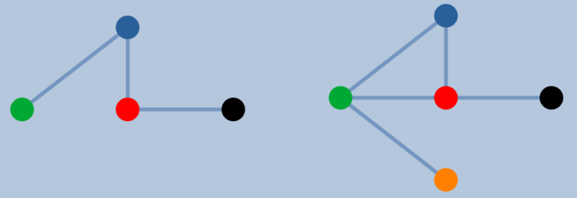
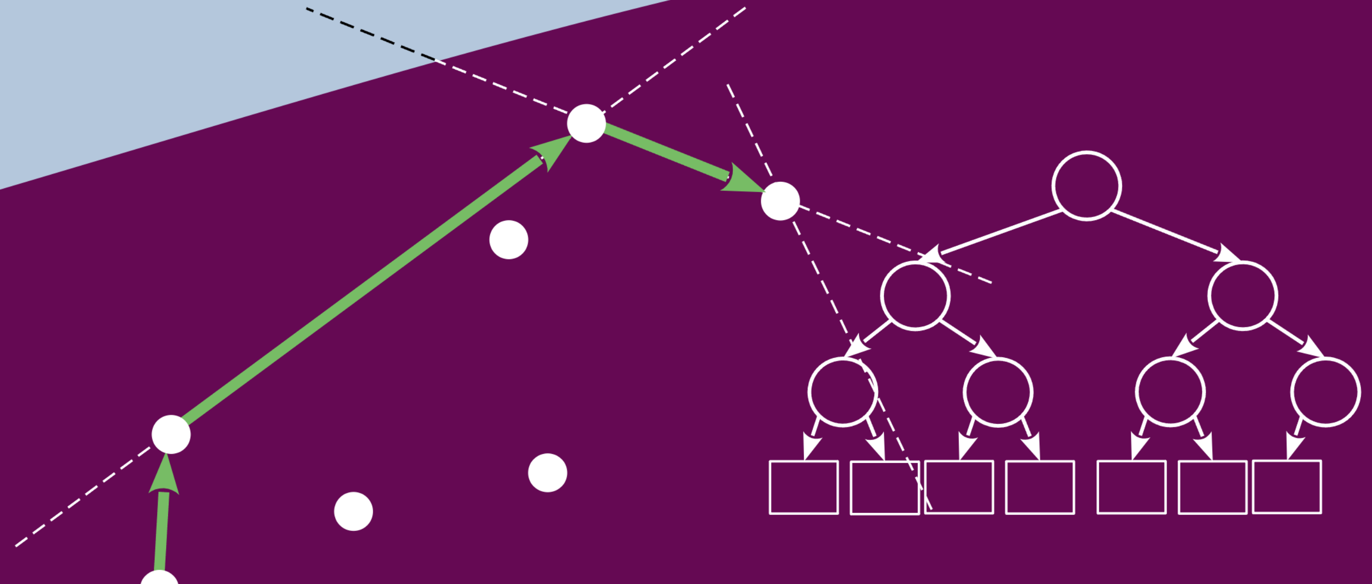
Napredni algoritmi i strukture podataka

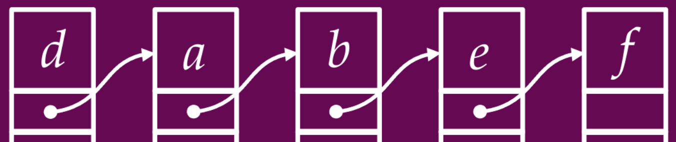
Tjedan 10: Algoritmi nad grafovima

Detekcija ciklusa, Min. razapinjujući grafovi,

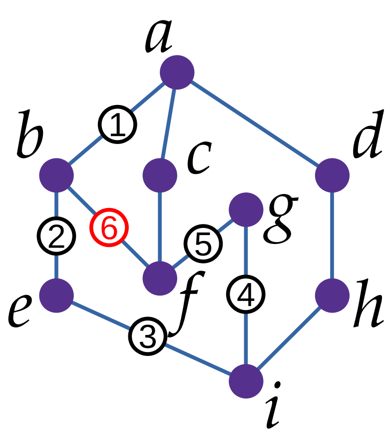
Eulerovi grafovi







Detekcija ciklusa



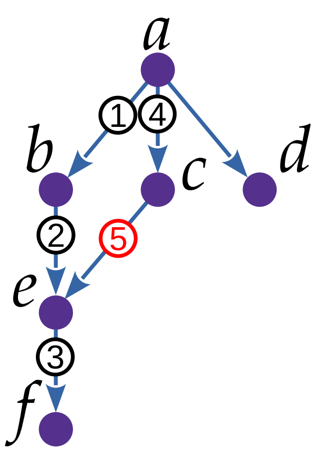
Detekcija ciklusa DFS-om

- Detekcija ciklusa bitan je zadatak za mnoge graf algoritme: recimo algoritmi za pronalaženje minimalnog razapinjujućeg stabla ili za detekciju Hamiltonovih ciklusa ili Eulerovih krugova
- Najjednostavniji način je koristiti DFS ili BFS za obilazak grafa
  - Označimo svaki vrh koji obiđemo
  - Ako u obilasku naiđemo na već označeni vrh, tada smo detektirali ciklus u grafu

Alternativna literatura:

- Thomas H, Cormen, et al. "Introduction to Algorithms." (2016). Introduction to Algorithms., Chapter VI

Detekcija ciklusa



- Ovaj pristup ima veliki nedostatak (recimo za DFS)
  - Nakon povratka iz rekurzije počinjemo se ponovo spuštati kroz graf
  - Oznake da smo obišli neke od vrhova nismo brisali
  - Ako u tom spuštanju naiđemo na označeni vrh – da li smo detektirali ciklus?
  - To zapravo nikad nismo sigurni
  - Ako bismo uklanjali oznake, to bi moglo prouzročiti problem s pronalaženjem ciklusa u nepovezanim grafovima
- Rješenje je uvesti stog s kojim pratimo našu trenutnu putanju
  - Kada se rekurzijom spuštamo kroz graf, vrhove stavljamo na stog
  - Kada se vraćamo iz rekurzije, vrhove uklanjamo sa stoga
  - Znamo da smo se vratili u već obiđeni vrh ako je on na stogu

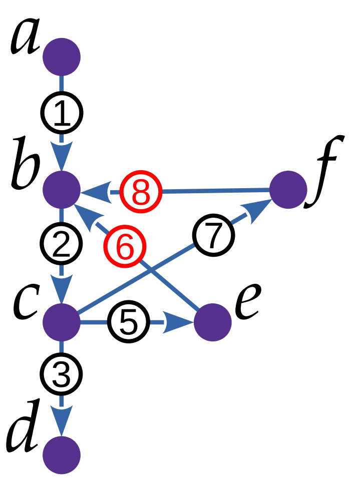
Detekcija ciklusa

```
procedure FindCycle(G)
    initialize all vertices in G as not visited
    S ← empty stack
    while there is an unvisited vertex u0 in G do
        FindCycle_recursive(G, u0, S)

procedure FindCycle_recursive(G, u, S)
    mark u as visited
    S.push(u)
    for v in adjacent vertices of u do
        if v not in S then
            set predecessor of v to u
            FindCycle_recursive(G, v, S)
        else
            if predecessor of u is not v then
                initialize cycle c ← {}
                repeat
                    retrieve vertex vx backwards from the stack S
                                                    ▷ Do not pop edges
                    add vertex vx to the cycle c
                until vx = v
                report the cycle c
    S.pop()
```

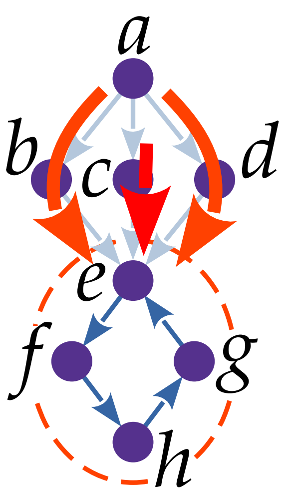
- Istovremeno koristimo dva mehanizma: označavamo vrhove obiđenim i imamo stog S s kojim pratimo trenutnu putanju
  - Označavanjem vrhova izbjegavamo da više puta obradimo istu particiju vrhova u nepovezanom grafu
- Ulaskom u rekurzivni poziv stavljamo vrh na stog S, a izlaskom ga mičemo sa stoga
  - Pazimo na to da se u neusmjerenim grafovima ne vratimo na prethodni vrh u trenutnoj putanji
  - U trenutku kada vidimo da je vrh v na stogu S, računamo ciklus – svi vrhovi na stogu unatrag sve do prethodne pojave vrha v
    - Prilikom računanja ciklusa ne mičemo vrhove sa stoga kako bismo omogućili daljnje napredovanje algoritma

Detekcija ciklusa - primjer



Iteration	1	2	3	4	5	6	7	8
vertex u	a	b	c	d	c	e	c	f
vertex v	b	c	d		e	b	f	b
stack S	a	b a	c b a	d c b a	e c b a	e c b a	f c b a	f c b a
cycle c						e,c,b		f,c,b

Detekcija ciklusa



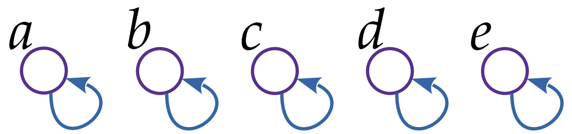
- Niti ovaj pristup nije bez problema
  - S obzirom da pratimo samo trenutnu putanju, moguće se otkrivanje jednog te istog ciklusa više puta
  - Recimo, prva putanja na izloženom primjeru je $abefhg$
    - Zadnjim bridom otkrivamo ciklus $efhge$
  - Vraćamo se iz rekurzije sve do a
  - Ponovno ulazimo u rekurziju i punimo stog, sve do $acefhg$
    - Zadnjim bridom ponovno otkrivamo ciklus $efhge$

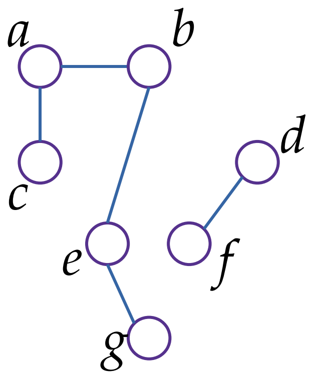
- Metoda je u biti malo modificirani DFS, čime je kompleksnost algoritma $O(V + E)$

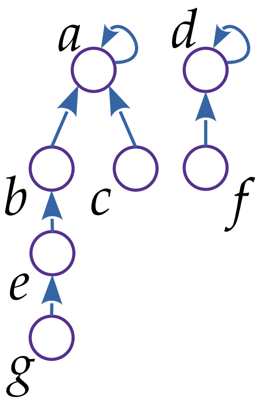
Union-Find

- Koristi bilježenje particija neusmjerenog grafa za izbjegavanje ciklusa
- Jedna od jednostavnijih metoda je korištenje grafova kao reprezentacije
- Svaki graf nazivamo stablom nepovezanog skupa (disjoint-set tree)
    - Svaki vrh stabla predstavlja vrh grafa
    - Vrh stabla usmjerenim bridom pokazuje na roditelja
    - Korijenski vrh u stablu pokazuje sam na sebe
- Skup svih stabala nepovezanih skupova naziva se šumom nepovezanih skupova (disjoint-set forest)
- Zamislimo da imamo graf $G = (V, E)$ i da ga obrađujemo brid po brid – što se često događa u algoritmima koji koriste detekciju ciklusa

Union-Find

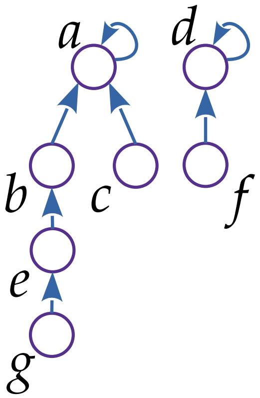


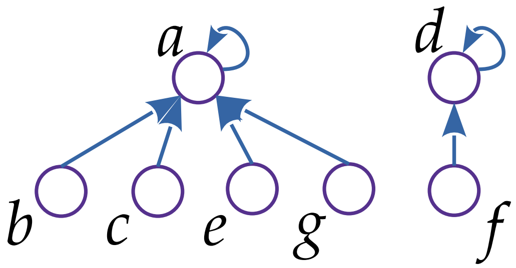




- Na početku imamo stablo nepovezanog skupa za svaki vrh grafa
  - Svako stablo očito ima jedan vrh koji je korijenski i pokazuje sam na sebe
  - Šuma nepovezanih skupova ima stabala koliko je i vrhova u grafu
- Kako dobivamo bridove grafa, imamo dvije situacije:
  1. Novi brid povezuje vrhove koji se nalaze u dva različita stabla nepovezanih skupova – znači da brid ne tvori ciklus
     - Pozivajućem algoritmu vraćamo da brid ne tvori ciklus
     - Dva stabla spajamo prema bridu koji smo dobili, tako da jedan vrh ovisi o drugome
  2. Novi brid povezuje vrhove u istom stablu nepovezanog skupa – znači da brid tvori ciklus
     - Pozivajućem algoritmu vraćamo da brid tvori ciklus

Union-Find





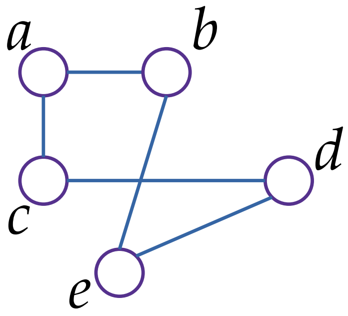
- Stabla nepovezanih skupova mogu se sažimati, kako bi se omogućilo što je kraće moguće traženje vrha po stablu
- Sažimanjem svodimo pretraživanje po stablu na O(1), a nimalo ne narušavamo funkcionalnost

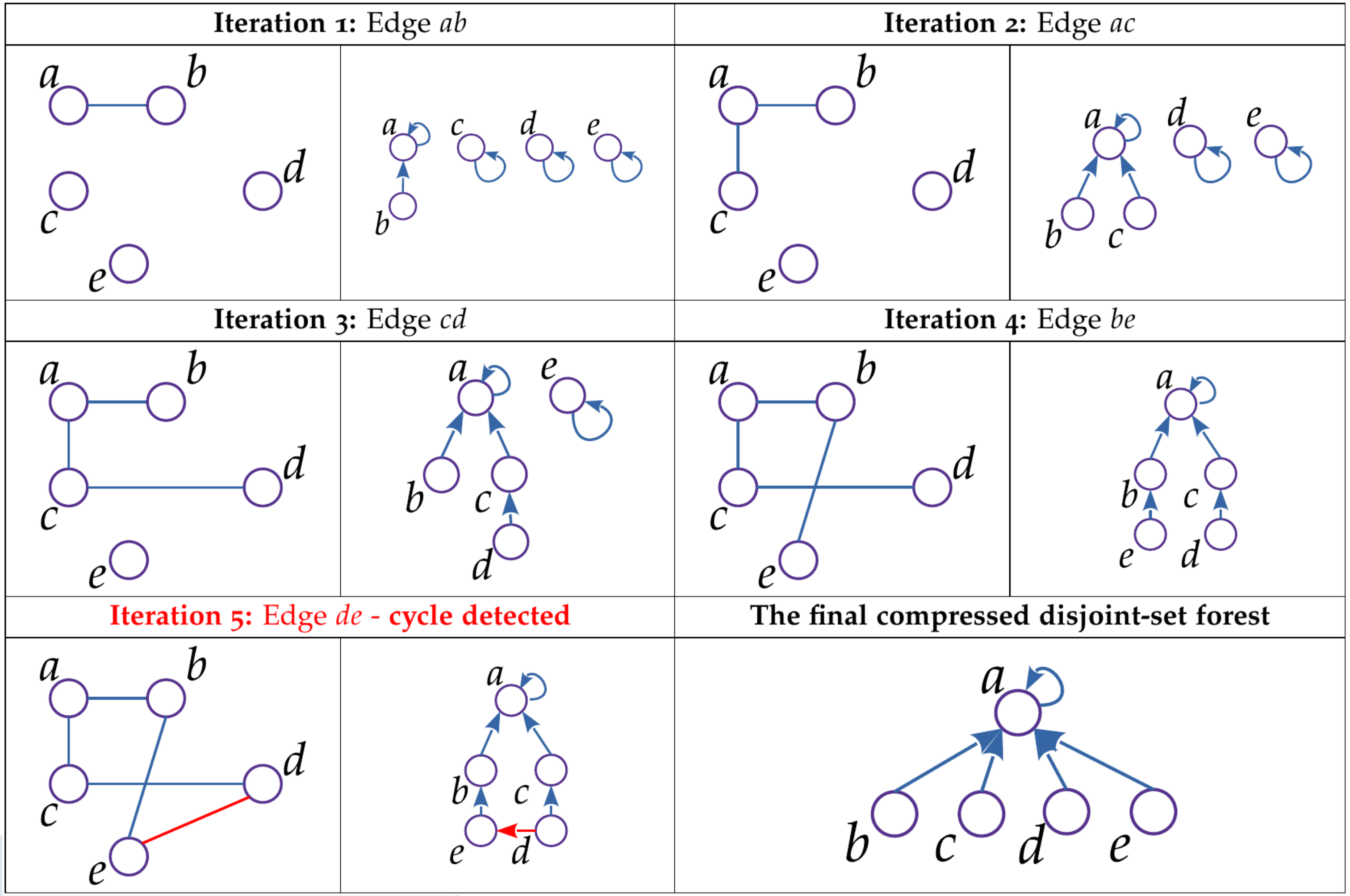
Union-Find

```
function MAKESET(F, x)
    if x not in F then
        x.parent = x
        add tree x to the forest F
    return F
function FIND(x)
    while x ≠ x.parent do
        x.parent = x.parent.parent          ▷ The compression
        x = x.parent
    return x
procedure UNION(x, y)
    x = Find(x)
    y = Find(y)
    if x ≠ y then
        x.parent = y
```

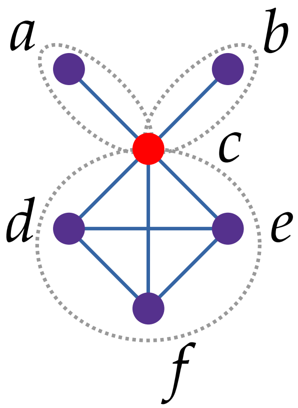
- **MakeSet** – procedura koja u šumu nepovezanih skupova F dodaje vrh x koji je sam svoj roditelj
- **Find** – procedura koja traži roditelja vrha x – u isto vrijeme radi sažimanje stabla
- **Union** – procedura koja provjerava da li su vrhovi x i y u istom stablu, pa ako nisu stavlja ih u isto stablo
- Jednostavna provjera da li su vrhovi u istom ciklusu vidi se u proceduri **Union**

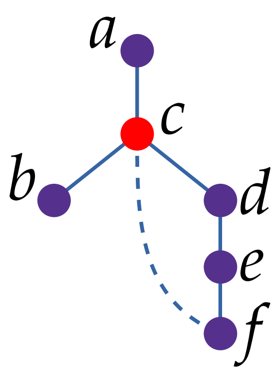
Union-Find


a
b
c
d
e


Iteration 1: Edge ab
Iteration 2: Edge ac
Iteration 3: Edge cd
Iteration 4: Edge be
Iteration 5: Edge de - cycle detected
The final compressed disjoint-set forest
a
b
c
d
e

Detekcija blokova u grafu





- Radi se u neusmjerenim grafovima
- Blok (biconnected component) je podgraf G' grafa G koji u sebi ne sadrži prijelomne točke (articulation points). To znači da ako iz bloka uklonimo bilo koji vrh, blok i dalje ostaje povezan.
- Pronalaženje prijelomnih točaka u grafu je osnova za detekciju blokova – problem je sličan detekciji ciklusa
  - Blokove dobivamo prekidanjem grafa u prijelomnoj točci
  - Na primjeru imamo blokove *ac*, *bc*, *cdef*
- Korištenjem DFS tražimo cikluse u grafu
  - Kada pronađemo ciklus, korijenski vrh predstavlja prijelomnu točku
- Na primjeru vidimo ciklus *cdefc*, gdje je *c* korijenski vrh, a time i prijelomna točka

Detekcija blokova u grafu

- Detekciju blokova radimo na razapinjujućem stablu koje potječe od proširene definicije grafa koji definiramo kao uređenu trojku

$$G = (V, E, p)$$

  - gdje je p funkcija mapiranja

$$p\colon V \to \mathbb{N}\times\mathbb{N}$$

  - koja mapira svaki vrh grafa na uređenu dvojku (n, p), gdje
    - n predstavlja broj vrha u razapinjujućem stablu
    - p predstavlja najvišeg prethodnika tog vrha – očito kandidata za prijelomnu točku
- Definiramo osnovna pravila na takav graf
  - Ne postoje dva vrha u grafu koji imaju isti broj

$$\nexists u, v \in V\colon u \neq v, n(u) = n(v)$$

Detekcija blokova u grafu

- Definiramo osnovna pravila na takav graf
  - Prethodnik vrha v definira se
    $$p(v) = \min(n(v), n(u_1), n(u_2), \dots, n(u_k))$$
    - gdje su $u_1, u_2, \dots, u_k$ vrhovi na koje se vraćaju potomci iz podstabla vrha v
    - traži se **minimalni** zajednički prethodnik cijelog podstabla vrha v
  - Drukčije definirano
    $$p(v) = \min(n(v), p(w_1), p(w_2), \dots, p(w_k))$$
    - gdje su $w_1, w_2, \dots, w_k$ potomci u podstablu vrha v
    - **minimalni** zajednički prethodnik vrha v je ili sam vrh v, ili minimalni prethodnik u podstablu vrha

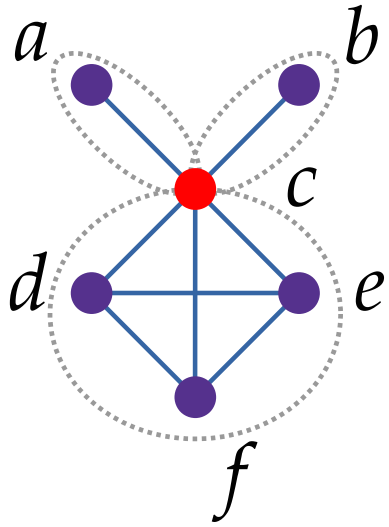
Detekcija blokova u grafu

```
procedure BlockSearch(G)
    initialize all vertices in G as n(v) = 0
    step ← 1
    S ← empty stack
    while there is a vertex u in G, having n(u) = 0 do
        BlockSearch_recursive(G, u, step, S)

procedure BlockSearch_recursive(G, u, step, S)
    p(u) = n(u) = step
    step ← step + 1
    for v in adjacent vertices of u do
        if n(v) = 0 then
            if not edge uv on the stack S then
                S.push(uv)
            BlockSearch_recursive(G, v, step, S)
            if p(v) ≥ n(u) then
                pop edges until uv and form a block
            else
                p(u) = min(p(u), p(v))
        else if S is not empty and vu is not the last element then
            p(u) = min(p(u), n(v))
```

- Svakom vrhu inicijaliziramo jedinstveni $p(u) = n(u)$
- Krećemo se prema susjednim vrhovima od u
  - Stavljamo brid uv na stog S
  - Ako je susjedni vrh v novi, to jest $n(v) = 0$, tada rekurzivno zovemo njegovu obradu
    - Pri povratku iz rekurzije provjeravamo da li je njegov prethodnik u ili neki potomaka od u
    - Ako je, formiramo blok sa stoga S na koji smo stavljali bridove
    - Ako nije, ažuriramo minimalnog prethodnika
  - Ako smo susjedni vrh v već obišli, moguće je da se radi o povratnom bridu
    - Ažuriramo minimalnog prethodnika vrha u, a koji može biti vrh v

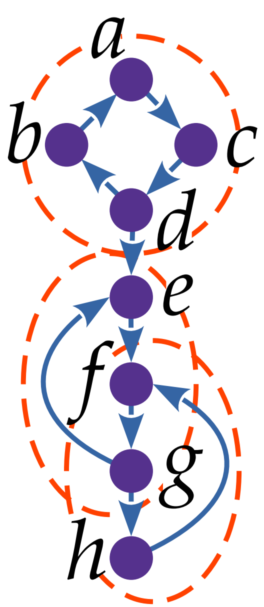
Detekcija blokova u grafu



	0	1	2	3	4	5	6	7	8	9	10	11	12
u		a	c	b	c	c	d	e	f	e	d	c	a
v		c	b		b	d	e	f	c	f	e	d	c
	n/p	n/p	n/p	n/p	n/p	n/p	n/p	n/p	n/p	n/p	n/p	n/p	n/p
a	$0/0$	$1/1$											pop
b	$0/0$			$3/3$									
c	$0/0$		$2/2$		pop							pop	
d	$0/0$						$4/4$				$4/2$		
e	$0/0$							$5/5$		$5/2$			
f	$0/0$								$6/6$ $6/2$				
stack S		ac	cb ac	cb ac	ac	cd ac	de cd ac	ef de cd ac	ef de cd ac	ef de cd ac	ef de cd ac	ac	

$$b_1 = G[V = \{a, c\}], b_2 = G[V = \{b, c\}], b_3 = G[V = \{c, d, e, f\}]$$

Detekcija komponenti u usmjerenom grafu



- Definicija čvrsto povezane komponente (SCC) podrazumijeva da su svi parovi vrhova u toj komponenti međusobno dohvatljivi
- Naivni pokušaj rješavanja ovog problema je NP-kompleksan jer bismo trebali stvarati particije vrhova u kojima su svi vrhovi međusobno dohvatljivi
- Problem se može pojednostavniti znajući da ciklusi tvore takve komponente
  - Na primjeru imamo tri ciklusa: $c1 = acdba$, $c2 = efge$ i $c3 = fghf$
  - Dva ciklusa dijele zajedničke vrhove i bridove, čime dobivamo dvije čvrsto povezane komponente: $c1$ i $c2 \cup c3$
- Blokovi u neusmjerenim grafovima dijele prijelomnu točku, dok čvrsto povezane komponente u usmjerenim grafovima ne mogu dijeliti vrhove

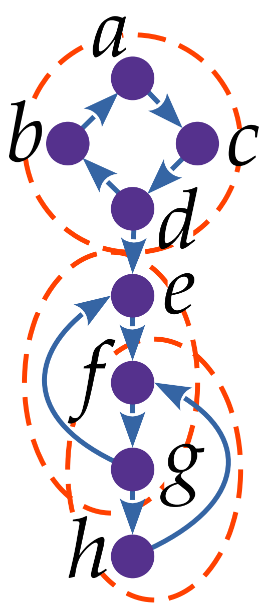
Tarjanov algoritam (SCC)

procedure SCCSEARCH(G)
    initialize all vertices in G as $n(v) = 0$
    $step \leftarrow 1$
    $S \leftarrow$ empty stack
    while there is a vertex u in G, having $n(u) = 0$ **do**
        SCCSearch_recursive($G, u, step, S$)

procedure SCCSEARCH_RECURSIVE($G, u, step, S$)
    $p(u) = n(u) = step$
    $step \leftarrow step + 1$
    $S.push(u)$
    for v in adjacent vertices of u **do**
        if $n(v) = 0$ **then**
            $SCCSearch\_recursive(G, v, step, S)$
            $p(u) = min(p(u), p(v))$
        else if $n(v) < n(u)$ and v is on the stack S **then**
            $p(u) = min(p(u), n(v))$
    if $p(u) = n(u)$ **then** ▷ this is the SCC root vertex
        pop vertices from the stack S until u is popped off

- Svakom vrhu inicijaliziramo jedinstveni $p(u) = n(u)$
- Krećemo se prema susjednim vrhovima od u
  - Stavljamo vrh u na stog S
  - Ako je susjedni vrh v novi, to jest $n(v) = 0$, tada rekurzivno zovemo njegovu obradu
    - Pri povratku iz rekurzije ažuriramo minimalnog prethodnika vrha u
  - Ako smo susjedni vrh v već obišli, moguće je da se radi o povratnom bridu
    - Ažuriramo minimalnog prethodnika vrha u, a koji može biti vrh v
  - Kada se vraćamo natrag pozivatelju, testiramo prethodnika vrha u. Ako vrh u nema prethodnika tako da je $p(u) < n(u)$, tada vrh u smatramo korijenom komponente
    - Ovaj je korak malo drukčiji od algoritma za detekciju blokova

Tarjanov algoritam (SCC)



	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
u		a	c	d	b	d	d	e	f	g	g	h	g	f	e	d	c	a
v		c	d	b	a	b	e	f	g	e	h	f	h	g	f	e	d	c
	n/p	n/p	n/p	n/p	n/p	n/p	n/p	n/p	n/p	n/p	n/p	n/p	n/p	n/p	n/p	n/p	n/p	n/p
a	0/0	1/1																pop
b	0/0				4/4 4/1													
c	0/0		2/2														2/1	
d	0/0			3/3		3/1												
e	0/0							5/5							pop			
f	0/0								6/6					6/5				
g	0/0									7/7 7/5								
h	0/0											8/8 8/6						
stack S		a	c a	d c a	b d c a	b d c a	b d c a	e b d c a	f e b d c a	g f e b d c a	g f e b d c a	h g f e b d c a	h g f e b d c a	h g f e b d c a	h g f e b d c a	b d c a	b d c a	b d c a

$scc_1 = G[V = \{h, g, f, e\}]$

$scc_2 = G[V = \{b, d, c, a\}]$

Minimalno razapinjujuće stablo (MST)

- Prethodno smo definirali razapinjujuće stablo kao produkt obilaska grafa raznim algoritmima, poput BFS i DFS
  - Zbog tog raznolikog pristupa, graf može imati više različitih razapinjujućih stabala
  - Za težinski graf $G = (V, E, w)$ tako možemo definirati skup svih razapinjujućih stabala

$$ST(G) = \{G_i' = (V, E_i, w) : E_i \subseteq E(G)\}$$

  - Pronalaženje minimalnog razapinjujućeg stabla rezultat je optimizacije

$$MST(G) = \arg min_{G_i' \in ST(G)} \sum_{e \in E_i(G_i')} w(e)$$

  - Tražimo razapinjujuće stablo čija je suma težina bridova minimalna u skupu svih razapinjujućih stabala
- Svi algoritmi koji se koriste u ovu svrhu spadaju i u pohlepne algoritme

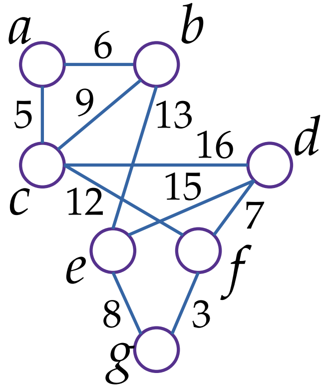
Kruskalov algoritam

```
procedure KRUSKAL(G)
    MST ← (V = V(G), E = ∅)
    sort edges E(G) ascending by weights
    for e_i ∈ E(G) do
        if |E(MST)| < |V(G)| − 1 then
            if no cycle in G' = (V(MST), E(MST) ∪ {e_i}) then
                add e_i to E(MST)
    return MST
```

- Kompleksnost *Union-Find* metode je $O(1)$
- Zbog sortiranja, kompleksnost Kruskalovog algoritma je $O(V^2 \log_2 V)$

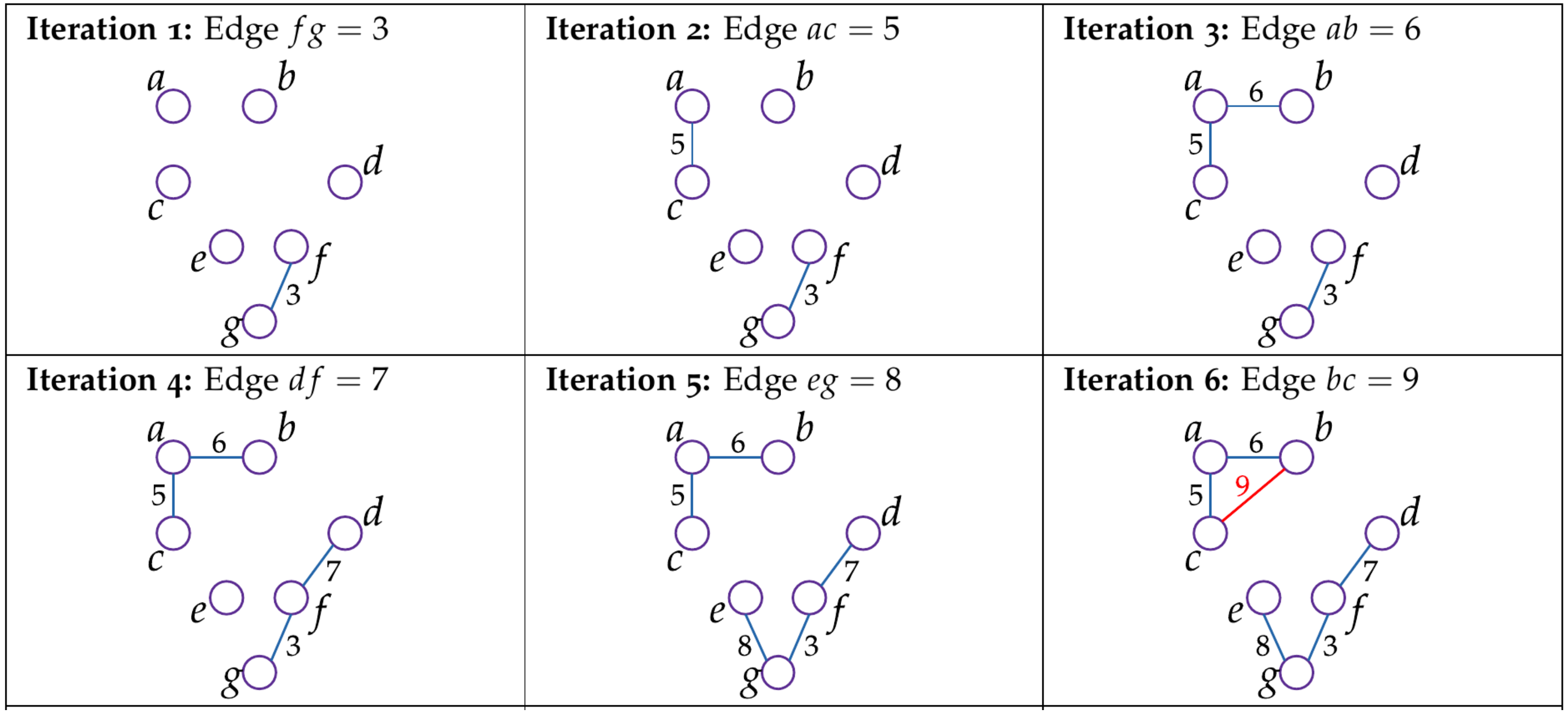
- Isključivo za neusmjerene grafove!
- Inicijaliziramo min. razapinjujuće stablo tako da preselimo sve vrhove i ostavimo skup bridova praznim
- Sortiramo bridove ulaznog grafa uzlazno po težinama – visoka kompleksnost!
- Prolazimo po bridovima tako dugo dok u min. razapinjujućem stablu imamo manje bridova od $|V(G)| - 1$
    - Ako brid koji smo uzeli iz sortiranog skupa bridova ne tvori ciklus u min. razapinjujućem stablu, dodajemo ga u bridove min. razapinjujućeg stabla
- Za detekciju ciklusa možemo koristiti *UnionFind* metodu.

Kruskalov algoritam

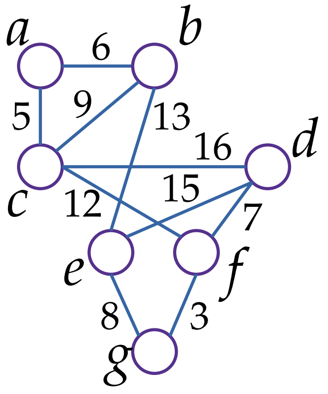


$$
\mathbf{G} = \begin{array}{c} \\ a \\ b \\ c \\ d \\ e \\ f \\ g \end{array}
\begin{array}{c}
\begin{array}{ccccccc} a & b & c & d & e & f & g \end{array} \\
\begin{bmatrix}
0 & 6 & 5 & 0 & 0 & 0 & 0 \\
6 & 0 & 9 & 0 & 13 & 0 & 0 \\
5 & 9 & 0 & 16 & 0 & 12 & 0 \\
0 & 0 & 16 & 0 & 15 & 7 & 0 \\
0 & 13 & 0 & 15 & 0 & 0 & 8 \\
0 & 0 & 12 & 7 & 0 & 0 & 3 \\
0 & 0 & 0 & 0 & 8 & 3 & 0
\end{bmatrix}
\end{array}
$$

$fg = 3, ac = 5, ab = 6, df = 7,$
$eg = 8, bc = 9, cf = 12,$
$be = 13, de = 15, cd = 16$



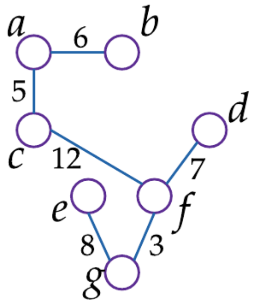
Kruskalov algoritam



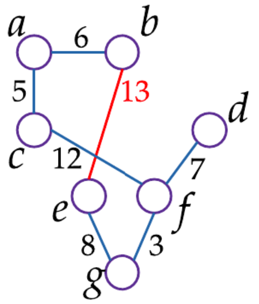
$$\mathbf{G} = \begin{array}{c} \\ a \\ b \\ c \\ d \\ e \\ f \\ g \end{array} \begin{array}{c} \begin{array}{ccccccc} a & b & c & d & e & f & g \end{array} \\ \begin{bmatrix} 0 & 6 & 5 & 0 & 0 & 0 & 0 \\ 6 & 0 & 9 & 0 & 13 & 0 & 0 \\ 5 & 9 & 0 & 16 & 0 & 12 & 0 \\ 0 & 0 & 16 & 0 & 15 & 7 & 0 \\ 0 & 13 & 0 & 15 & 0 & 0 & 8 \\ 0 & 0 & 12 & 7 & 0 & 0 & 3 \\ 0 & 0 & 0 & 0 & 8 & 3 & 0 \end{bmatrix} \end{array}$$

$fg = 3, ac = 5, ab = 6, df = 7,$
$eg = 8, bc = 9, cf = 12,$
$be = 13, de = 15, cd = 16$

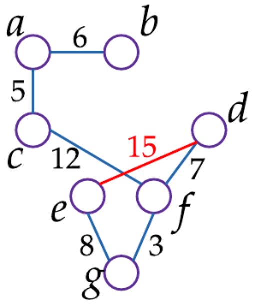
Iteration 7: Edge $cf = 12$



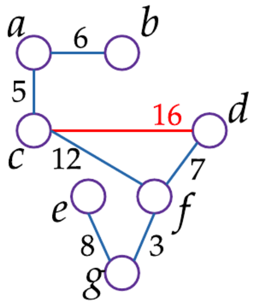
Iteration 8: Edge $be = 13$



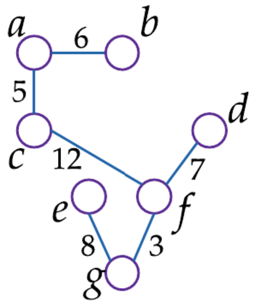
Iteration 9: Edge $de = 15$



Iteration 10: Edge $cd = 16$



Final MST



Final MST adjacency matrix

$$\begin{array}{c} \\ a \\ b \\ c \\ d \\ e \\ f \\ g \end{array} \begin{array}{c} \begin{array}{ccccccc} a & b & c & d & e & f & g \end{array} \\ \begin{bmatrix} 0 & 6 & 5 & 0 & 0 & 0 & 0 \\ 6 & 0 & 0 & 0 & 0 & 0 & 0 \\ 5 & 0 & 0 & 0 & 0 & 12 & 0 \\ 0 & 0 & 0 & 0 & 0 & 7 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 8 \\ 0 & 0 & 12 & 7 & 0 & 0 & 3 \\ 0 & 0 & 0 & 0 & 8 & 3 & 0 \end{bmatrix} \end{array}$$

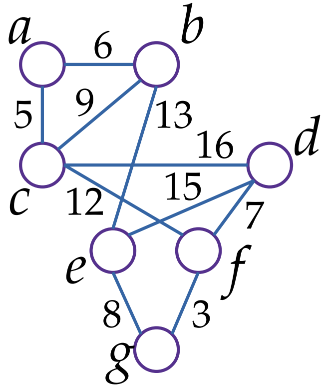
Dijkstrin algoritam (MST)

```
procedure DijkstraMST(G)
    MST ← (V = V(G), E = ∅)
    for e_i ∈ E(G) do
        add e_i to E(MST)
        if there is a cycle in MST then
            remove the maximal weight edge from the cycle
    return MST
```

- Kompleksnost osnovne iteracije je $O(E)$. DFS algoritam je $O(V+E)$, što postaje $O(2V)$, to jest $O(V)$ za min. razapinjujuće stablo.
- U konačnici je kompleksnost Dijsktrinog algoritma $O(E*V)$.

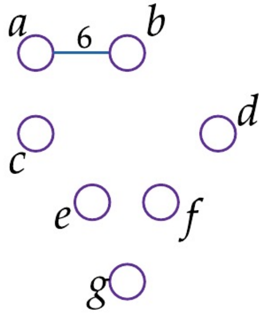
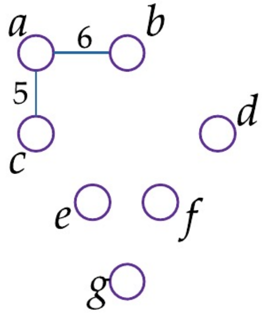
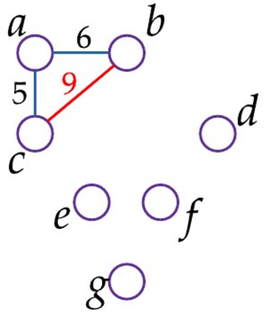
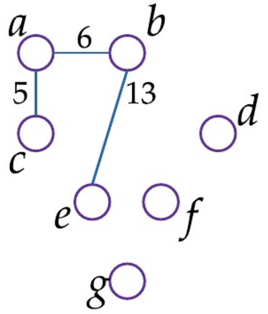
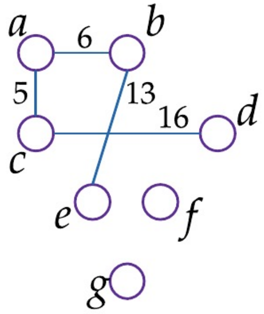
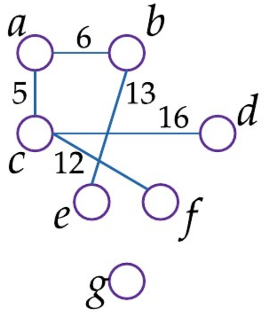
- Nedostatak Kruskalovog algoritma je visoke kompleksnost zbog potrebe za sortiranjem bridova
- Dijkstrina metoda je nešto drukčija
  - Nema sortiranja
  - U trenutku kada detektiramo ciklus, iz ciklusa uklanjamo brid najveće težine
- Ovdje nam *UnionFind* metoda ne funkcionira, pa se moramo poslužiti metodom koja koristi prošireni DFS (prikazano pred nekoliko prikaznica)
  - Nakon što dodamo brid uv, uzmemo jedan od ta dva vrha kao početni vrh
  - Ako se vratimo u taj početni vrh, tada imamo ciklus
  - U stogu možemo spremati i težine bridova – radi detekcije brida koji ima najveći težinu

Dijkstrin algoritam (MST)

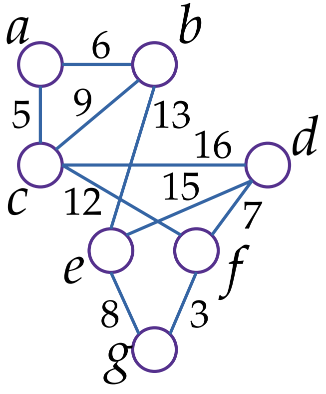


$$
\mathbf{G} = \begin{array}{c} \\ a \\ b \\ c \\ d \\ e \\ f \\ g \end{array}
\begin{array}{c}
\begin{array}{ccccccc} a & b & c & d & e & f & g \end{array} \\
\begin{bmatrix}
0 & 6 & 5 & 0 & 0 & 0 & 0 \\
6 & 0 & 9 & 0 & 13 & 0 & 0 \\
5 & 9 & 0 & 16 & 0 & 12 & 0 \\
0 & 0 & 16 & 0 & 15 & 7 & 0 \\
0 & 13 & 0 & 15 & 0 & 0 & 8 \\
0 & 0 & 12 & 7 & 0 & 0 & 3 \\
0 & 0 & 0 & 0 & 8 & 3 & 0
\end{bmatrix}
\end{array}
$$

$ab, ac, bc, be, cd, cf, de, df, eg, fg$

Iteration 1: Edge $ab = 6$ 	**Iteration 2:** Edge $ac = 5$ 	**Iteration 3:** Edge $bc = 9$, Cycle, Remove bc 
Iteration 4: Edge $be = 13$ 	**Iteration 5:** Edge $cd = 16$ 	**Iteration 6:** Edge $cf = 12$ 

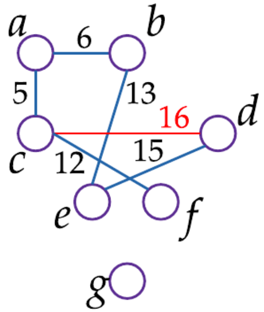
Dijkstrin algoritam (MST)



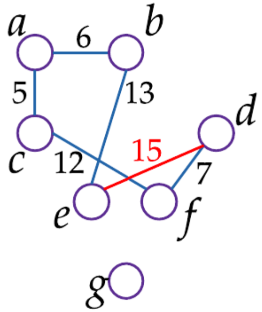
$$
\mathbf{G} = \begin{array}{c} \\ a \\ b \\ c \\ d \\ e \\ f \\ g \end{array}
\begin{array}{c}
\begin{array}{ccccccc} a & b & c & d & e & f & g \end{array} \\
\begin{bmatrix}
0 & 6 & 5 & 0 & 0 & 0 & 0 \\
6 & 0 & 9 & 0 & 13 & 0 & 0 \\
5 & 9 & 0 & 16 & 0 & 12 & 0 \\
0 & 0 & 16 & 0 & 15 & 7 & 0 \\
0 & 13 & 0 & 15 & 0 & 0 & 8 \\
0 & 0 & 12 & 7 & 0 & 0 & 3 \\
0 & 0 & 0 & 0 & 8 & 3 & 0
\end{bmatrix}
\end{array}
$$

$ab, ac, bc, be, cd, cf, de, df, eg, fg$

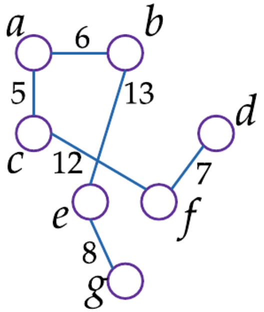
Iteration 7: Edge $de = 15$, Cycle, Remove cd



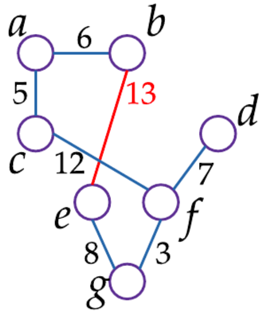
Iteration 8: Edge $df = 7$, Cycle, Remove de



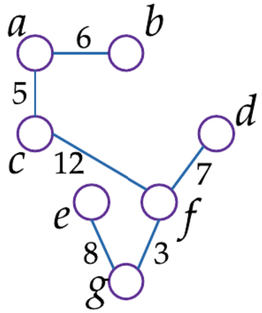
Iteration 9: Edge $eg = 8$



Iteration 10: Edge $fg = 3$, Cycle, Remove be



Final MST



Final MST adjacency matrix

$$
\begin{array}{c} \\ a \\ b \\ c \\ d \\ e \\ f \\ g \end{array}
\begin{array}{c}
\begin{array}{ccccccc} a & b & c & d & e & f & g \end{array} \\
\begin{bmatrix}
0 & 6 & 5 & 0 & 0 & 0 & 0 \\
6 & 0 & 0 & 0 & 0 & 0 & 0 \\
5 & 0 & 0 & 0 & 0 & 12 & 0 \\
0 & 0 & 0 & 0 & 0 & 7 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 8 \\
0 & 0 & 12 & 7 & 0 & 0 & 3 \\
0 & 0 & 0 & 0 & 8 & 3 & 0
\end{bmatrix}
\end{array}
$$

Primov algoritam

- Zamislimo da smo graf G razgradili na skup elementarnih razapinjujućih stabala ST(G), tako da svaki vrh čini jedno stablo

$$ST(G) = \{ST_i: 1 \leq i \leq n, ST_i = (V = \{v_i\}, E = \emptyset), v_i \in V(G)\}$$

  - dodavanjem brida između dva elementarna razapinjujuća stabla dobivamo novo jedinstveno razapinjujuće stablo

$$ST_i \cup ST_j = (V(ST_i) \cup V(ST_j), E(ST_i) \cup E(ST_j) \cup \{e_n\})$$

- Osnovna ideja Primovog algoritma je **rastuće** minimalno razapinjujuće stablo $ST_g \in ST(G)$ i ostatak elementarnih razapinjujućih stabala $ST_r = ST(G) \setminus \{ST_g\}$

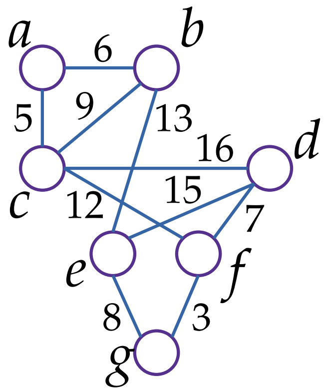
Primov algoritam

```
procedure PRIM(G, v_s)
    MST ← (V_g = {v_s}, E = ∅)
    V_r ← V(G) \ {v_s}
    while V_r ≠ ∅ do
        choose minimal weighted edge e_n = uv ∈ E(G) such that
                    u ∈ V(MST) and v ∈ V_r
        add v to V_g(MST)
        remove v from V_r
        add e_n = uv to E(MST)
    return MST
```

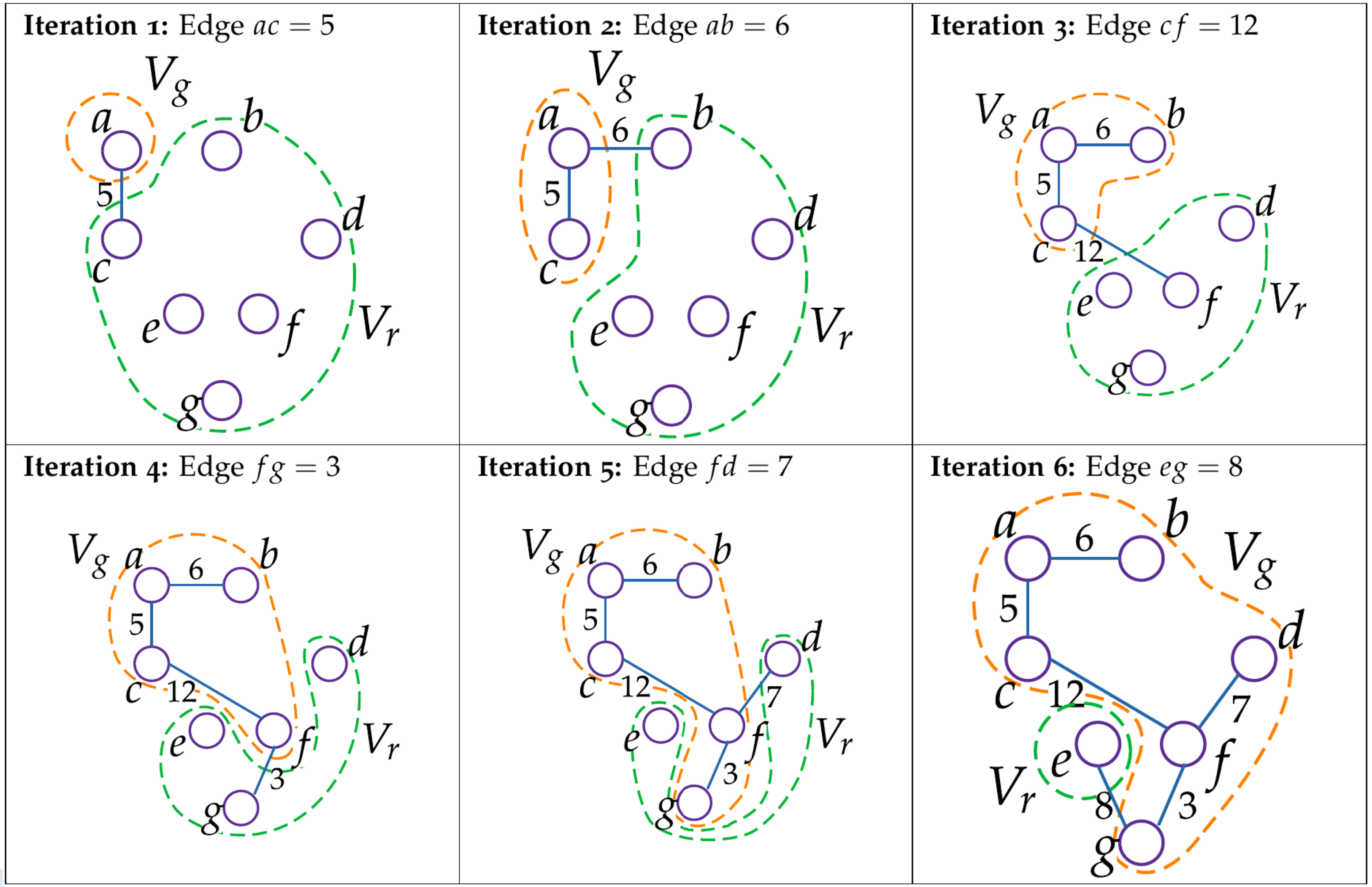
- Kompleksnost osnovne iteracije je $O(V)$.
- Problem predstavlja pretraživanje brida minimalne težine.
  - U sekvencijalnoj implementaciji ukupna kompleksnost je $O(V^2)$.
  - Ako koristimo gomilu, tada je ukupna kompleksnost $O(E + V\log_2 V)$.

- U inicijalno min. razapinjujuće stablo dodajemo samo početni vrh v_s, bez bridova
- Definiramo skup V_r koji inicijalno sadržava sve vrhove osim početnog vrha v_s
- Uzimamo prvi vrh v iz skupa V_r. To radimo sve do dok imamo vrhova u tom skupu
  - Pronađemo brid $e_n = vu$ minimalne težine između vrha v i nekog od vrhova u rastućem razapinjujućem stablu V_g
  - Dodamo vrh v i brid e_n u rastuće razapinjujuće stablo V_g
  - Vrh v uklanjamo iz skupa vrhova V_r

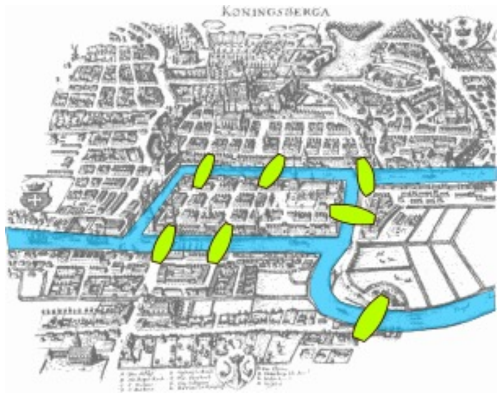
Primov algoritam



$$
\mathbf{G} = \begin{array}{c} \\ a \\ b \\ c \\ d \\ e \\ f \\ g \end{array}
\begin{array}{c}
\begin{array}{ccccccc} a & b & c & d & e & f & g \end{array} \\
\begin{bmatrix}
0 & 6 & 5 & 0 & 0 & 0 & 0 \\
6 & 0 & 9 & 0 & 13 & 0 & 0 \\
5 & 9 & 0 & 16 & 0 & 12 & 0 \\
0 & 0 & 16 & 0 & 15 & 7 & 0 \\
0 & 13 & 0 & 15 & 0 & 0 & 8 \\
0 & 0 & 12 & 7 & 0 & 0 & 3 \\
0 & 0 & 0 & 0 & 8 & 3 & 0
\end{bmatrix}
\end{array}
$$

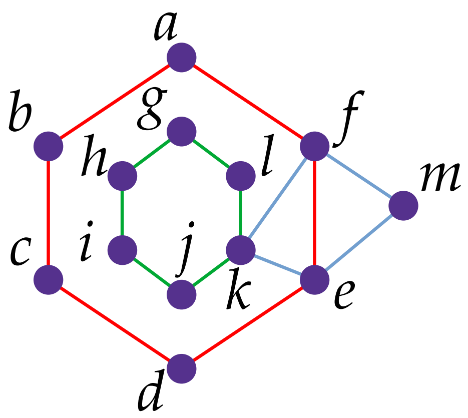


Eulerovi grafovi



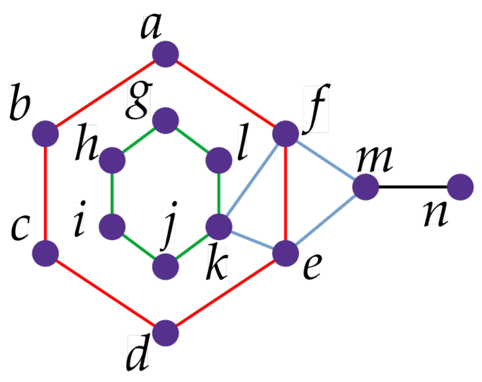
- Leibniz prvi predlaže granu *pozicijske geometrije*
- Euler prvi postulira *pozicijsku geometriju* na problemu sedam mostova Königsberga (danas Kaliningrad)
- Problem je bio pronaći rutu kroz Königsberg tako da se svaki od sedam mostova prijeđe samo jednom
  - Euler je prvo prozreo problem jer je smatrao da je trivijalan i nije htio trošiti vrijeme ne to
  - Kasnije je iz njegovih promatranja na tom problemu nastala teorija grafova
  - Euler je dokazao da je problem nerješiv, ali je ponudio i klasu problema koji jesu rješivi
- Kasnije se na originalnom problemu sedam mostova Königsberga izvode neki generički problemi, kao problem kineskog poštara

Eulerovi grafovi



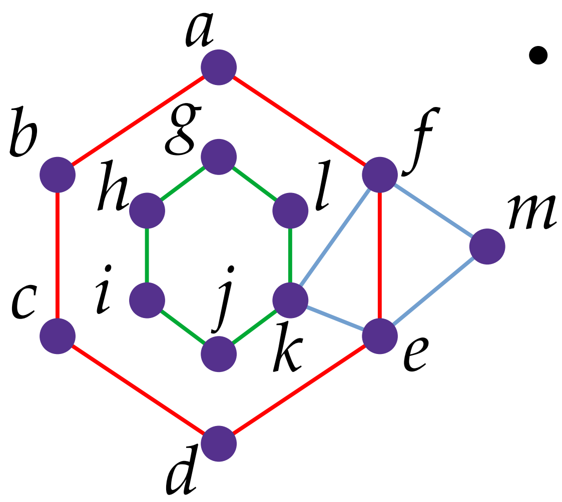
- **Eulerova putanja** (*trail*) – je putanja kroz graf koja svakim bridom grafa prolazi samo jednom
- **Eulerov krug** (*circuit*) – je Eulerova putanja koja počinje i završava u istom vrhu
- **Eulerov graf** – je graf koji je sačinjen od Eulerovog kruga ili putanje
- **Teorem** – Spojeni neusmjereni graf koji ima sve vrhove parnog stupnja je Eulerov graf
  - **Dokaz**
    - Zamislimo da radimo obilazak grafa. Za svaki ulazak u vrh koristimo jedan priležeći brid, dok za izlazak iz vrha koristimo drugi priležeći brid.
    - Svaki vrh možemo obići više od jednom, što znači da svaki vrh Eulerovog grafa mora imati *broj obilazaka* * 2 prileżećih bridova
    - <u>Time je Eulerov graf sačinjen od Eulerovog kruga</u>

Eulerovi grafovi



- Eulerov graf može biti sačinjen i od Eulerove putanje koja ne mora početi i završiti u istom vrhu !?
- Za razliku od Eulerovog grafa koji je sačinjen od Eulerovog kruga, u ovom slučaju točno dva vrha u grafu smiju biti neparnog stupnja
  - Bez obzira na to, postoji obilazak koji prolazi svakim bridom grafa točno jednom

Eulerovi krugovi



- Možemo uočiti da se Eulerov graf u primjeru sastoji od nekoliko ne-Eulerovih krugova

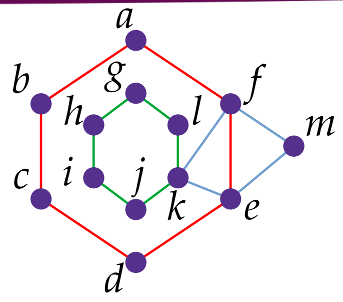
$$C_1 = abcdefa, C_2 = klghijk, C_3 = emfke$$

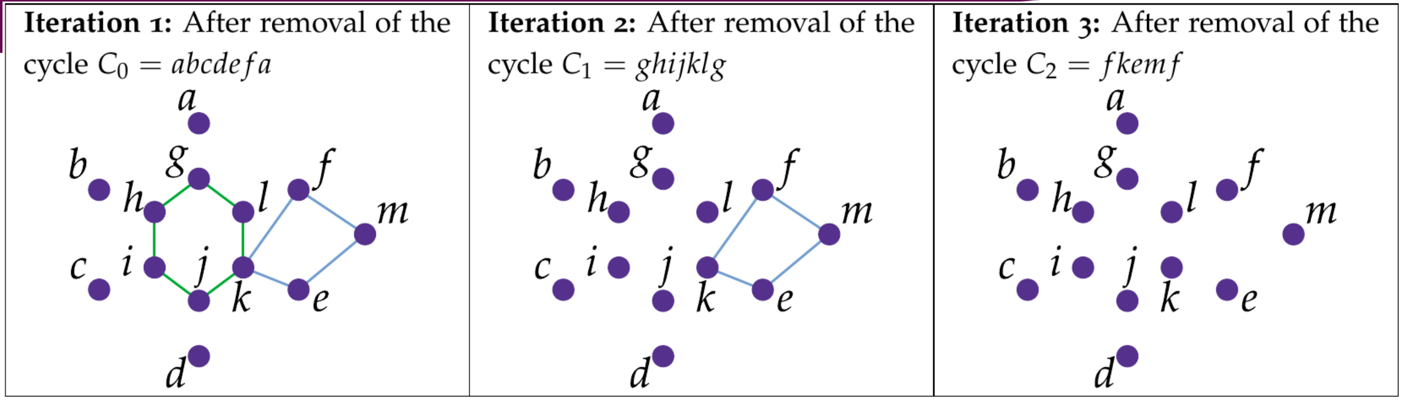
  - Ti krugovi dijele samo zajedničke vrhove, te se mogu povezati kroz te zajedničke vrhove
  - Tako je Eulerov krug onda

$$C = C_1 \cup C_2 \cup C_3$$
$$C = abcdemfklghijkefa$$

- Ideja detekcije Eulerovih krugova temelji se na obilasku grafa s uklanjanjem bridova koji su se obišli

Eulerovi krugovi





- Obilazimo graf i uklanjamo vrhove. U trenutku kada smo se našli u početnom vrhu, imamo ili Eulerov krug ili jedan od ne-Eulerovih krugova
  - S obzirom da je Eulerov graf povezan, ne-Eulerovi krugovi dijele vrhove
  - Dijeljeni vrhovi moraju imati paran stupanj i nakon uklanjanja ne-Eulerovog kruga
  - Nastavljamo obilazak grafa sve do dok možemo detektirati još ne-Eulerovih krugova u grafu i do dok nismo uklonili sve bridove grafa
- Ako nam ostanu vrhovi koji ne čine ne-Eulerov krug, a imamo još preostalih bridova u grafu – tada očito neki od vrhova nisu imali paran stupanj i nije moguće pronaći Eulerov krug u grafu

Eulerovi krugovi

```
procedure IsEulerCircuit(G, u)
    while true do
        u0 ← u
        while there is an edge uv in G do
            add uv to the circuit
            remove uv from the graph G
            u ← v
        if u ≠ u0 then
            return false
        if there are edges in G then
            pick a vertex u that has incident edges
        else
            return true
```

- Kompleksnost ovog algoritma je $O(V+E)$
- Obilazak grafa nam prolazi kroz sve bridove, dok traženje novog vrha koji još ima brid zahtijeva prolazak kroz sve vrhove

- Započinjemo nekim proizvoljnim početnim vrhom $u = u_0$
  - Tako dugo dok imamo brid uv koji vodi iz vrha u stvaramo krug
    - Ukloni brid uv
    - Vrh v prebacujemo u vrh u
  - Ako više nemamo bridova koji vode iz vrha u i ako u nije početni vrh u_0, tada ovaj graf **NIJE** sačinjen od Eulerovog kruga
    - U ovom trenutku znamo da imamo ili Eulerov krug ili jedan od ne-Eulerovih krugova
- Ako još postoji bridova, tada odabiremo novi početni vrh u_0 i ponovno se vraćamo na detekciju kruga
- Ako više ne postoji bridova u grafu, tada imamo Eulerov graf sačinjen od Eulerovog kruga!

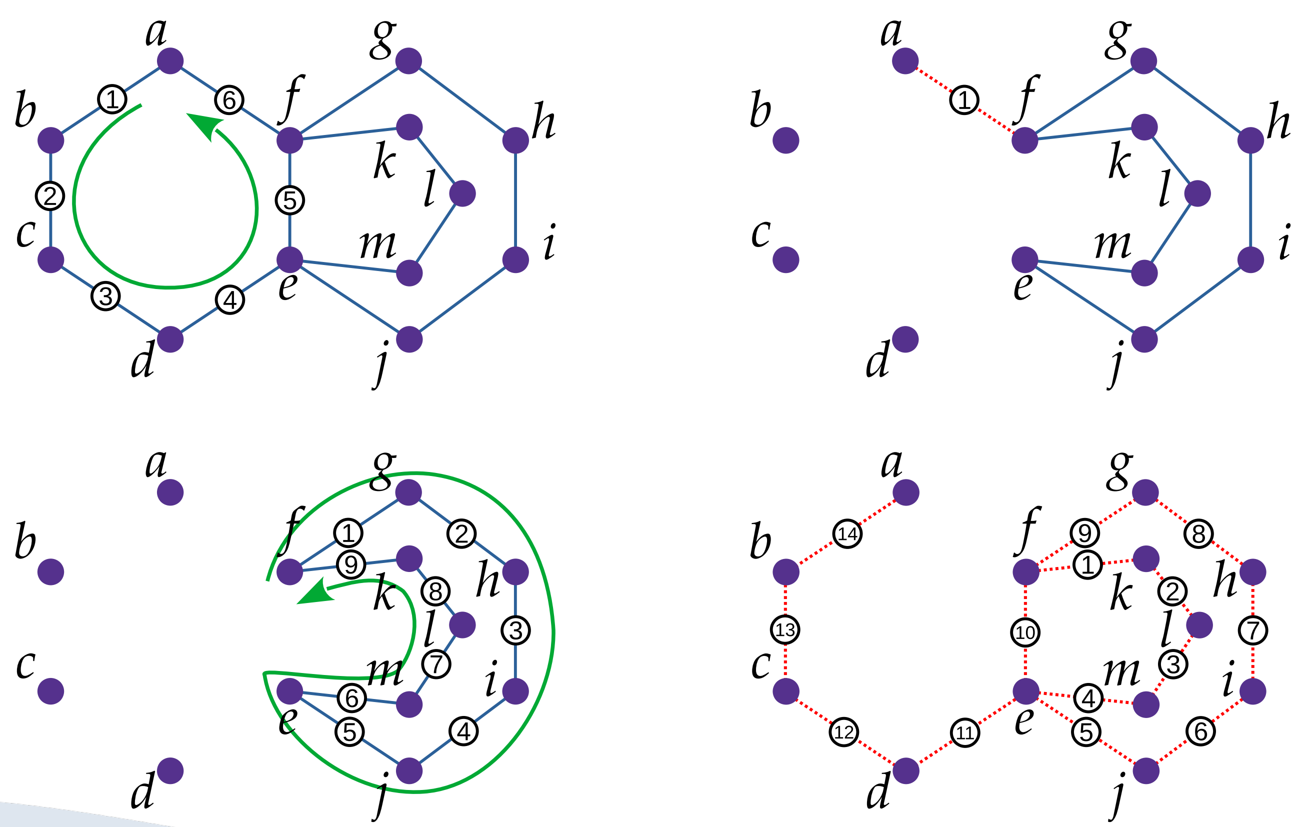
Hierholzerov algoritam

```
procedure HIERHOLZER(G, u)
    s ← new empty stack
    cycle ← ∅
    s.push(u)
    while s not empty do
        u ← last element on the stack s
        if u has adjacent vertices then
            v ← one of the adjacent vertices of u
            s.push(v)
            remove edge uv from G
        else
            e ← edge uv from last two vertices on the stack
            add e to cycle
            s.pop()
    return cycle
```

- Kompleksnost ovog algoritma je $O(E)$

- Koristeći činjenicu da je Eulerov graf povezan možemo upotrijebiti stog za izbjegavanje slijepog pretraživanja svih vrhova u potrazi za novim bridom
  - Kako napredujemo po krugu tako vrhove stavljamo na stog
  - Kada dođemo do kraja kruga, tako se vraćamo po obiđenih vrhovima uzimajući ih sa stoga
    - Ako je prethodni krug bio Eulerov, tada se vratimo natrag na početni vrh
    - Ako prethodni krug nije bio Eulerov, a s obzirom da je Eulerov graf povezan, na putu natrag naići ćemo na barem jedan vrh koji još ima bridova
      - Krećemo tim bridom, napredujući kroz novi ne-Eulerov krug.

Hierholzerov algoritam



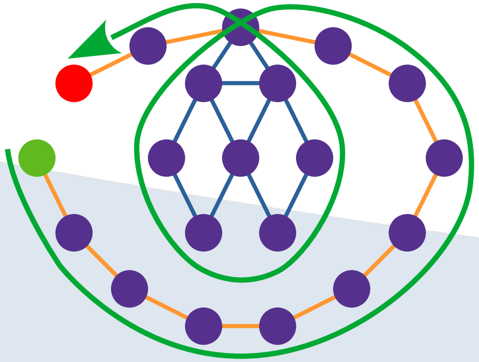
- Krajnji Eulerov krug je

  fa, kf, lk, ml, em, je, il, hi, gh, fg, ef, de, cd, bc, ab

Fleuryev algoritam

```
function FLEURY(G, u)
    cycle ← ∅
    while there are edges in G do
        pick an edge uv from G prioritizing
            non-bridge edges over bridge edges
        add uv to the cycle
        remove uv from the graph G
        u ← v
    return cycle
```

- Kompleksnost ovog algoritma je $O(E)$
- No, detekcija da li je brid most ili ne, diže kompleksnost algoritma na $O(E^2)$, što je sporije od Hierholzerovog algoritma



- Koncept Fleuryevog algoritma temelji se na odabiru sljedećeg brida kojim se krećemo po Eulerovom krugu
  - Ako iz trenutnog vrha vodi brid koji nije most (*bridge*), tada odabiremo njega
  - Tek kada iz vrha vodi samo vrh koji je most, krećemo se njime
  - Sjetimo se – most je brid čijim uklanjanjem činimo graf nepovezanim
    - To se može desiti kada se krećemo Eulerovim grafom
    - Ako su nam ostali bridovi u obje particije grafa, ne možemo se vratiti natrag
    - Na taj način nećemo detektirati Eulerov krug
- Sve se temelji na poučku (*lemma*), kojom utvrđujemo da uklanjanjem prvog brida na početnom vrhu, stupanj tog vrha postaje neparan i samim time to postaje i posljednji vrh kojeg ćemo obići