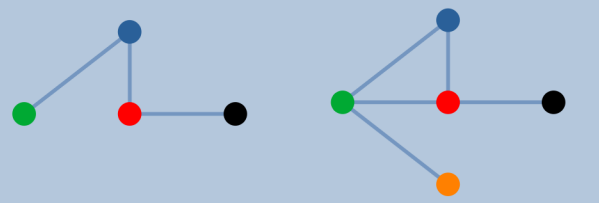
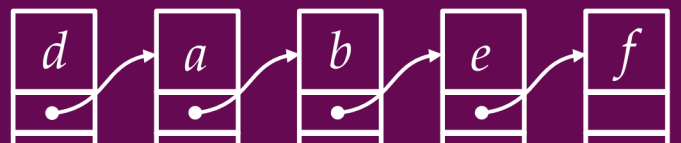
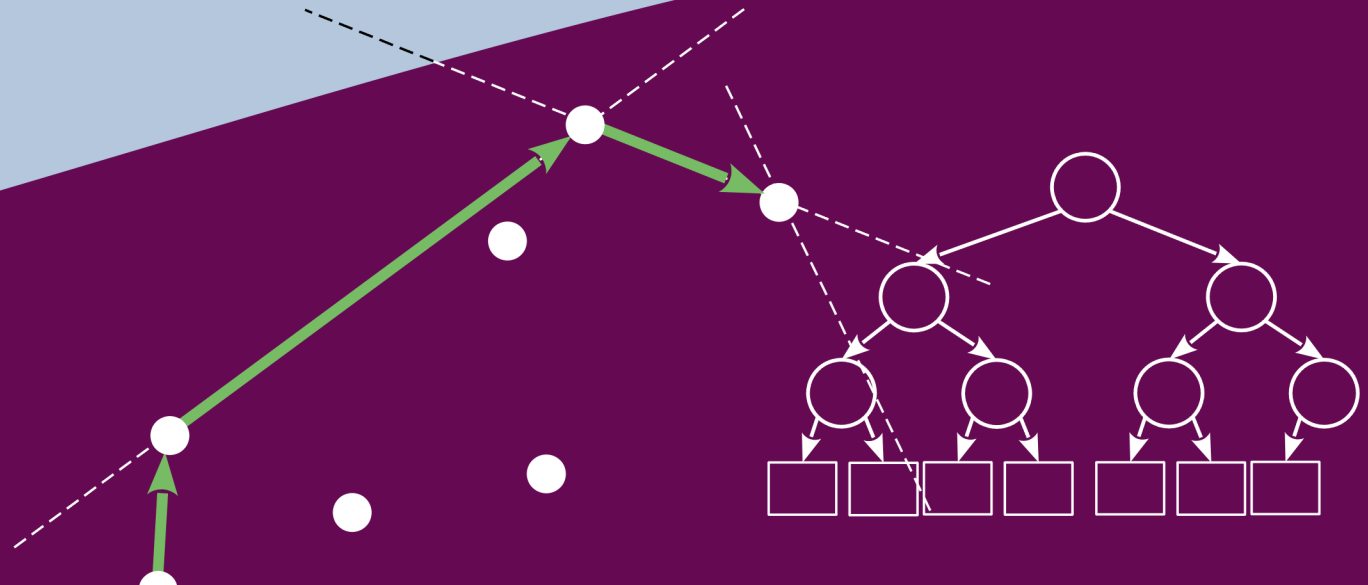
Advanced algorithms and structures data

Week 10: Algorithms over graphs Cycle detection, Min. spanning graphs, Euler graphs

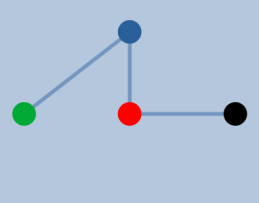


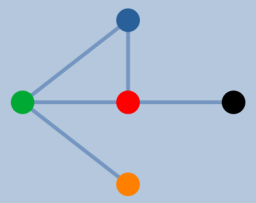


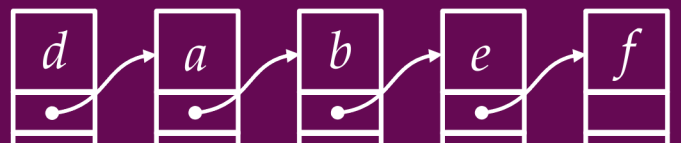


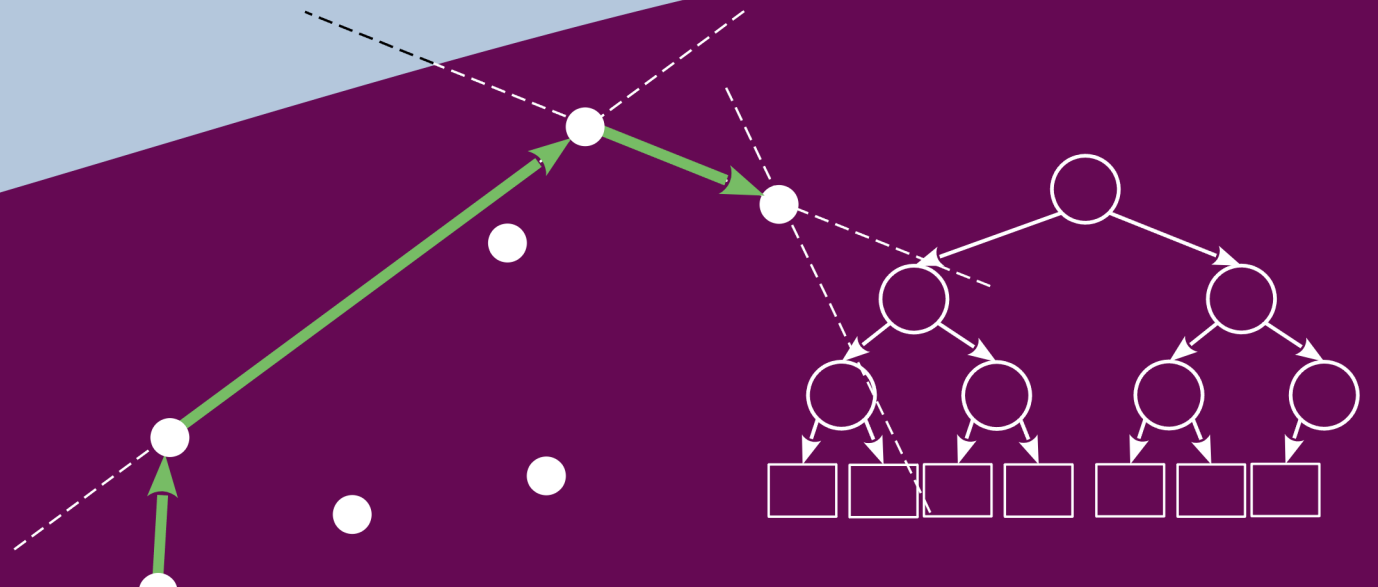
Napredni algoritmi i strukture podataka

Tjedan 10: Algoritmi nad grafovima
Detekcija ciklusa, Min. razapinjujući grafovi,
Eulerovi grafovi

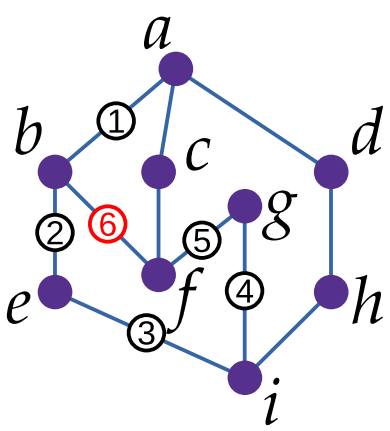








Cycle detection



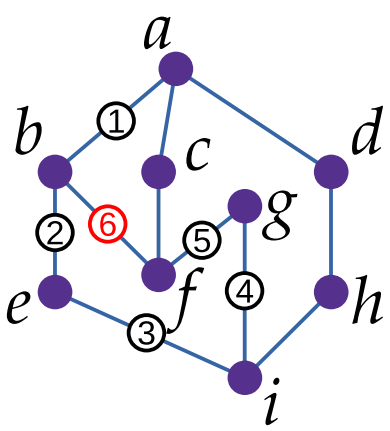
Cycle detection by DFS

- Cycle detection is an essential task for many graph algorithms: say algorithms for finding a minimum spanning tree or for detecting Hamiltonian cycles or Euler circles
- The simplest way is to use DFS or BFS to traverse the graph
  - Let's mark each peak we visit
  - If in the tour we come across an already marked peak, then we have detected a cycle in the graph

Alternative literature:

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

Detekcija ciklusa



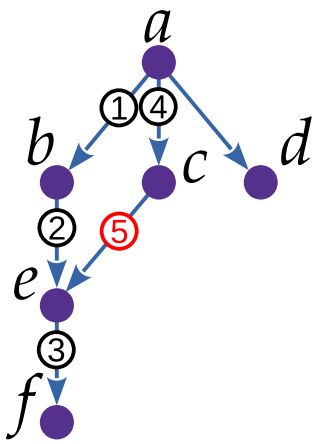
Detekcija ciklusa DFS-om

- 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:

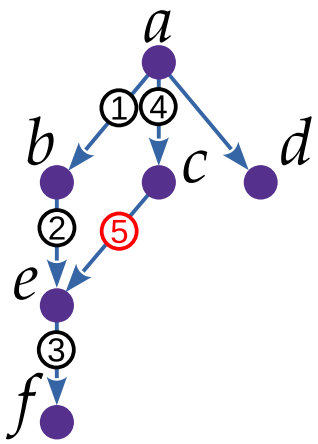
- **Thomas H, Cormen, et al. "Introduction to Algorithms." (2016). Introduction to Algorithms., Chapter VI**

Cycle detection

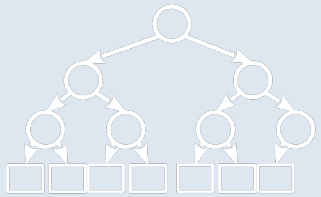


- This approach has a big drawback (say for DFS)
  - After returning from the recursion, we start descending through the graph again
  - We did not erase the marks that we visited some of the peaks
  - If in that descent we encounter a marked peak - have we detected a cycle?
  - We're never really sure
  - If we were to remove labels, this could cause a problem with finding cycles in disconnected graphs
- The solution is to introduce a stack with which we track our current path
  - When we recurse down through the graph, we put the vertices on the stack
  - When we return from the recursion, we remove the vertices from the stack
  - We know that we have returned to an already visited peak if it is on the stack

Detekcija ciklusa



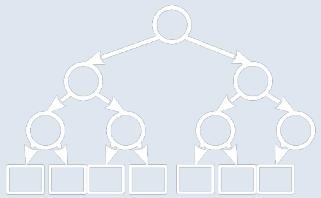
- Ovaj pristup ima veliki nedostatak (recimo za DFS)
  - Nakon povratka iz rekurzije počinjemo se ponovo spuštati kroz graf
  - Oznake da smo obišli neke od vrhova nismo brisali
  - Ako u tom spuštanju naiđemo na označeni vrh – da li smo detektirali ciklus?
  - To zapravo nikad nismo sigurni
  - Ako 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



Cycle detection

```
procedure FindCycle(G)
    initialize all vertices in G as not visited
    S ← empty stack
    while there is an unvisited vertex u_0 in G do
        FindCycle_recursive(G, u_0, 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()
```



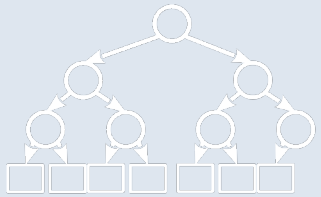
- At the same time, we use two mechanisms: we mark vertices as visited and we have a stack with which we follow the current trajectory
  - By marking vertices, we avoid having to process the same partition of vertices more than once in a disconnected graph
- By entering the recursive call, we put the top on the stack ,and when leaving, we move it from the stack
  - We take care not to return to the previous vertex in the current path in undirected graphs
  - At the moment when we see that the top on the stack , we count the cycle - all vertices on the stack backwards until the previous occurrence of the vertices
    - When calculating the cycles, we do not move the vertices from the stack to allow further progression of the algorithm

Detekcija ciklusa

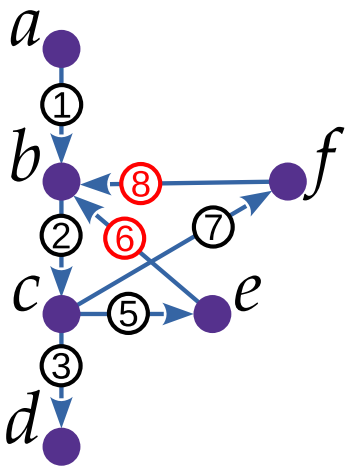
```
procedure FindCycle(G)
    initialize all vertices in G as not visited
    S ← empty stack
    while there is an unvisited vertex u_0 in G do
        FindCycle_recursive(G, u_0, S)

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

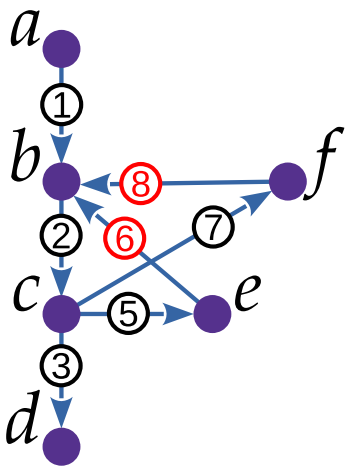


Cycle detection - example



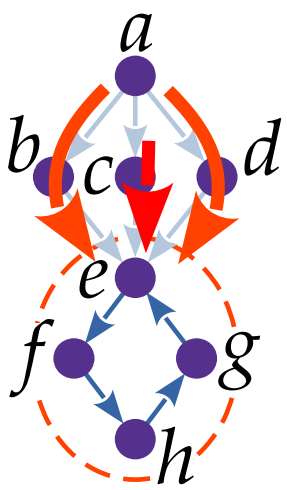
Iteration	1	2	3	4	5	6	7	8
vertex u	a	b	c	d	c	e	c	f
vertex v	b	c	d		e	b	f	b
stack S	a	b a	c b a	d c b a	c b a	e c b a	c b a	f c b a
cycle c						e, c, b		f, c, b

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	c b a	e c b a	c b a	f c b a
cycle c						e, c, b		f, c, b

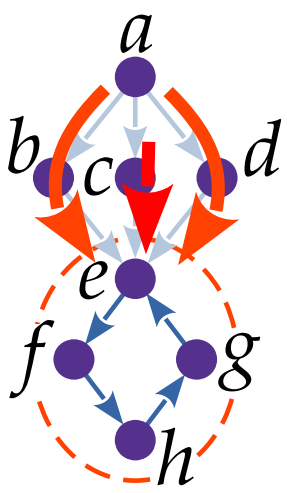
Cycle detection



- This approach is not without problems either
  - Given that we only follow the current path, it is possible to detect one and the same cycle more than once
  - Let's say, the first trajectory in the exposed example is h
    - The last edge reveals the cycle h
  - We return from the recursion until
  - We enter the recursion again and fill the stack, up to h
    - With the last edge, we rediscover the cycle h

- The method is essentially a slightly modified DFS, thus increasing the complexity of the algorithm (+)

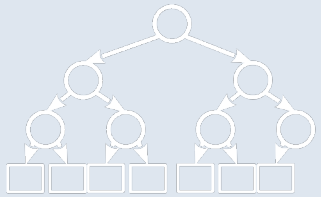
Detekcija ciklusa



- 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 $O(V + E)$

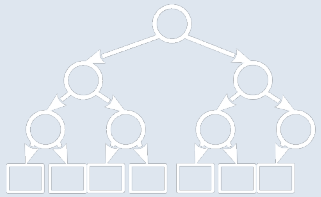
Union-Find

- It uses notation of partitions of an undirected graph to avoid cycles
- One of the simpler methods is to use graphs as a representation
- We call each graph a tree of a disconnected set(disjoint-set tree)
  - Each vertex of the tree represents a vertex of the graph
  - The top of the tree points to the parent with a pointed edge
  - The root tip in the tree points to itself
- The set of all trees of disjoint sets is called a disjoint forest meetings(disjoint-set forest)
- Let's imagine we have a graph = (,)and to process it edge by edge – which often happens in algorithms that use cycle detection

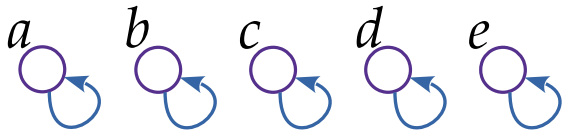


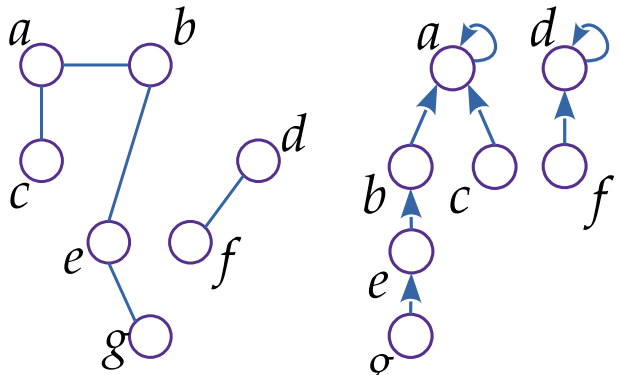
Union-Find

- 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

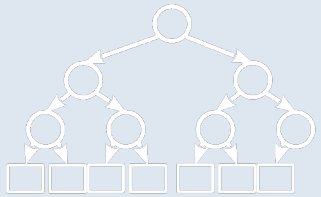


Union-Find

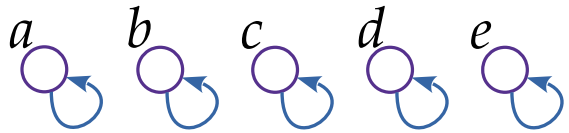


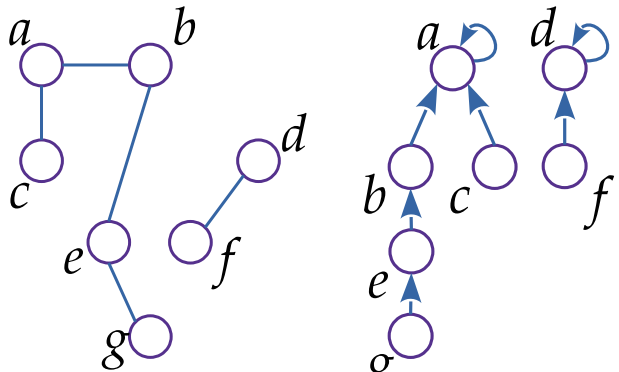


- At the beginning, we have a tree of the disconnected set for each vertex of the graph
  - Each tree obviously has one tip which is the root and points to itself
  - A forest of disconnected sets has as many trees as there are vertices in the graph
- As we get the edges of the graph, we have two situations:
  1. A new edge connects vertices located in two different trees of disjoint sets - means that the edge does not form a cycle
     - We return to the calling algorithm that the edge does not form a cycle
     - We join the two trees according to the edge we got, so that one top depends on the other
  2. A new edge connects vertices in the same tree of a disjoint set – means that the edge forms a cycle
     - We return to the calling algorithm that the edge forms a cycle

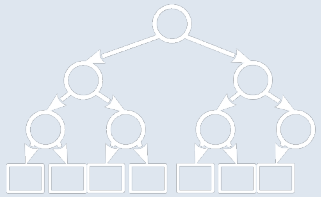


Union-Find

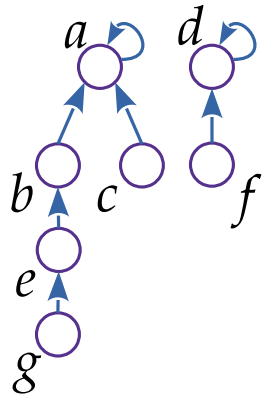


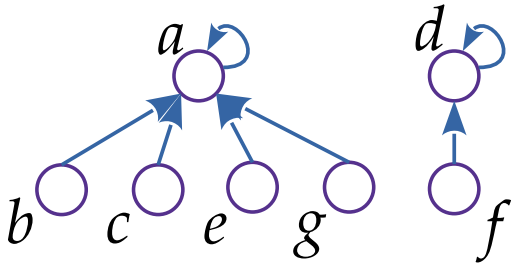


- Na početku imamo stablo nepovezanog skupa za svaki vrh grafa
  - Svako stablo očito ima jedan vrh koji je korijenski i pokazuje sam na sebe
  - Šuma nepovezanih skupova ima stabala koliko je i vrhova u grafu
- Kako dobivamo bridove grafa, imamo dvije situacije:
  1. Novi brid povezuje vrhove koji se 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

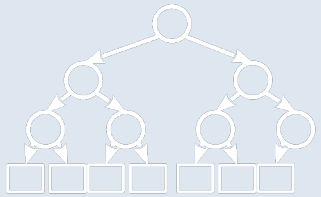


Union-Find

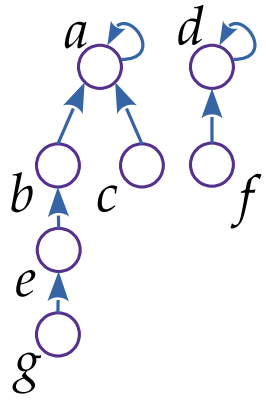


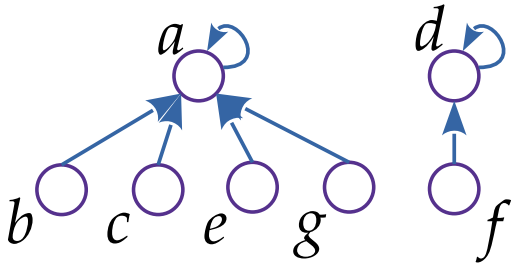


- Trees of disjoint sets can be collapsed, to allow the shortest possible search for a vertex in the tree
- By compressing, we reduce the tree search to O(1), and do not impair the functionality at all
- Merging summary sets takes more time, since restructuring of these merging trees is done

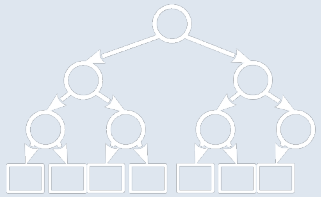


Union-Find





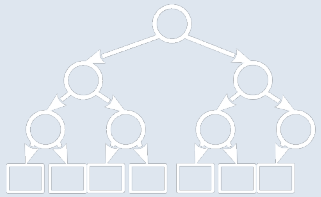
- Stabla nepovezanih skupova mogu se sažimati, kako bi se omogućilo što je kraće moguće traženje vrha po stablu
- Sažimanjem svodimo pretraživanje po stablu na O(1), a nimalo ne narušavamo funkcionalnost
- Spajanje sažetih skupova zahtijeva više vremena, s obzirom da se radi restrukturiranje ova stabla koja se spajaju



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
```

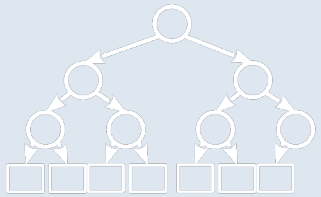
- **MakeSet**–a procedure that in a forest of unrelated sets adds the top who is his own parent
- **Find**–a procedure that searches for the parent of the vertex –at the same time it does tree compaction
- **Union**–procedure that checks if vertices and in the same tree, so if they are not, it puts them in the same tree
- A simple check whether the peaks are in the same cycle can be seen in the procedure**Union**



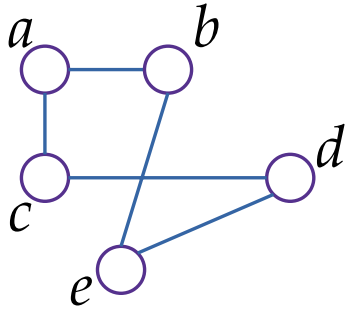
Union-Find

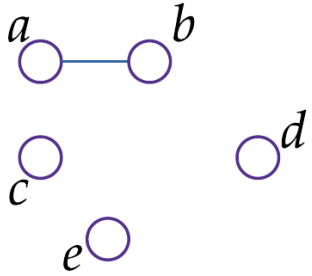
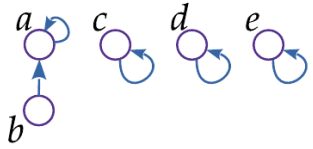
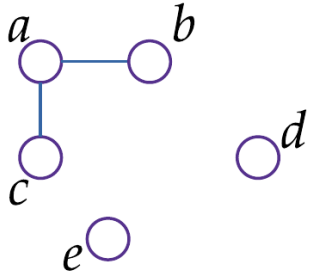
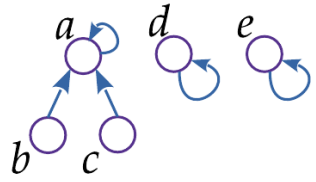
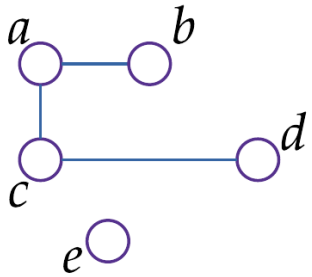
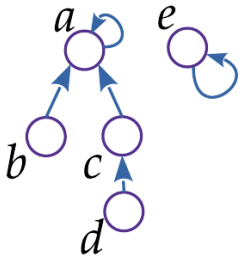
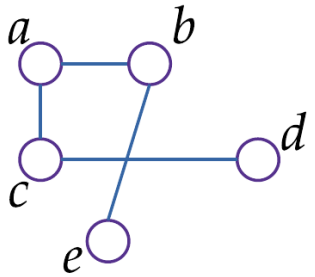
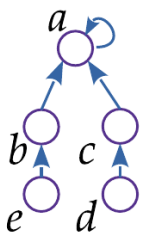
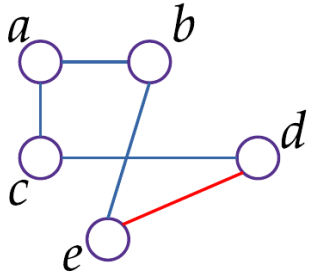
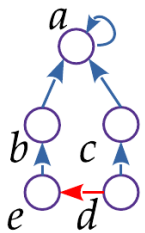
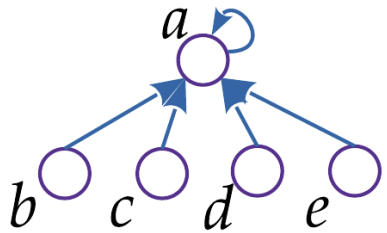
```
function MakeSet(F, x)
    if x not in F then
        x.parent = x
        add tree x to the forest F
    return F
function Find(x)
    while x ≠ x.parent do
        x.parent = x.parent.parent          ▷ The compression
        x = x.parent
    return x
procedure Union(x, y)
    x = Find(x)
    y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

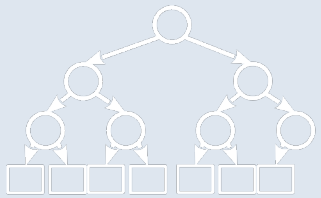
- **MakeSet** – procedura koja u šumu nepovezanih skupova F dodaje vrh x koji je sam svoj roditelj
- **Find** – procedura koja traži roditelja vrha x – u isto vrijeme radi sažimanje stabla
- **Union** – procedura koja provjerava da li su vrhovi x i y u istom stablu, pa ako nisu stavlja ih u isto stablo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



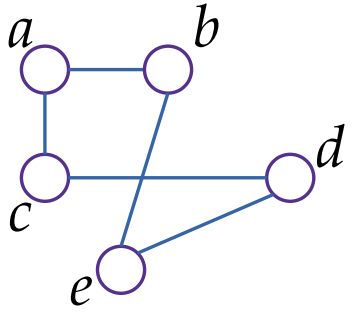
Union-Find

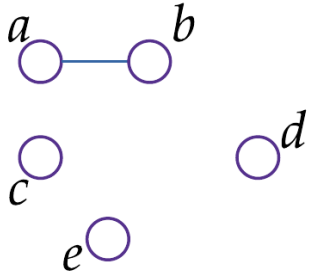
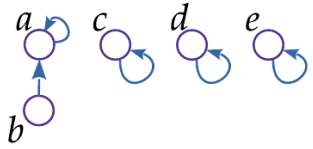
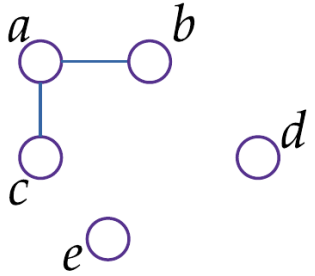
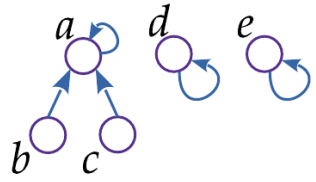
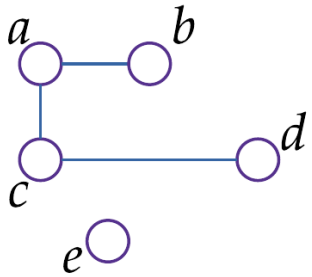
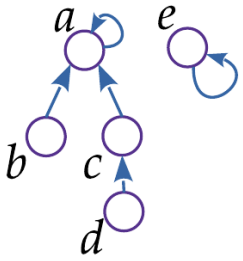
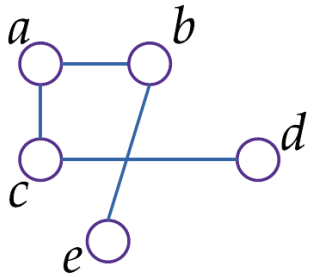
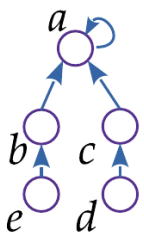
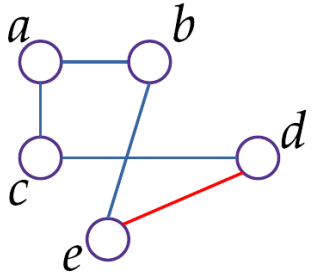
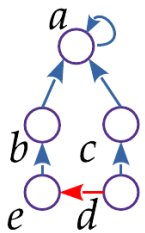
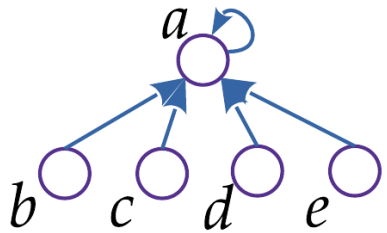


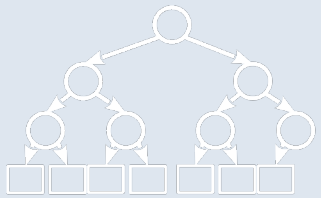
Iteration 1: Edge *ab*		**Iteration 2:** Edge *ac*	
			
Iteration 3: Edge *cd*		**Iteration 4:** Edge *be*	
			
Iteration 5: Edge *de* - **cycle detected**		**The final compressed disjoint-set forest**	
			



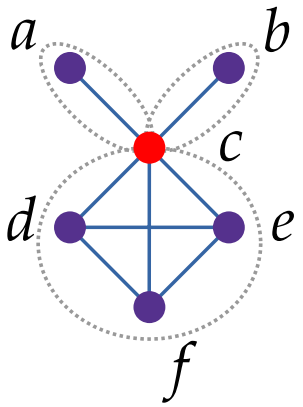
Union-Find

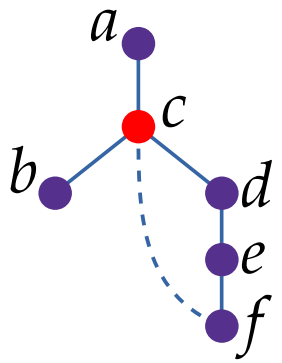


Iteration 1: Edge *ab*		**Iteration 2:** Edge *ac*	
			
Iteration 3: Edge *cd*		**Iteration 4:** Edge *be*	
			
Iteration 5: Edge *de* - **cycle detected**		**The final compressed disjoint-set forest**	
			



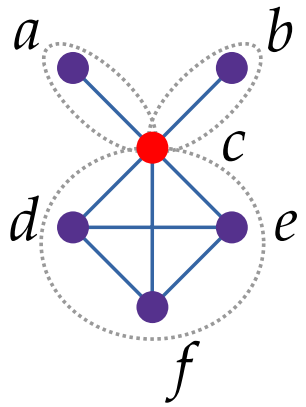
Detection of blocks in the graph

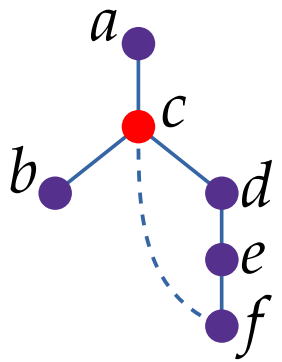




- It is done in undirected graphs
- Block(biconnected component)is a subgraph ′graph which does not contain breaking points(articulation points). This means that if we remove any vertex from the block, the block still remains connected.
- Finding breakpoints in a graph is the basis for block detection - the problem is similar to cycle detection
  - Blocks are obtained by interrupting the graph at a turning point
  - For example, we have blocks*a.c*,*b.c*,*cdef*
- Using DFS we search for cycles in the graph
  - When we find the cycle, the root tip represents the breaking point
- For example, we see a cycle*cdefc*, where is*c*the root tip, and thus the breaking point

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

Detection of blocks in the graph

- Block detection is performed on a spanning tree derived from the extended definition of a graph, which we define as an ordered triple

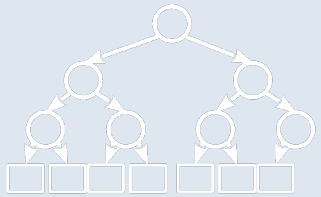
$$= (, ,)$$

  - where is mapping function

$$: \rightarrow \mathbb{N}\times\mathbb{N}$$

  - which maps each vertex of the graph to an ordered pair (,),where
    - represents the number of the vertex in the spanning tree
    - represents the highest predecessor of that peak - obviously a candidate for the turning point
- We define basic rules on such a graph
  - No two vertices in the graph have the same number

$$\nexists , \in : \neq , \quad = (())$$



Detekcija blokova u grafu

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

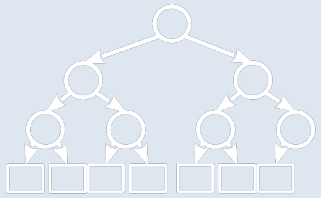
$$G = (V, E, p)$$

  - gdje je p funkcija mapiranja

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

  - 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

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



Detection of blocks in the graph

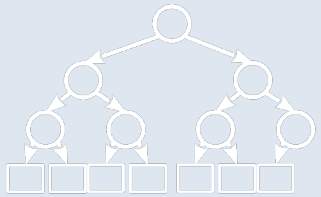
- We define basic rules on such a graph
  - The predecessor of the vertex v is defined

$$\in m)\min(\quad,\quad !,(\quad '),\ldots,(\quad(\quad \#)))\quad(\quad\quad)$$

    - where are they !, ",..., #the vertices to which descendents from the subtree of the vertex return
    - wanted**minimal**the common ancestor of the entire subtree of the vertex
  - Defined differently

$$\in m)\min(\quad,\quad !,(\quad '),\ldots,(\quad(\quad \#)))\quad(\quad\quad)$$

    - where are they !, ",..., #descendants in the vertex subtree
    - **minimal**common ancestor of the peak is either the top itself ,or the minimal predecessor in the vertex's subtree



Detekcija blokova u grafu

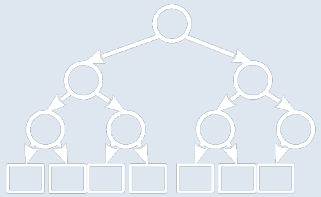
- Definiramo osnovna pravila na takav graf
  - Prethodnik vrha v definira se

$$p(v) = \min(n(v), n(u_1), n(u_2), \dots, n(u_k))$$

    - 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

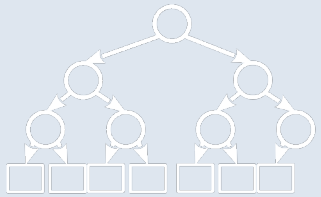


Detection of blocks in the graph

```
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))
```

- We initialize each vertex uniquely ( ) = ( )
- We are moving towards the neighboring peaks of
  - We put the edge on the stack
  - If the adjacent vertex new, that is ( ) = 0, then we recursively call its processing
    - When returning from recursion, we check whether it is its predecessor or some descendants of
    - If it is, we form a block with a stack on which we put the edges
    - If not, we update the minimal predecessor
  - If we are an adjacent vertex already visited, it is possible that it is a return edge
    - We update the minimal predecessor of the vertex ,and which can be the top

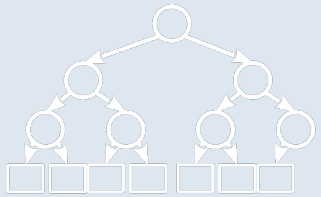


Detekcija blokova u grafu

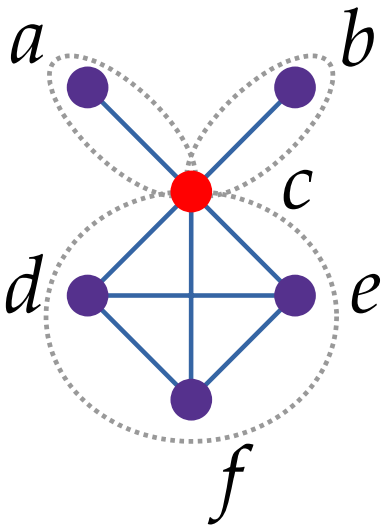
```
procedure BlockSearch(G)
    initialize all vertices in G as n(v) = 0
    step ← 1
    S ← empty stack
    while there is a vertex u in G, having n(u) = 0 do
        BlockSearch_recursive(G, u, step, S)

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



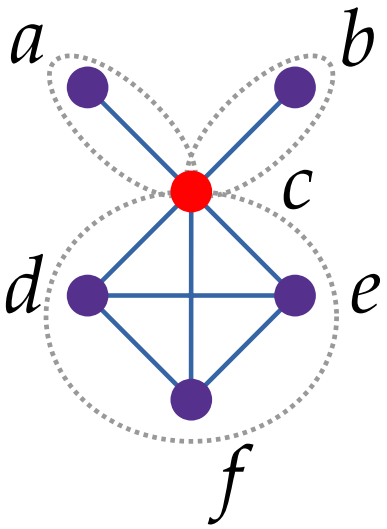
Detection of blocks in the graph



	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	

!= = [, "{ =}], , #= [= , { , }] [{ }]

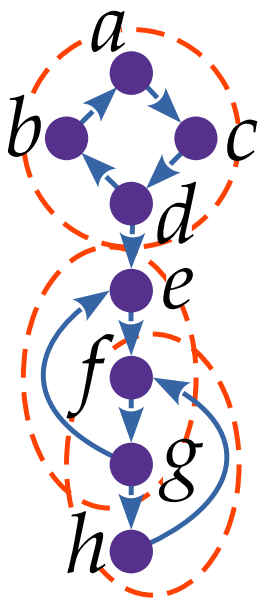
Detekcija blokova u grafu



	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*	

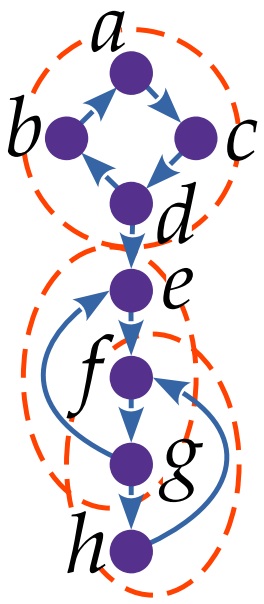
$$b_1 = G[V = \{a, c\}], b_2 = G[V = \{b, c\}], b_3 = G[V = \{c, d, e, f\}]$$

Detection of components in a directed graph



- The definition of a tightly connected component (SCC) implies that all pairs of vertices in that component are mutually accessible
- A naive attempt to solve this problem is NP-complex because we should create partitions of vertices in which all vertices are mutually reachable
- The problem can be simplified by knowing that cycles form such components
  - For example, we have three cycles: 1 =*acdba*, 2 =*efge*and 3 =*fghf*
  - The two cycles share common vertices and edges, which gives us two tightly connected components: 1and 2 ∪ 3
- Blocks in undirected graphs share a breakpoint, while tightly connected components in directed graphs cannot share vertices

Detekcija komponenti u usmjerenom grafu



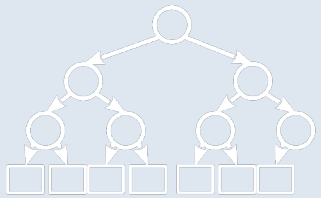
- Definicija čvrsto povezane komponente (SCC) podrazumijeva da su svi parovi vrhova u toj komponenti međusobno dohvatljivi
- Naivni pokušaj rješavanja ovog problema je NP-kompleksan jer bismo trebali stvarati particije vrhova u kojima su svi vrhovi međusobno dohvatljivi
- Problem se može pojednostavniti 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

Tarjan's Algorithm (SCC)

```
procedure SCCSearch(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
        SCCSearch_recursive(G, u, step, S)

procedure SCCSearch_recursive(G, u, step, S)
    p(u) = n(u) = step
    step ← 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
```

- We initialize each vertex uniquely ( ) = ( )
- We are moving towards the neighboring peaks of
  - We put the top on the stack
  - If the adjacent vertex new, that is ( ) = 0,then we recursively call its processing
    - When returning from recursion, we update the minimal predecessor of the vertex
  - If we are an adjacent vertex already visited, it is possible that it is a return edge
    - We update the minimal predecessor of the vertex ,and which can be the top
  - When returning back to the caller, we test the vertex's predecessor .If the top has no predecessor so it is ( ) < ( ),then the top we consider the root of the component
    - This step is slightly different from the block detection algorithm

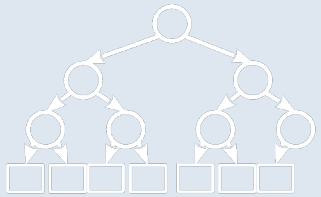


Tarjanov algoritam (SCC)

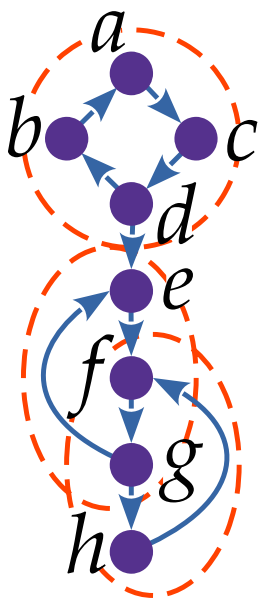
```
procedure SCCSearch(G)
    initialize all vertices in G as n(v) = 0
    step ← 1
    S ← empty stack
    while there is a vertex u in G, having n(u) = 0 do
        SCCSearch_recursive(G, u, step, S)

procedure SCCSearch_recursive(G, u, step, S)
    p(u) = n(u) = step
    step ← 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



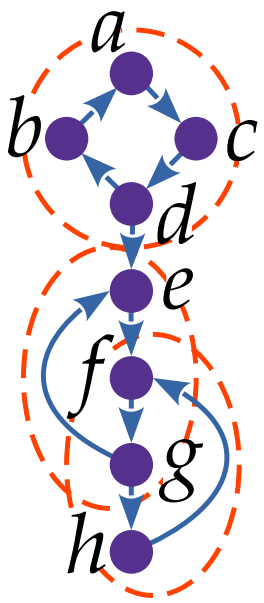
Tarjan's Algorithm (SCC)



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

!= [= {*h*, , , }] "= [= { , , , }]

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

$$scc_1 = G[V = \{h, g, f, e\}]$$
$$scc_2 = G[V = \{b, d, c, a\}]$$

Minimum spanning tree (MST)

- Previously, we defined a spanning tree as the product of graph traversal with various algorithms, such as BFS and DFS
  - Because of this diverse approach, a graph can have many different spanning trees
  - For the weight graph = (, ,)thus we can define the set of all spanning trees

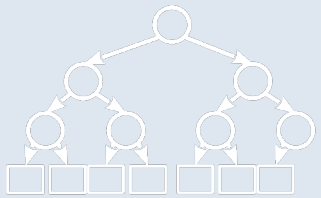
$$( ) = \{ " \quad != , (!, \ : \ !\subseteq \ )( )\}$$

  - Finding the minimum spanning tree is the result of optimization

$$= ar(g\ ) \ \#\% \qquad \$\in\%\&(\#)> \qquad ( )$$

$$) \in *\$(\#\$)^{\%}$$

  - We are looking for a spanning tree whose sum of edge weights is minimal in the set of all spanning trees
- All algorithms used for this purpose are greedy algorithms



Minimalno razapinjujuće stablo (MST)

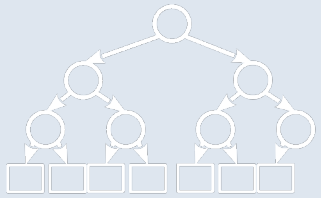
- Prethodno smo definirali razapinjujuće stablo kao produkt obilaska grafa raznim algoritmima, poput BFS i DFS
  - Zbog tog raznolikog pristupa, graf može imati više različitih razapinjujućih stabala
  - Za težinski graf $G = (V, E, w)$ tako možemo definirati skup svih razapinjujućih stabala

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

  - 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

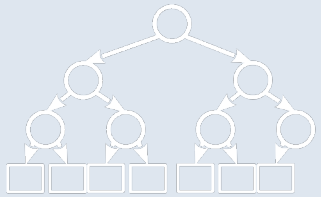


Kruskal's algorithm

```
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
```

- Search complexity summarized *Union-Find* method is (1)
- Due to sorting, the complexity of Kruskal's algorithm is log(.)
- Given that is in a complete undirected graph < !, complexity *for* loops with *Union-Find* it's a method + −(1 , (which for the complete graph is less complex than sorting.

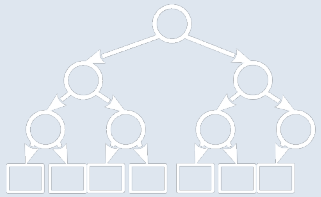
- Exclusively for undirected graphs!
- We initialize the min. spanning tree by moving all vertices and leaving the set of edges empty
- We sort the edges of the input graph in ascending order of weight(log$!)
- We pass along the edges for so long that in min. the crucifix tree has fewer edges than (−)1
  - If the edge we took from the sorted set of edges does not form a cycle in min. to the crucifying tree, we add it to the edges min. crucifixion tree
- For cycle detection we can use summary *UnionFind* method.



Kruskalov algoritam

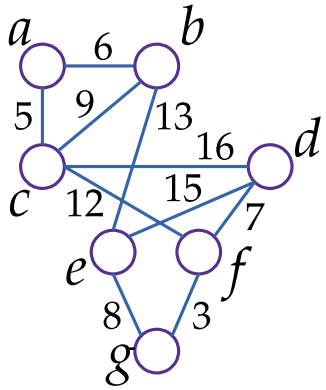
```
procedure KRUSKAL(G)
    MST ← (V = V(G), E = ∅)
    sort edges E(G) ascending by weights
    for e_i ∈ E(G) do
        if |E(MST)| < |V(G)| − 1 then
            if no cycle in G' = (V(MST), E(MST) ∪ {e_i}) then
                add e_i to E(MST)
    return MST
```

- Kompleksnost pretraživanja sažete *Union-Find* metode je $O(1)$
- Zbog sortiranja, kompleksnost Kruskalovog algoritma je $O(E\log_2 E)$.
- S obzirom da je u kompletnom neusmjerenom grafu $E < V^2$, kompleksnost *for* petlje s *Union-Find* metodom je $O(E + (V-1)V)$, što je za kompletni graf manje kompleksnosti od sortiranja.



- 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 – $O(E\log_2 E)$!
- 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 sažetu *UnionFind* metodu.

Kruskal's algorithm

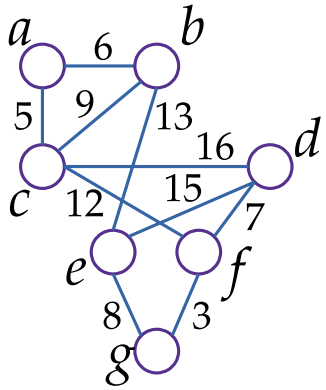


$$\mathbf{G} = \begin{array}{c} \\ a \\ b \\ c \\ d \\ e \\ f \\ g \end{array} \begin{array}{c} \begin{array}{ccccccc} a & b & c & d & e & f & g \end{array} \\ \begin{bmatrix} 0 & 6 & 5 & 0 & 0 & 0 & 0 \\ 6 & 0 & 9 & 0 & 13 & 0 & 0 \\ 5 & 9 & 0 & 16 & 0 & 12 & 0 \\ 0 & 0 & 16 & 0 & 15 & 7 & 0 \\ 0 &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

= 3, = 5, = 6, = 7,
= 8, = 9, = 12, = 13,
= 15, = 16

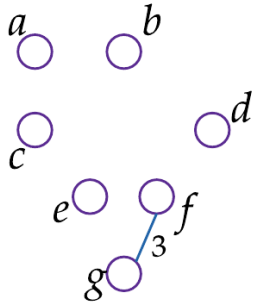
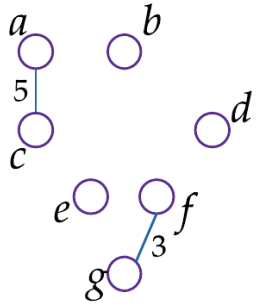
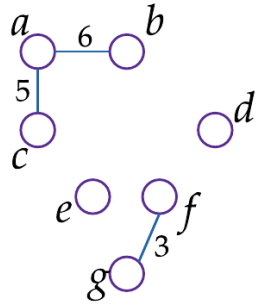
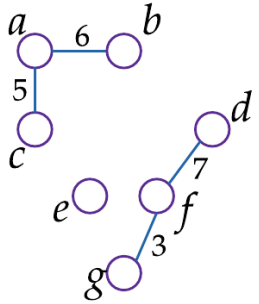
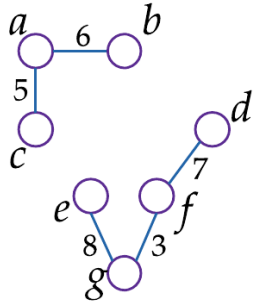
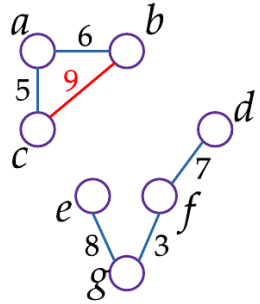
Iteration 1: Edge $fg = 3$	**Iteration 2:** Edge $ac = 5$	**Iteration 3:** Edge $ab = 6$
Iteration 4: Edge $df = 7$	**Iteration 5:** Edge $eg = 8$	**Iteration 6:** Edge $bc = 9$

Kruskalov algoritam

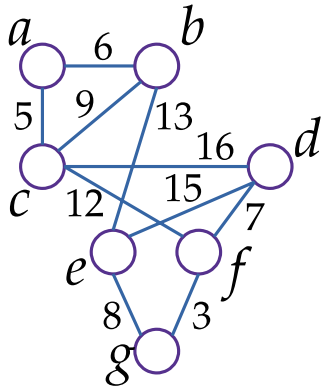


$$\mathbf{G} = \begin{array}{c} \\ a \\ b \\ c \\ d \\ e \\ f \\ g \end{array} \begin{array}{c} \begin{array}{ccccccc} a & b & c & d & e & f & g \end{array} \\ \begin{bmatrix} 0 & 6 & 5 & 0 & 0 & 0 & 0 \\ 6 & 0 & 9 & 0 & 13 & 0 & 0 \\ 5 & 9 & 0 & 16 & 0 & 12 & 0 \\ 0 & 0 & 16 & 0 & 15 & 7 & 0 \\ 0 &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

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

Iteration 1: Edge $fg = 3$ 	**Iteration 2:** Edge $ac = 5$ 	**Iteration 3:** Edge $ab = 6$ 
Iteration 4: Edge $df = 7$ 	**Iteration 5:** Edge $eg = 8$ 	**Iteration 6:** Edge $bc = 9$ 

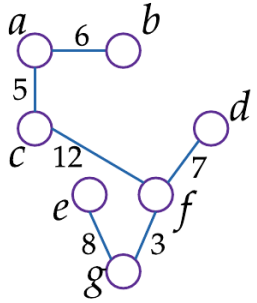
Kruskal's algorithm



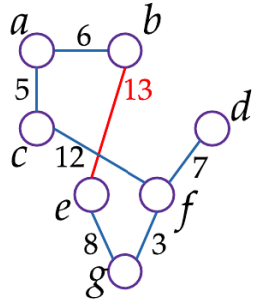
$$\mathbf{G} = \begin{array}{c} \\ a \\ b \\ c \\ d \\ e \\ f \\ g \end{array} \begin{array}{c} \begin{array}{ccccccc} a & b & c & d & e & f & g \end{array} \\ \begin{bmatrix} 0 & 6 & 5 & 0 & 0 & 0 & 0 \\ 6 & 0 & 9 & 0 & 13 & 0 & 0 \\ 5 & 9 & 0 & 16 & 0 & 12 & 0 \\ 0 & 0 & 16 & 0 & 15 & 7 & 0 \\ 0 &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

= 3, = 5, = 6, = 7,
= 8, = 9, = 12, = 13,
= 15, = 16

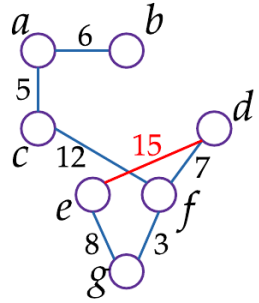
Iteration 7: Edge $cf = 12$



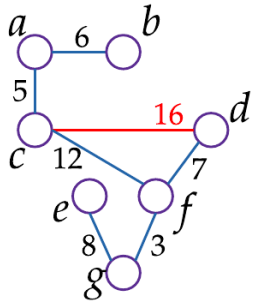
Iteration 8: Edge $be = 13$



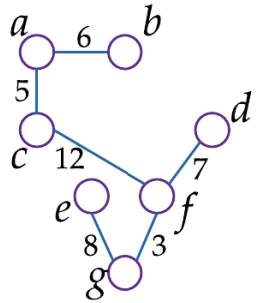
Iteration 9: Edge $de = 15$



Iteration 10: Edge $cd = 16$



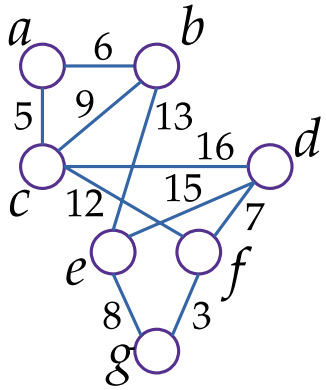
Final MST



Final MST adjacency matrix

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

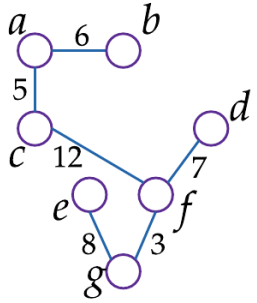
Kruskalov algoritam



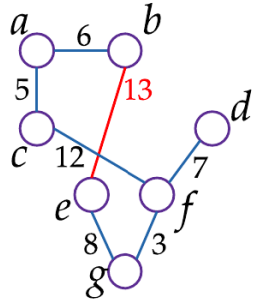
$$\mathbf{G} = \begin{array}{c} \\ a \\ b \\ c \\ d \\ e \\ f \\ g \end{array} \begin{array}{c} \begin{array}{ccccccc} a & b & c & d & e & f & g \end{array} \\ \begin{bmatrix} 0 & 6 & 5 & 0 & 0 & 0 & 0 \\ 6 & 0 & 9 & 0 & 13 & 0 & 0 \\ 5 & 9 & 0 & 16 & 0 & 12 & 0 \\ 0 & 0 & 16 & 0 & 15 & 7 & 0 \\ 0 &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

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

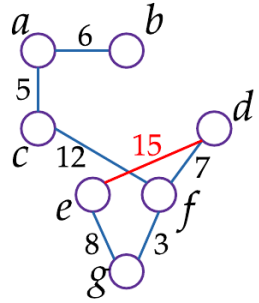
Iteration 7: Edge $cf = 12$



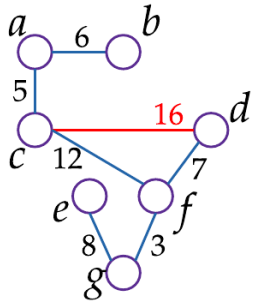
Iteration 8: Edge $be = 13$



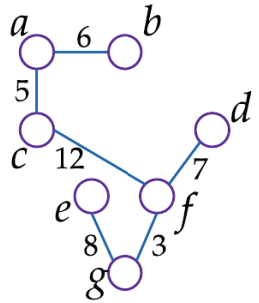
Iteration 9: Edge $de = 15$



Iteration 10: Edge $cd = 16$



Final MST



Final MST adjacency matrix

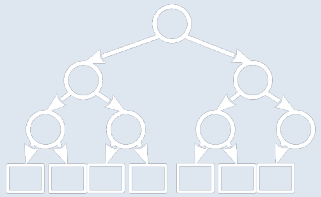
$$\begin{array}{c} \\ a \\ b \\ c \\ d \\ e \\ f \\ g \end{array} \begin{array}{c} \begin{array}{ccccccc} a & b & c & d & e & f & g \end{array} \\ \begin{bmatrix} 0 & 6 & 5 & 0 & 0 & 0 & 0 \\ 6 & 0 & 0 & 0 & 0 & 0 & 0 \\ 5 & 0 & 0 & 0 & 0 & 12 & 0 \\ 0 & 0 & 0 & 0 & 0 & 7 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0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

Dijkstra's Algorithm (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
```

- The complexity of the base iteration is ( ). DFS algorithm is (+),what is becoming (2),that is ( )for min. crucifixion tree.
- Ultimately, the complexity of Dijkstra's algorithm (*).

- The disadvantage of Kruskal's algorithm is its high complexity due to the need to sort edges
- Dijkstra's method is somewhat different
  - No sorting
  - The moment we detect a cycle, we remove the heaviest edge from the cycle
- Here to us *UnionFind* method does not work, so we have to use the method that uses extended DFS (shown in front of several displays)
  - After we add the edge ,let's take one of those two vertices as the initial vertex
  - If we return to that initial peak, then we have a cycle
  - We can also save edge weights in the stack - to detect the edge with the highest weight

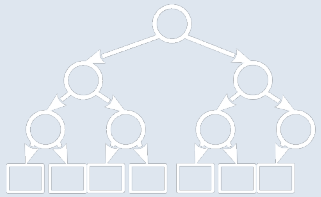


Dijkstrin algoritam (MST)

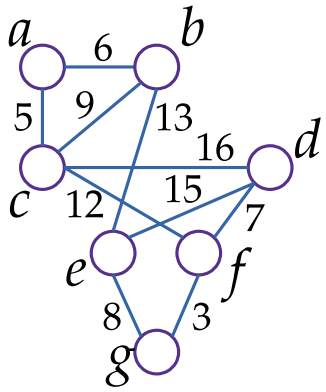
```
procedure DijkstraMST(G)
    MST ← (V = V(G), E = ∅)
    for e_i ∈ E(G) do
        add e_i to E(MST)
        if there is a cycle in MST then
            remove the maximal weight edge from the cycle
    return MST
```

- 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 Dijkstrinog 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

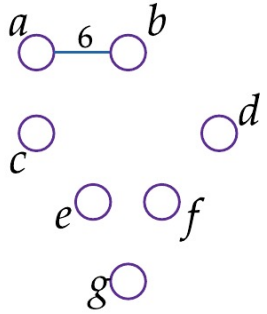
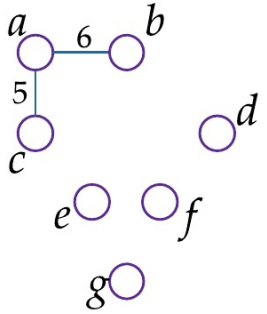
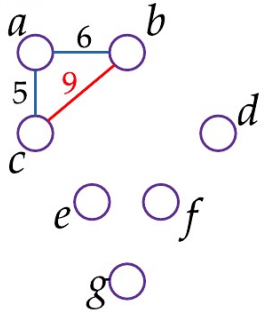
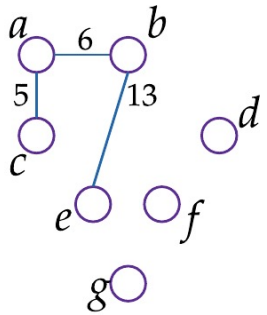
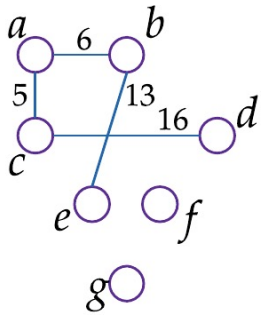
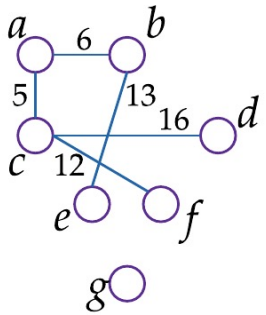


Dijkstra's Algorithm (MST)

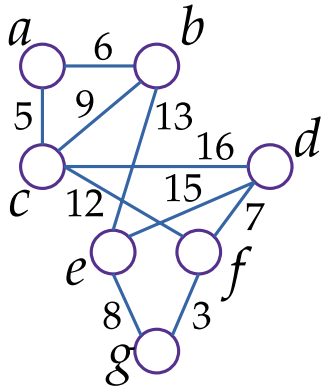


$$\mathbf{G} = \begin{array}{c} \\ a \\ b \\ c \\ d \\ e \\ f \\ g \end{array} \begin{array}{c} \begin{array}{ccccccc} a & b & c & d & e & f & g \end{array} \\ \begin{bmatrix} 0 & 6 & 5 & 0 & 0 & 0 & 0 \\ 6 & 0 & 9 & 0 & 13 & 0 & 0 \\ 5 & 9 & 0 & 16 & 0 & 12 & 0 \\ 0 & 0 & 16 & 0 & 15 & 7 & 0 \\ 0 &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

, , , , , , , , , ,

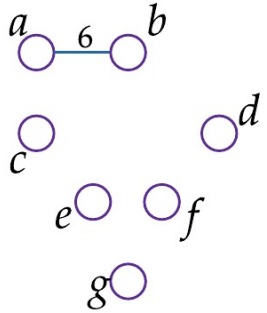
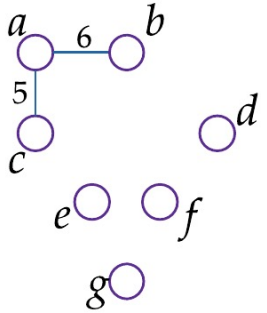
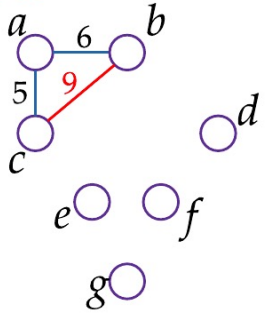
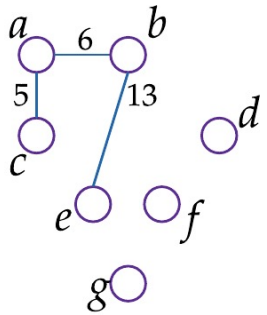
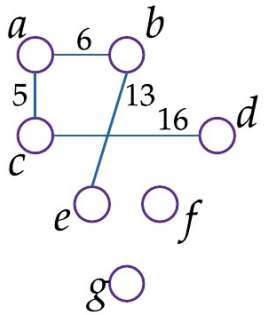
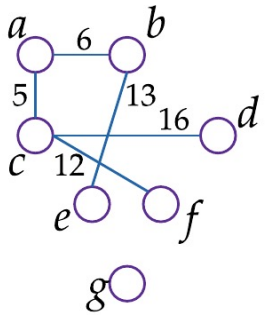
Iteration 1: Edge $ab = 6$ 	**Iteration 2:** Edge $ac = 5$ 	**Iteration 3:** Edge $bc = 9$, Cycle, Remove bc 
Iteration 4: Edge $be = 13$ 	**Iteration 5:** Edge $cd = 16$ 	**Iteration 6:** Edge $cf = 12$ 

Dijkstrin algoritam (MST)

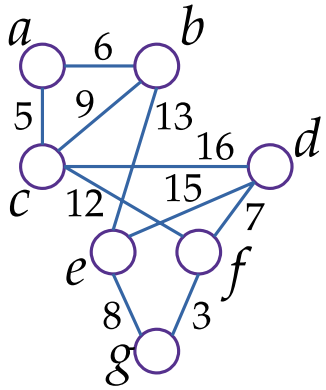


$$\mathbf{G} = \begin{array}{c} \\ a \\ b \\ c \\ d \\ e \\ f \\ g \end{array} \begin{array}{c} \begin{array}{ccccccc} a & b & c & d & e & f & g \end{array} \\ \begin{bmatrix} 0 & 6 & 5 & 0 & 0 & 0 & 0 \\ 6 & 0 & 9 & 0 & 13 & 0 & 0 \\ 5 & 9 & 0 & 16 & 0 & 12 & 0 \\ 0 & 0 & 16 & 0 & 15 & 7 & 0 \\ 0 &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

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

Iteration 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 $cf = 12$ 

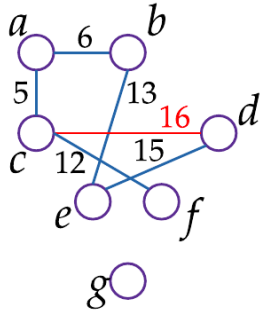
Dijkstra's Algorithm (MST)



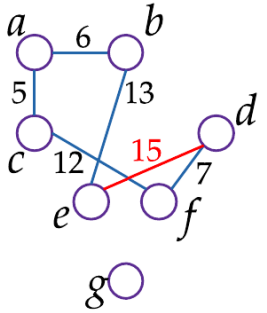
$$\mathbf{G} = \begin{array}{c} \\ a \\ b \\ c \\ d \\ e \\ f \\ g \end{array} \begin{array}{c} \begin{array}{ccccccc} a & b & c & d & e & f & g \end{array} \\ \begin{bmatrix} 0 & 6 & 5 & 0 & 0 & 0 & 0 \\ 6 & 0 & 9 & 0 & 13 & 0 & 0 \\ 5 & 9 & 0 & 16 & 0 & 12 & 0 \\ 0 & 0 & 16 & 0 & 15 & 7 & 0 \\ 0 &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

, , , , , , , , , ,

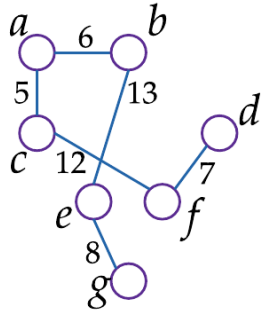
Iteration 7: Edge $de = 15$, Cycle, Remove cd



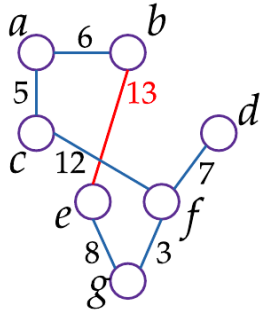
Iteration 8: Edge $df = 7$, Cycle, Remove de



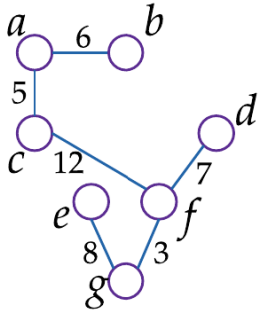
Iteration 9: Edge $eg = 8$



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



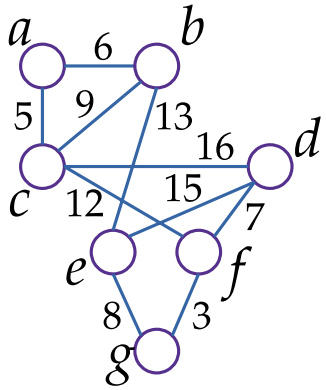
Final MST



Final MST adjacency matrix

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

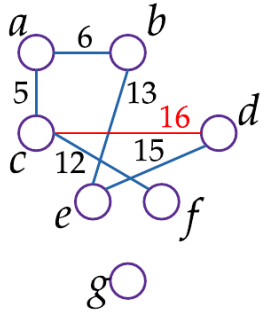
Dijkstrin algoritam (MST)



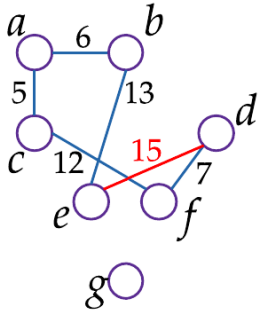
$$\mathbf{G} = \begin{array}{c} \\ a \\ b \\ c \\ d \\ e \\ f \\ g \end{array} \begin{array}{c} \begin{array}{ccccccc} a & b & c & d & e & f & g \end{array} \\ \begin{bmatrix} 0 & 6 & 5 & 0 & 0 & 0 & 0 \\ 6 & 0 & 9 & 0 & 13 & 0 & 0 \\ 5 & 9 & 0 & 16 & 0 & 12 & 0 \\ 0 & 0 & 16 & 0 & 15 & 7 & 0 \\ 0 &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

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

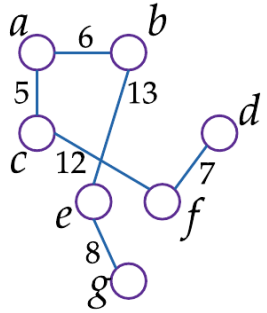
Iteration 7: Edge $de = 15$, Cycle, Remove cd



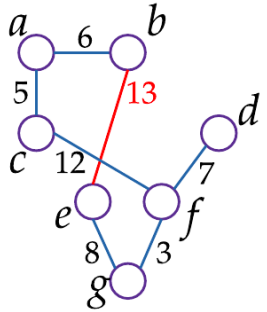
Iteration 8: Edge $df = 7$, Cycle, Remove de



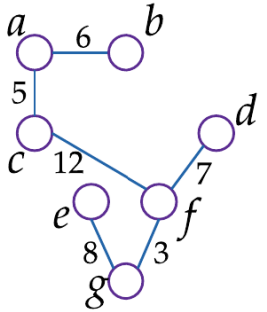
Iteration 9: Edge $eg = 8$



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



Final MST



Final MST adjacency matrix

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

Prim's algorithm

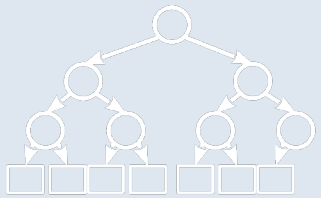
- Let's imagine that we decomposed the graph G into a set of elementary spanning trees ST(G), so that each vertex forms one tree

  $= (\{ 2): 1 \leq \leq , \ 2= \ = 2, \ = \varnothing (2\in (\{)\}\}$)

  - by adding an edge between two elementary spanning trees we get a new unique spanning tree

    $\text{AND} \cup \ B = (\ \text{AN}(\text{D} \cup)(\ B), \ (\ \text{AND}) \cup \ (\ B) \cup \{ C\})$

- The basic idea of Prim's algorithm is **growing** minimal spanning tree $(3\in)$ and the rest of the elemental crucifixion trees $4=(\ ) \setminus \{ 3\}$



Primov algoritam

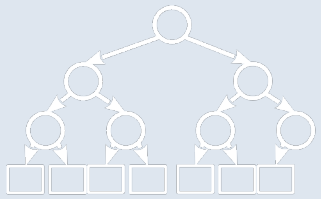
- Zamislimo da smo graf G razgradili na skup elementarnih razapinjujućih stabala ST(G), tako da svaki vrh čini jedno stablo

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

  - 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 $ST_r = ST(G) \setminus \{ST_g\}$

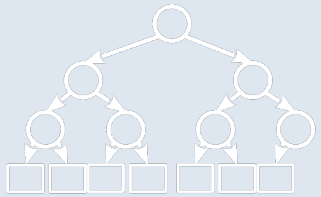


Prim's algorithm

```
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
```

- The complexity of the base iteration is ( ).
- The problem is the search for edges of minimum weight.
  - In a sequential implementation, the total complexity is (!).
  - If we use heap, then the total complexity is (+ log!).

- In the initial min. we add only the initial vertex to the spanning tree 5, without edges
- We define a set 4which initially contains all vertices except the initial vertex 5
- We take the first peak from the set 4. We do this until we have peaks in that set
  - Let's find the edge C= minimum weight between the top and one of the peaks in the growing crucifixion tree D
  - Let's add the top and the edge Cinto a growing crucifix tree D
  - Top we remove from the set of vertices E

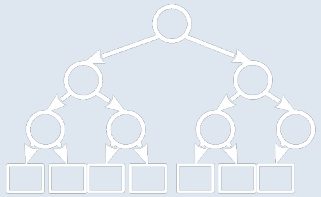


Primov algoritam

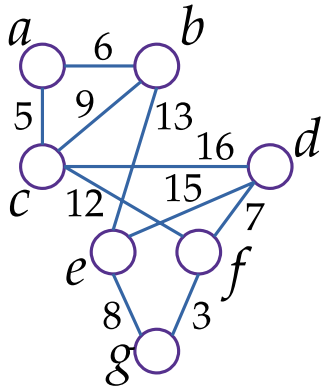
```
procedure PRIM(G, v_s)
    MST ← (V_g = {v_s}, E = ∅)
    V_r ← V(G) \ {v_s}
    while V_r ≠ ∅ do
        choose minimal weighted edge e_n = uv ∈ E(G) such that
                                    u ∈ V(MST) and v ∈ V_r
        add v to V_g(MST)
        remove v from V_r
        add e_n = uv to E(MST)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

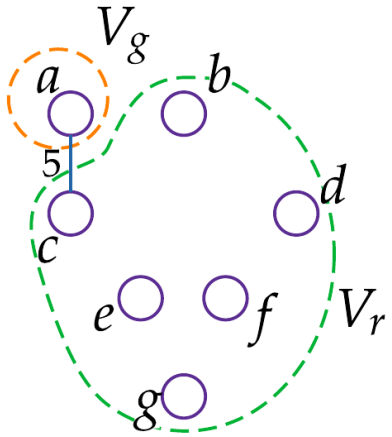


Prim's algorithm

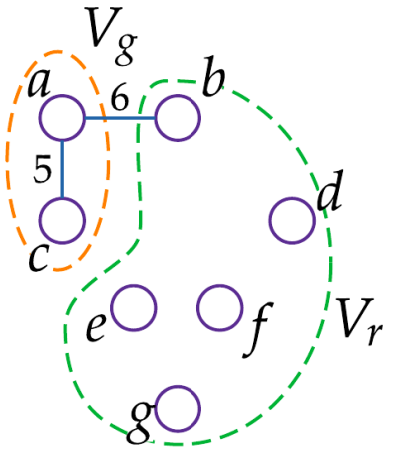


$$\mathbf{G} = \begin{array}{c} \\ a \\ b \\ c \\ d \\ e \\ f \\ g \end{array} \begin{array}{c} \begin{array}{ccccccc} a & b & c & d & e & f & g \end{array} \\ \begin{bmatrix} 0 & 6 & 5 & 0 & 0 & 0 & 0 \\ 6 & 0 & 9 & 0 & 13 & 0 & 0 \\ 5 & 9 & 0 & 16 & 0 & 12 & 0 \\ 0 & 0 & 16 & 0 & 15 & 7 & 0 \\ 0 &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

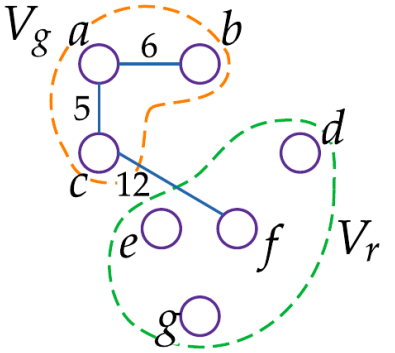
Iteration 1: Edge $ac = 5$



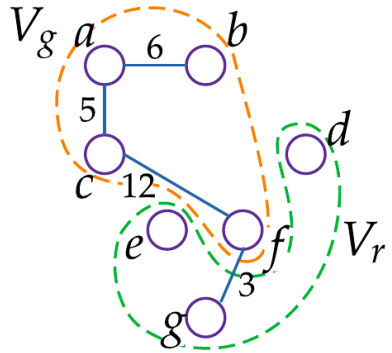
Iteration 2: Edge $ab = 6$



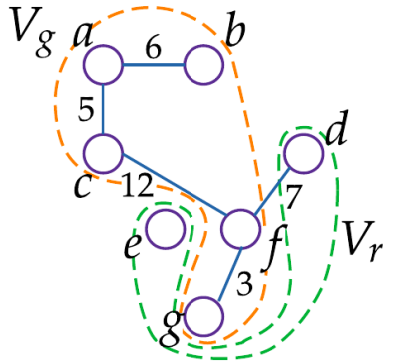
Iteration 3: Edge $cf = 12$



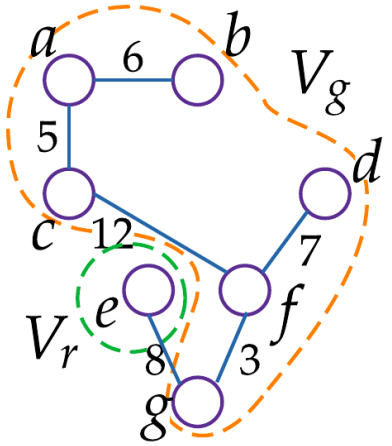
Iteration 4: Edge $fg = 3$

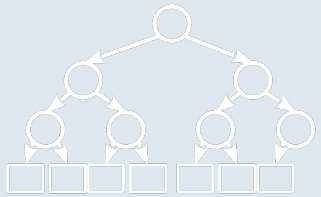


Iteration 5: Edge $fd = 7$

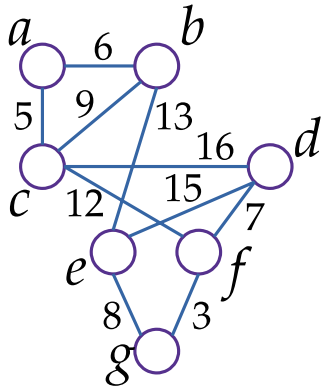


Iteration 6: Edge $eg = 8$



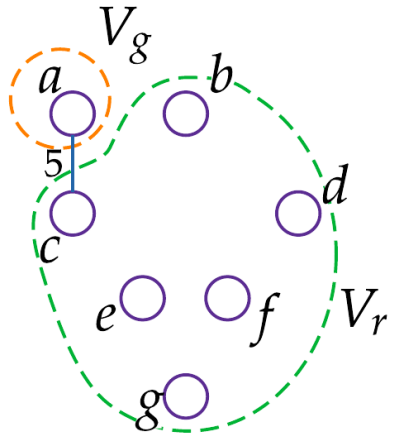


Primov algoritam

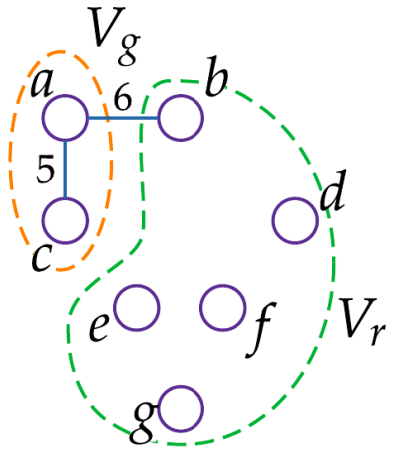


$$\mathbf{G} = \begin{array}{c} \\ a \\ b \\ c \\ d \\ e \\ f \\ g \end{array} \begin{array}{c} \begin{array}{ccccccc} a & b & c & d & e & f & g \end{array} \\ \begin{bmatrix} 0 & 6 & 5 & 0 & 0 & 0 & 0 \\ 6 & 0 & 9 & 0 & 13 & 0 & 0 \\ 5 & 9 & 0 & 16 & 0 & 12 & 0 \\ 0 & 0 & 16 & 0 & 15 & 7 & 0 \\ 0 &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

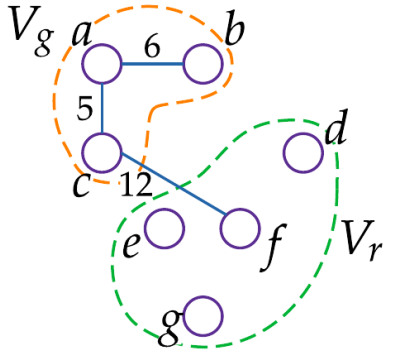
Iteration 1: Edge $ac = 5$



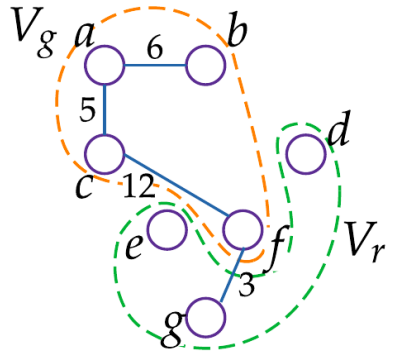
Iteration 2: Edge $ab = 6$



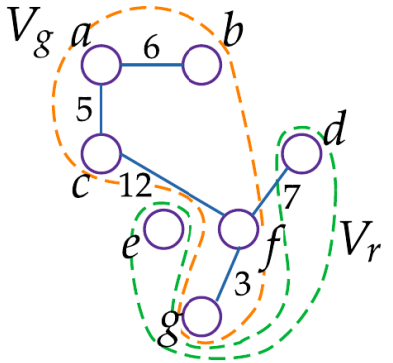
Iteration 3: Edge $cf = 12$



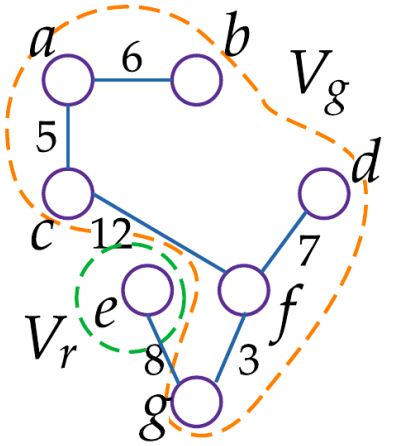
Iteration 4: Edge $fg = 3$



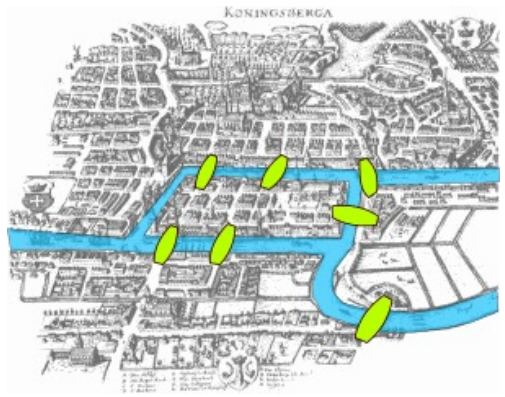
Iteration 5: Edge $fd = 7$



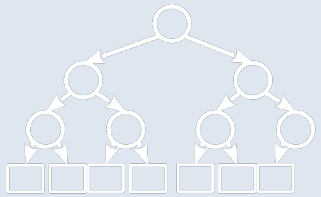
Iteration 6: Edge $eg = 8$



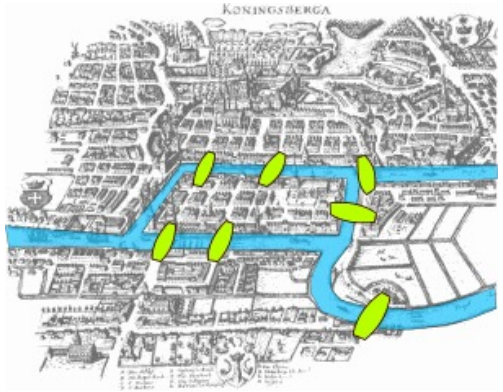
Euler graphs



- Leibniz is the first to propose a branch*position geometry*
- Euler was the first to postulate*positional geometry*on the problem of the seven bridges of Königsberg (today Kaliningrad)
- The problem was to find a route through Königsberg so that each of the seven bridges was crossed only once
  - Euler first saw through the problem because he thought it was trivial and didn't want to waste time on it
  - Later, his observations on this problem gave rise to the theory of graphs
  - Euler proved that the problem is unsolvable, but he also offered a class of problems that are solvable
- Later, on the original problem of the seven bridges of Königsberg, some generic problems are derived, such as the Chinese postman problem

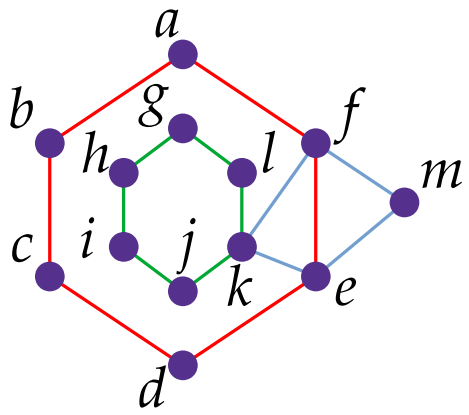


Eulerovi grafovi



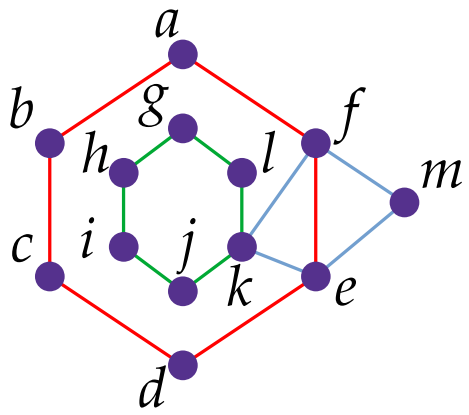
- Leibniz prvi predlaže granu *pozicijske geometrije*
- Euler prvi postulira *pozicijsku geometriju* na problemu sedam mostova Königsberga (danas Kaliningrad)
- Problem je bio pronaći rutu kroz Königsberg tako da se svaki od sedam mostova prijeđe samo jednom
  - Euler je prvo prozreo problem jer je smatrao 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

Euler graphs



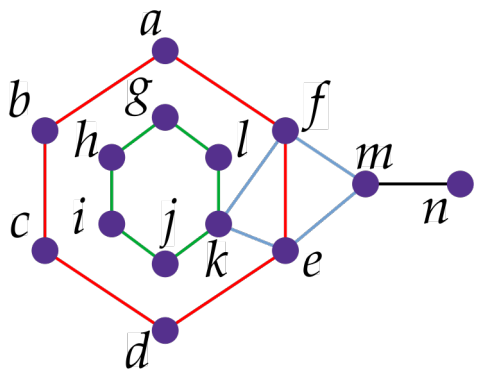
- **Euler trajectory**(*trail*) is a path through the graph that passes each edge of the graph only once
- **Euler's circle**(*circuit*) is an Euler path that starts and ends at the same vertex
- **Euler graph**–is a graph that is made up of an Euler circle or path
- **Theorem**–A connected undirected graph that has all vertices of even degree is an Euler graph
  - **Evidence**
    - Let's imagine that we are doing a tour of the graph. For each entry into the top we use one adjacent edge, while for exiting the top we use another adjacent edge.
    - We can visit each vertex more than once, which means that each vertex of the Euler graph must have*number of tours**2 adjacent edges
    - Thus, the Euler graph is made of the Euler circle

Eulerovi grafovi



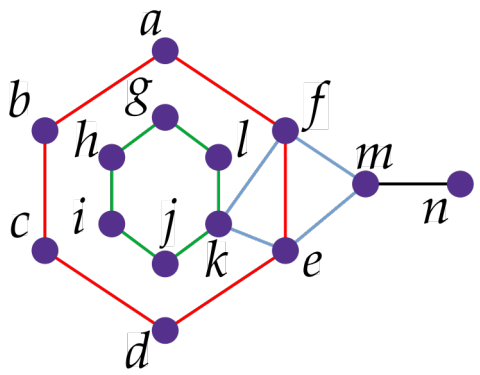
- **Eulerova putanja** (*trail*) – je putanja kroz graf koja svakim bridom grafa prolazi samo jednom
- **Eulerov krug** (*circuit*) – je Eulerova putanja koja počinje i završava u istom vrhu
- **Eulerov graf** – je graf koji je sačinjen od Eulerovog kruga ili putanje
- **Teorem** – Spojeni neusmjereni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

Euler graphs



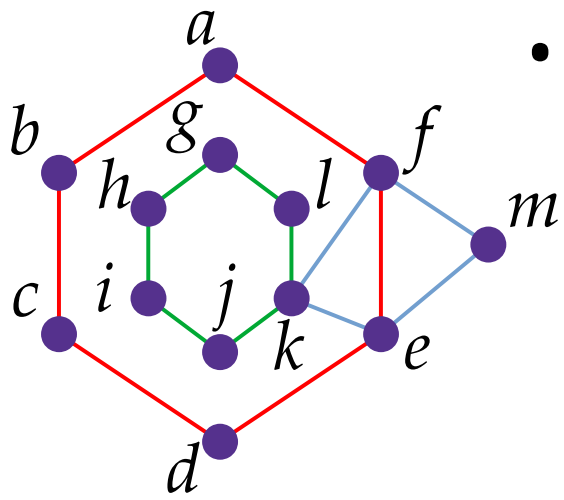
- An Euler graph can also be made of an Euler path that does not have to start and end at the same vertex!?
- Unlike the Euler graph, which is made up of the Euler circle, in this case exactly two vertices in the graph may be of odd degree
  - Regardless, there is a tour that traverses each edge of the graph exactly once

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

Euler circles



- We can see that the Euler graph in the example consists of several non-Eulerian circles

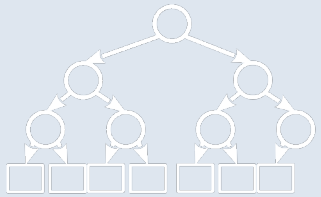
  6= , 7= *h* , 8=

  - These circles share only common vertices, and can be connected through these common vertices
  - So the Euler circle is then

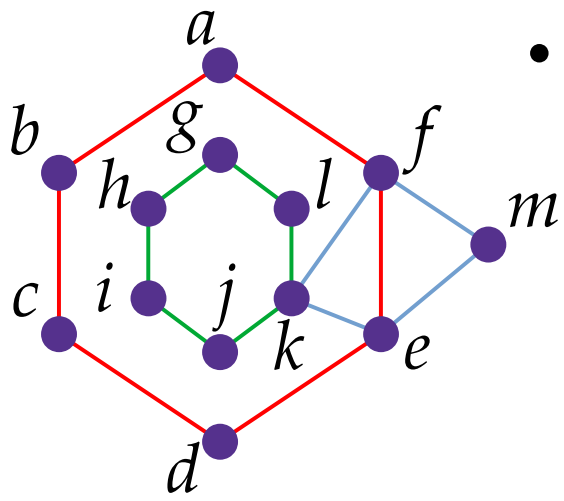
    = !∪ "∪ F

    = *h*

- The idea of detecting Euler circles is based on the traversal of the graph with the removal of edges that have been traversed



Eulerovi krugovi



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

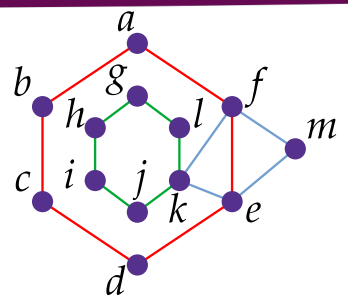
$$C_1 = abcdefa, C_2 = klghijk, C_3 = emfke$$

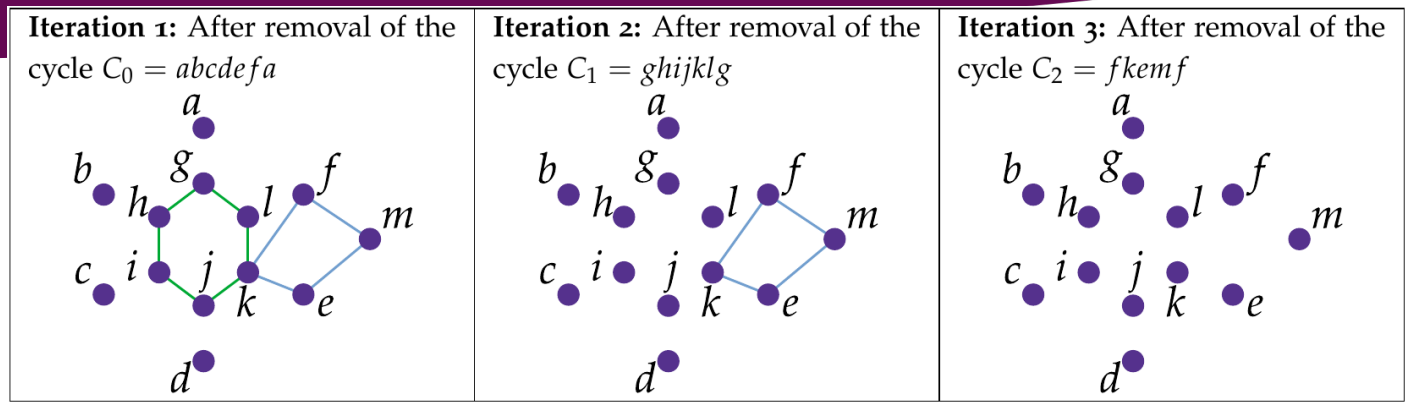
  - Ti krugovi dijele samo zajedničke vrhove, te se mogu povezati kroz te zajedničke vrhove
  - Tako je Eulerov krug onda

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

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

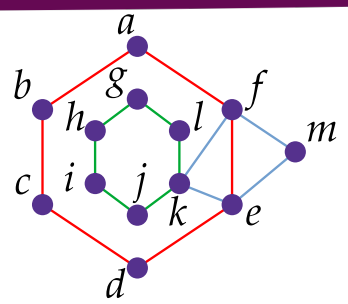
Euler circles

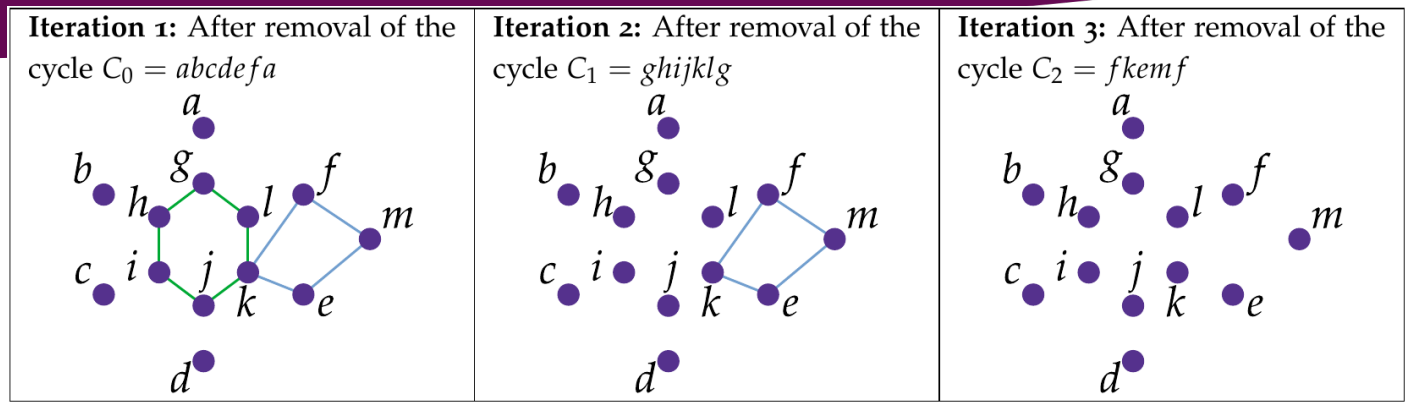




- We go around the graph and remove vertices. At the moment when we are in the initial vertex, we have either an Euler circle or one of the non-Eulerian circles
  - Since the Euler graph is connected, non-Eulerian circles share vertices
  - Shared vertices must have an even degree even after removing the non-Eulerian circle
  - We continue traversing the graph until we can detect more non-Eulerian circles in the graph and until we have removed all edges of the graph
- If we are left with vertices that do not form a non-Eulerian circle, and we still have remaining edges in the graph - then obviously some of the vertices did not have an even degree and it is not possible to find an Euler circle in the graph

Eulerovi krugovi





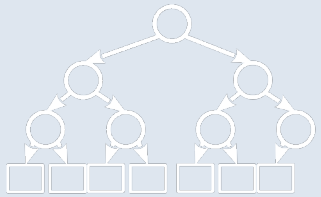
- Obilazimo graf i uklanjamo vrhove. U trenutku kada smo se našli u početnom vrhu, imamo ili Eulerov krug ili jedan od ne-Eulerovih krugova
  - S obzirom da je Eulerov graf povezan, ne-Eulerovi krugovi dijele vrhove
  - Dijeljeni vrhovi moraju imati paran stupanj i nakon uklanjanja ne-Eulerovog kruga
  - 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

Euler circles

```
procedure IsEulerCircuit(G, u)
    while true do
        u_0 ← 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 ≠ u_0 then
            return false
        if there are edges in G then
            pick a vertex u that has incident edges
        else
            return true
```

- The complexity of this algorithm is (+)
- Traversing the graph goes through all the edges, while searching for a new vertex that still has an edge requires going through all the vertices

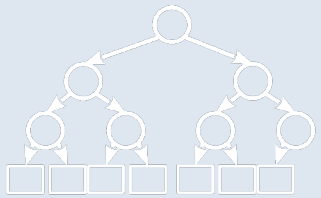
- We start with some arbitrary initial vertex = MR
  - As long as we have the edge which leads from the top we create a circle
    - Remove the edge
    - Top we transfer to the top
  - If we no longer have edges leading from the top although not the initial peak $, then this graph**It's NOT**made of Euler's circle
    - At this point we know that we have either an Euler circuit or one of the non-Eulerian circuits
- If there are still edges, then we select a new starting vertex MRand we're back to circle detection again
- If there are no more edges in the graph, then we have an Euler graph made of an Euler circle!



Eulerovi krugovi

```
procedure IsEulerCircuit(G, u)
    while true do
        u_0 ← 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 ≠ u_0 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



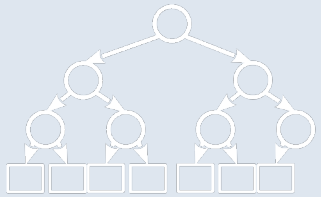
- Započinjemo nekim proizvoljnim početnim vrhom $u = u_0$
  - Tako dugo dok imamo brid uv koji vodi iz vrha u stvaramo krug
    - Ukloni brid uv
    - Vrh v prebacujemo u vrh u
  - Ako više nemamo bridova koji vode iz vrha u i ako u nije početni vrh u_0, tada ovaj graf **NIJE** sačinjen od Eulerovog 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!

Hierholzer's algorithm

```
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
```

- The complexity of this algorithm is ( )

- Taking advantage of the fact that the Euler graph is connected, we can use a stack to avoid blindly searching all vertices for a new edge
  - As we progress around the circle, we put the tops on the stack
  - When we reach the end of the circuit, we return to the visited vertices by taking them from the stack
    - If the previous circle was Euler, then we return back to the initial vertex
    - If the previous circle was not Euler, and given that the Euler graph is connected, on the way back we will encounter at least one vertex that still has edges
      - We start along that edge, progressing through a new non-Eulerian circle.

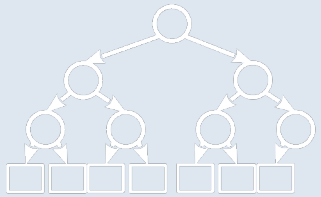


Hierholzerov algoritam

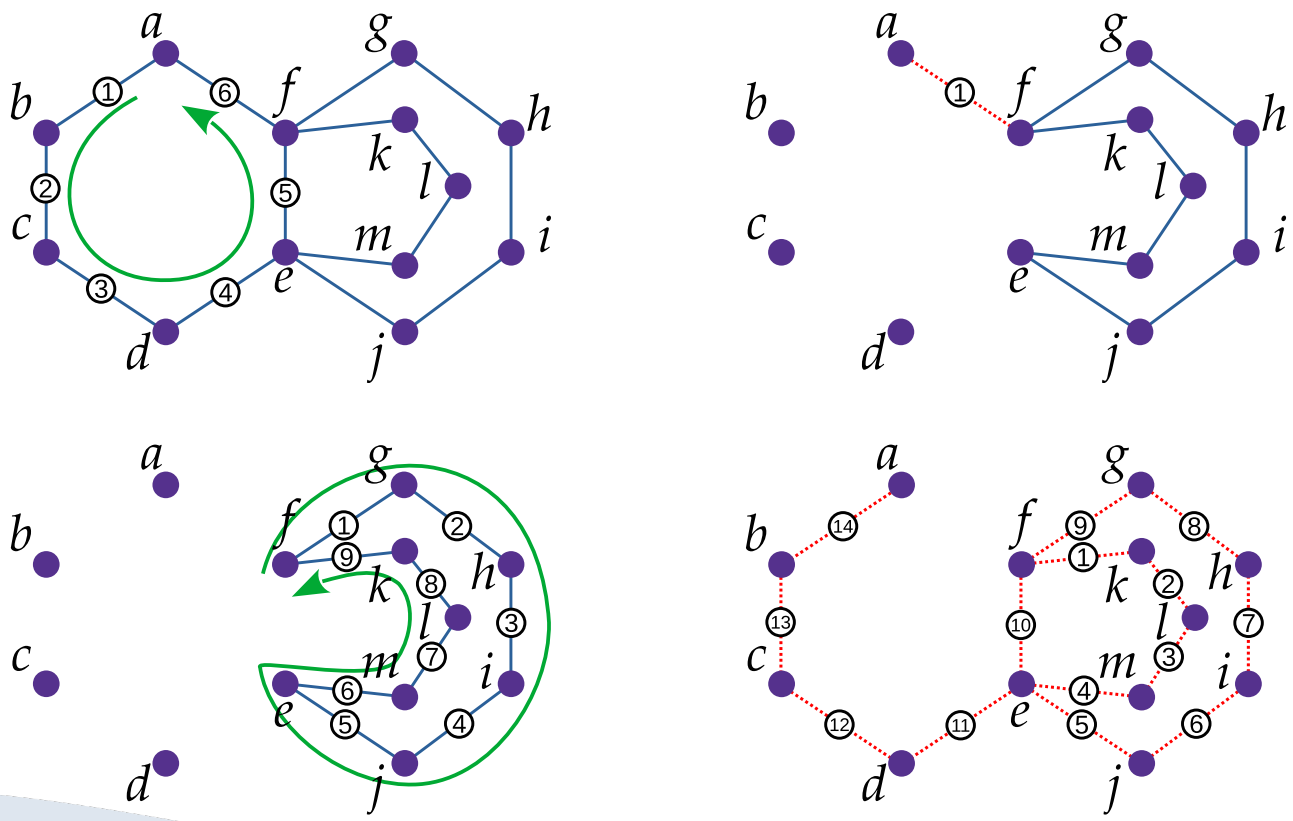
```
procedure HIERHOLZER(G, u)
    s ← new empty stack
    cycle ← ∅
    s.push(u)
    while s not empty do
        u ← last element on the stack s
        if u has adjacent vertices then
            v ← one of the adjacent vertices of u
            s.push(v)
            remove edge uv from G
      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



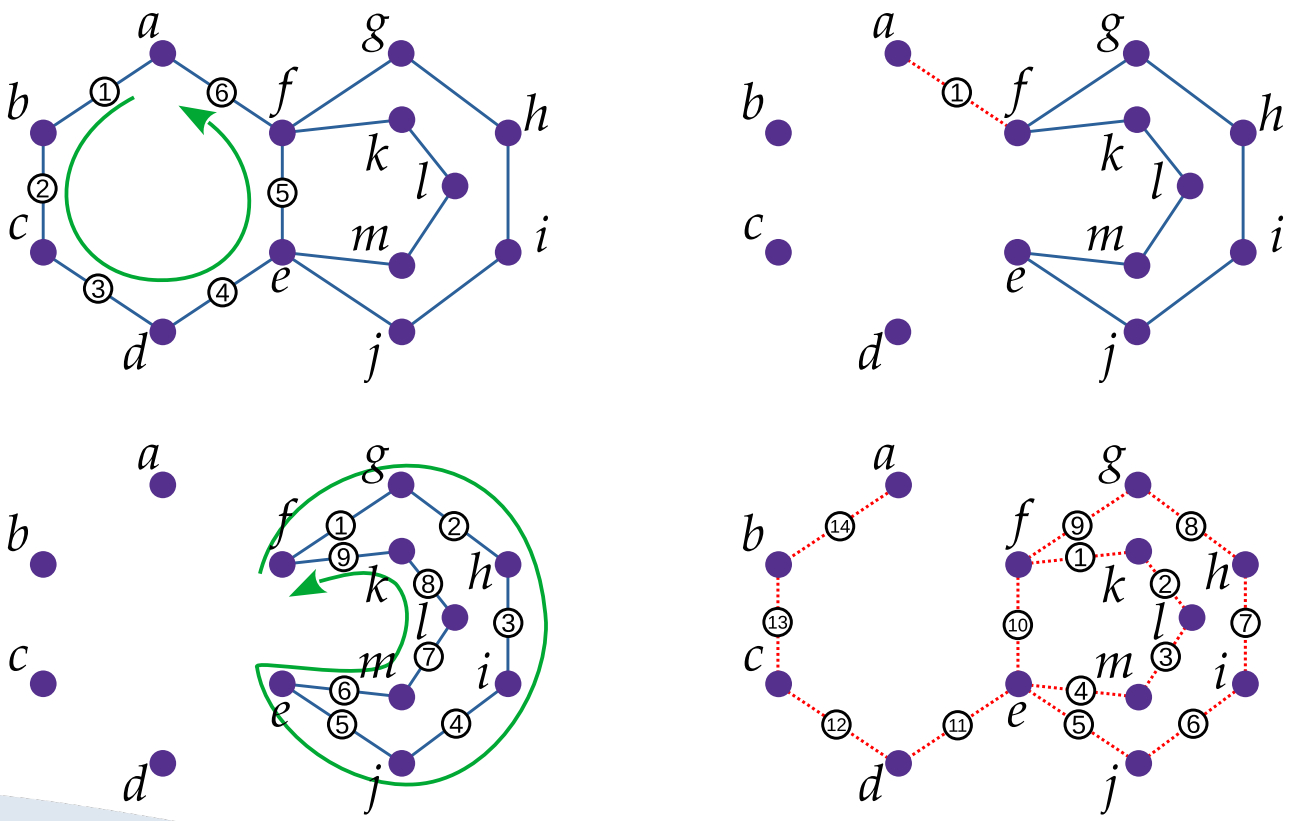
Hierholzer's algorithm



- The ultimate Euler circle is

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

Hierholzerov algoritam



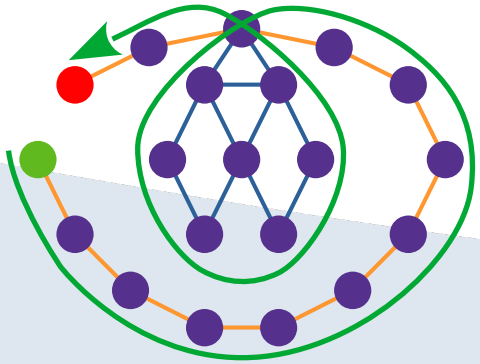
- Krajnji Eulerov krug je

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

Fleury's algorithm

```
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
```

- The complexity of this algorithm is ( )
- However, the detection of whether an edge is a bridge or not raises the complexity of the algorithm to ("), which is slower than the Hierholzer algorithm

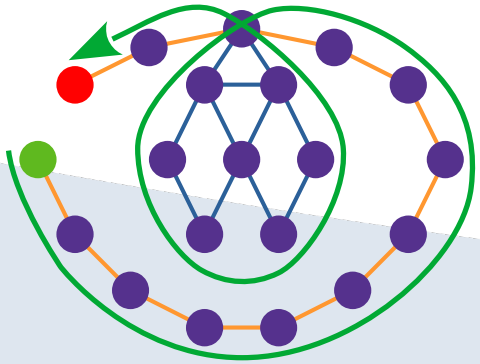


- The concept of Fleury's algorithm is based on the selection of the next edge by which we move along the Euler circle
  - If an edge that is not a bridge leads from the current vertex (*bridge*), then we select it
  - Only when there is only a bridge leading from the top, we move along it
  - Let's remember – a bridge is an edge whose removal makes the graph disconnected
    - This can happen when we move along the Euler graph
    - If we have edges left in both partitions of the graph, we cannot go back
    - In this way, we will not detect the Euler circle
- Everything is based on instruction (*lemma*), by which we determine that by removing the first edge on the initial vertex, the degree of that vertex becomes odd and thus it becomes the last vertex that we will visit

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