepoznamo da li je graf acikličan ili ne. Uopšteno, usmeren graf ima topološko uređenje ako i samo ako je acikličan.

Prvi korak algoritma je određivanje ulaznih stepena čvorova. Njih ćemo čuvati u vektoru *indeg* čije ćemo vrednosti na početku postaviti na 0. Za svaki čvor grafa posmatramo sve njegove susede i svakom od njih uvećavamo ulazni stepen za 1. Nakon takvog prolaska kroz sve čvorove grafa se u vektoru *indeg* na indeksu v nalazi ulazni stepen čvora v.

Nakon toga jednim prolazom kroz vektor *indeg* pronalazimo čvorove sa ulaznim stepenom 0 i smeštamo ih u vektor *sortirani*.

Zatim prolazimo kroz vektor sortiranih čvorova. Svakom susedu trenutno posmatranog čvora umanjujemo ulazni stepen za jedan i ukoliko je nakon te promene njegov ulazni stepen postao 0 dodajemo ga na kraj vektora *sortirani*.

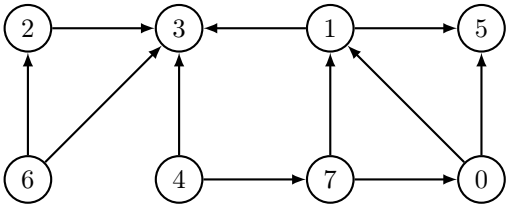
```
vector<int> topsort(vector< vector<int> >& g) {
  int n = g.size();

  // Određujemo ulazne stepene svih čvorova - za svaki čvor
  // brojimo koliko puta se našao kao sused nekog drugog čvora
  vector<int> indeg(n);
  for(auto& lista : g)
    for(int sused : lista)
      indeg[sused]++;

  // U niz smeštamo sve čvorove čiji je ulazni stepen 0
  vector<int> sortirani;
  for(int i = 0; i < n; i++)
    if(indeg[i] == 0)
      sortirani.push_back(i);

  // Prolazimo kroz sve čvorove u nizu...
  for(int i = 0; i < sortirani.size(); i++)
    // ... i svakom susedu trenutnog čvora koji obrađujemo
    // smanjujemo ulazni stepen i dodajemo ga u niz ukoliko
    // ima potrebe
    for(int sused : g[sortirani[i]])
      if(--indeg[sused] == 0)
        sortirani.push_back(sused);

  return sortirani;
}
```

Određivanje ulaznih stepena čvorova podrazumeva prolazak kroz sve čvorove i njihove susede pa je složenost tog dela postupka $O(|V| + |E|)$. Pronalaženje čvorova sa ulaznim stepenom 0 je složenosti $O(|V|)$. Poslednji deo algoritma zahteva prolazak kroz sve čvorove i njihove susede pa je i njegova složenost $O(|V| + |E|)$. Dakle, ukupna složenost algoritma je $O(|V| + |E|)$.

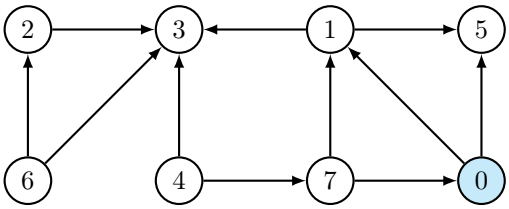
Pretraga u dubinu

Drugi način za određivanje topološkog sortiranja grafa je modifikovanom pretragom u dubinu. Pokretanjem pretrage iz nekog čvora v možemo jednostavno odrediti sve čvorove koji se u topološkom uređenju moraju naći nakon čvora v - to su svi čvorovi njegovog DFS podstabla. Prikažimo na primeru kako ćemo

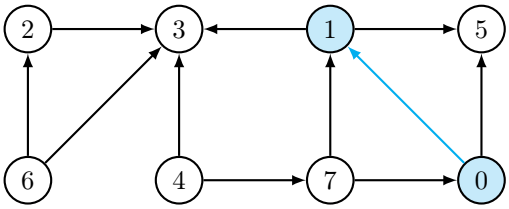
to iskoristiti.



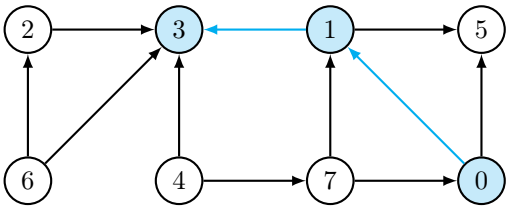
Započinjemo pretragu u dubinu od prvog neobrađenog čvora. To je čvor 0.



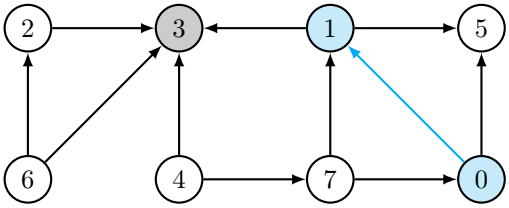
Pretragom ćemo obići sve potomke čvora 0 i dodati ih u niz. Nakon toga ćemo dodati čvor 0 na početak tog niza kako bi se on našao pre svih svojih potomaka u topološkom uređenju.



Pretragom ćemo obići sve potomke čvora 1 i dodati ih u niz. Nakon toga ćemo dodati čvor 1 na početak tog niza kako bi se on našao pre svih svojih potomaka u topološkom uređenju.

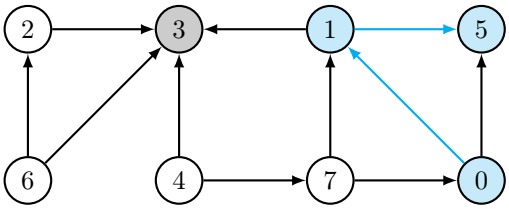


Pretragom ćemo obići sve potomke čvora 3 (takvih nema) i dodati ih u niz. Nakon toga ćemo dodati čvor 3 na početak tog niza kako bi se on našao pre svih svojih potomaka u topološkom uređenju.



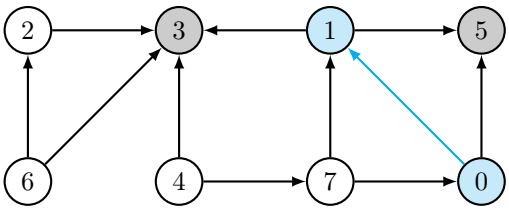
Čvor 3 dodajemo na početak niza.

3



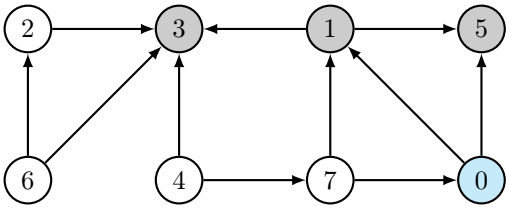
Pretragom ćemo obići sve potomke čvora 5 (takvih nema) i dodati ih u niz. Nakon toga ćemo dodati čvor 5 na početak tog niza kako bi se on našao pre svih svojih potomaka u topološkom uređenju.

3



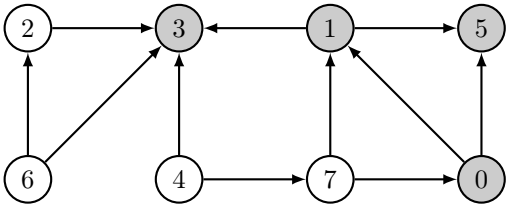
Čvor 5 dodajemo na početak niza.

5 3



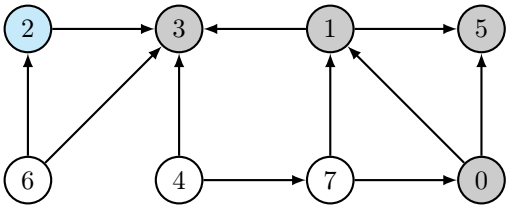
Nakon što smo obišli sve potomke čvora 1 dodajemo i njega u niz, na početak. Primetimo da se sada svi njegovi potomci nalaze iza njega u nizu.

1 5 3



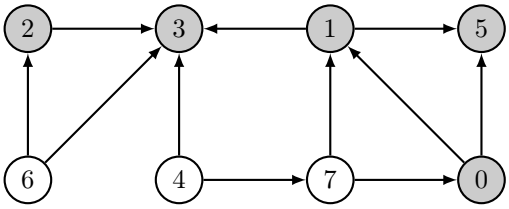
0 1 5 3

Nakon što smo obišli sve potomke čvora 0 dodajemo i njega u niz, na početak. Primetimo da se sada svi njegovi potomci nalaze iza njega u nizu.



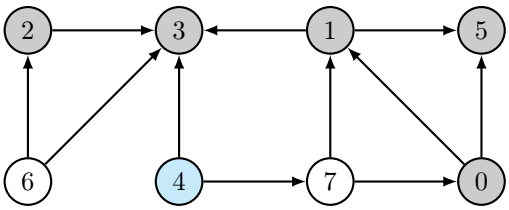
0 1 5 3

Pošto smo završili sa pretragom iz 0, biramo prvi sledeći neobiđen čvor. Pretragom ćemo obići sve **neobiđene** potomke čvora 2 (takvih nema) i dodati ih u niz. Nema potrebe da ponovo obilazimo obiđene potomke jer se oni već nalaze u nizu, a kada dodamo 2 na početak niza oni će se nalaziti iza čvora 2.



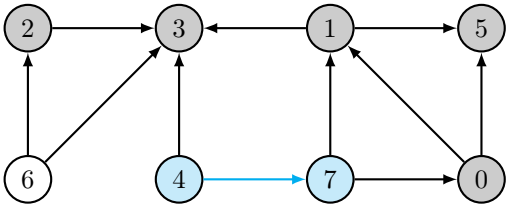
2 0 1 5 3

Čvor 2 dodajemo na početak niza. Primećujemo da je čvor 3 već u nizu i to iza čvora 2.



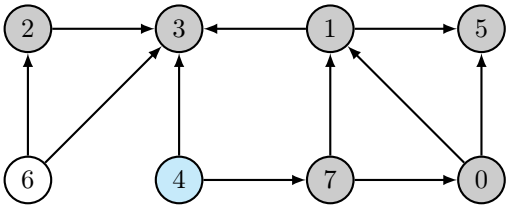
2 0 1 5 3

Biramo prvi sledeći neobiđen čvor. Pretragom ćemo obići sve neobiđene potomke čvora 4 i dodati ih u niz.



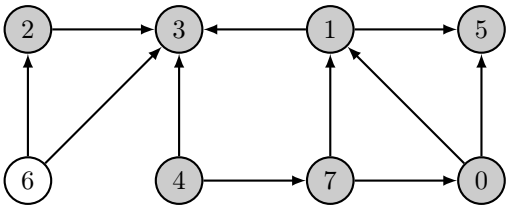
Pretragom ćemo obići sve neobiđene potomke čvora 7 (takvih nema) i dodati ih u niz.

2	0	1	5	3



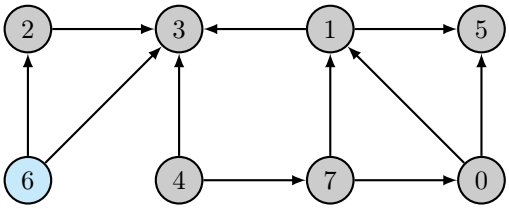
Čvor 7 dodajemo na početak niza.

7	2	0	1	5	3



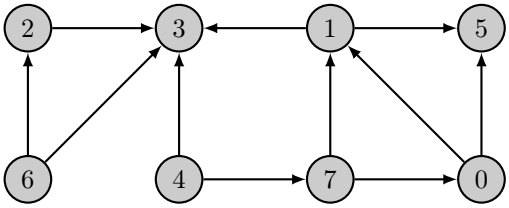
Čvor 4 dodajemo na početak niza.

4	7	2	0	1	5	3



Biramo prvi sledeći neobiđen čvor. Pretragom ćemo obići sve neobiđene potomke čvora 6 (takvih nema) i dodati ih u niz.

4	7	2	0	1	5	3



Kako su svi čvorovi obiđeni i smešteni u niz algoritam je završen i dobili smo jedno topološko uređenje.

6	4	7	2	0	1	5	3

Potrebno je da pamtimo koje smo čvorove obišli i za to ćemo koristiti vektor *mark*. Redom prolazimo kroz sve čvorove i ukoliko naiđemo na neki koji nije obiđen, puštamo pretragu iz njega. U samoj pretrazi je potrebno da rekurzivno obiđemo sve neobiđene susede trenutnog čvora. Nakon svih rekurzivnih poziva pretrazi dodajemo trenutni čvor na kraj vektora *sortirani*. Na samom kraju algoritma obrćemo taj vektor i time dobijamo topološko uređenje.

```
void dfs(int u, vector<bool>& mark,
                vector<int>& sortirani,
                vector< vector<int> >& g) {
  // Markiramo čvor u da je obiđen
  mark[u] = true;

  // Pozivamo pretragu za sve neobiđene susede od u
  for(int v : g[u])
    if(!mark[v])
      dfs(v, mark, sortirani, g);

  // Na kraju dopisujemo čvor u niz
  sortirani.push_back(u);
}

vector<int> topsort(vector< vector<int> >& g) {
  int n = g.size();

  // Nizovi potrebni za DFS
  vector<bool> mark(n);
  vector<int> sortirani;

  // Prolazimo redom kroz čvorove i puštamo pretragu iz svakog neobiđenog
  for(int i = 0; i < n; i++)
    if(!mark[i])
      dfs(i, mark, sortirani, g);

  reverse(sortirani.begin(), sortirani.end());
```

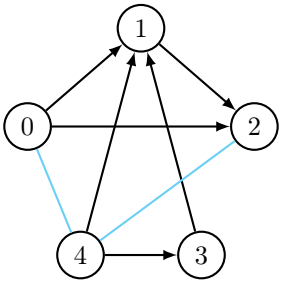
```
  return sortirani;
}
```

Kako svaka pretraga ima vremensku složenost određenu brojem obiđenih čvorova i grana ukupna složenost za obilazak celog grafa je $O(|V| + |E|)$.

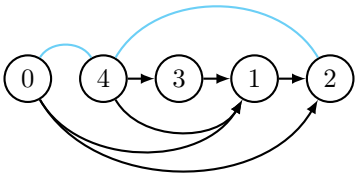
Napomenimo da je, kao i kod Kanovog algoritma, moguće odrediti da li je graf acikličan ili ne. Nakon što smo obradili sve čvorove i odredili niz, potrebno je da proverimo da li su sve grane usmerene udesno. Ukoliko jesu, niz predstavlja topološko uređenje. Ukoliko nisu, graf ima ciklus.

Zadatak - Usmeravanje grana

Dat je usmeren acikličan graf G i skup neusmerenih grana U nad čvorovima grafa G. Potrebno je svakoj grani iz U dodeliti smer tako da dobijeni graf bude acikličan.



Kako je G acikličan znamo da postoji topološko uređenje. Poređajmo čvorove grafa po jednom topološkom uređenju.



Primećujemo da je sada dovoljno usmeriti svaku granu skupa U udesno jer time ne narušavamo acikličnost grafa.

Jedno zanimljivo zapažanje je da granu od 0 do 4 možemo usmeriti i ulevo. To je zato što zamenom njihovih mesta dobijamo drugo topološko uređenje ovog grafa. Granu od 4 do 2 ipak ne smemo da usmerimo ulevo jer time nastaju ciklusi, npr. $4 - 1 - 2$.

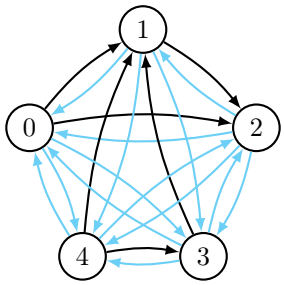
Prvo određujemo topološko sortiranje grafa. Prolazimo zatim kroz sve grane skupa U i svakoj dodeljujemo onaj smer koji ide od čvora koji se ranije pojavljuje u topološkom sortiranju do onog koji se kasnije javlja. Kako bismo brzo mogli da odredimo koji čvor se javlja ranije u topološkom sortiranju, nakon što smo odredili topološko uređenje možemo proći kroz vektor koji sadrži uređenje i

na indeks v pomoćnog vektora *pozicija* upisati poziciju čvora v u topološkom sortiranju.

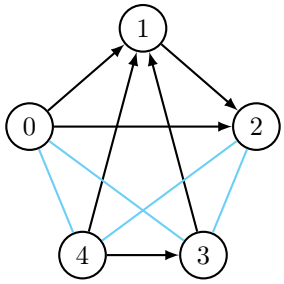
Zadatak - Dodavanje grana

Dat je usmeren acikličan graf G. Potrebno je dodati mu maksimalan broj grana tako da on ostane acikličan.

Posmatrajmo na primeru graf sa svim granama koje potencijalno možemo da dodamo.



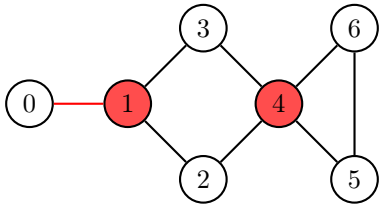
Primetimo da nema smisla dodavati dve grane suprotnog smera koje povezuju isti par čvorova jer bismo time napravili ciklus. Kada njih eliminišemo, ostaje pitanje da li svaku od preostalih grana možemo da dodamo i, ako možemo, kako ih usmeravamo.



Zapravo smo ovim zapažanjem ovaj zadatak sveli na prethodni. Znamo da sve preostale grane možemo da dodamo u graf, a usmeravamo ih na osnovu topološkog uređenja početnog grafa G. Time garantujemo da se u grafu neće pojaviti ciklus.

Artikulacione tačke i mostavi

Ukoliko analiziramo neku mrežu, bila to mreža računara, električna mreža, telefonska mreža, korisno je da umemo da odredimo koji čvorovi i veze te mreže mogu da naprave prekid u komunikaciji u slučaju da se pokvare.



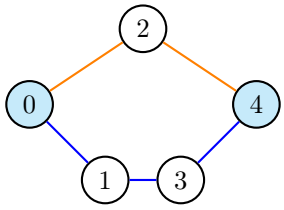
U navedenom primeru vidimo da bi prekid rada veze između čvorova 0 i 1 doveo do toga da čvor 0 gubi mogućnost komunikacije sa ostatkom mreže. Prekid rada čvora 1 bi doveo do iste situacije. Prekid rada čvora 4 bi onemogućio komunikaciju čvorova 5 i 6 sa ostatkom mreže. Sa druge strane, prekid rada čvora 3 ne bi napravio veliki problem jer su svi ostali čvorovi idalje povezani. To je zato što sva komunikacija koja se vrši preko čvora 3 ima i alternativni put, npr. preko čvora 2.

Formalnije, čvor povezanog neusmerenog grafa G je artikulaciona tačka (artikulacioni čvor) ukoliko nakon njegovog uklanjanja (zajedno sa njemu incidentnim granama) dobijeni graf nije povezan. Grana povezanog neusmerenog grafa G je most ukoliko nakon njenog uklanjanja dobijeni graf nije povezan.

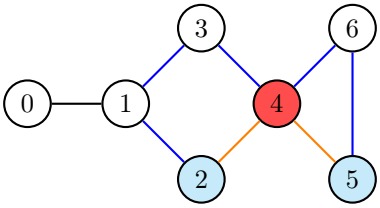
Vidimo da je korisno imati algoritam kojim je moguće odrediti sve artikulacione tačke odnosno sve mostove. Jedan postupak za određivanje svih artikulacionih tačaka u grafu jeste da redom za svaki čvor direktno proverimo da li je artikulacioni - uklonimo ga iz grafa i proverimo da li je graf idalje povezan. Tu proveru možemo vršiti pretragom grafa i njena složenost je $O(|V|+|E|)$. Kako je to potrebno uraditi za svaki čvor složenost celog postupka je $O(|V|(|V|+|E|))$. Postavlja se pitanje: da li možemo bolje?

Artikulacioni čvorovi

Odgovor je da. Ispostavlja se da postoji efikasniji algoritam za određivanje svih artikulacionih tačaka grafa zasnovan na pretrazi u dubinu.



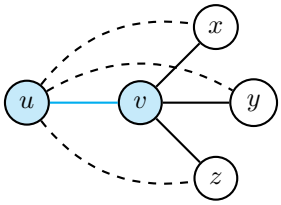
Čvor 2 nije artikulaciona tačka. To jednostavno vidimo iz toga što njenim uklanjanjem čvorovi 0 i 4 ostaju povezani. To je zato što osim puta od čvora 0 do čvora 4 koji ide preko čvora 2 postoji i alternativni put koji ne ide preko čvora 2.



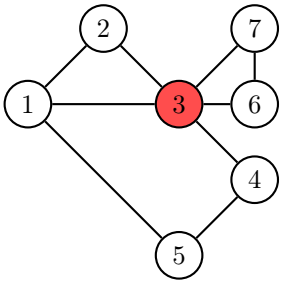
Čvor 4 jeste artikulaciona tačka. Vidimo da za dva njegova suseda, čvorove 2 i 5, ne postoji put koji ih povezuje, a ne ide preko čvora 4. Na slici su označena dva puta koji povezuju čvorove 2 i 5 (naravno postoji još takvih puteva) i vidimo da oba puta moraju da sadrže čvor 4. Štaviše, to važi za svaki put koji povezuje čvorove 2 i 5.

U opštem slučaju važi da je čvor v artikulaciona tačka ako i samo ako za bar jedan par njegovih suseda ne postoji alternativni put koji ih povezuje (odnosno put koji ih povezuje i ne sadrži čvor v). Ekvivalentno, čvor v **nije** artikulaciona tačka ako i samo ako za svaki par njegovih suseda postoji alternativni put.

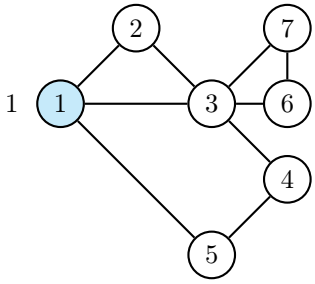
Primetimo još da je ovo ekvivalentno sledećem. Neka je u sused čvora v. Čvor v nije artikulaciona tačka ako i samo ako svi njegovi susedi različiti od u imaju alternativni put do čvora u. Ovo svojstvo će nam omogućiti da konstruišemo efikasan algoritam za pronalaženje svih artikulacionih tačaka.



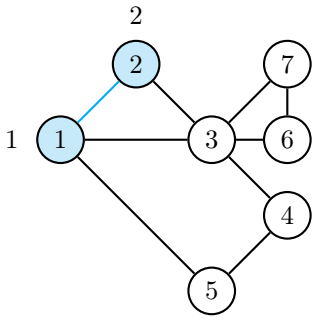
Ukoliko v nije artikulacioni čvor, znamo da svaki par njegovih suseda ima alternativni put. Prema tome svaki sused čvora v različit od u ima alternativni put do čvora u. Sa druge strane, ukoliko svaki sused čvora v ima alternativni put do u, onda ima i alternativni put do svakog drugog suseda - preko čvora u (taj put preko u ne mora biti prost, ali to ne menja tačnost tvrđenja).



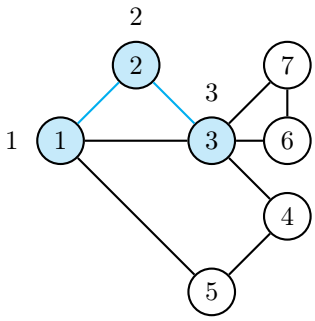
Razmotrimo na ovom primeru kako ćemo iskoristiti to svojstvo da pokažemo da je čvor 3 artikulaciona tačka.



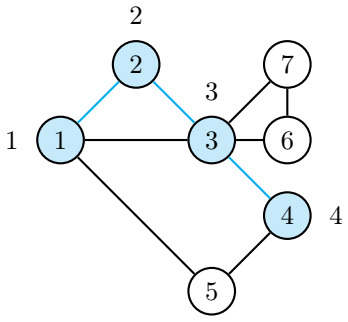
Puštamo pretragu u dubinu iz bilo kog čvora. Pri tome beležimo ulaznu numeraciju za svaki čvor.



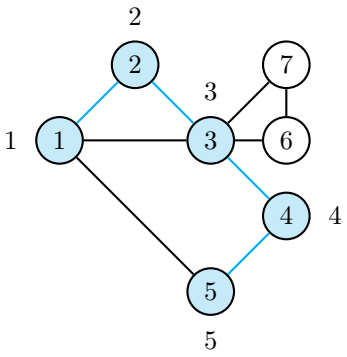
Želimo da proverimo da li svaki sused čvora $v = 3$ različit od $u = 2$ ima alternativni put do čvora 2. Dovoljno je proveriti da li svaki od njih ima put do bilo kog do sada obiđenog čvora jer su svi oni povezani sa čvorom 2 (pripadaju istom DFS stablu). To su zapravo čvorovi čiji je broj DFS numeracije manji od broja DFS numeracije čvora 3.



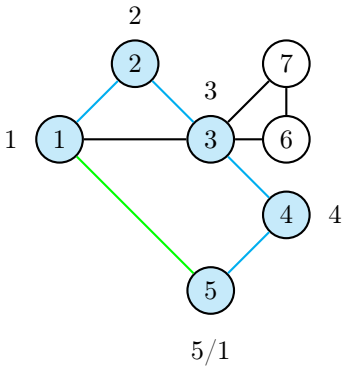
To radimo tako što puštamo pretragu iz svakog suseda i računamo koji je najraniji čvor (čvor sa najmanjim DFS brojem) do kog možemo doći. Prvo puštamo pretragu iz čvora 4.



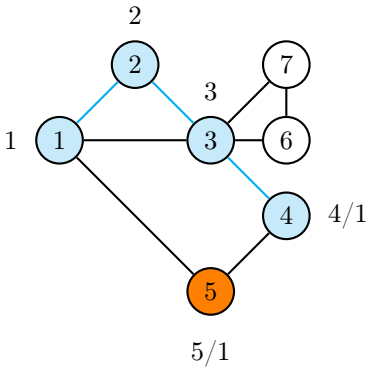
Obilazimo sve susede čvora 4 osim roditelja - čvora 3. Puštamo pretragu iz čvora 5.



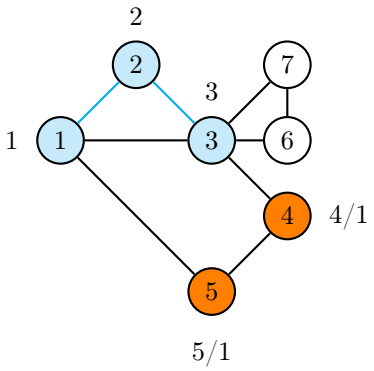
Obilazimo sve susede čvora 5 osim čvora 4.



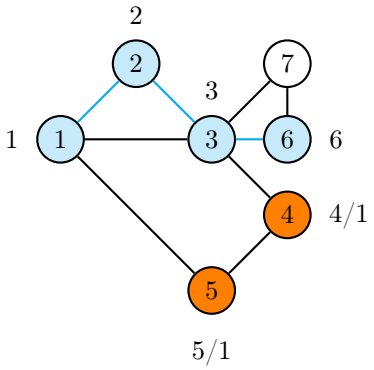
Čvor 1 je već posećen. Pošto je on najraniji čvor do kog smo došli iz čvora 5, beležimo to (beleži se DFS broj, a ne *id* čvora).



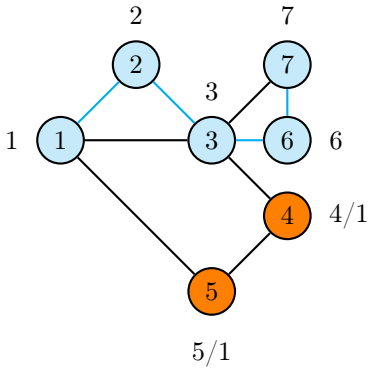
Kako smo obišli sve susede čvora 5, završavamo sa njegovom pretragom i kao rezultat (najraniji pronađen čvor) vraćamo 1. To je najraniji čvor do kog smo došli iz čvora 4, pa beležimo to.



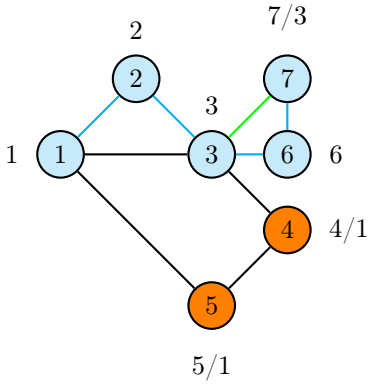
Kako smo obišli sve susede čvora 4, završavamo sa njegovom pretragom i kao rezultat (najraniji pronađen čvor) vraćamo 1. Kako je 1 manje od 3 znamo da postoji alternativni put od čvora 4 do čvora 2.



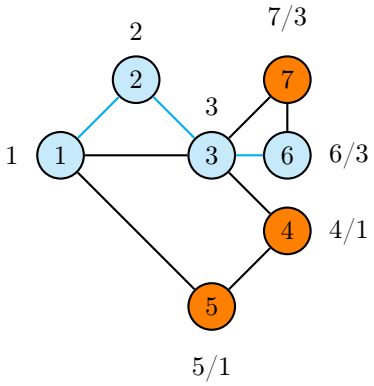
Obilazimo sledećeg suseda čvora 3. To je čvor 6.



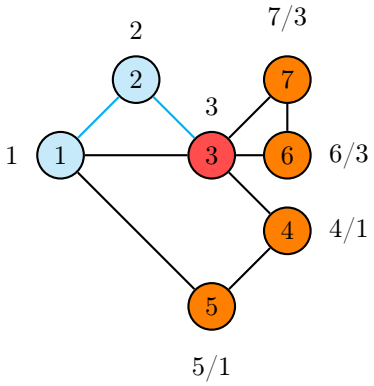
Obilazimo čvor 7.



Čvor 3 je već obiđen. On je najraniji čvor do kog smo došli iz čvora 7 pa to beležimo.



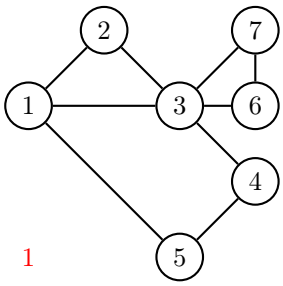
Kako smo obišli sve susede čvora 7, završavamo sa njegovom pretragom i kao rezultat (najraniji pronađen čvor) vraćamo 3. To je najraniji čvor do kog smo došli iz čvora 6, pa beležimo to.

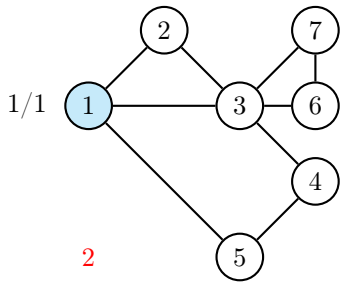


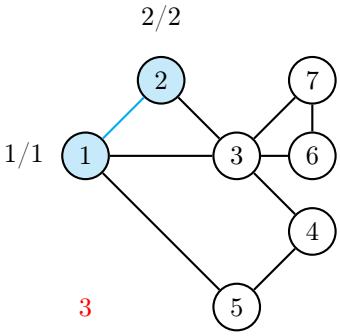
Kako smo obišli sve susede čvora 6, završavamo sa njegovom pretragom i kao rezultat (najraniji pronađen čvor) vraćamo 3. Kako nismo pronašli put od 6 do 2 (jer nije $3 < 3$), čvor 3 je artikulaciona tačka.

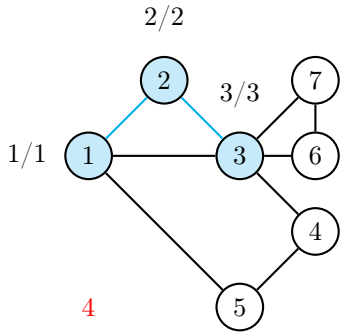
Ovaj postupak je moguće uopštiti tako da istu proveru koju je radio za čvor 3 sada radi za sve čvorove tokom jednog obilaska grafa. Za svaki čvor pored

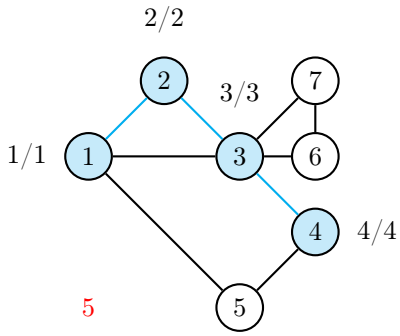
njegove DFS numeracije pamtimo i koja je DFS numeracija najranijeg čvora do kog možemo doći iz njega (u daljem tekstu *low*). Prilikom obilaska suseda čvorova postoje dve mogućnosti. Ukoliko je sused već obiđen i njegov DFS broj je manji od trenutne *low* vrednosti, ažuriramo *low*. Ukoliko sused nije obiđen rekurzivno računamo *low* vrednost suseda i ukoliko je ona manja od *low* vrednosti trenutnog čvora ažuriramo *low* vrednost trenutnog. Ukoliko je *low* vrednost suseda veća ili jednaka od DFS numeracije trenutnog čvora, trenutni čvor je artikulaciona tačka. Nakon što smo obišli sve susede trenutnog čvora, kao rezultat vraćamo njegovu *low* vrednost.

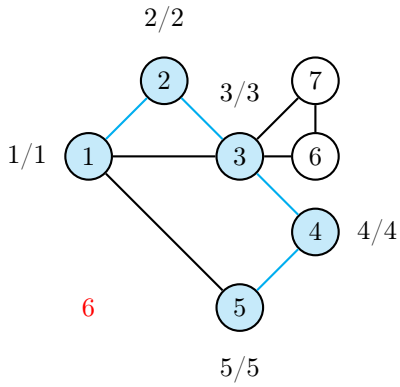


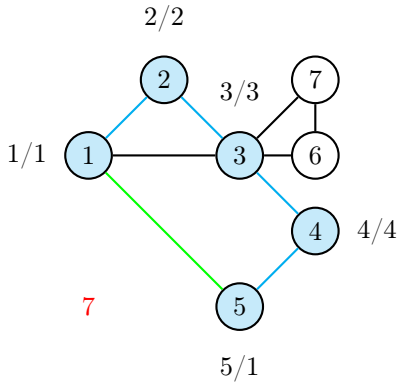


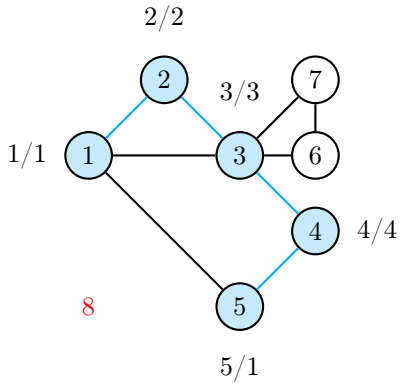


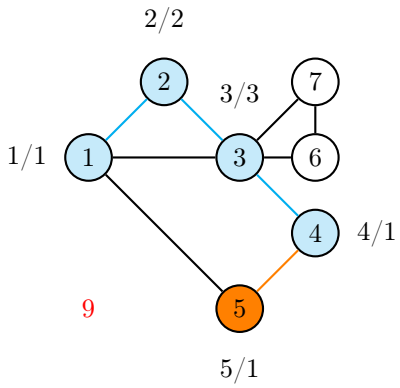


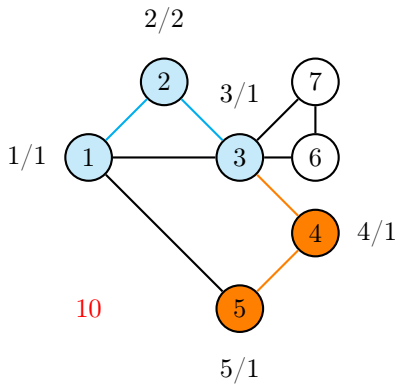

2/2
2
3/3
7
1/1
1
3
6
4
4/4
5
5

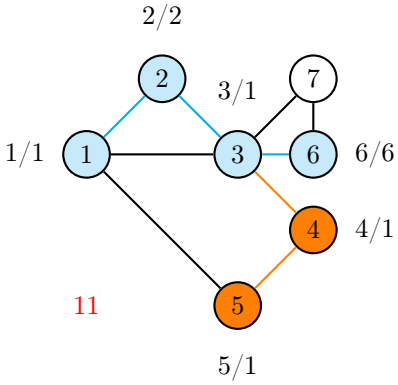

2/2
2
3/3
7
1/1
1
3
6
4
4/4
6
5
5/5

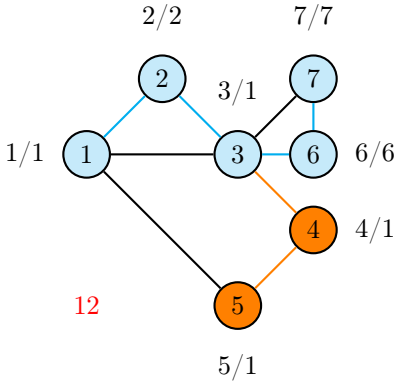

2/2
2
3/3
7
1/1
1
3
6
4
4/4
7
5
5/1

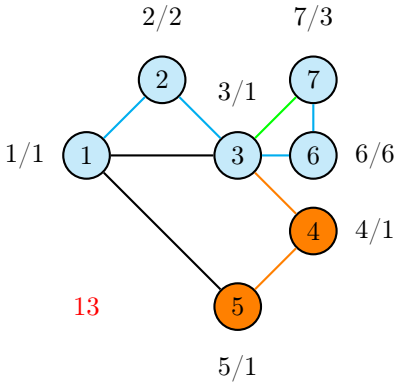

2/2
2
3/3
7
1/1
1
3
6
4
4/4
8
5
5/1

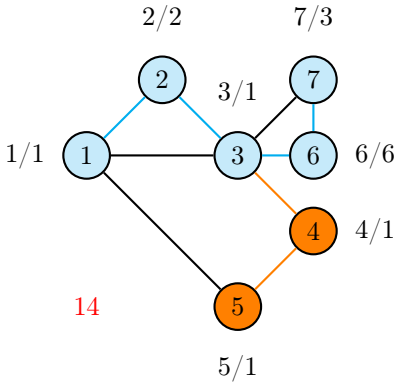

2/2
2
3/3
7
1/1
1
3
6
4
4/1
9
5
5/1

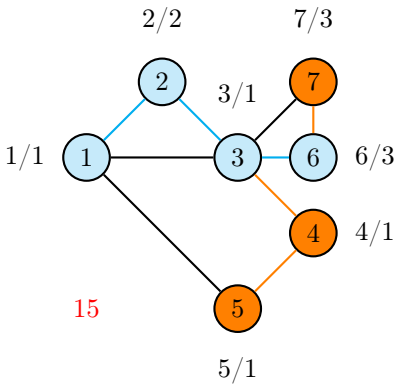

2/2
2
3/1
7
1/1
1
3
6
4
4/1
10
5
5/1

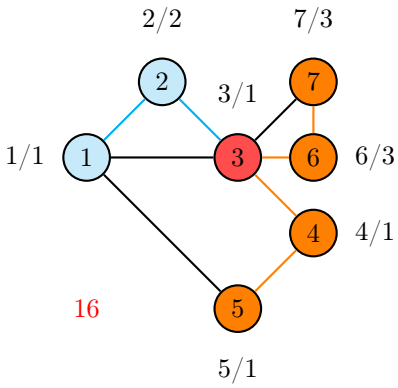

2/2
2
3/1
7
1/1
1
3
6
6/6
4
4/1
11
5
5/1

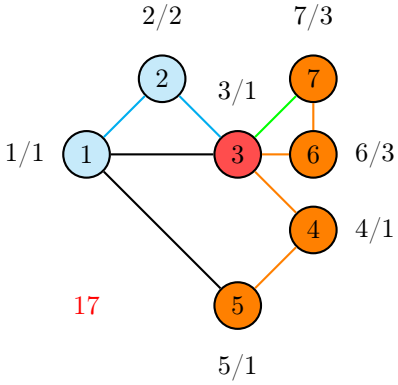

2/2
7/7
2
3/1
7
1/1
1
3
6
6/6
4
4/1
12
5
5/1

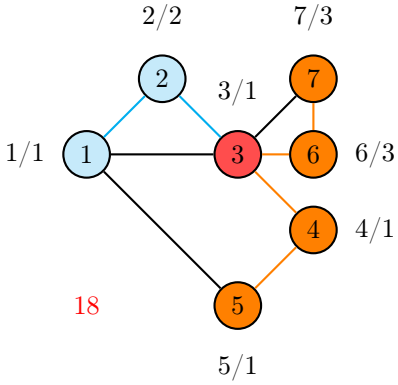

2/2
7/3
2
3/1
7
1/1
1
3
6
6/6
4
4/1
13
5
5/1

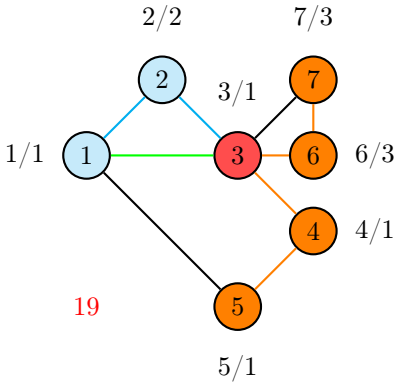

2/2
7/3
2
3/1
7
1/1
1
3
6
6/6
4
4/1
14
5
5/1

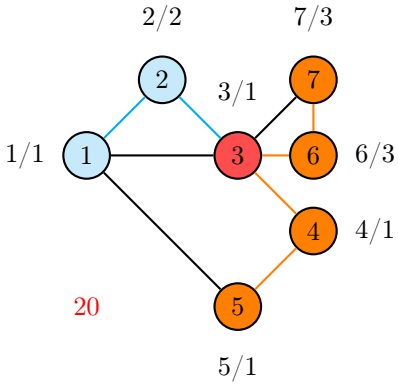

2/2
7/3
2
3/1
7
1/1
1
3
6
6/3
4
4/1
15
5
5/1

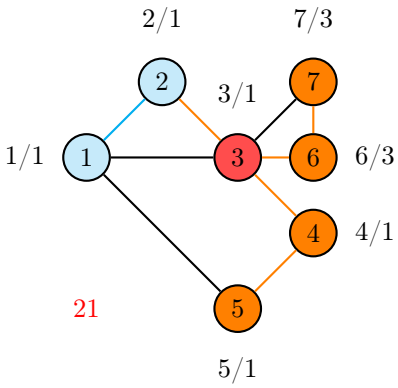

2/2
7/3
2
3/1
7
1/1
1
3
6
6/3
4
4/1
16
5
5/1

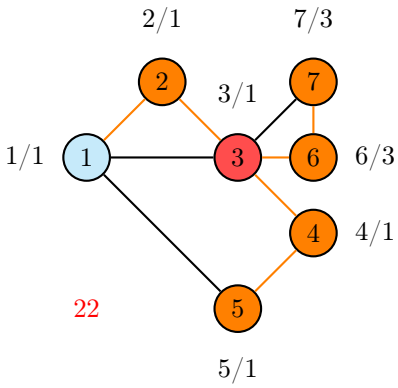

2/2
7/3
2
3/1
7
1/1
1
3
6
6/3
4
4/1
17
5
5/1

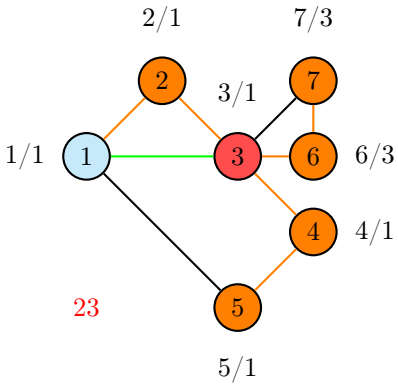

2/2
7/3
2
3/1
7
1/1
1
3
6
6/3
4
4/1
18
5
5/1

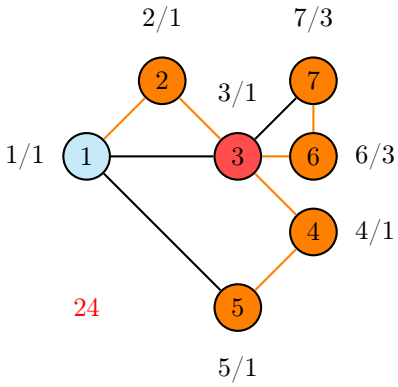

2/2
7/3
2
3/1
7
1/1
1
3
6
6/3
4
4/1
19
5
5/1

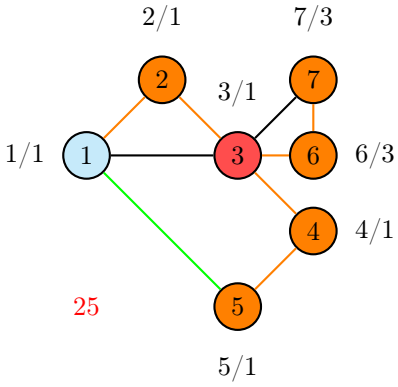

2/2
7/3
2
3/1
7
1/1
1
3
6
6/3
4
4/1
20
5
5/1

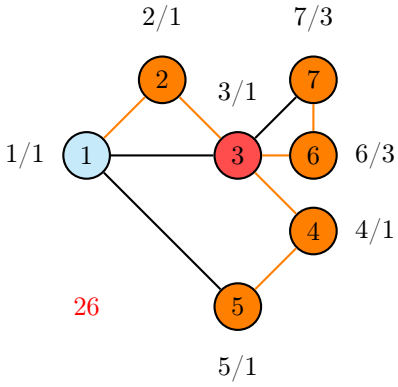

2/1
7/3
2
3/1
7
1/1
1
3
6
6/3
4
4/1
21
5
5/1

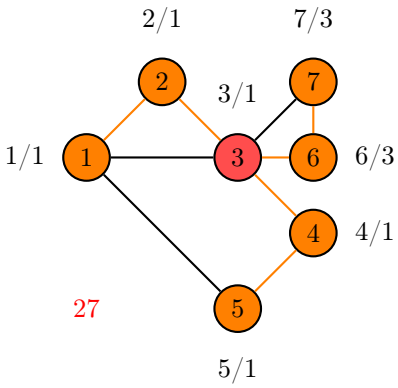

2/1
7/3
2
3/1
7
1/1
1
3
6
6/3
4
4/1
22
5
5/1



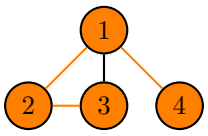








Obratimo pažnju na specijalan slučaj: čvor iz kog započnemo pretragu ima DFS numeraciju 1 pa samim tim je *low* vrednost svakog njegovog suseda veća ili jednaka 1. Po prethodnom algoritmu bi taj čvor trebalo da bude artikulaciona tačka, ali kao što vidimo u prethodnom primeru to nije slučaj. Razlog je to što čvor 1 ima samo jedno dete u DFS stablu. U slučaju da ih ima 2 ili više, to je onda zaista artikulacioni čvor.



U ovom primeru čvor 1 ima više dece u DFS stablu. Njegovim uklanjanjem bi sva njegova deca postala međusobno nepovezana, pa je on artikulaciona tačka.

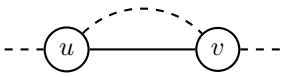
Pri implementaciji nam je potrebno da čuvamo DFS numeraciju svakog čvora. U vektoru *mark* će biti upisano 0 ukoliko čvor nije obiđen, odnosno DFS numeracija tog čvora ukoliko jeste. U svakom DFS pozivu ćemo prosleđivati iz kog smo čvora došli (odnosno roditelja). Prvom čvoru za koji pozivamo DFS ćemo proslediti -1. Funkcija će vraćati *low* vrednost čvora kako bismo je mogli računati rekurzivno.

```
int dfs(int u, int p, int& t, vector<int>& mark,
                              vector< vector<int> >& g) {
  int low = mark[u] = ++t;
  bool ap = false;
  int count = 0; // Za specijalan slučaj - broj DFS podstabala
  for(auto v : g[u])
    if(!mark[v]) {
      int vlow = dfs(v, u, t, mark, g);

      // Ako nije koren DFS stabla ispitujemo uslov
      if(p != -1 && vlow >= mark[u])
        ap = true;

      low = min(low, vlow);
      count++; // Uvećavamo broj DFS podstabala
    }
    else if(v != p)
      low = min(low, mark[v]);

  // Proveravamo da li je artikulaciona tačka ili specijalan
  // slučaj za koren stabla
  if(ap || (p == -1 && count > 1))
    cout << u << ' ';

  return low;
}
```

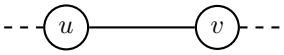
Uobičajena je i implementacija u kojoj *low* vrednosti čuvamo u nizu, pa onda povratnu vrednost funkcije možemo koristiti za nešto drugo. Jasno je da je složenost algoritma $O(|V| + |E|)$ - vršimo jednu pretragu u dubinu.

Mostovi

Algoritam određivanja svih mostova u grafu je veoma sličan algoritmu za određivanje artikulacionih tačaka. Zasnovan je na zapažanju da je grana uv most ako i samo ako ne postoji alternativni put (put koji ne sadrži granu uv) koji povezuje čvorove u i v.



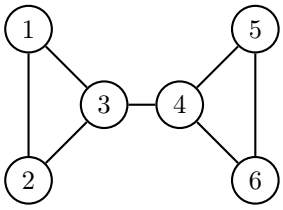
Postoji put od u do v koji ne sadrži granu uv, pa oni ostaju povezani i nakon njenog izbacivanja.



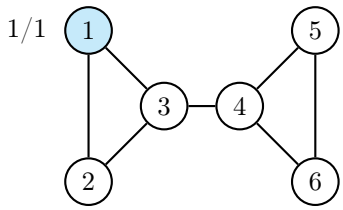
Ne postoji put od u do v koji ne sadrži granu uv, pa njenim izbacivanjem ova dva čvora postaju nepovezana.

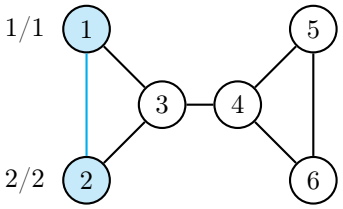
Prilikom DFS obilaska grafa za svaki čvor računamo DFS numeraciju i *low* vrednost na isti način kao i kod artikulacionih tačaka. Prilikom obilaska neposećenog suseda v čvora u proveravamo da li je grana uv most, odnosno da li postoji alternativni put od v do u. To ponovo radimo poređenjem *low* vrednosti čvora v sa DFS numeracijom čvora u. Ukoliko je veća, alternativni put ne postoji i grana uv je most.

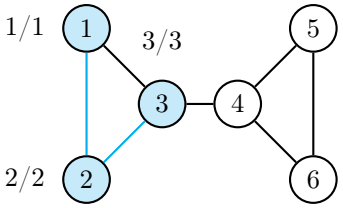
Napomenimo još da u slučaju posećenog suseda ne moramo da proveravamo da li je uv most. To je zato što takve grane u DFS obilasku neusmerenog grafa mogu biti ili direktne ili povratne, pa sigurno postoji alternativni put - granama DFS stabla.



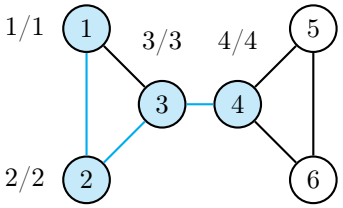
Pokrećemo pretragu iz čvora 1.

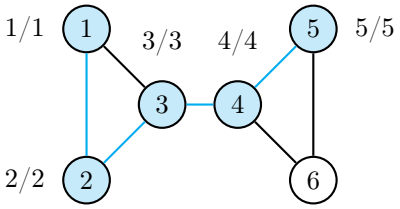


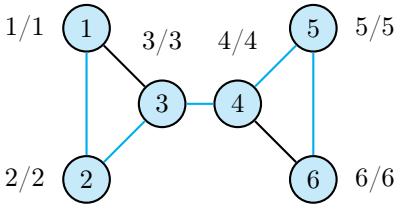


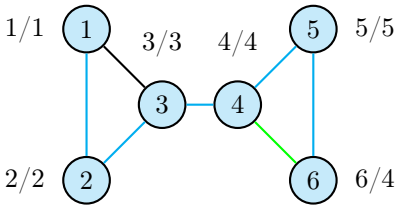


Želimo da proverimo da li postoji put od čvora 4 koji ne sadrži granu $(3, 4)$ odnosno da li od čvora 4 možemo doći do nekog od čvorova koje smo obišli do sada. Drugim rečima, želimo da proverimo da li je $low(4) < 4$.

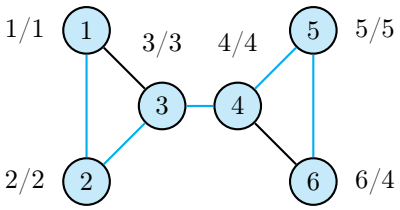


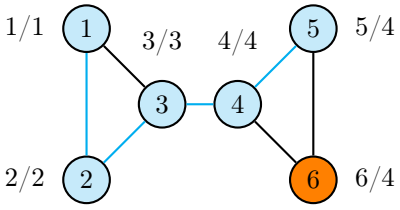


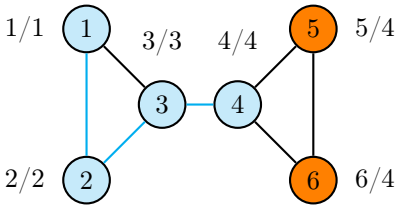


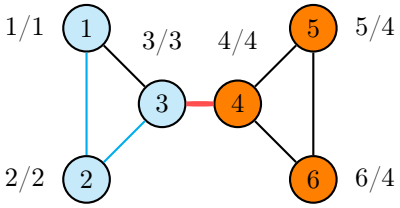


Želimo da proverimo da li postoji put od čvora 4 koji ne sadrži granu $(3, 4)$ odnosno da li od čvora 4 možemo doći do nekog od čvorova koje smo obišli do sada. Drugim rečima, želimo da proverimo da li je $low(4) < 4$.









Kako $low(4)$ nije manje od 4, ne postoji alternativni put pa je grana $(3, 4)$ most.

U implementaciji je potrebno da računamo DFS numeraciju i *low* vrednosti čvorova kao kod artikulacionih tačaka. Pored toga preostaje samo ispitivanje jednog uslova za proveru da li je grana most. Za razliku od artikulacionih tačaka, nema specijalnih slučajeva koje je potrebno obraditi.

```
int dfs(int u, int p, int& t, vector<int>& mark,
```

```
                                vector< vector<int> >& g) {
  int low = mark[u] = ++t;
  for(auto v : g[u])
    if(!mark[v]) {
      int vlow = dfs(v, u, t, mark, g);

      if(vlow > mark[u])
        cout << u << ' ' << v << '\n';

      low = min(low, vlow);
    }
    else if(v != p)
      low = min(low, mark[v]);

  return low;
}
```