

Grafovi

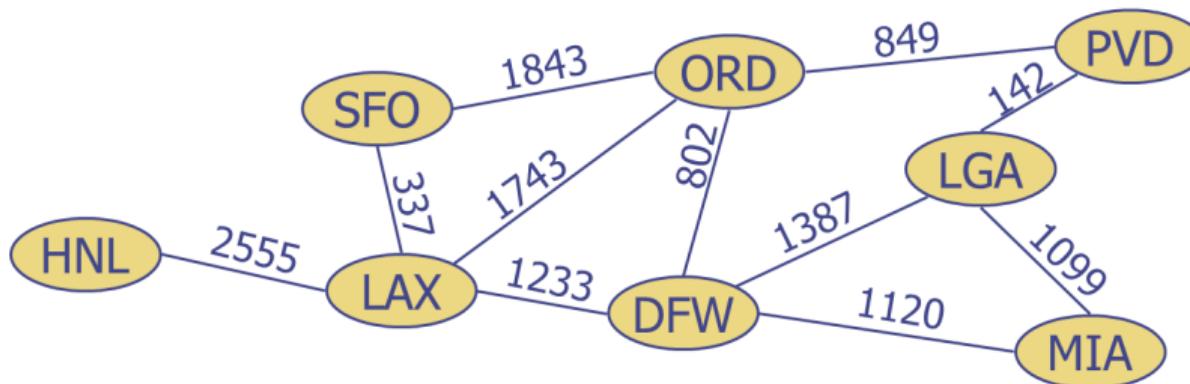
© Goodrich, Tamassia, Goldwasser

Katedra za informatiku, Fakultet tehničkih nauka, Univerzitet u Novom Sadu

2023.

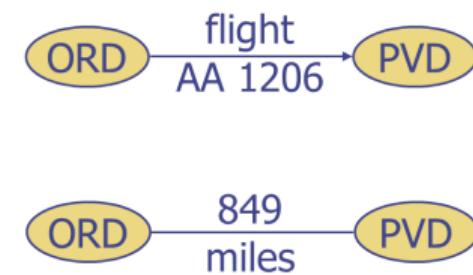
Pojam grafa

- **graf** je par (V, E)
 - V je set čvorova (*vertices*)
 - E je skup grana (*edges*)
 - čvorovi i grane čuvaju elemente
- primer:
 - čvorovi predstavljaju aerodrome i čuvaju šifre aerodroma
 - grane predstavljaju letove između aerodroma i čuvaju dužinu puta



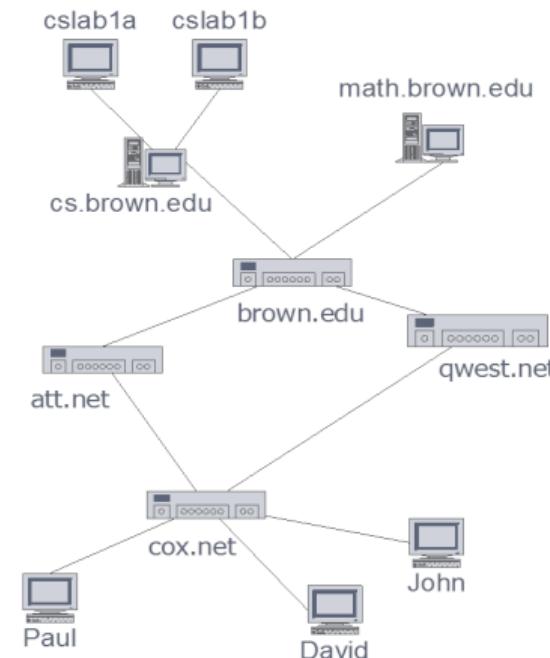
Vrste grana

- usmerena grana
 - uređeni par čvorova (u, v)
 - prvi čvor u je polazište
 - drugi čvor v je odredište
 - npr. konkretan let aviona
- neusmerena grana
 - neuređeni par čvorova (u, v)
 - npr. putanja leta
- usmereni graf
 - sve grane su usmerene
 - npr. mreža letova
- neusmereni graf
 - sve grane su neusmerene
 - npr. mreža putanja



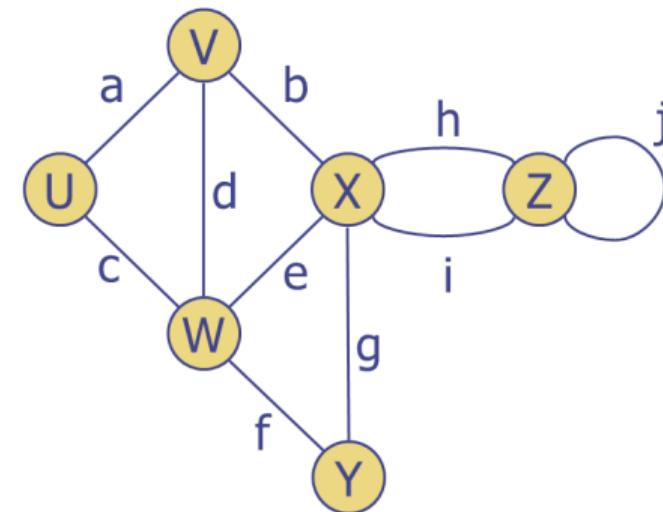
Primene

- elektronska kola
 - štampane ploče
 - integrisana kola
- transportne mreže
 - putevi
 - avionski letovi
- računarske mreže
 - lokalne mreže
 - Internet
- baze podataka
 - ER dijagrami



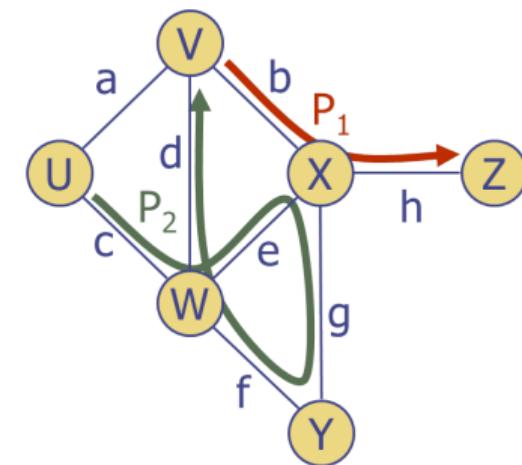
Terminologija 1

- krajevi grane
 - U i V su krajevi a
- grane incidentne na čvoru
 - a , d i b su incidentni na V
- susedni čvorovi
 - povezani granom
 - U i V su susedni
- stepen čvora
 - broj grana kojima je on kraj
 - stepen of X je 5
- paralelne grane: h i i
- petlja: j



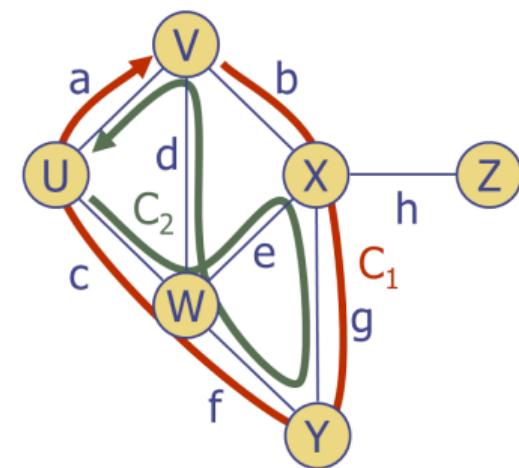
Terminologija 2

- putanja
 - sekvenca naizmenično čvorova i grana
 - počinje sa čvorom
 - završava sa čvorom
 - svakoj grani prethodi i sledi njen kraj
- prosta putanja
 - svi čvorovi i grane su različiti
- primeri
 - $P_1 = (V, b, X, h, Z)$ je prosta
 - $P_2 = (U, c, W, e, X, g, Y, f, W, d, V)$ nije prosta



Terminologija 3

- petlja
 - cirkularna sekvenca naizmenično čvorova i grana
 - svakoj grani prethodi i sledi njen kraj
- prosta petlja
 - s vi čvorovi i grane su različiti
- primeri
 - $C_1 = (V, b, X, g, Y, f, W, c, U, a, V)$ je prosta
 - $C_2 = (U, c, W, e, X, g, Y, f, W, d, V, a, U)$ nije prosta



Osobine

- suma stepena čvorova

$$\sum_v \deg(v) = 2m$$

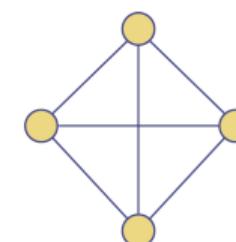
(svaka grana se broji dva puta)

- u neusmerenom grafu bez petlji i višestrukih grana

$$m < n(n - 1)/2$$

(svaki čvor ima stepen najviše $n - 1$)

- n – broj čvorova
- m – broj grana
- $\deg(v)$ – stepen čvora



- $n = 4$
- $m = 6$
- $\deg(v) = 3$

Čvorovi i grane

- graf je kolekcija čvorova i grana
- prikazaćemo ga pomoću tri tipa: Vertex, Edge i Graph
- Vertex je „lagani“ objekat koji čuva sadržaj (npr. šifru aerodroma)
 - ima metodu `element()` kojom se može dobiti taj sadržaj
- Edge čuva sadržaj (npr. broj leta, rastojanje)
 - `element()` vraća taj sadržaj
 - `endpoints()` vraća par (u, v) polazište i odredište
 - `opposite(v)` vraća suprotni kraj grane

Graf ATP

<code>vertex_count()</code>	broj čvorova
<code>vertices()</code>	lista svih čvorova
<code>edge_count()</code>	broj grana
<code>edges()</code>	lista svih grana
<code>get_edge(u,v)</code>	vraća granu između u i v ako postoji, inače None
<code>degree(v,out=True)</code>	broj izlaznih/ulaznih grana iz v
<code>incident_edges(v,out=True)</code>	lista izlaznih/ulaznih grana iz v
<code>insert_vertex(x=None)</code>	dodaj novi čvor sa sadržajem x
<code>insert_edge(u,v,x=None)</code>	dodaj novu granu od u ka v sa sadrža- jem x
<code>remove_vertex(v)</code>	ukloni čvor v i sve vezane grane
<code>remove_edge(e)</code>	ukloni granu e

Implementacija₁: Lista čvorova i lista grana

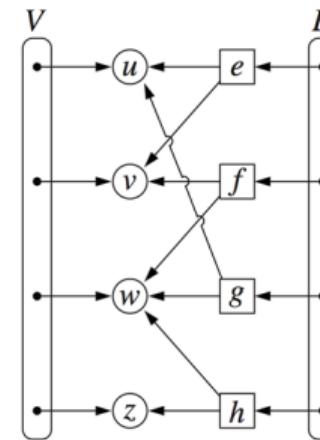
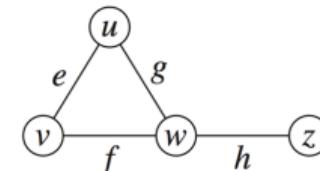
- Vertex

- čuva sadržaj
- element liste

- Edge

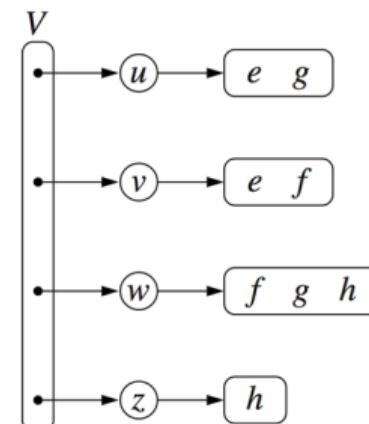
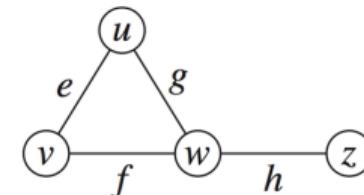
- čuva sadržaj
- referenca na polazište
- referenca na odredište
- element liste

- lista čvorova
- lista grana



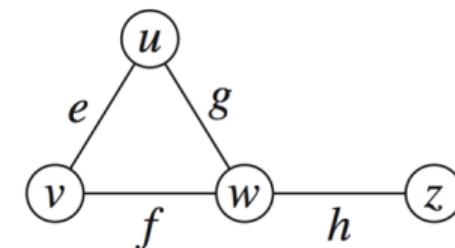
Implementacija₂: Lista suseda

- lista grana za svaki čvor
- grane mogu imati reference na drugu pojavu iste grane (za drugi krajnji čvor)



Implementacija₃: Matrica incidencije

- svakom čvoru dodeljen int ključ
- matrica sadrži referencu na granu ukoliko ona povezuje dva čvora, ili None
- ili samo 0/1 u matrici



	0	1	2	3	
u	→	0	e	g	
v	→	1	e	f	
w	→	2	g	f	h
z	→	3			h

Performanse

n čvorova, m grana, bez paralelnih grana, bez petlji

	lista grana	lista suseda	matrica incidencije
<i>prostor</i>	$n + m$	$n + m$	n^2
<i>incident_edges(v)</i>	m	$\deg(v)$	n
<i>are_adjacent(v, w)</i>	m	$\min\{\deg(v), \deg(w)\}$	1
<i>insert_vertex(v)</i>	1	1	n^2
<i>insert_edge(v, w, e)</i>	1	1	1
<i>remove_vertex(v)</i>	m	$\deg(v)$	n^2
<i>remove_edge(e)</i>	1	1	1

Python implementacija

- koristićemo mapu susedstva
- za čvor v čuvamo Python rečnik sa susedima $I(v)$
- listu čvorova zamenićemo rečnikom D koji mapira svaki čvor na njegovu mapu suseda $I(v)$
 - možemo proći kroz sve čvorove sa $D.keys()$
- čvor ne mora da čuva svoj položaj u D jer se to izračuna za $O(1)$
- performanse iste kao za listu suseda ali u **očekivanom** slučaju

Klasa Vertex

```
class Vertex:  
    """Lightweight vertex structure for a graph."""  
    __slots__ = '_element'  
  
    def __init__(self, x):  
        """Do not call constructor directly.  
        Use Graph's insert_vertex(x).  
        """  
        self._element = x  
  
    def element(self):  
        """Return element associated with this vertex."""  
        return self._element  
  
    def __hash__(self): # will allow vertex to be a map/set key  
        return hash(id(self))  
  
    def __str__(self):  
        return str(self._element)
```

Klasa Edge

```
class Edge:  
    __slots__ = '_origin', '_destination', '_element'  
  
    def __init__(self, u, v, x):  
        self._origin = u  
        self._destination = v  
        self._element = x  
  
    def endpoints(self):  
        return (self._origin, self._destination)  
  
    def opposite(self, v):  
        if not isinstance(v, Graph.Vertex):  
            raise TypeError('v must be a Vertex')  
        return self._destination if v is self._origin else self._origin  
        raise ValueError('v not incident to edge')  
  
    def element(self):  
        return self._element  
  
    def __hash__(self): # will allow edge to be a map/set key  
        return hash( (self._origin, self._destination) )  
  
    def __str__(self):  
        return '({0},{1},{2})'.format(self._origin, self._destination, self._element)
```

Klasa Graph 1

```
class Graph:  
    def __init__(self, directed=False):  
        self._outgoing = {}  
        self._incoming = {} if directed else self._outgoing  
  
    def _validate_vertex(self, v):  
        if not isinstance(v, self.Vertex):  
            raise TypeError('Vertex expected')  
        if v not in self._outgoing:  
            raise ValueError('Vertex does not belong to this graph.')  
  
    def is_directed(self):  
        return self._incoming is not self._outgoing  
  
    def vertex_count(self):  
        return len(self._outgoing)  
  
    def vertices(self):  
        return self._outgoing.keys()
```

Klasa Graph 2

```
def edge_count(self):
    total = sum(len(self._outgoing[v]) for v in self._outgoing)
    return total if self.is_directed() else total // 2

def edges(self):
    result = set() # avoid double-reporting edges of undirected graph
    for secondary_map in self._outgoing.values():
        result.update(secondary_map.values()) # add edges to resulting set
    return result

def get_edge(self, u, v):
    self._validate_vertex(u)
    self._validate_vertex(v)
    return self._outgoing[u].get(v) # returns None if v not adjacent

def degree(self, v, outgoing=True):
    self._validate_vertex(v)
    adj = self._outgoing if outgoing else self._incoming
    return len(adj[v])
```

Klasa Graph 3

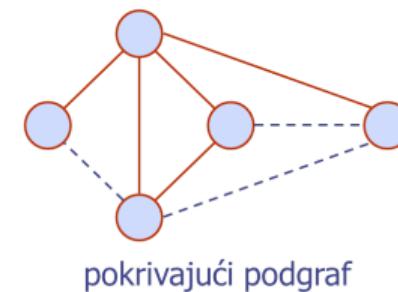
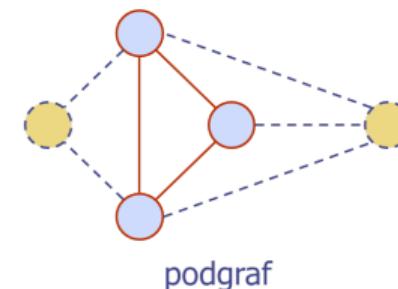
```
def incident_edges(self, v, outgoing=True):
    self._validate_vertex(v)
    adj = self._outgoing if outgoing else self._incoming
    for edge in adj[v].values():
        yield edge

def insert_vertex(self, x=None):
    v = self.Vertex(x)
    self._outgoing[v] = {}
    if self.is_directed():
        self._incoming[v] = {} # need distinct map for incoming edges
    return v

def insert_edge(self, u, v, x=None):
    if self.get_edge(u, v) is not None: # includes error checking
        raise ValueError('u and v are already adjacent')
    e = self.Edge(u, v, x)
    self._outgoing[u][v] = e
    self._incoming[v][u] = e
```

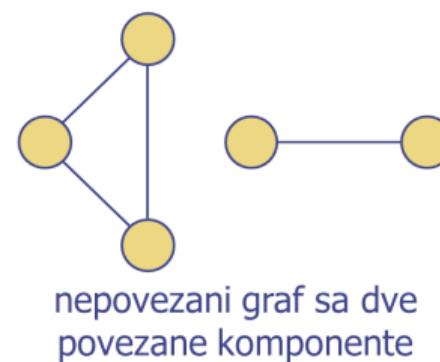
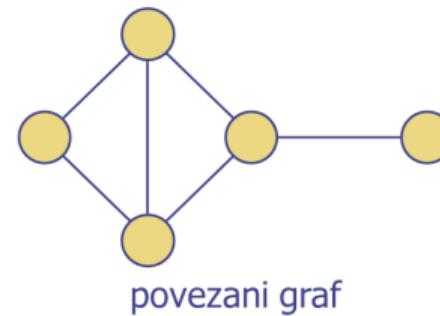
Podgraf

- **podgraf** S grafa G je graf takav da
 - čvorovi S su podskup čvorova G
 - grane S su podskup grana G
- **pokrivajući podgraf** sadrži sve čvorove G



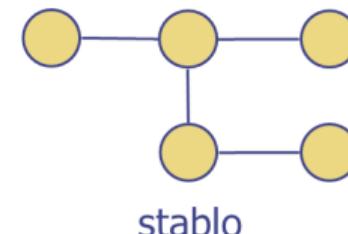
Povezanost

- graf je **povezan** ako postoji putanja između svaka dva čvora
- **povezana komponenta** je maksimalni povezani podgraf

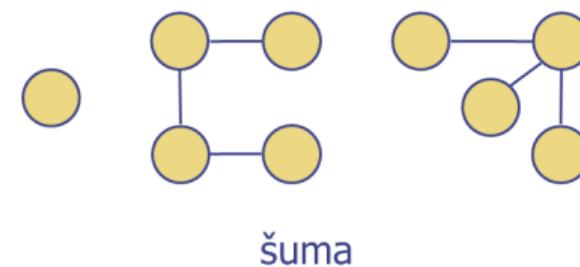


Stablo i šuma

- **stablo** je neusmereni graf T takav da
 - T je povezan
 - T nema petlje
- **šuma** je neusmereni graf takav da
 - nema petlje
 - povezane komponente su stabla



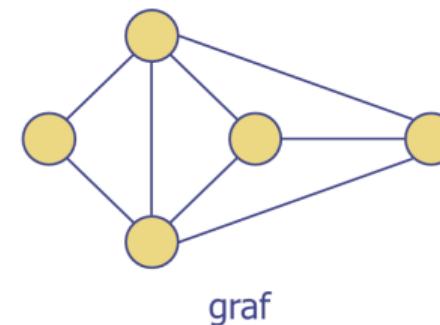
stablo



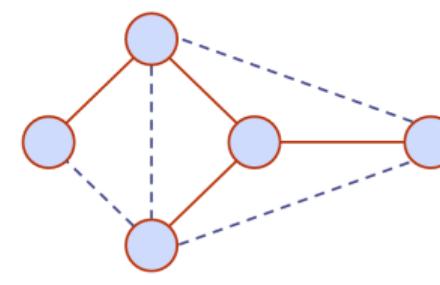
šuma

Pokrivaće stablo i šuma

- **pokrivaće stablo** je podgraf koji
 - je stablo i
 - pokriva graf
- pokrivaće stablo nije jedinstveno ako graf nije stablo
- **pokrivaća šuma** je podgraf koji
 - je šuma i
 - pokriva graf



graf



pokrivaće stablo

Obilazak grafa po dubini

- depth-first-search (**DFS**) je opšti metod za obilazak grafa
 - obilazi sve čvorove i grane
 - određuje da li je graf povezan
 - određuje povezane komponente grafa
 - određuje pokrivajuću šumu grafa
- DFS na grafu sa n čvorova i m grana traje $O(n + m)$
- može se proširiti za rešavanje drugih problema
 - nađi putanju između dva čvora
 - nađi petlju u grafu
- DFS je za graf isto što i Ojlerov obilazak za binarna stabla

DFS algoritam

dodeljuje oznake (labele) čvorovima i granama

DFS(G)

Input: graf G

Output: oznake na granama: DISCOVERY ili BACK

```

for all  $u \in G.\text{vertices}()$  do
    setLabel( $u$ , UNEXPLORED)
for all  $e \in G.\text{edges}()$  do
    setLabel( $e$ , UNEXPLORED)
for all  $v \in G.\text{vertices}()$  do
    if label( $v$ ) = UNEXPLORED then
        DFS( $G, v$ )
    
```

DFS(G, v)

Input: graf G i početni čvor v

Output: oznake na granama: DISCOVERY ili BACK

```

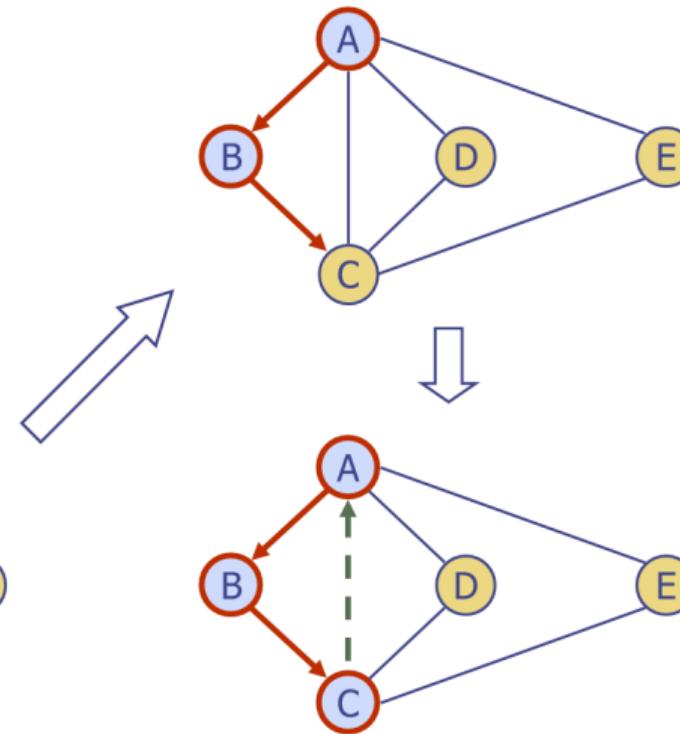
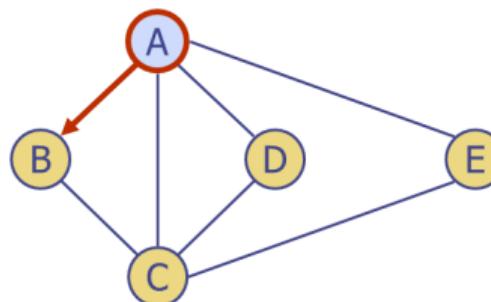
setLabel( $v$ , VISITED)
for all  $e \in G.\text{incidentEdges}(v)$  do
    if label( $e$ ) = UNEXPLORED then
         $w \leftarrow \text{opposite}(v, e)$ 
        if label( $w$ ) = UNEXPLORED then
            setLabel( $e$ , DISCOVERY)
            DFS( $G, w$ )
        else
            setLabel( $e$ , BACK)
    
```

DFS u Pythonu

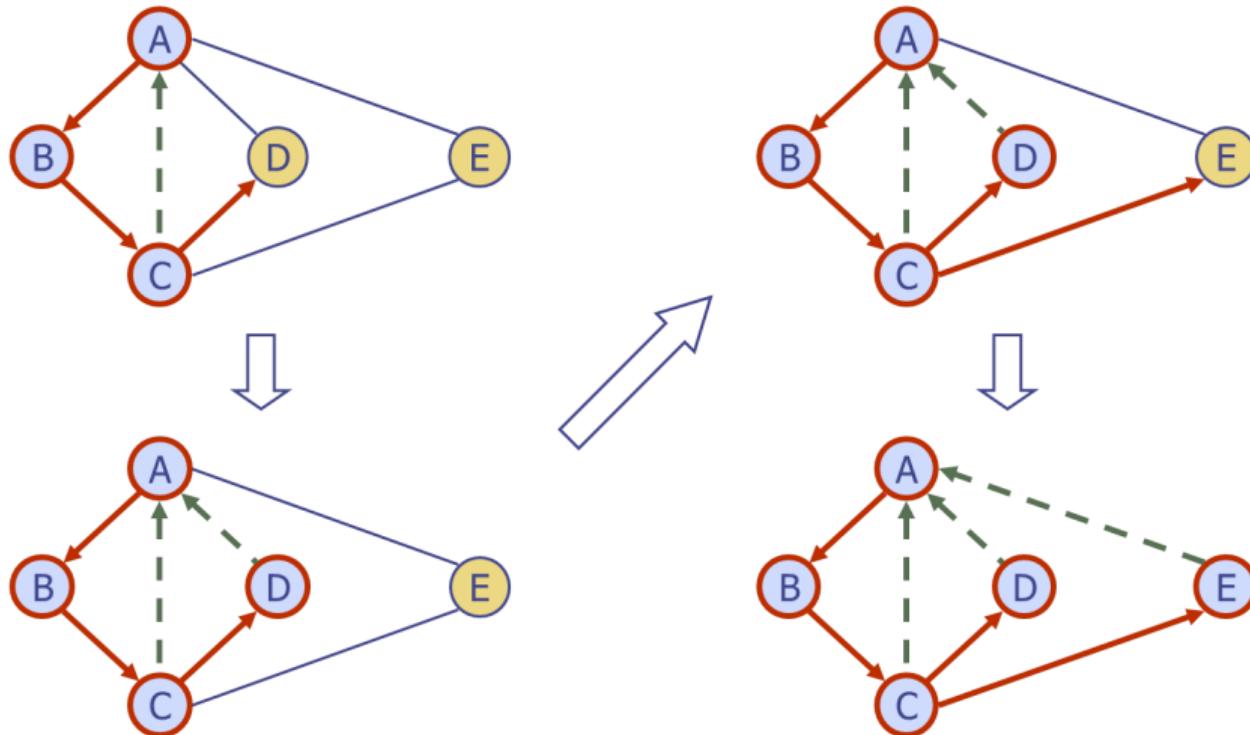
```
def DFS(g, u, discovered):
    for e in g.incident_edges(u): # for every outgoing edge from u
        v = e.opposite(u)
        if v not in discovered: # v is an unvisited vertex
            discovered[v] = e      # e is the tree edge that discovered v
            DFS(g, v, discovered)  # recursively explore from v
```

DFS primer

-  unexplored čvor
-  visited čvor
- unexplored grana
- discovery grana
- - - → back grana

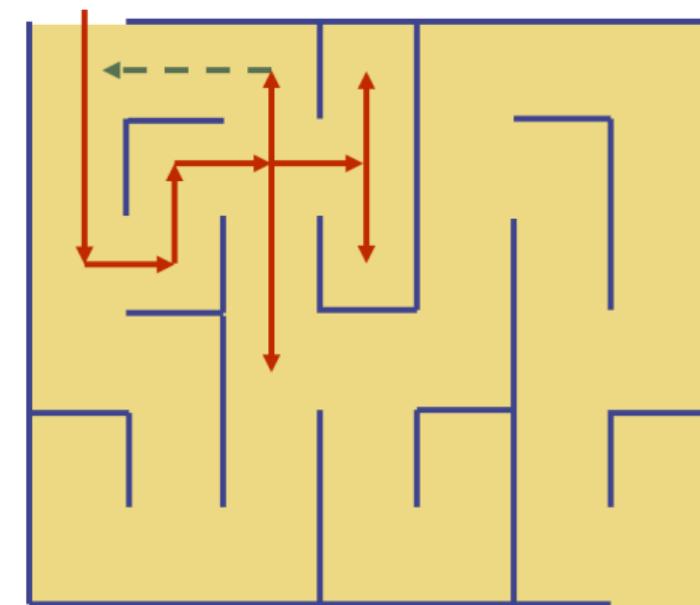


DFS primer (još)



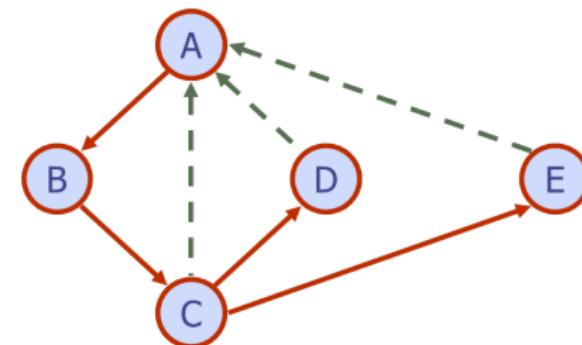
DFS i prolazak labyrintha

- svaku raskrsnicu, ugao (skretanje) i kraj puta označimo kao posećen čvor
- svaki hodnik (granu) kao posećenu
- pamtimo odakle smo počeli pomoću steka rekurzije



Osobine DFS

- 1 $\text{DFS}(G, v)$ obilazi sve čvorove i grane u povezanoj komponenti od v
- 2 grane označene kao DISCOVERY čine pokrivajuće stablo povezane komponente od v



Performanse DFS

- stavljanje labele na čvor/granu traje $O(1)$
- svaki čvor se označi **dva** puta, jednom kao UNEXPLORED, drugi put kao VISITED
- svaka grana se označi **dva** puta, jednom kao UNEXPLORED, drugi put kao DISCOVERY ili BACK
- metoda `incident_edges` se poziva jednom za svaki čvor
- DFS traje $O(n + m)$ ako je graf predstavljen listom suseda

$$\sum_v \deg(v) = 2m$$

DFS za pronalaženje putanje

- DFS se može upotrebiti za pronalaženje **putanje** između dva čvora u i z
- pozivamo DFS sa u kao početnim čvorom
- discovery grane definišu pokrivajuće stablo: postoji putanja od u do z
- na steku S čuvamo grane od početnog do tekućeg čvora
- kada dođemo do z na steku je cela putanja

```

pathDFS( $G, v, z$ )
setLabel( $v$ , VISITED)
if  $v = z$  then
    return  $S$ .elements()
for all  $e \in G.\text{incidentEdges}(v)$  do
    if label( $e$ ) = UNEXPLORED then
         $w \leftarrow \text{opposite}(v, e)$ 
        if label( $w$ ) = UNEXPLORED then
            setLabel( $e$ , DISCOVERY)
             $S$ .push( $e$ )
            pathDFS( $G, w$ )
             $S$ .pop( $e$ )
        else
            setLabel( $e$ , BACK)
    
```

DFS za pronalaženje petlje

- DFS se može upotrebiti za pronalaženje **petlje**
- na steku S čuvamo putanju od početnog do tekućeg čvora
- čim nađemo na BACK granu (v, w) vraćamo petlju kao sadržaj steka od vrha do čvora w

```

cycleDFS( $G, v$ )
setLabel( $v$ , VISITED)
for all  $e \in G.\text{incidentEdges}(v)$  do
    if label( $e$ ) = UNEXPLORED then
         $w \leftarrow \text{opposite}(v, e)$ 
         $S.\text{push}(e)$ 
        if label( $w$ ) = UNEXPLORED then
            setLabel( $e$ , DISCOVERY)
            pathDFS( $G, w$ )
             $S.\text{pop}(e)$ 
        else
             $T \leftarrow \text{new Stack}$ 
            repeat
                 $o \leftarrow S.\text{pop}()$ 
                 $T.\text{push}(o)$ 
            until  $o = w$ 
return  $T.\text{elements}()$ 

```

Obilazak grafa po širini

- breadth-first-search (**BFS**) je opšti metod za obilazak grafa
 - obilazi sve čvorove i grane
 - određuje da li je graf povezan
 - određuje povezane komponente grafa
 - određuje pokrivajuću šumu grafa
- BFS na grafu sa n čvorova i m grana traje $O(n + m)$
- može se proširiti za rešavanje drugih problema
 - nađi najkraću putanju između dva čvora
 - nađi prostu petlju

BFS algoritam

```
BFS( $G$ )
for all  $u \in G.\text{vertices}()$  do
    setLabel( $u$ , UNEXPLORED)
for all  $e \in G.\text{edges}()$  do
    setLabel( $e$ , UNEXPLORED)
for all  $v \in G.\text{vertices}()$  do
    if label( $v$ ) = UNEXPLORED
    then
        BFS( $G, v$ )
```

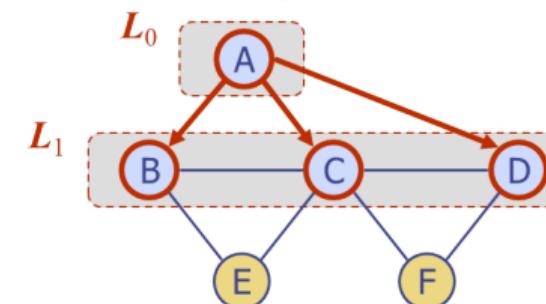
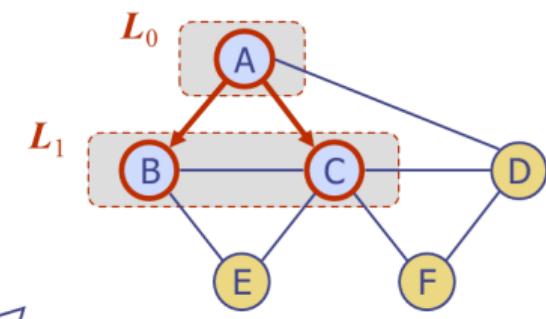
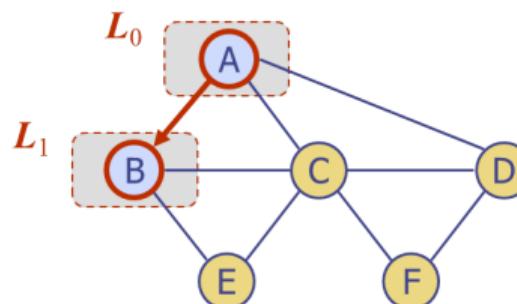
```
BFS( $G, s$ )
 $L_0 \leftarrow []$ 
 $L_0.\text{addLast}(s)$ 
setLabel( $s$ , VISITED)
 $i \leftarrow 0$ 
while  $\neg L_i.\text{isEmpty}()$  do
     $L_{i+1} \leftarrow []$ 
    for all  $v \in L_i.\text{elements}()$  do
        for all  $e \in G.\text{incidentEdges}(v)$  do
            if label( $e$ ) = UNEXPLORED then
                 $w \leftarrow \text{opposite}(v, e)$ 
                if label( $w$ ) = UNEXPLORED then
                    setLabel( $e$ , DISCOVERY)
                    setLabel( $w$ , VISITED)
                     $L_{i+1}.\text{addLast}(w)$ 
                else
                    setLabel( $e$ , CROSS)
     $i \leftarrow i + 1$ 
```

BFS u Pythonu

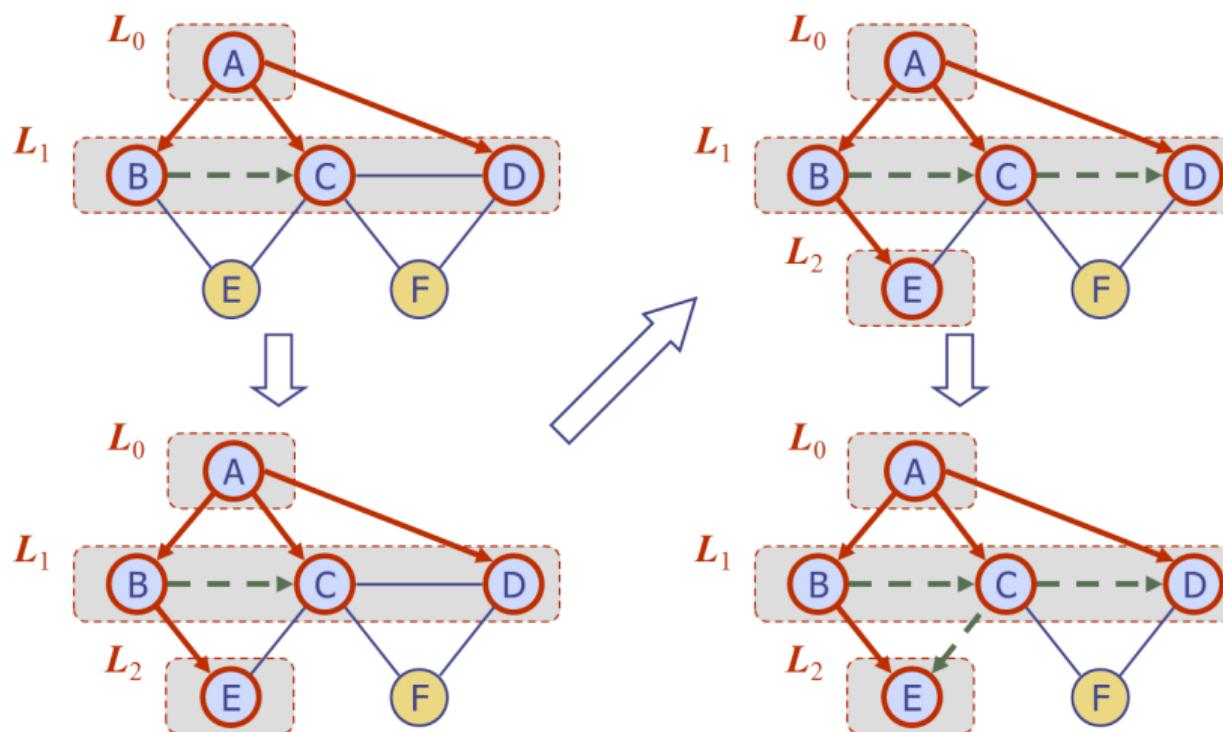
```
def BFS(g, s, discovered):
    level = [s]          # first level includes only s
    while len(level) > 0:
        next_level = []  # prepare to gather newly found vertices
        for u in level:
            for e in g.incident_edges(u): # for every outgoing edge from u
                v = e.opposite(u)
                if v not in discovered: # v is an unvisited vertex
                    discovered[v] = e    # e is the tree edge that discovered v
                    next_level.append(v) # v will be considered in next pass
        level = next_level         # relabel 'next' level to become current
```

BFS primer

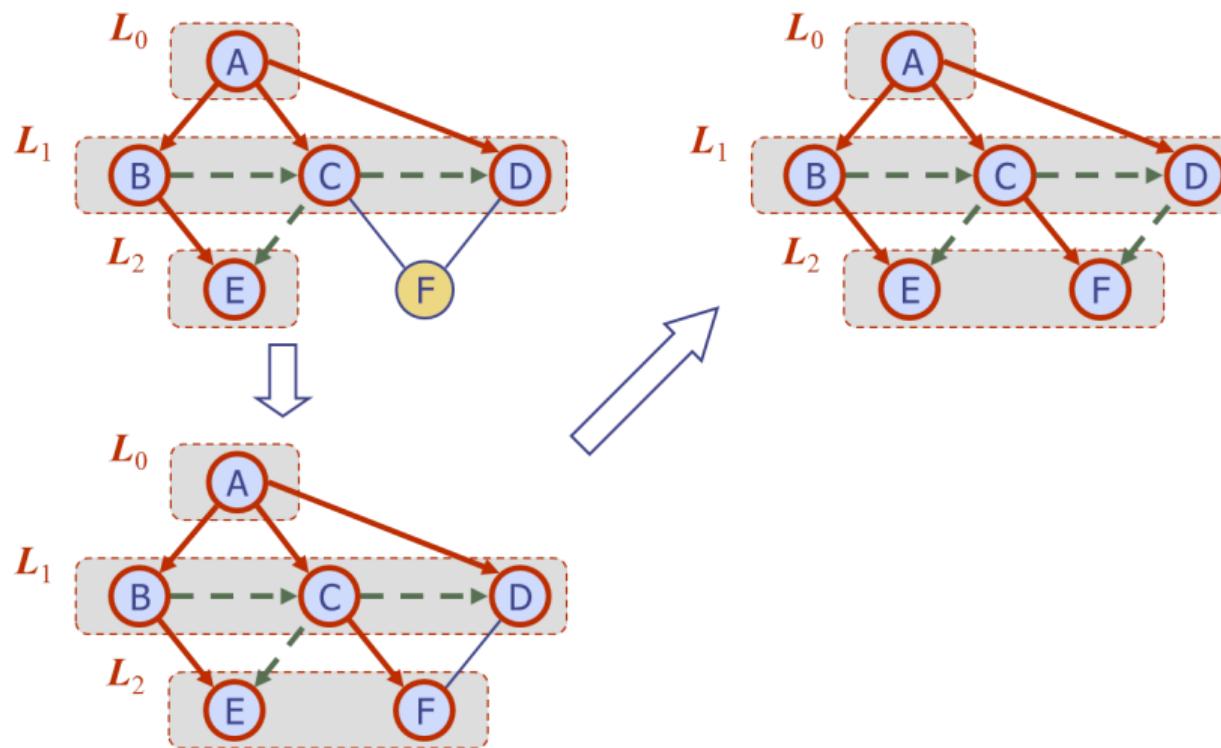
-  unexplored čvor
-  visited čvor
- Unexplored grana
- discovery grana
- - -> cross grana



BFS primer (još)



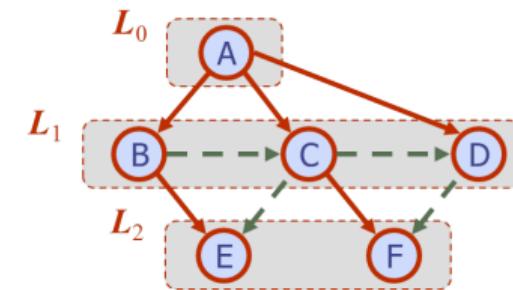
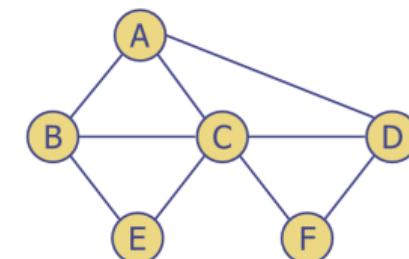
BFS primer (još još)



Osobine BFS

G_s : povezana komponenta od s

- 1 $\text{BFS}(G, v)$ obilazi sve čvorove i grane od G_s
- 2 grane označene kao DISCOVERY čine pokrivajuće stablo T_s za G_s
- 3 za svaki čvor u L_i
 - putanja iz T_s od s do v ima i čvorova
 - svaka putanja od s do v u G_s ima bar i čvorova



Performanse BFS

- stavljanje labele na čvor/granu traje $O(1)$
- svaki čvor se označi **dva** puta, jednom kao UNEXPLORED, drugi put kao VISITED
- svaka grana se označi **dva** puta, jednom kao UNEXPLORED, drugi put kao DISCOVERY ili CROSS
- svaki čvor se jednom dodaje u sekvencu L_i
- metoda `incident_edges` se poziva jednom za svaki čvor
- BFS traje $O(n + m)$ ako je graf predstavljen listom suseda

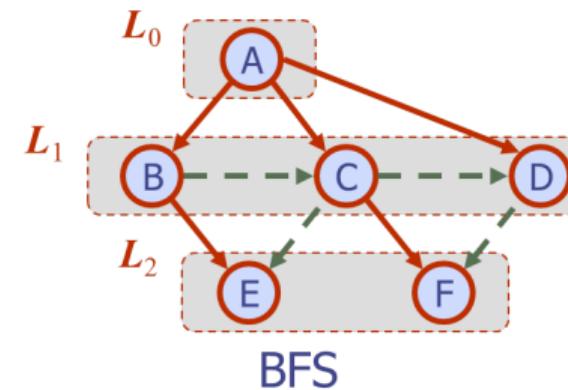
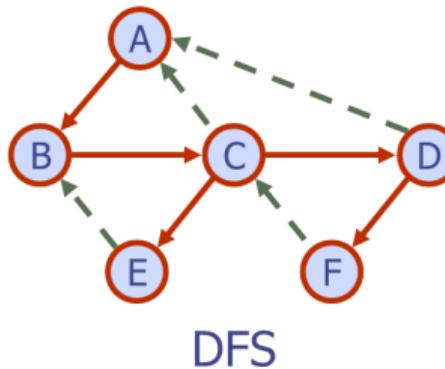
$$\sum_v \deg(v) = 2m$$

Primene BFS

- možemo prilagoditi BFS za rešavanje sledećih problema u trajanju $O(n + m)$:
 - određivanje povezanih komponenti
 - određivanje pokrivajuće šume
 - pronalaženje proste petlje ili je graf šuma
 - pronalaženje najkraće putanje između dva čvora, ili ne postoji

DFS vs BFS

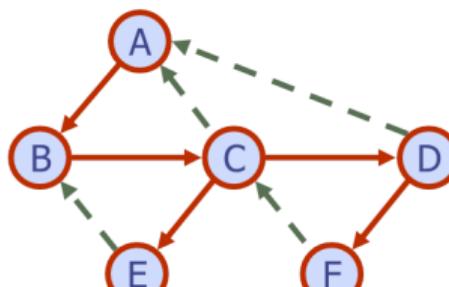
primena	DFS	BFS
pokrivajuća šuma, povezane komponente, putanje, petlje	✓	✓
najkraći put		✓
bipovezane komponente	✓	



DFS vs BFS

BACK grana (v, w)

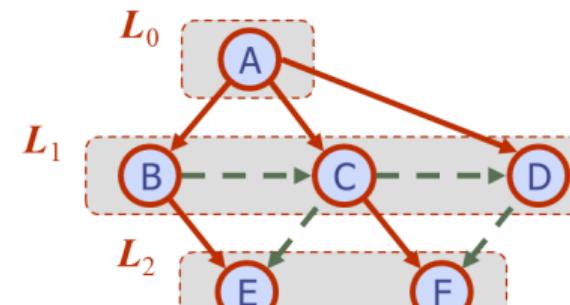
- w je predak od v u stablu koje čine DISCOVERY grane



DFS

CROSS grana (v, w)

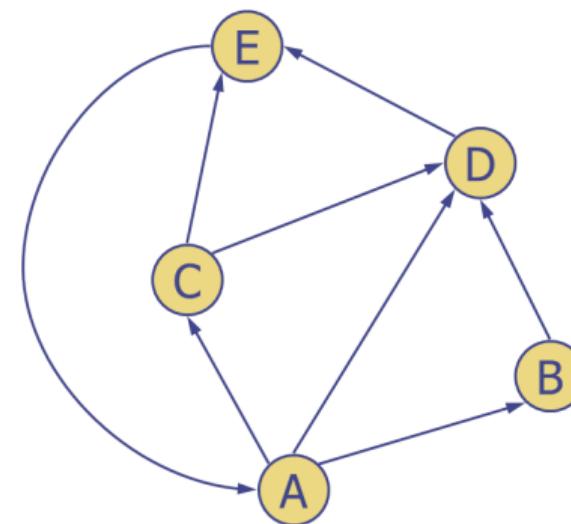
- w je na istom nivou ili na sledećem nivou u odnosu na v



BFS

Usmereni graf

- **usmereni graf** (directed graph, „digraph“) je graf čije su sve grane usmerene



Osobine usmerenog grafa

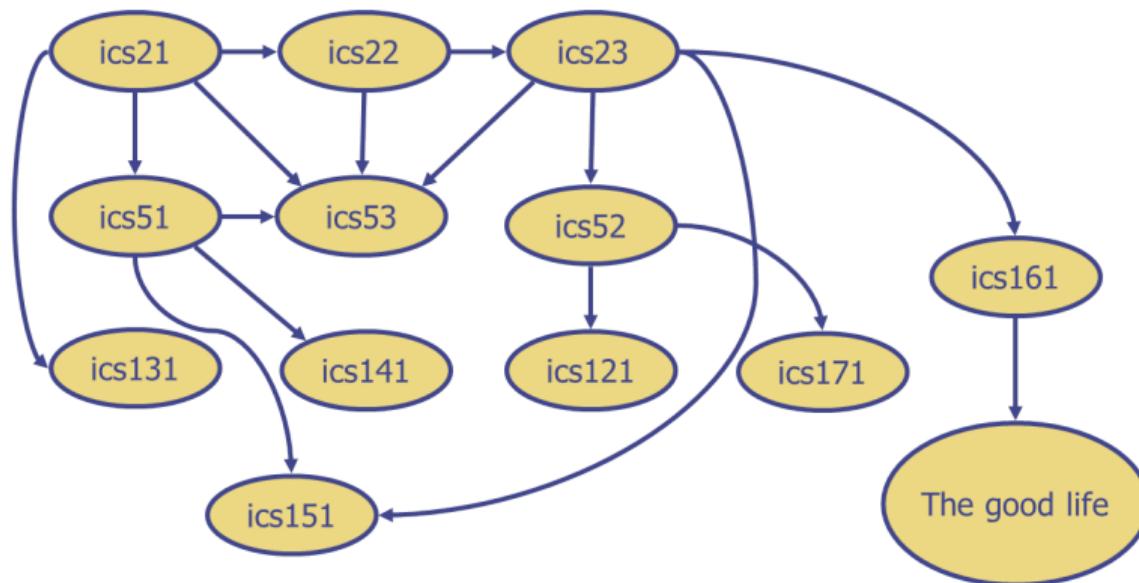
- graf $G = (V, E)$ takav da
 - svaka grana je usmerena
 - grana (a, b) ide od a ka b , i ne ide od b ka a
- ako je G prost (nema petlji i višestrukih grana):

$$m \leq n(n - 1)$$

- ako čuvamo ulazne i izlazne grane u posebnim listama, njihovo listanje traje proporcionalno njihovom broju

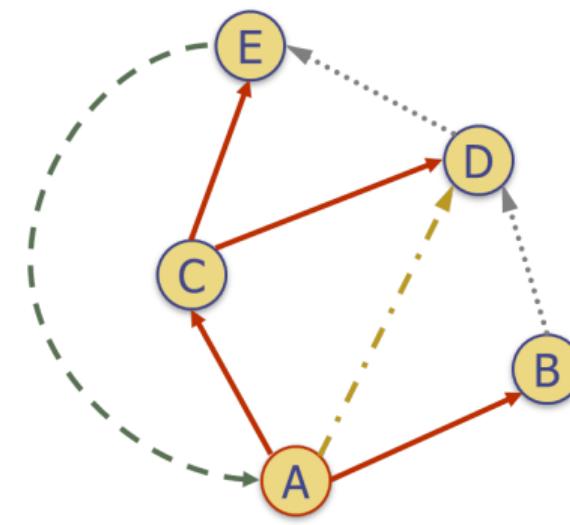
Primene usmerenog grafa

- raspoređivanje zadataka (scheduling)
- grana (a, b) označava da se zadatak a mora uraditi pre nego što počne zadatak b



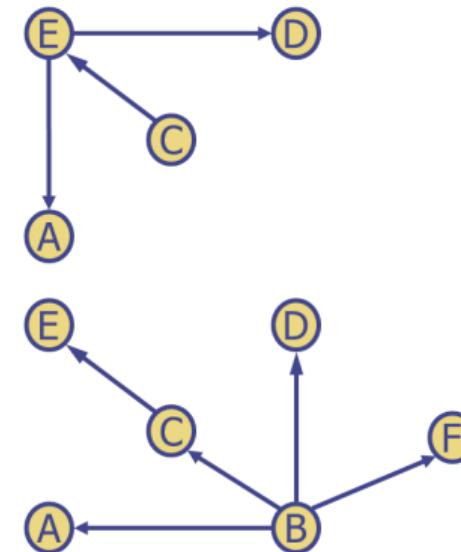
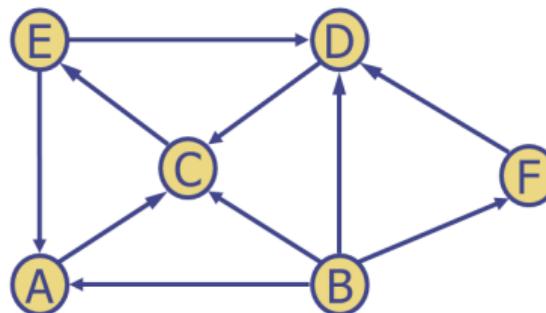
DFS za usmereni graf

- biće četiri vrste grana:
 - discovery
 - back
 - forward: vodi prema potomku u pokrivajućem stablu
 - cross: vodi prema čvoru u pokrivajućem stablu koji nije ni predak ni potomak
- DFS za usmereni graf polazeći od čvora s određuje čvorove **dostupne** iz s



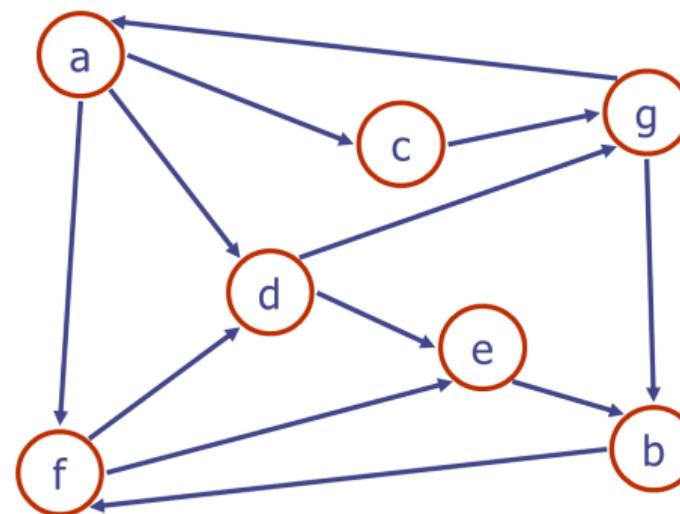
Dostupnost

- DFS stablo sa korenom u v : čvorovi dostupni iz v duž usmerenih grana



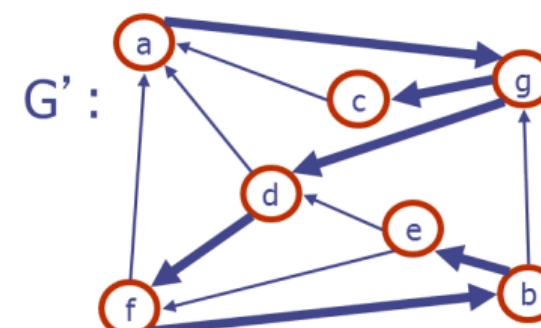
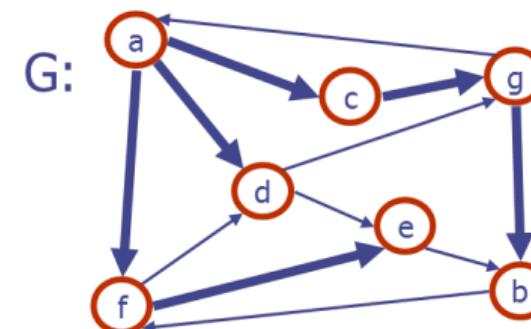
Jaka povezanost

- iz svakog čvora se može stići do svakog drugog čvora



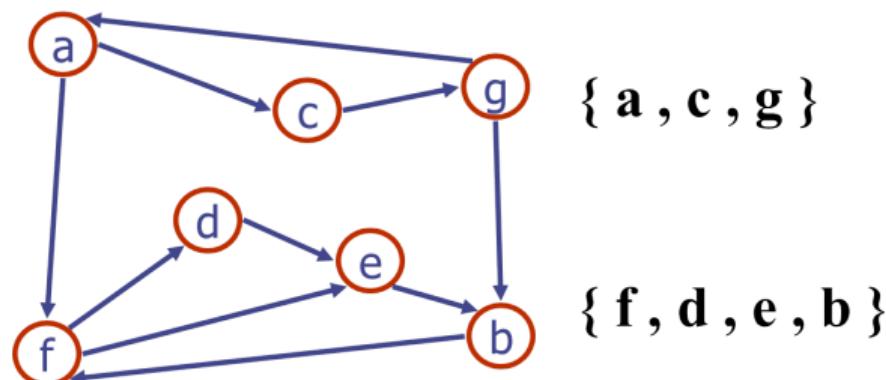
Jaka povezanost: algoritam

- izaberi čvor v iz G
- uradi DFS iz v
 - ako postoji w koji nije obiđen, ispiši „ne“
- neka je G' dobijen okretanjem grana u G
- uradi DFS iz v u G'
 - ako postoji w koji nije obiđen, ispiši „ne“
 - inače ispiši „da“
- vreme izvršavanja: $O(n + m)$



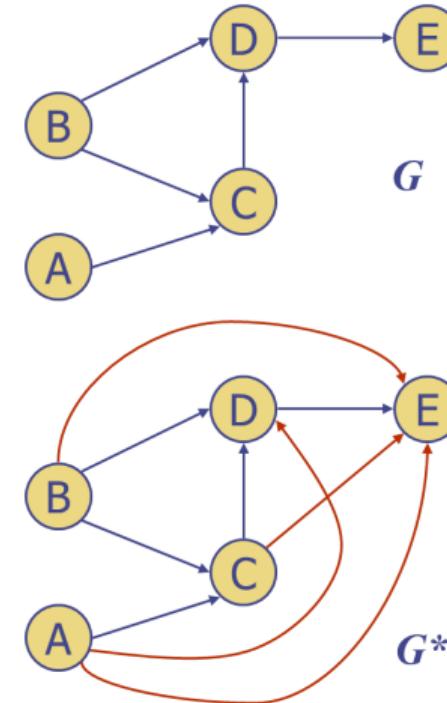
Jako povezana komponenta

- maksimalni podgraf takav da su iz svakog čvora dostupni svi drugi čvorovi
- takođe traje $O(n + m)$ pomoću DFS ali je složenije



Tranzitivno zatvorenje

- za dati usmereni graf G ,
tranzitivno zatvorenje od G je
 usmereni graf G^* takav da
 - G^* ima iste čvorove kao G
 - ako G ima putanju od u do v ($u \neq v$), G^* ima granu od u do v
- tranzitivno zatvorenje daje
 informaciju o dostupnosti unutar
 usmerenog grafa

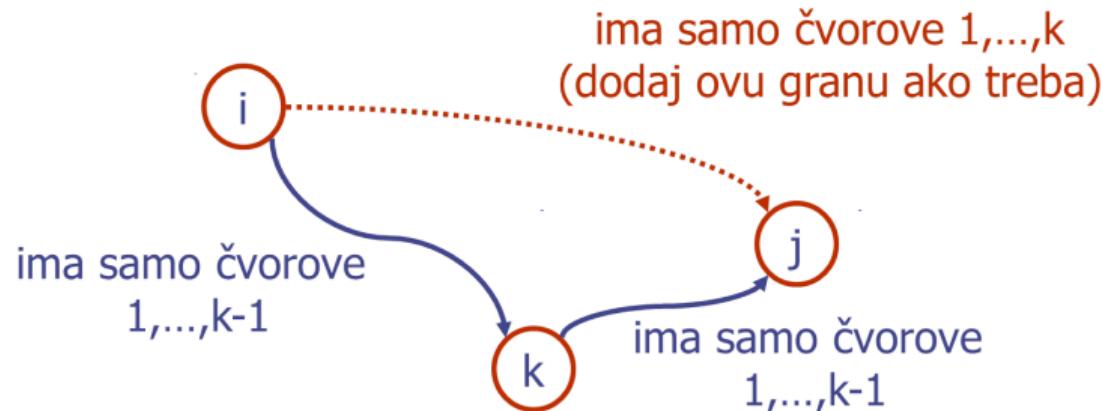


Izračunavanje tranzitivnog zatvorenja

- možemo raditi DFS iz svakog čvora – traje $O(n(n + m))$
- možemo koristiti dinamičko programiranje – Floyd-Warshall algoritam

Floyd-Warshall algoritam

- ideja #1: označi čvorove brojevima $1, 2, \dots, n$
- ideja #2: razmatramo putanje koje imaju samo čvorove $1, 2, \dots, k$ kao međučvorove



Floyd-Warshall algoritam

- numeriši čvorove $v_1 \dots v_n$
- izračunaj grafove $G_0 \dots G_n$
 - $G_0 = G$
 - G_k ima granu (v_i, v_j) ako G ima putanju $v_i \rightarrow v_j$ sa međučvorovima u $\{v_1 \dots v_k\}$
- na kraju će biti $G_n = G^*$
- u fazi k , G_k se računa iz G_{k-1}
- vreme je $O(n^3)$ ako `areAdjacent` je $O(1)$ (koristi se matrica incidencije)

Floyd-Warshall algoritam

FloydWarshall(G)

$i \leftarrow 1$

for all $v \in G.\text{vertices}()$ **do**

 označi v kao v_i

$i \leftarrow i + 1$

$G_0 \leftarrow G$

for $k \leftarrow 1$ **to** n **do**

$G_k \leftarrow G_{k-1}$

for $i \leftarrow 1$ **to** $n(i \neq k)$ **do**

for $j \leftarrow 1$ **to** $n(j \neq i, k)$ **do**

if $G_{k-1}.\text{areAdjacent}(v_i, v_k) \wedge G_{k-1}.\text{areAdjacent}(v_k, v_j)$ **then**

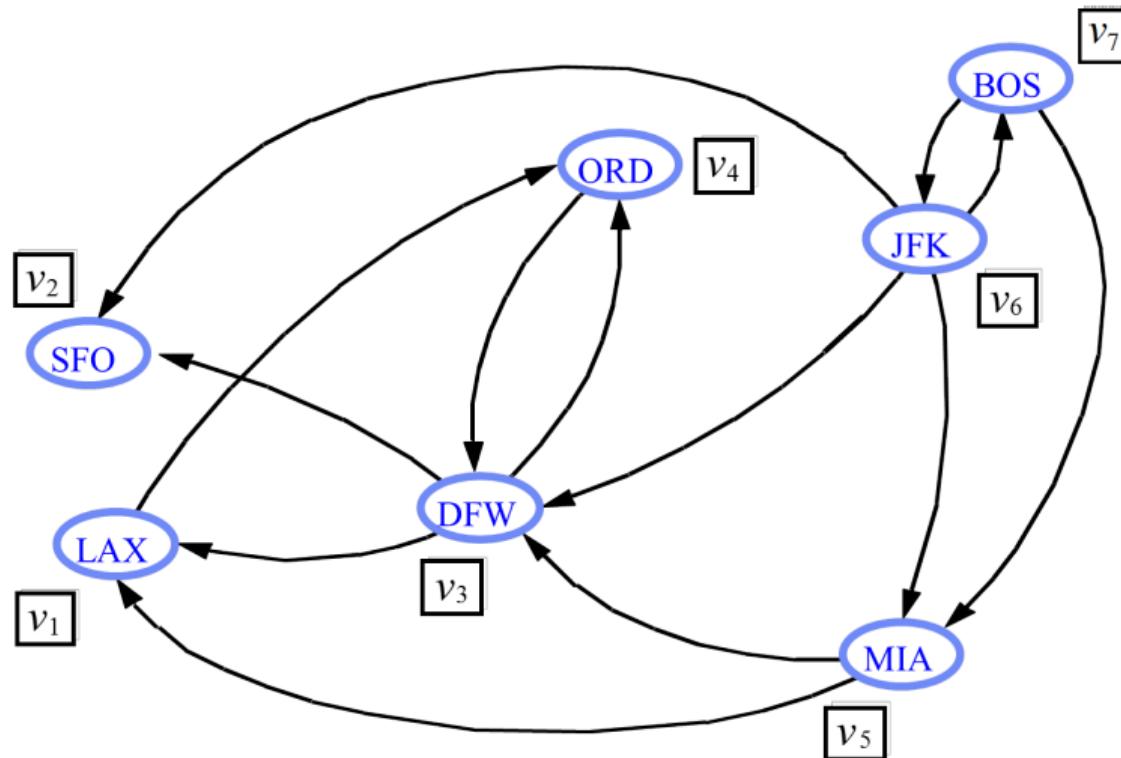
if $\neg G_{k-1}.\text{areAdjacent}(v_i, v_j)$ **then**

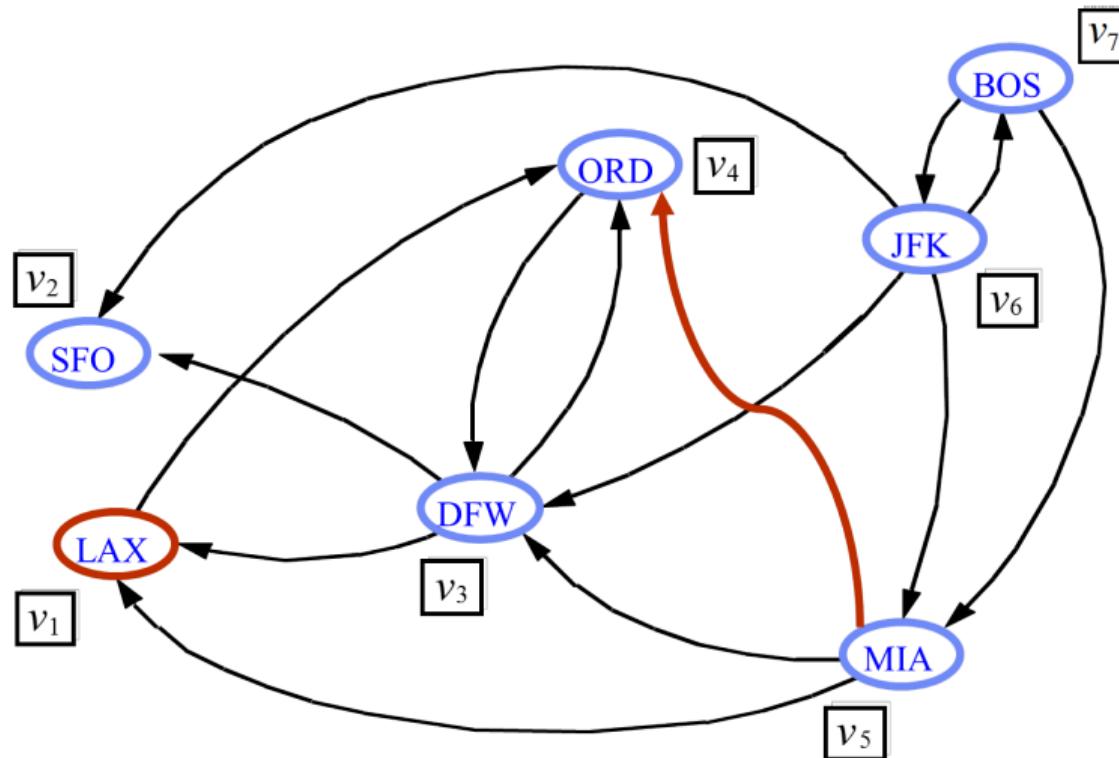
$G_k.\text{insertEdge}(v_i, v_j, k)$

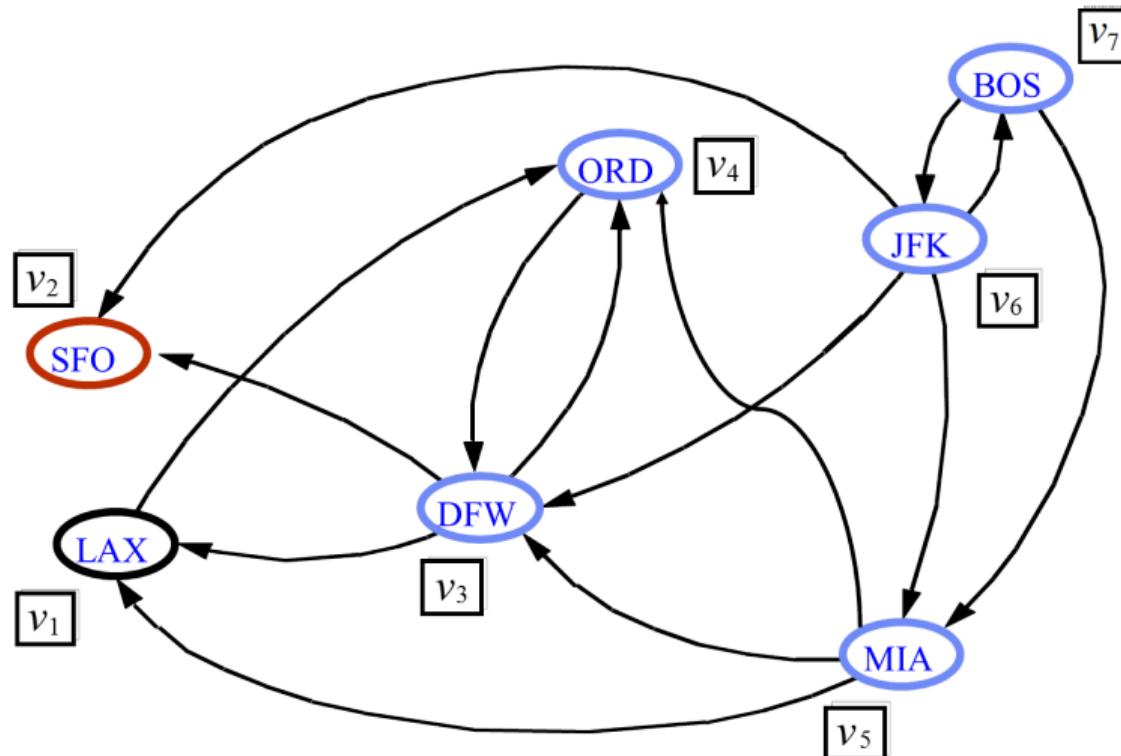
return G_n

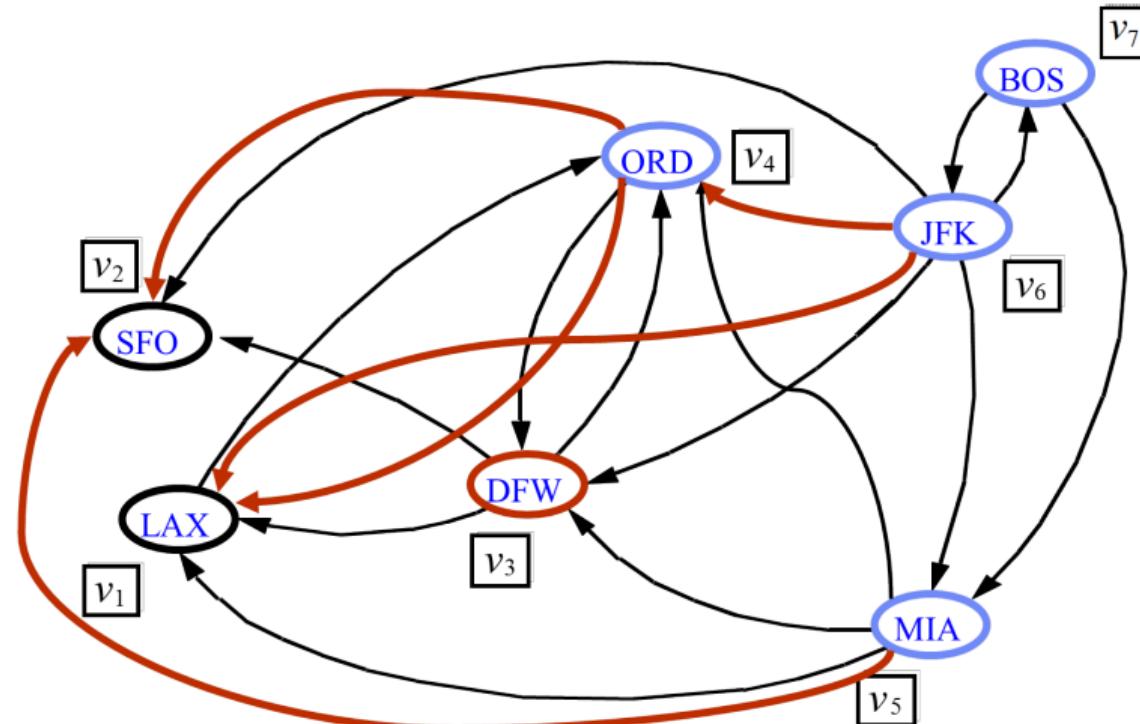
Floyd-Warshall u Pythonu

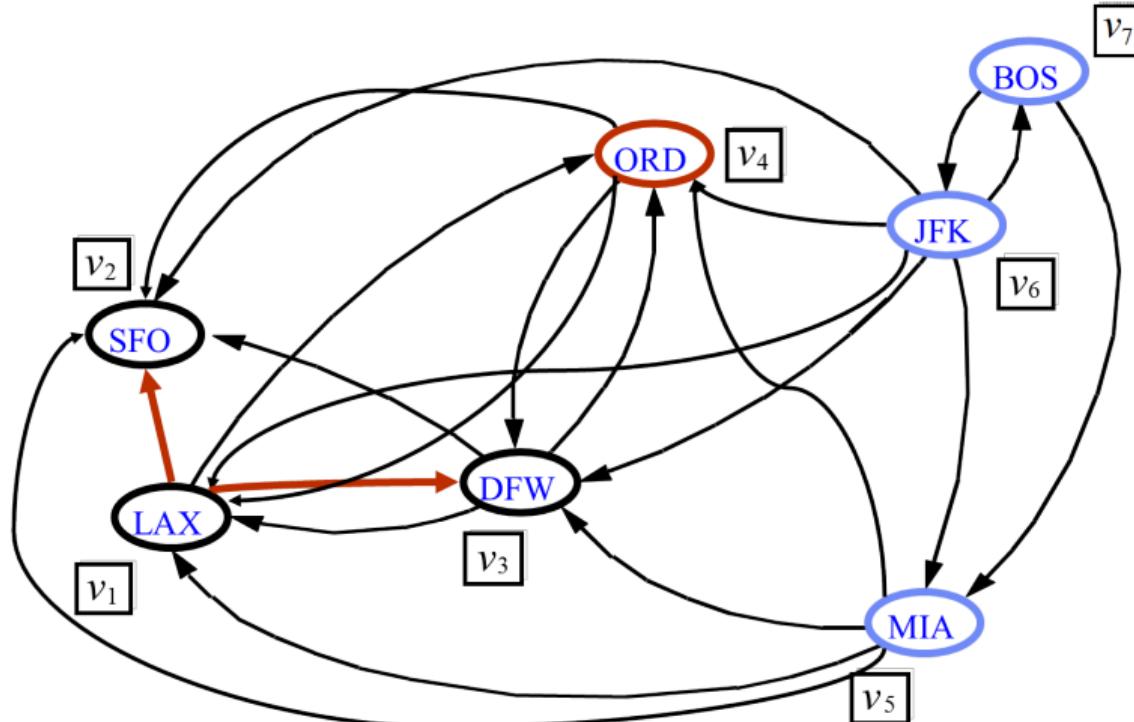
```
def floyd_marshall(g):
    """Return a new graph that is the transitive closure of g."""
    closure = deepcopy(g)
    verts = list(closure.vertices()) # make indexable list
    n = len(verts)
    for k in range(n):
        for i in range(n):
            # verify that edge (i,k) exists in the partial closure
            if i != k and closure.get_edge(verts[i],verts[k]) is not None:
                for j in range(n):
                    # verify that edge (k,j) exists in the partial closure
                    if i != j != k and closure.get_edge(verts[k],verts[j]) is not None:
                        # if (i,j) not yet included, add it to the closure
                        if closure.get_edge(verts[i],verts[j]) is None:
                            closure.insert_edge(verts[i],verts[j])
    return closure
```

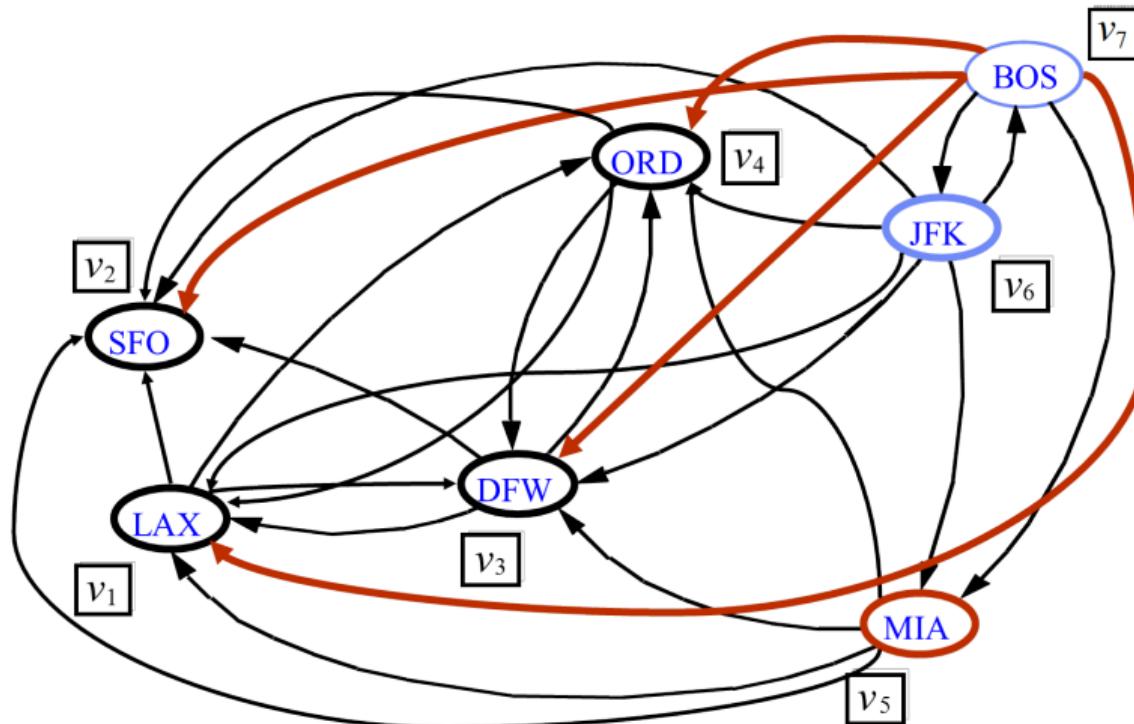
Floyd-Warshall primer: G_0 

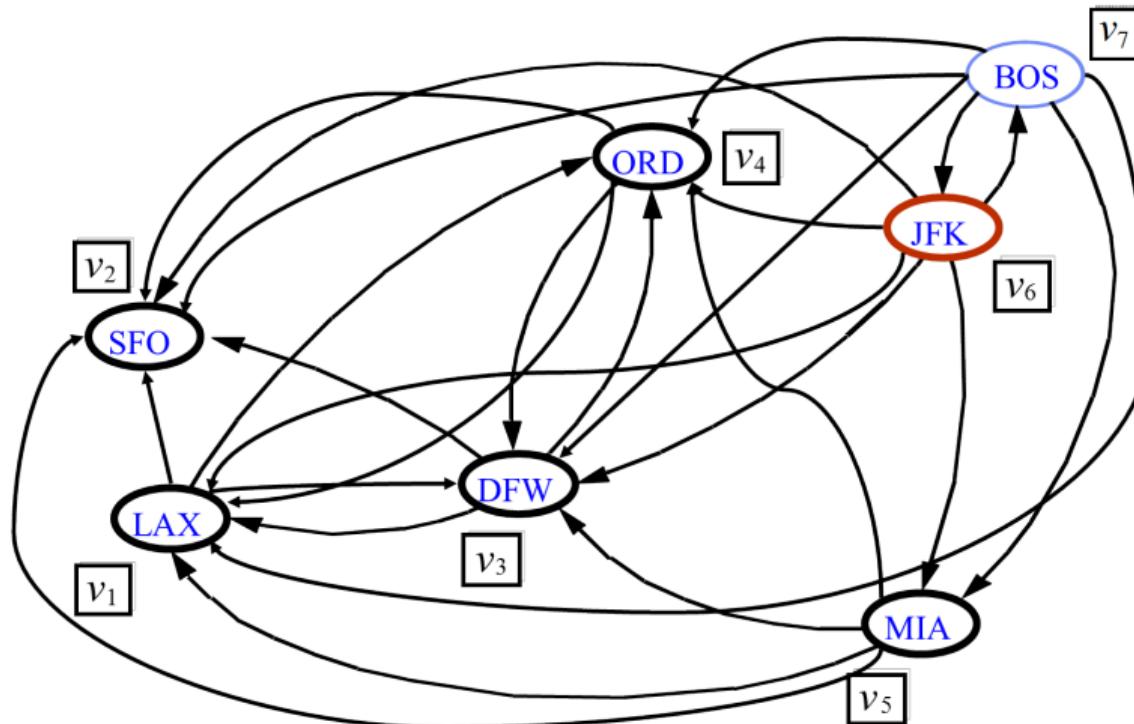
Floyd-Warshall primer: G_1 

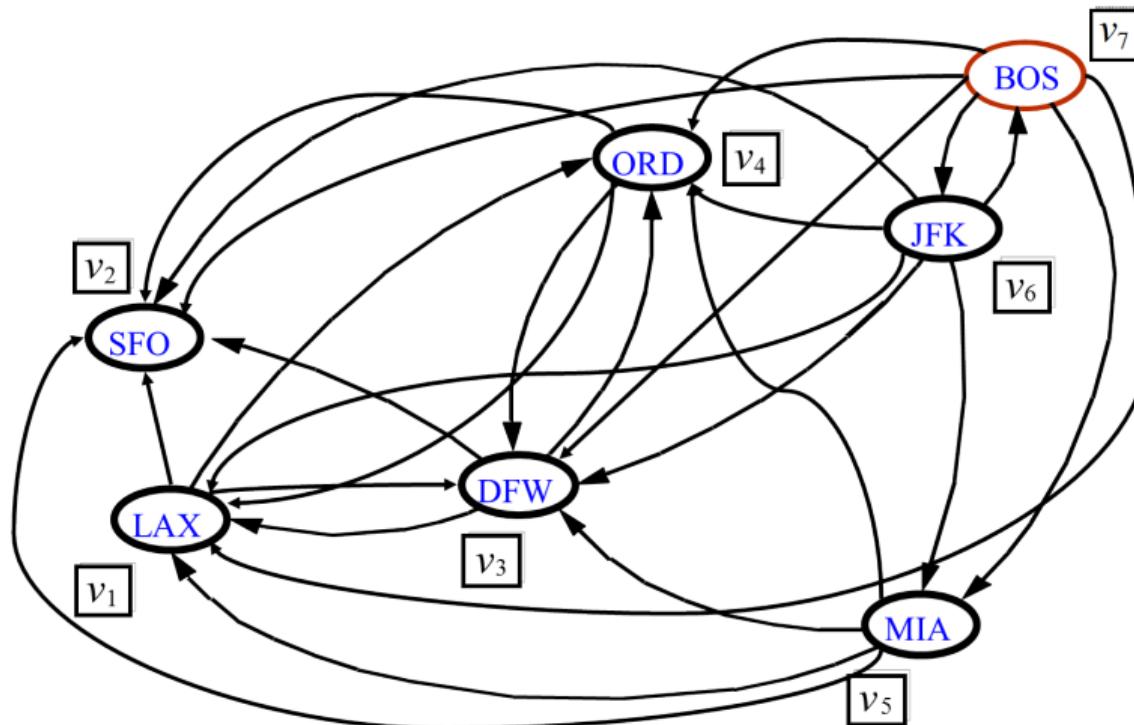
Floyd-Warshall primer: G_2 

Floyd-Warshall primer: G_3 

Floyd-Warshall primer: G_4 

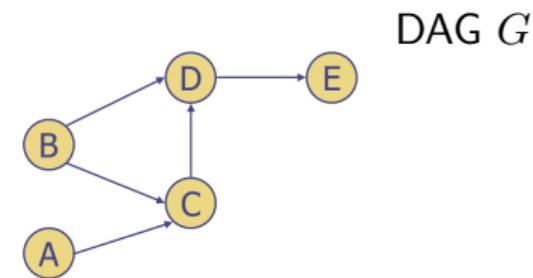
Floyd-Warshall primer: G_5 

Floyd-Warshall primer: G_6 

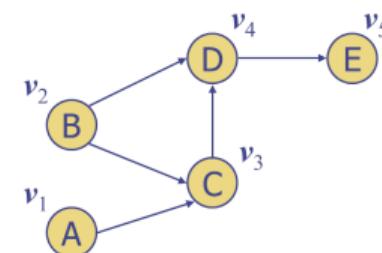
Floyd-Warshall primer: G_7 

Topološko uređenje usmerenog grafa

- usmereni aciklični graf (DAG) je aciklični graf koji nema petlje
- topološko uređenje je numerisanje $v_1 \dots v_n$ takvo da za svaku granu (v_i, v_j) važi $i < j$
- primer: u grafu raspodele zadataka topološko uređenje je sekvenca zadataka koji ispunjavaju uslove prethođenja
- usmereni graf ima topološko uređenje akko je acikličan

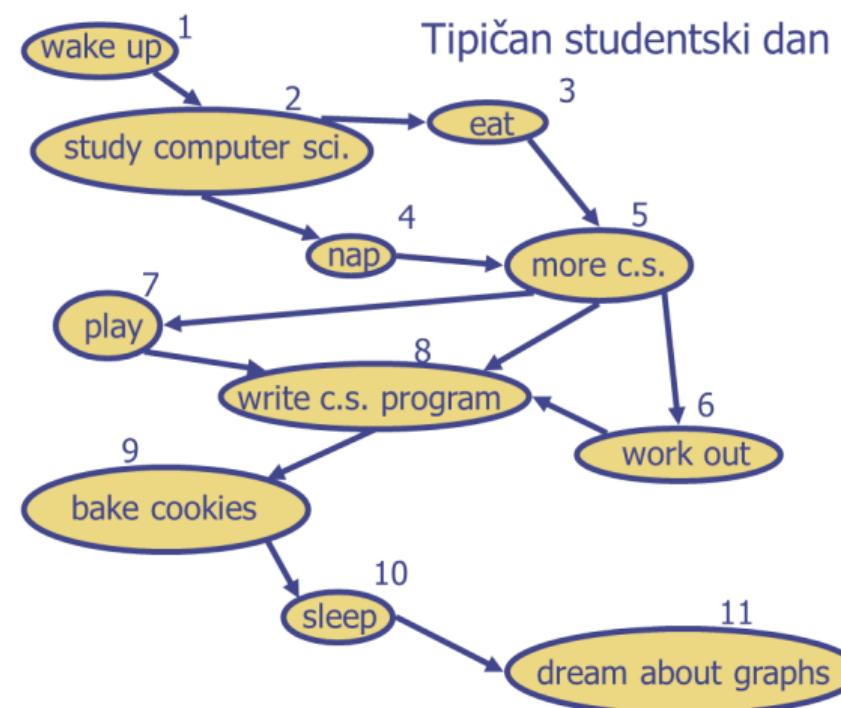


topološko uređenje G



Topološko sortiranje

- numeriši čvorove tako da $(u, v) \in E$ implicira $u < v$



Algoritam za topološko sortiranje

TopologicalSort(G)

$H \leftarrow G.\text{copy}()$

$n \leftarrow G.\text{numVertices}()$

while $\neg H.\text{isEmpty}()$ **do**

v je čvor bez izlaznih grana

 označi v sa n

$n \leftarrow n - 1$

$H.\text{remove}(v)$

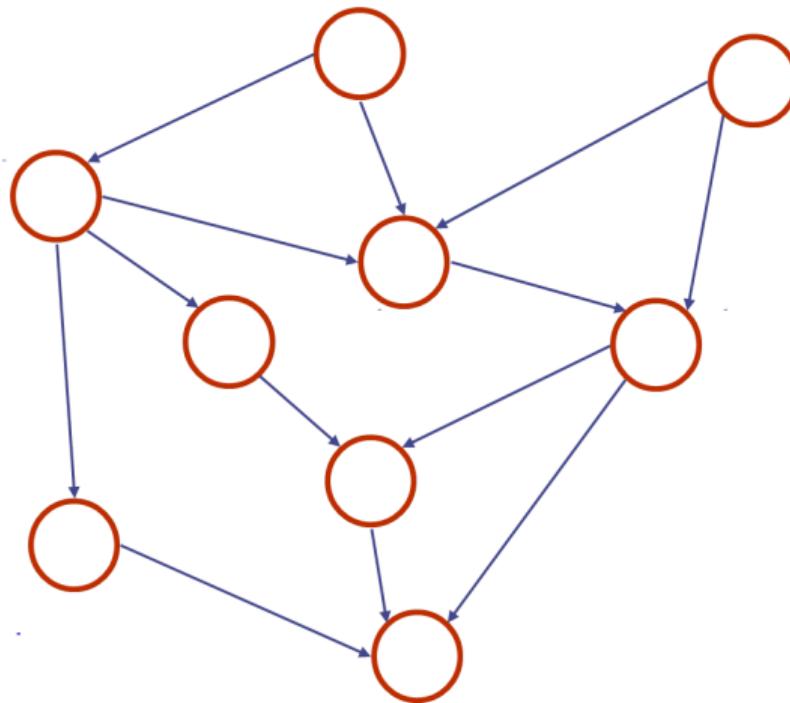
vreme je $O(n + m)$

Topološko sortiranje pomoću DFS

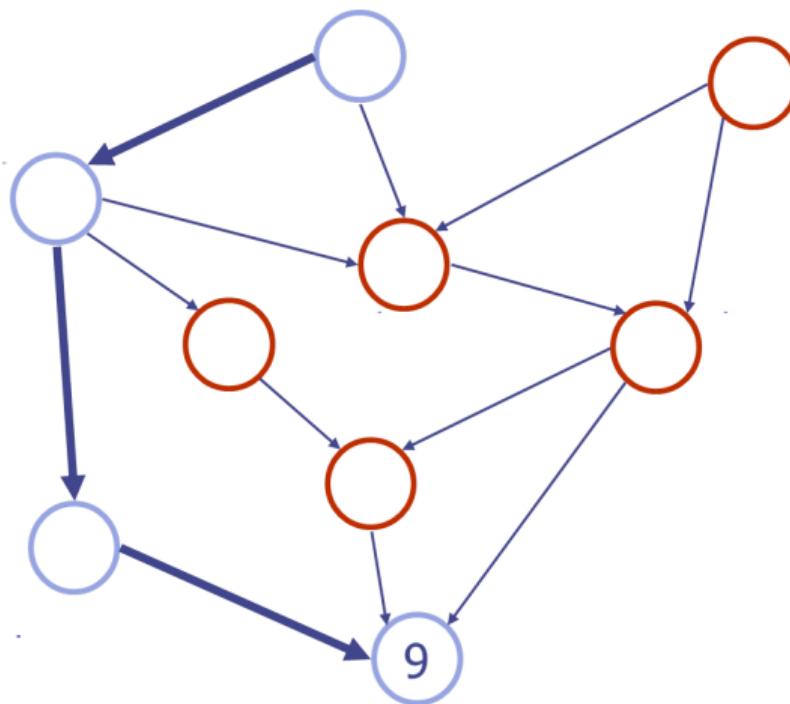
```
topologicalDFS( $G$ )
 $n \leftarrow G.\text{numVertices}()$ 
for all  $u \in G.\text{vertices}()$  do
    setLabel( $u$ , UNEXPLORED)
for all  $v \in G.\text{vertices}()$  do
    if label( $v$ ) = UNEXPLORED then
        topologicalDFS( $G, v$ )
```

```
topologicalDFS( $G, v$ )
setLabel( $v$ , VISITED)
for all  $e \in G.\text{outEdges}(v)$  do
     $w \leftarrow \text{opposite}(v, e)$ 
    if label( $w$ ) = UNEXPLORED then
        { $e$  je DISCOVERY}
        topologicalDFS( $G, w$ )
    else
        { $e$  je FORWARD ili CROSS}
        označi  $v$  brojem  $n$ 
     $n \leftarrow n - 1$ 
```

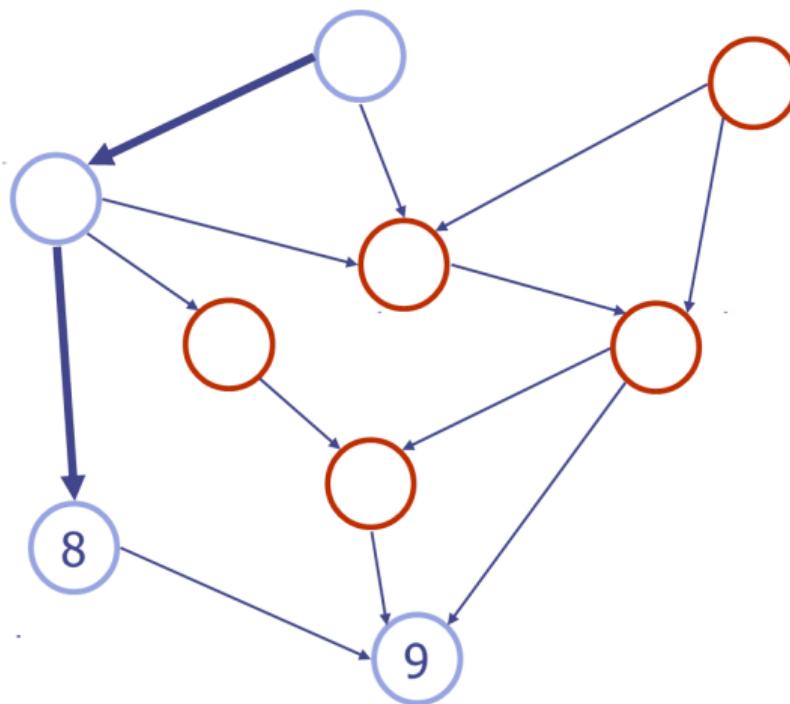
Topološko sortiranje: primer 1



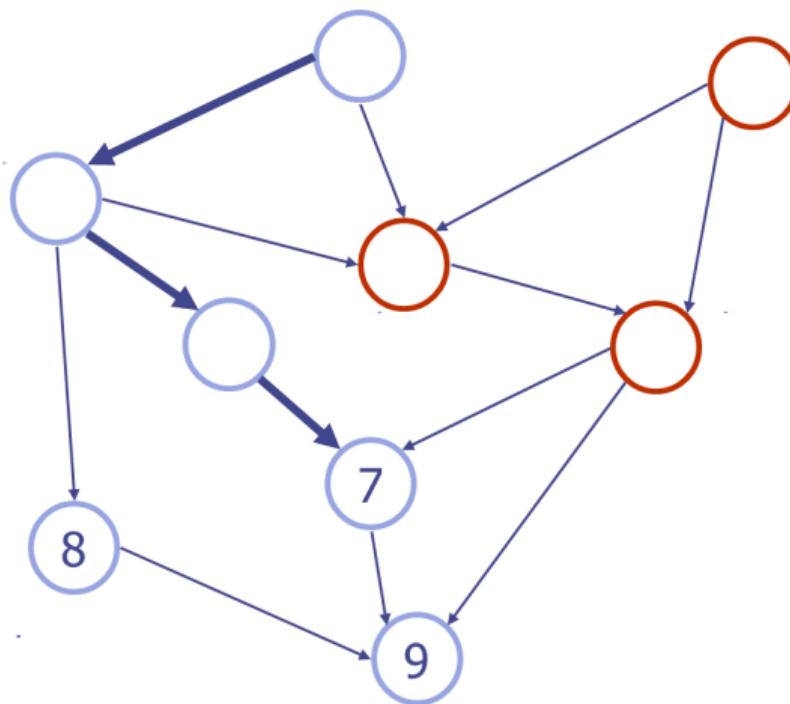
Topološko sortiranje: primer 2



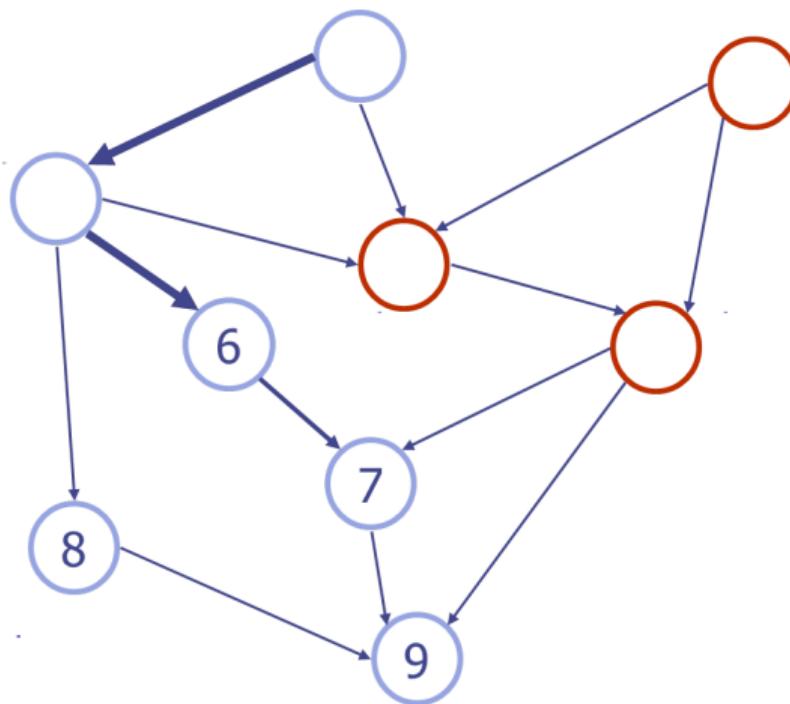
Topološko sortiranje: primer 3



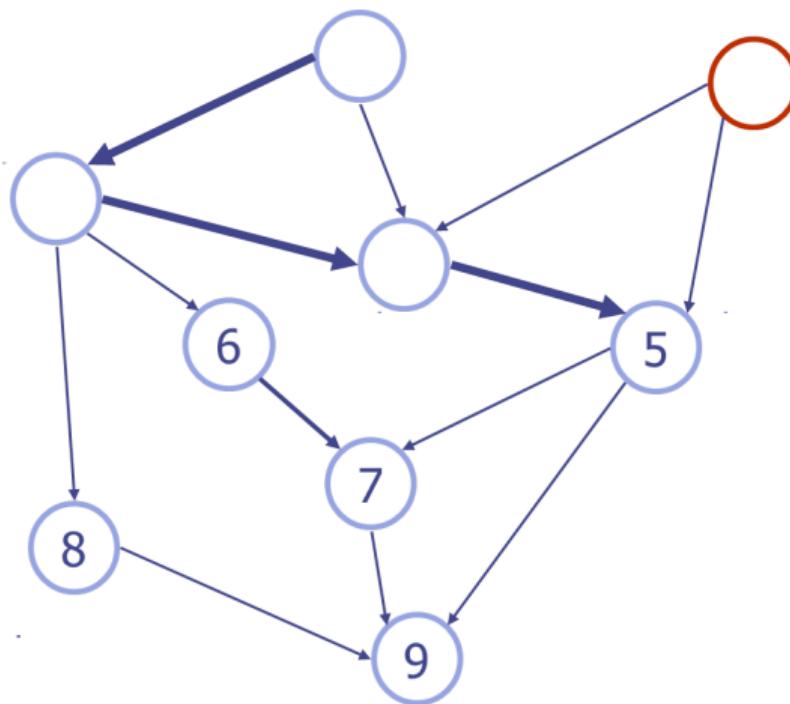
Topološko sortiranje: primer 4



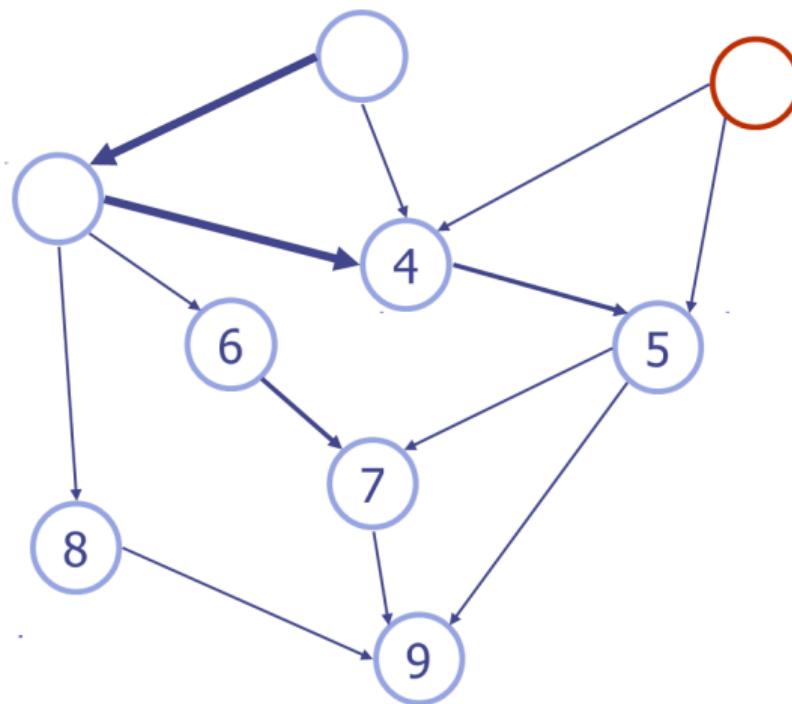
Topološko sortiranje: primer 5



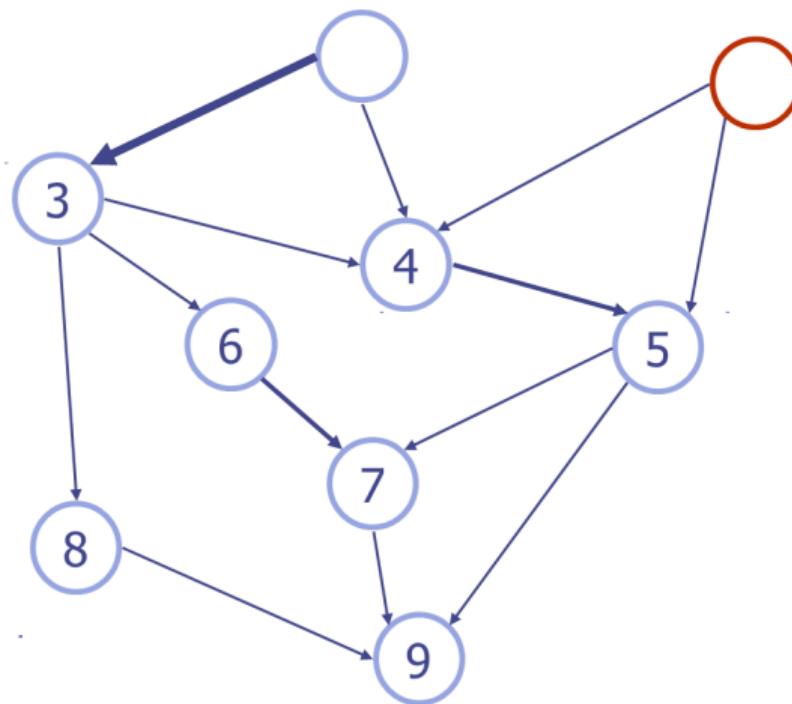
Topološko sortiranje: primer 6



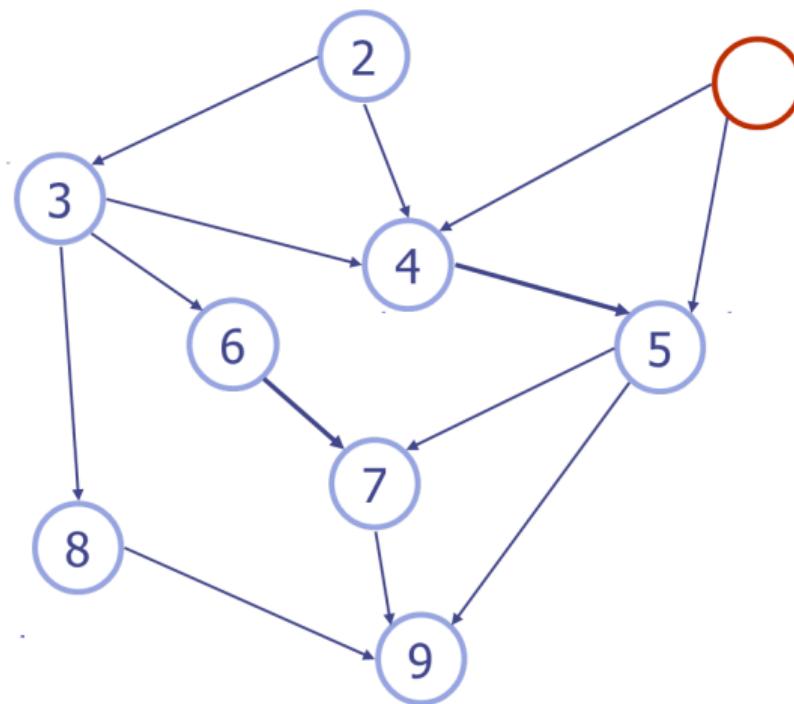
Topološko sortiranje: primer 7



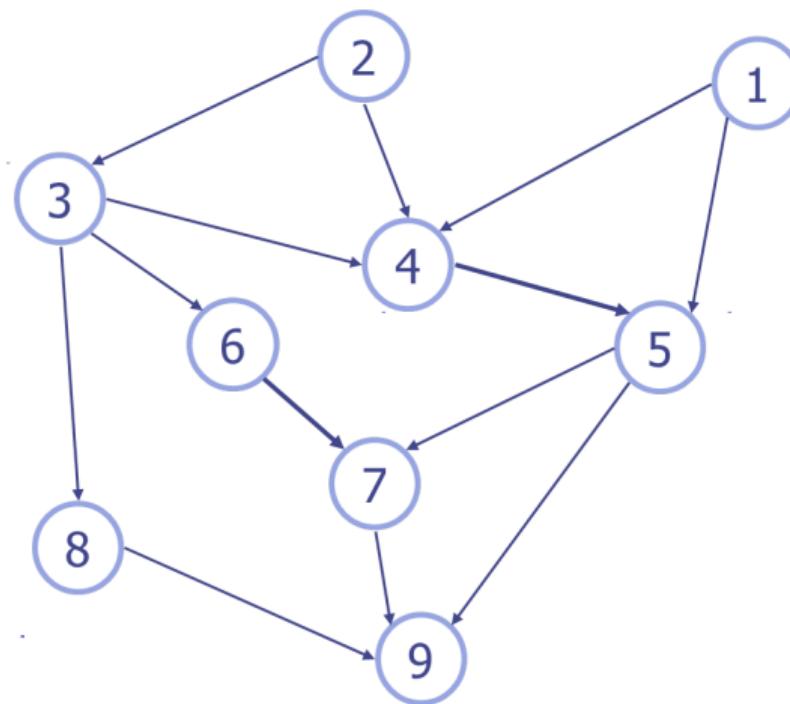
Topološko sortiranje: primer 8



Topološko sortiranje: primer 9

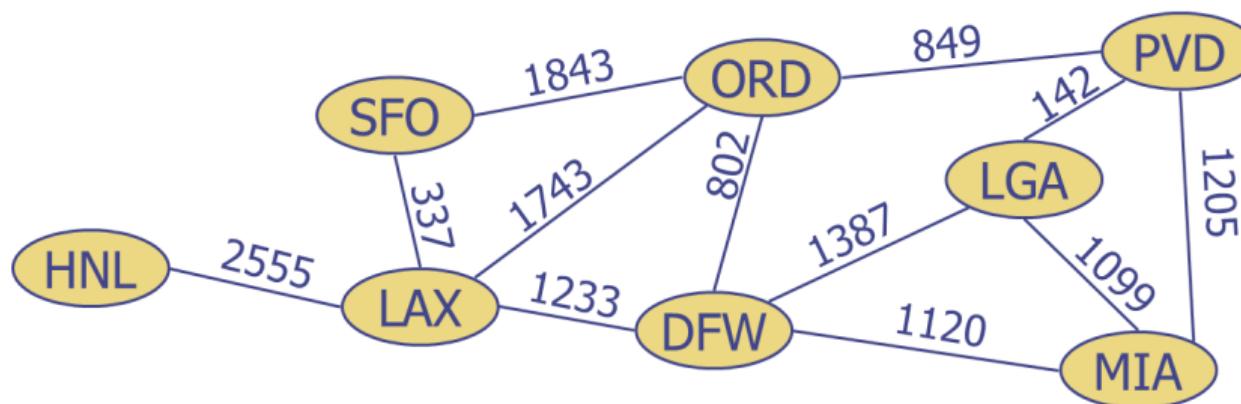


Topološko sortiranje: primer 10



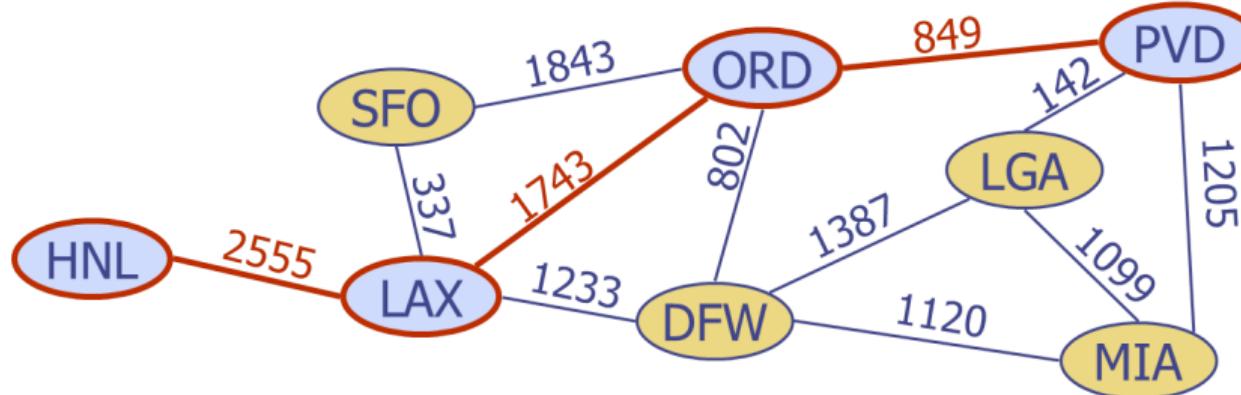
Težinski graf

- svaka grana ima dodeljenu **težinu**
- može da predstavlja rastojanje, trošak, itd.
- primer: rastojanje između aerodroma



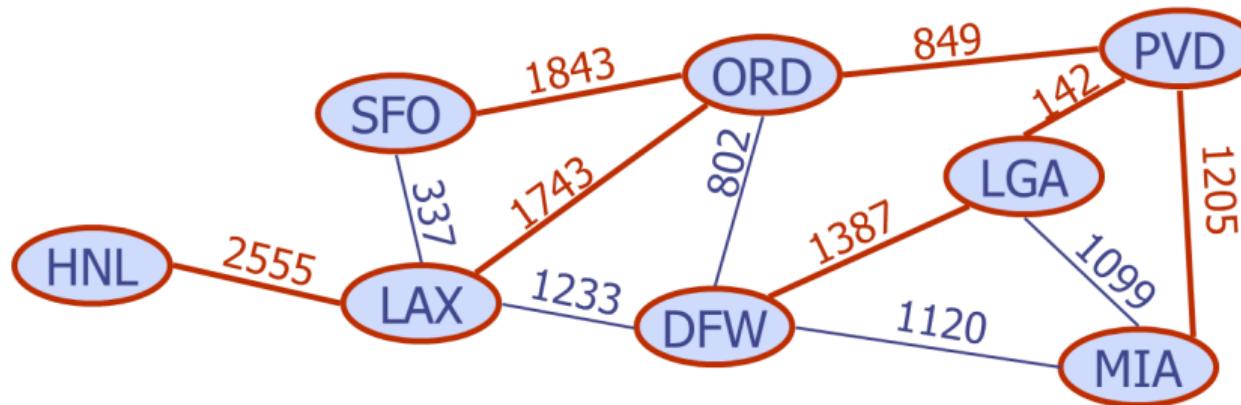
Najkraći put

- za dati težinski graf i čvorove u i v želimo da nađemo putanju sa najmanjom ukupnom težinom između u i v
- težina putanje je suma težina grana u putanji
- primer: rutiranje paketa u mreži



Osobine najkraćeg puta

- 1 podput najkraćeg puta je takođe najkraći put
- 2 postoji stablo najkraćih puteva od početnog čvora do svih ostalih čvorova



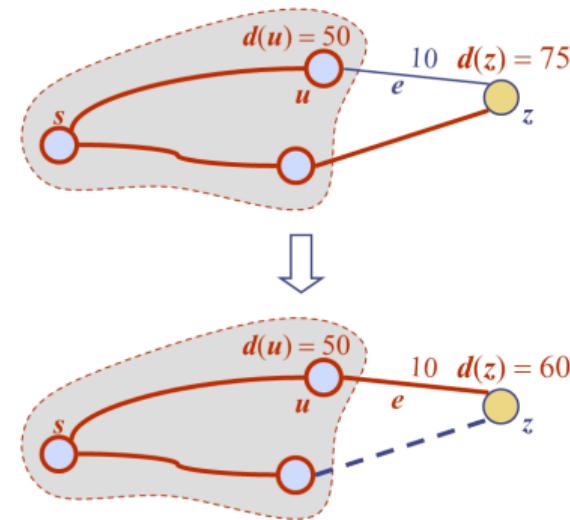
Dijkstra algoritam

- rastojanje od čvora v do čvora s je dužina najkraćeg puta $v \rightarrow s$
- Dajkstrin algoritam računa rastojanja do svih čvorova od datog čvora s
- pretpostavke:
 - graf je povezan
 - grane nisu usmerene
 - težine nisu negativne
- stvaramo „oblak“ čvorova počevši od s dok ne uključimo sve čvorove
- za svaki čvor v čuvamo labelu $d(v)$ koja predstavlja rastojanje od v do s u podgrafu sastavljenom od čvorova iz oblaka
- u svakom koraku:
 - u oblak dodamo čvor u sa najmanjim rastojanjem $d(u)$
 - ažuriramo labele čvorova susednih sa u

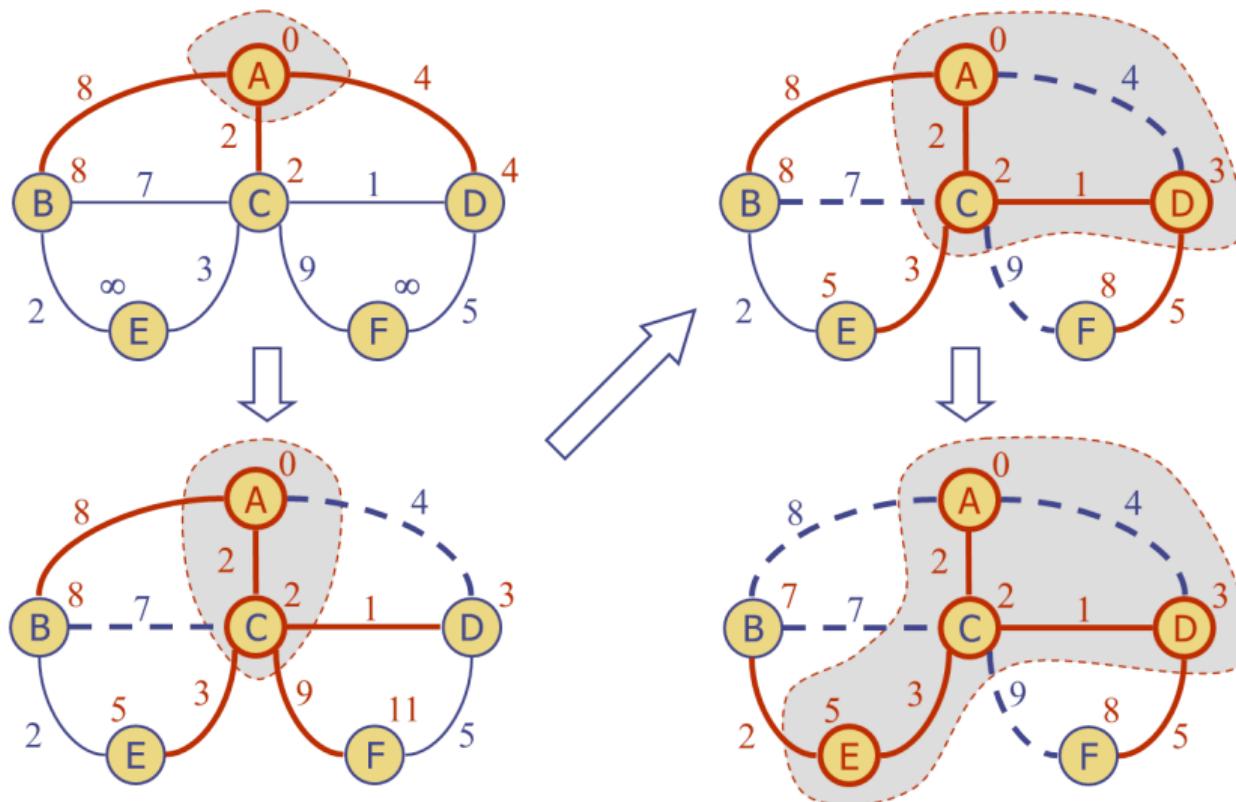
Relaksiranje grana

- posmatramo granu $e = (u, z)$ takvu da
 - u je čvor poslednji dodat u oblaku
 - z nije u oblaku
- reklassiranje grane e ažurira rastojanje $d(z)$:

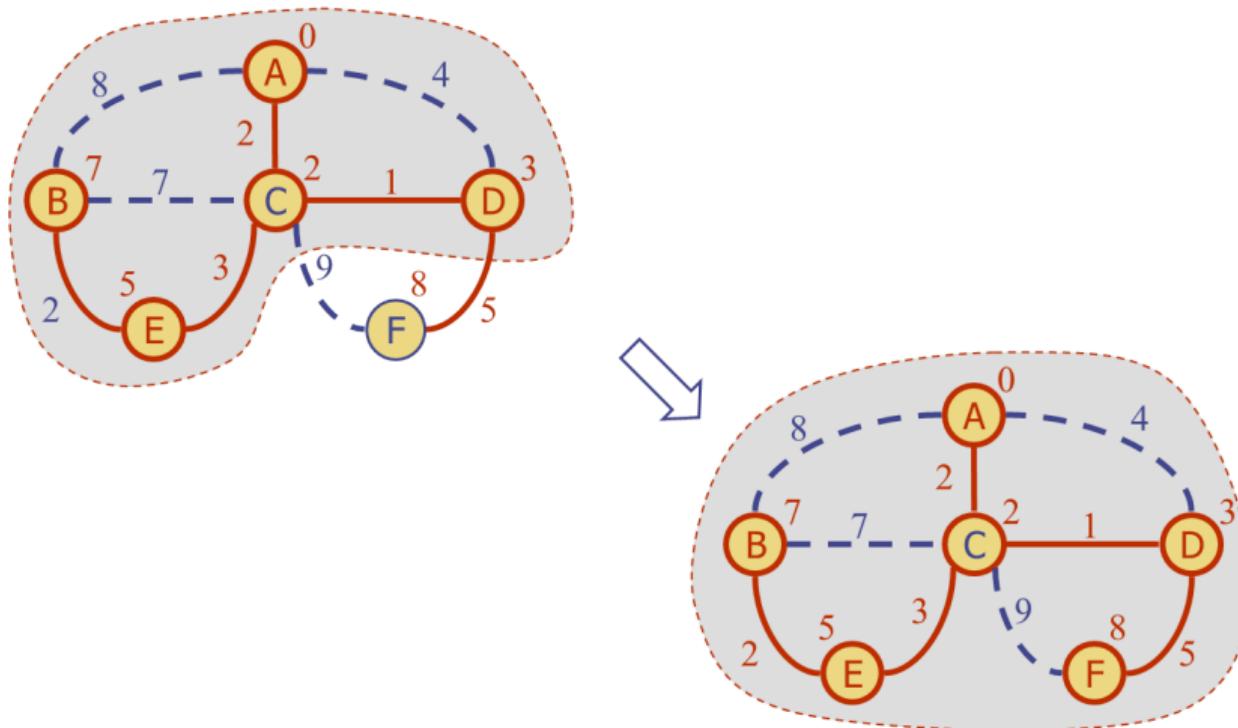
$$d(z) \leftarrow \min\{d(z), d(u) + \text{weight}(e)\}$$



Dijkstra primer 1



Dijkstra primer 2



Dijkstra algoritam

ShortestPath(G, s)

$D[s] \leftarrow 0, D[v] = \infty$ za svaki čvor $v \neq s$

red sa prioritetom Q sadrži sve čvorove iz G , ključevi su iz D

while $\neg Q.\text{isEmpty}()$ **do**

{dodaj novi čvor u u oblak}

$u \leftarrow Q.\text{removeMin}()$

for all $v \in Q$ koji je sused od u **do**

{relaksiraj granu (u, v) }

if $D[u] + w(u, v) < D[v]$ **then**

$D[v] \leftarrow D[u] + w(u, v)$

promeni ključ za v na $D[v]$ u Q

return $D[v]$ za svaki čvor v

Dijkstra algoritam: analiza

- operacije nad grafom
 - nalazimo susedne grane po jednom za svaki čvor
- operacije sa labelama
 - za čvor z postavljamo rastojanje i labele $O(\deg(z))$ puta
 - postavljanje labele traje $O(1)$
- operacije nad redom sa prioritetom
 - svaki čvor se dodaje jednom i uklanja jednom iz RSP, svaka operacija traje $O(\log n)$
 - ključ čvora u RSP se menja najviše $\deg(w)$ puta, svaka zamena ključa traje $O(\log n)$
 - postavljanje labele traje $O(1)$
- Dajkstrin algoritam traje $O((n + m) \log n)$ ako je u implementaciji korišćena lista suseda

Python implementacija

```

def shortest_path_lengths(g, src):
    d = {}                                #  $d[v]$  is upper bound from  $s$  to  $v$ 
    cloud = {}                             # map reachable  $v$  to its  $d[v]$  value
    pq = AdaptableHeapPriorityQueue()       # vertex  $v$  will have key  $d[v]$ 
    pqlocator = {}                         # map from vertex to its pq locator

    # for each vertex  $v$  of the graph, add an entry to the priority queue, with
    # the source having distance 0 and all others having infinite distance
    for v in g.vertices():
        if v is src:
            d[v] = 0
        else:
            d[v] = float('inf')           # syntax for positive infinity
            pqlocator[v] = pq.add(d[v], v) # save locator for future updates

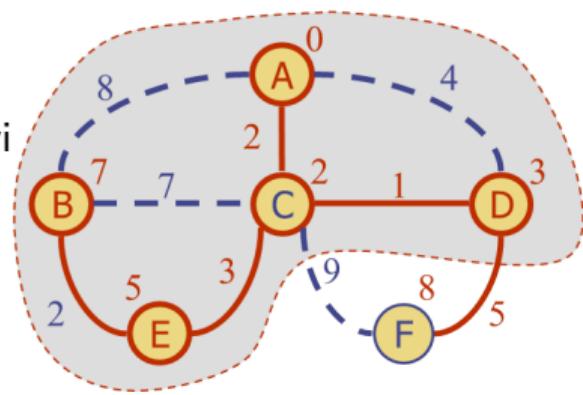
    while not pq.is_empty():
        key, u = pq.remove_min()
        cloud[u] = key                 # its correct  $d[u]$  value
        del pqlocator[u]               #  $u$  is no longer in pq
        for e in g.incident_edges(u):  # outgoing edges  $(u,v)$ 
            v = e.opposite(u)
            if v not in cloud:
                # perform relaxation step on edge  $(u,v)$ 
                wgt = e.element()
                if d[u] + wgt < d[v]:      # better path to  $v$ ?
                    d[v] = d[u] + wgt      # update the distance
                    pq.update(pqlocator[v], d[v], v) # update the pq entry

    return cloud                          # only includes reachable vertices

```

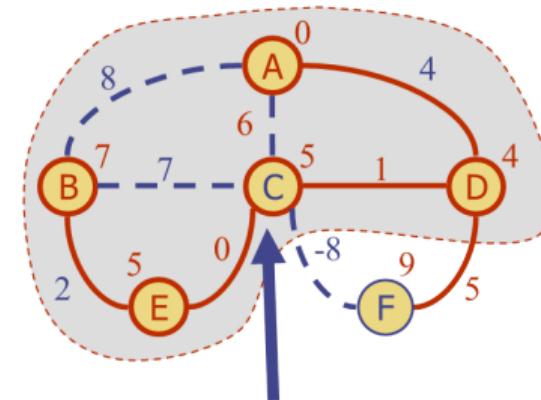
Dijkstra i pohlepa

- Dijkstra koristi pohlepni metod – dodaje čvorove po rastućem rastojanju
- prepostavimo da nije našao najkraća rastojanja; neka je F prvi pogrešan čvor
- najkraći put je bio OK za prethodni čvor D
- ali grana (D, F) je tada relaksirana
- prema tome, sve dok je $d(F) \geq d(D)$, $d(F)$ nije pogrešno, tj. nismo pogrešili čvor



Dijkstra ne radi za negativne težine

- Dijkstra koristi pohlepni metod – dodaje čvorove po rastućem rastojanju
- ako bi čvor sa negativnom granom bio dodat u oblak, pokvario bi rastojanja čvorova koji su već tamo



C-ovo pravo rastojanje je 1, ali već je u oblaku sa $d(C)=5!$

Bellman-Ford algoritam

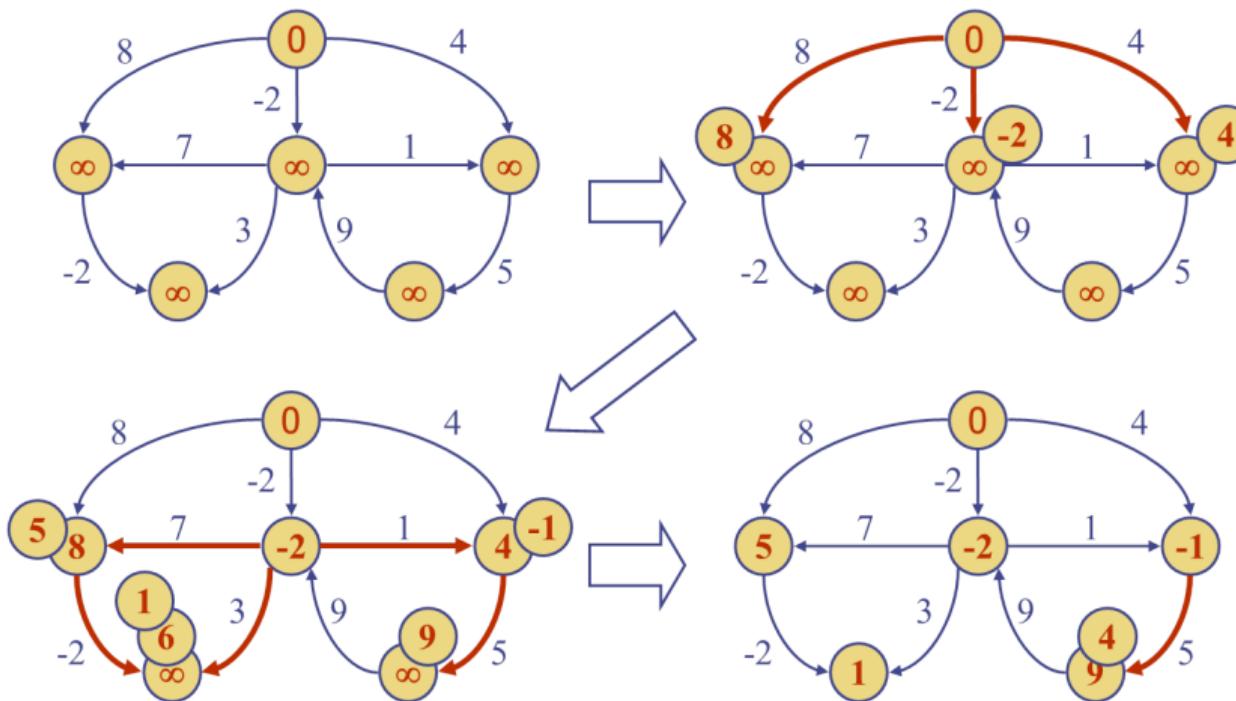
- radi i za negativne težine
- grane moraju biti usmerene – inače bismo imali negativne petlje!
- i -ta iteracija pronađi najkraće puteve sa i grana
- vreme: $O(nm)$

```

BellmanFord( $G, s$ )
for all  $v \in G.\text{vertices}()$  do
    if  $v = s$  then
        setDistance( $v, 0$ )
    else
        setDistance( $v, \infty$ )
for  $i \leftarrow 1$  to  $n - 1$  do
    for all  $e \in G.\text{edges}()$  do
        {relaksiraj granu  $e$ }
         $u \leftarrow G.\text{origin}(e)$ 
         $z \leftarrow G.\text{opposite}(u, e)$ 
         $r \leftarrow \text{getDistance}(u) + \text{weight}(e)$ 
        if  $r < \text{getDistance}(z)$  then
            setDistance( $z, r$ )

```

Bellman-Ford primer

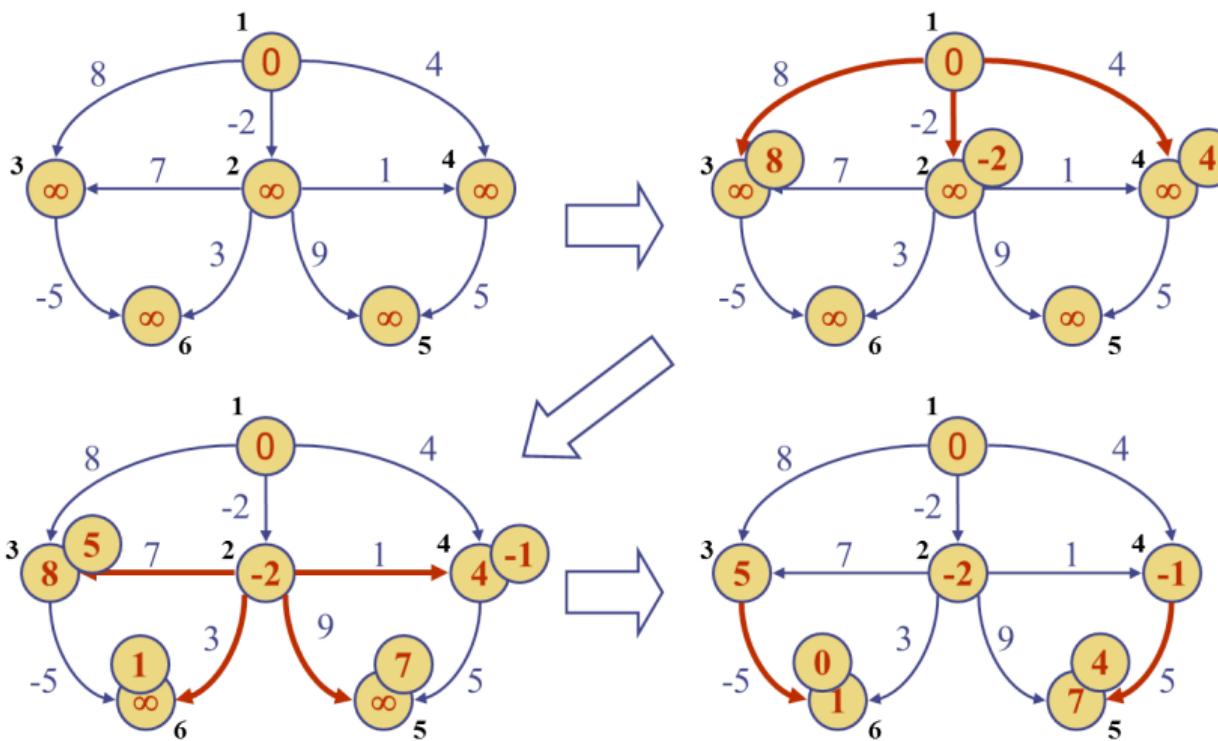
 \checkmark vorovi su označeni sa $d(v)$

Algoritam za usmereni aciklični graf

- radi i za negativne težine
- koristi topološko uređenje
- ne koristi posebnu strukturu podataka
- znatno brži od Dijkstre:
 $O(n + m)$

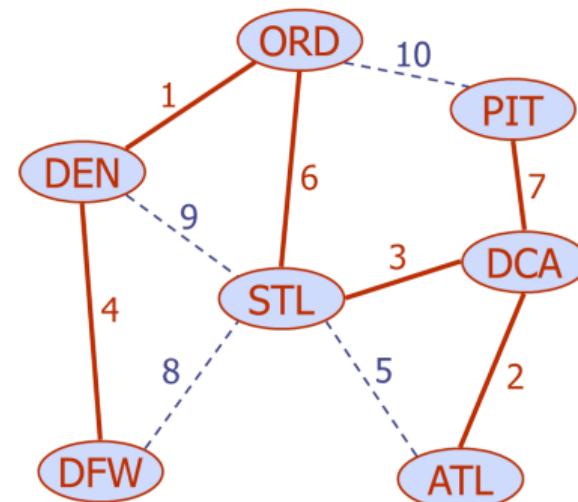
```
DagDistances( $G, s$ )
for all  $v \in G.\text{vertices}()$  do
    if  $v = s$  then
        setDistance( $v, 0$ )
    else
        setDistance( $v, \infty$ )
    {topološki sortiraj čvorove}
for  $u \leftarrow 1$  to  $n$  do
    {u topološkom redosledu}
    for all  $e \in G.\text{outEdges}(u)$  do
        {relaksiraj granu e}
         $z \leftarrow G.\text{opposite}(u, e)$ 
         $r \leftarrow \text{getDistance}(u) + \text{weight}(e)$ 
        if  $r < \text{getDistance}(z)$  then
            setDistance( $z, r$ )
```

DAG primer



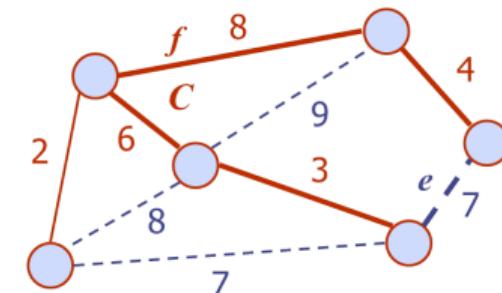
Minimalno pokrivajuće stablo

- **pokrivajući podgraf**: podgraf od G koji sadrži sve čvorove iz G
- **pokrivajuće stablo**: pokrivajući podgraf koji je stablo
- **minimalno pokrivajuće stablo**: pokrivajuće stablo sa najmanjom sumom težina grana
- „minimum spanning tree“ (MST)

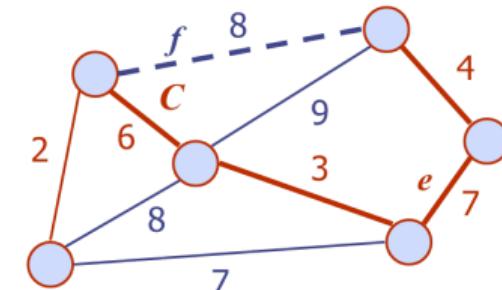


MST i petlje

- T je MST grafa G
- e je grana koja nije u T
- C je petlja koju e pravi sa T
- za svaku granu f iz C važi:
 $weight(f) \leq weight(e)$
- dokaz kontradikcijom: ako bi bilo
 $weight(f) > weight(e)$ imali bismo
MST sa manjom težinom zamenom e
sa f

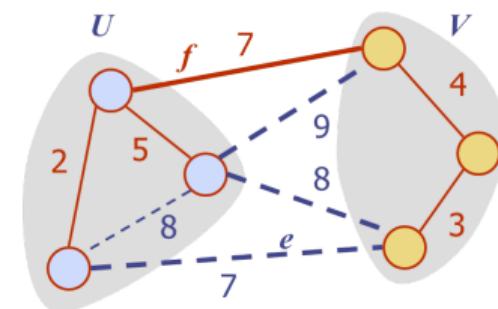


Zamena f sa e daje
bolje stablo

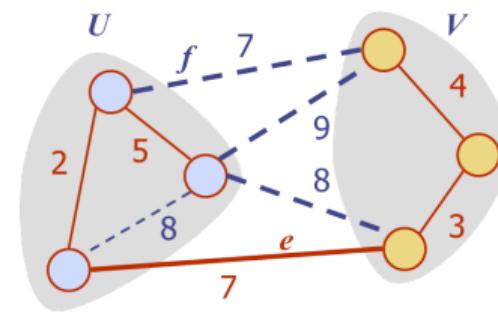


MST i particije

- posmatramo podelu čvorova iz G na podskupove U i V
- e je grana između podskupova sa najmanjom težinom
- postoji MST od G koji sadrži e
- dokaz: T je MST od G
- ako T ne sadrži e , posmatramo petlju C koju čine T i e i neka je f grana iz C koja prelazi particiju
- (prethodni slajd):
 $weight(f) \leq weight(e)$
- prema tome: $weight(f) = weight(e)$



Zamena f sa e daje
još jedno MSP



Prim-Jarnik algoritam

- sličan Dijkstra algoritmu
- izaberemo čvor s i pravimo MST kao oblak čvorova počevši od s
- čvor v čuva labelu $d(v)$ – najmanja težina grane koja povezuje v sa oblakom
- u svakom koraku:
 - dodamo u oblak čvor u sa najmanjim $d(u)$
 - ažuriramo labele čvorova susednih sa u

Prim-Jarnik algoritam

PrimJarnik(G)

izaberi bilo koji s iz G

$D[s] \leftarrow 0$

for all čvor $v \neq s$ **do**

$D[v] \leftarrow \infty$

$T \leftarrow \emptyset$

napuni RSP Q elementima $(D[v], (v, \text{None}))$ za svaki čvor v

while $\neg Q.\text{isEmpty}()$ **do**

$(u, e) \leftarrow Q.\text{removeMin}()$

poveži u sa T preko e

for all $e' = (u, v) | v \in Q$ **do**

{da li (u, v) bolje povezuje v sa T }

if $w(u, v) < D[v]$ **then**

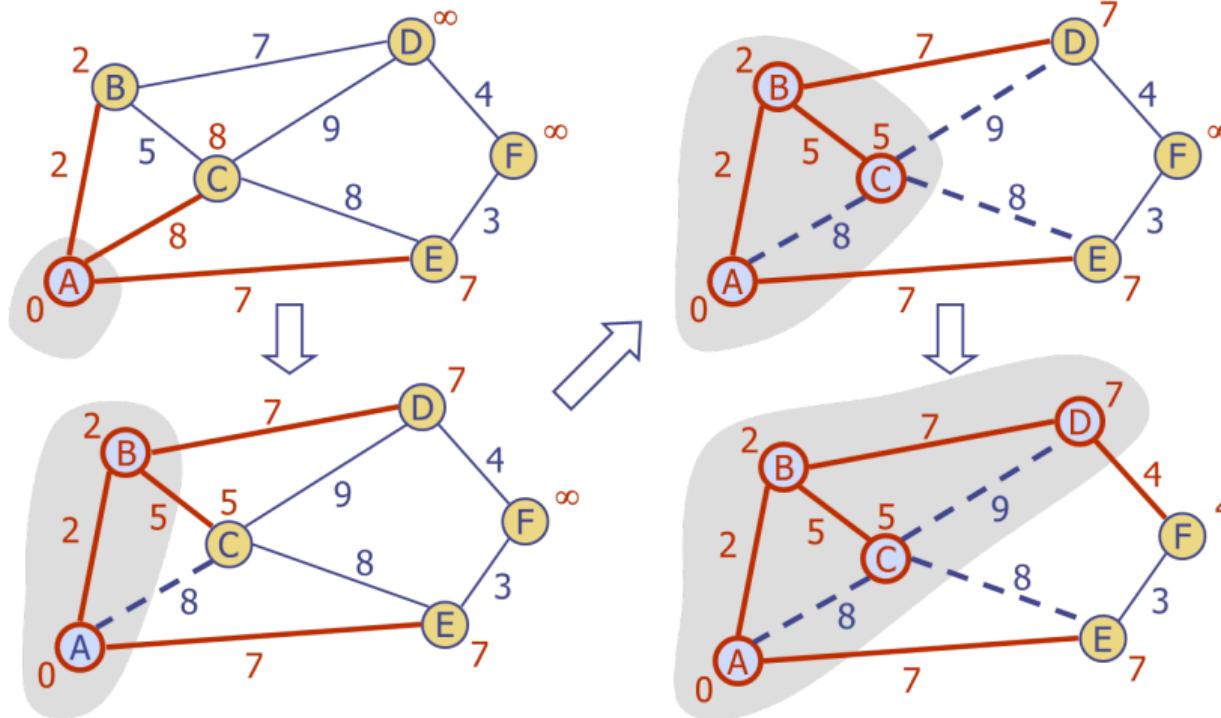
$D[v] \leftarrow w(u, v)$

promeni ključ za v u Q na $D[v]$

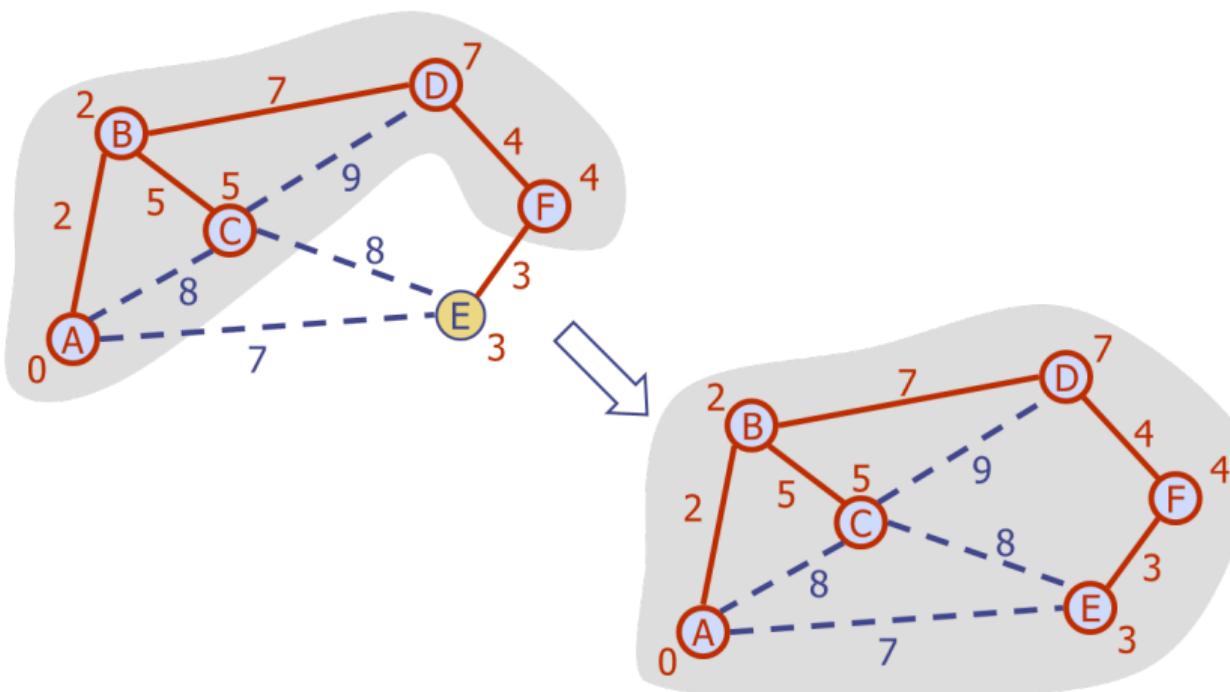
promeni vrednost за v у Q на (v, e')

return T

Prim-Jarnik primer 1



Prim-Jarnik primer 2



Prim-Jarnik algoritam: analiza

- operacije nad grafom
 - prolazimo kroz susedne grane po jednom za svaki čvor
- operacije sa labelama
 - za čvor z postavljamo rastojanje i labele $O(\deg(z))$ puta
 - postavljanje labele traje $O(1)$
- operacije nad redom sa prioritetom
 - svaki čvor se dodaje jednom i uklanja jednom iz RSP, svaka operacija traje $O(\log n)$
 - ključ čvora u RSP se menja najviše $\deg(w)$ puta, svaka zamena ključa traje $O(\log n)$
 - postavljanje labele traje $O(1)$
- Prim-Jarnik algoritam traje $O((n + m) \log n)$ ako je u implementaciji korišćena lista suseda

Prim-Jarnik u Pythonu

```

def MST_PrimJarnik(g):
    d = {}                      #  $d[v]$  is bound on distance to tree
    tree = []                    # list of edges in spanning tree
    pq = AdaptableHeapPriorityQueue() #  $d[v]$  maps to value  $(v, e=(u,v))$ 
    pqlocator = {}               # map from vertex to its pq locator

    # for each vertex  $v$  of the graph, add an entry to the priority queue, with
    # the source having distance 0 and all others having infinite distance
    for v in g.vertices():
        if len(d) == 0:           # this is the first node
            d[v] = 0              # make it the root
        else:
            d[v] = float('inf')   # positive infinity
        pqlocator[v] = pq.add(d[v], (v, None))

    while not pq.is_empty():
        key,value = pq.remove_min()
        u,edge = value          # unpack tuple from pq
        del pqlocator[u]         #  $u$  is no longer in pq
        if edge is not None:
            tree.append(edge)    # add edge to tree
        for link in g.incident_edges(u):
            v = link.opposite(u)
            if v in pqlocator:    # thus  $v$  not yet in tree
                # see if edge  $(u,v)$  better connects  $v$  to the growing tree
                wgt = link.element()
                if wgt < d[v]:      # better edge to  $v$ ?
                    d[v] = wgt      # update the distance
                    pq.update(pqlocator[v], d[v], (v, link)) # update the pq entry

    return tree

```

Kruskal algoritam

- particija čvorova u klastere (grozdove)
 - inicijalno klasteri sa po jednim čvorom
 - održava se MST za svaki klaster
 - spajanje „najbližih“ klastera i njihovih MST
- red sa prioritetom čuva grane izvan klastera
 - ključ: težina
 - vrednost: grana
- na kraju rada algoritma: jedan klaster i jedno MST

Kruskal algoritam

Kruskal(G)

for all $v \in G$ **do**

postavi osnovni klaster $C(v) = \{v\}$

napuni RSP Q granama iz G koristeći težine kao ključeve

$T \leftarrow \emptyset$

while T ima manje od $n - 1$ grana **do**

$(u, e) \leftarrow Q.\text{removeMin}()$

$C(u)$ je klaster koji sadrži u

$C(v)$ je klaster koji sadrži v

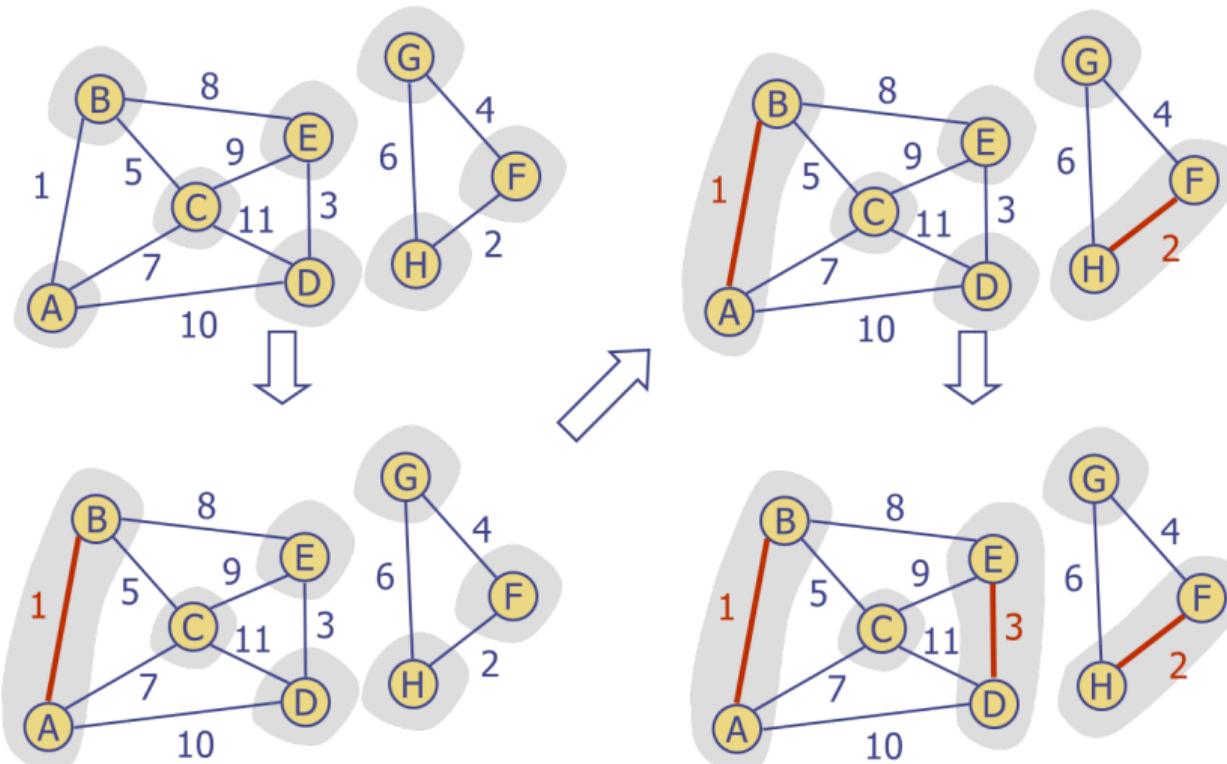
if $C(u) \neq C(v)$ **then**

dodaj granu (u, v) u T

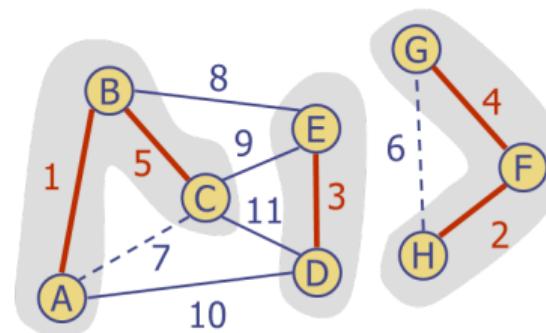
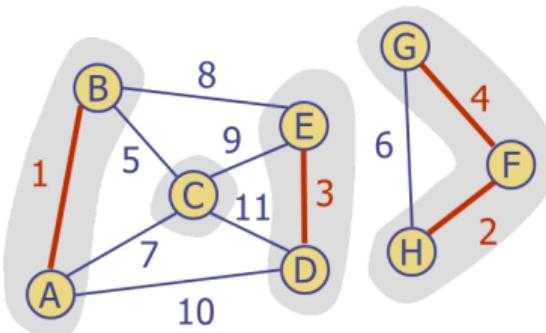
spoj $C(u)$ i $C(v)$ u jedan klaster

return T

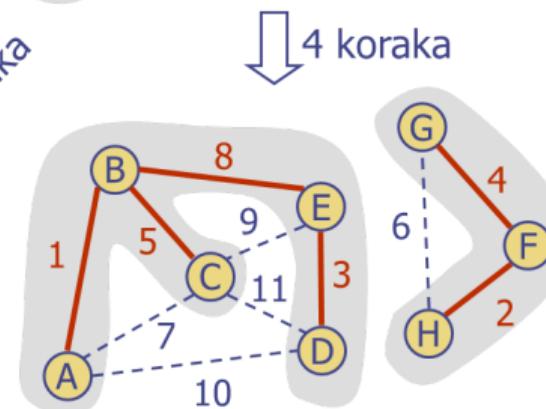
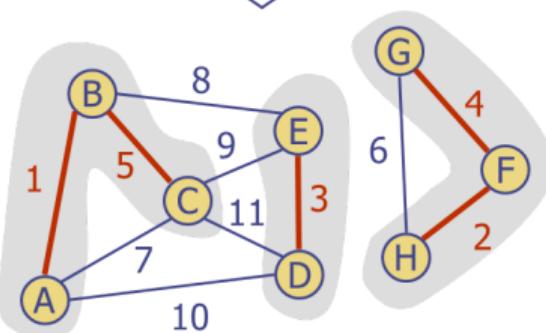
Kruskal primer 1



Kruskal primer 2



dva koraka

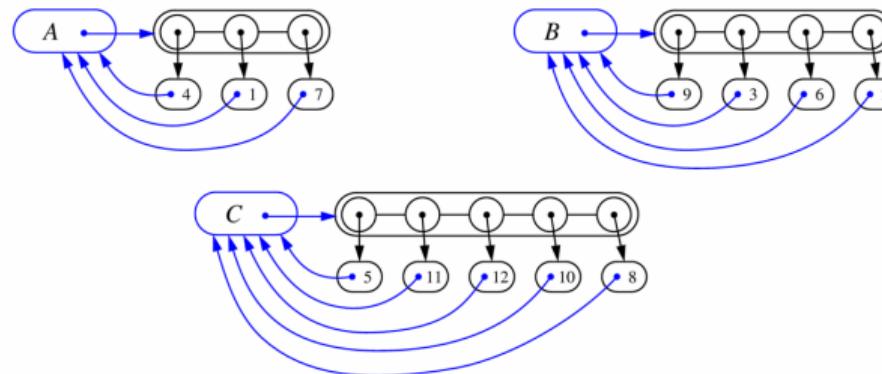


Struktura podataka za Kruskal algoritam: particija

- algoritam rukuje šumom stabala
- RSP daje grane u redosledu rastućih težina
- grana se prihvata ako spaja različita stabla
- potrebna je struktura podataka koja čuva **particiju**, tj. kolekciju disjunktnih skupova, sa operacijama
 - **makeSet(u)**: kreiraj skup koji sadrži u
 - **find(u)**: nađi skup koji sadrži u
 - **union(A, B)**: zameni skupove A i B njihovom unijom

Particija pomoću liste

- svaki skup se čuva u sekvenci
- svaki element ima referencu na skup
 - `find(u)` traje $O(1)$ i vraća skup koji sadrži u
 - `union(A, B)`: premeštamo elemente iz manjeg skupa u veći i ažuriramo reference na skup; traje $O(\min\{|A|, |B|\})$
- kada se element obradi, prelazi u skup koji je bar duplo veći, dakle svaki element se obrađuje najviše $\log n$ puta



Kruskal pomoću particije

- spajanje klastera pomoću **union**
- pronalaženje klastera pomoću **find**
- brzina je $O((n + m) \log n)$
 - operacije sa RSP: $O(m \log n)$
 - union-find operacije: $O(n \log n)$

Kruskal u Pythonu

```
def MST_Kruskal(g):
    tree = []                      # list of edges in spanning tree
    pq = HeapPriorityQueue()        # entries are edges in G, with weights as key
    forest = Partition()           # keeps track of forest clusters
    position = {}                  # map each node to its Partition entry

    for v in g.vertices():
        position[v] = forest.make_group(v)

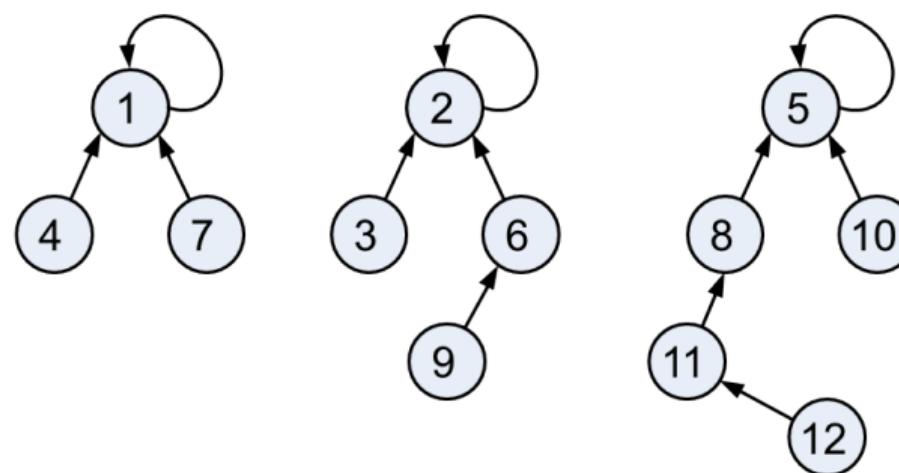
    for e in g.edges():
        pq.add(e.element(), e)      # edge's element is assumed to be its weight

    size = g.vertex_count()
    while len(tree) != size - 1 and not pq.is_empty():
        # tree not spanning and unprocessed edges remain
        weight,edge = pq.remove_min()
        u,v = edge.endpoints()
        a = forest.find(position[u])
        b = forest.find(position[v])
        if a != b:
            tree.append(edge)
            forest.union(a,b)

    return tree
```

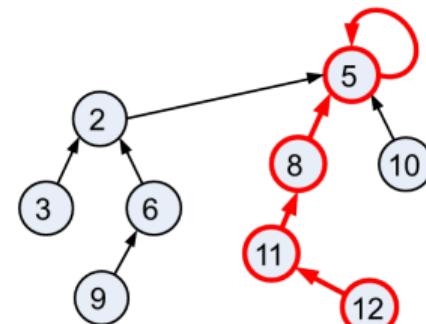
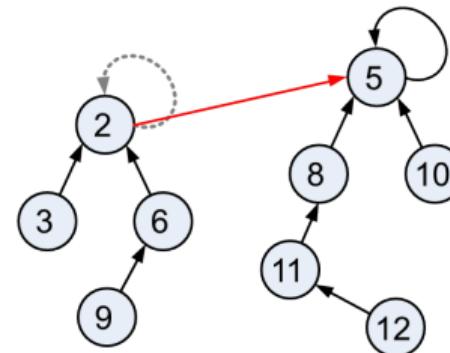
Particija pomoću stabla

- čvor stabla čuva element i referencu na skup
- čvor v čija referenca pokazuje na v je istovremeno i skup
- svaki skup je stablo, njegov koren čuva referencu na sebe
- na primer, skupovi „1“, „2“ i „5“



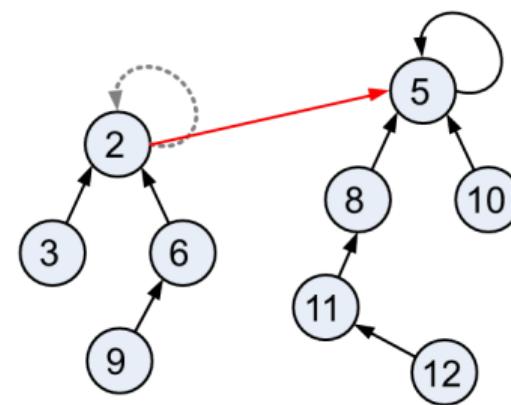
Particija pomoću stabla: union i find

- **union**: koren jednog stabla treba da pokazuje na koren drugog
- **find**: prati reference do korena koji pokazuje na samog sebe



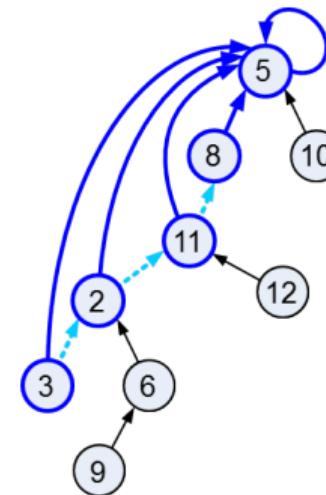
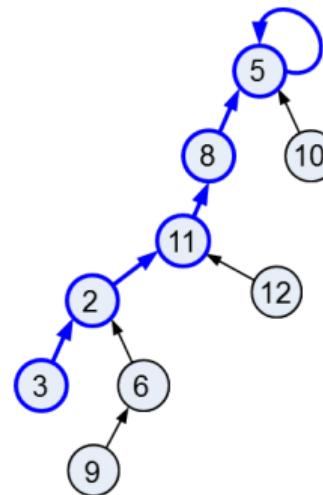
Recepti za union-find

- kada se radi union, neka koren manjeg stabla pokaže na koren većeg
- $O(n \log n)$ vreme za izvođenje n union-find operacija
 - svaki put kada pratimo referencu, idemo prema stablu koje je bar dvostruko veće od trenutnog
 - prema tome, pratimo najviše $O(\log n)$ referenci za svaki find



Recepti za union-find

- **kompresija putanje:** nakon find, neka sve reference čvorova koje smo upravo obišli pokazuju na koren



Particija u Pythonu 1

```
class Partition:
    class Position:
        __slots__ = '_container', '_element', '_size', '_parent'

    def __init__(self, container, e):
        """Create a new position that is the leader of its own group."""
        self._container = container # reference to Partition instance
        self._element = e
        self._size = 1
        self._parent = self           # convention for a group leader

    def element(self):
        """Return element stored at this position."""
        return self._element

    def _validate(self, p):
        if not isinstance(p, self.Position):
            raise TypeError('p must be proper Position type')
        if p._container is not self:
            raise ValueError('p does not belong to this container')
```

Particija u Pythonu 2

```
def make_group(self, e):
    """Makes a new group containing element e, and returns its Position."""
    return self.Position(self, e)

def find(self, p):
    """Finds the group containing p and return the position of its leader."""
    self._validate(p)
    if p._parent != p:
        p._parent = self.find(p._parent) # overwrite p._parent after recursion
    return p._parent

def union(self, p, q):
    """Merges the groups containing elements p and q (if distinct)."""
    a = self.find(p)
    b = self.find(q)
    if a is not b:                      # only merge if different groups
        if a._size > b._size:
            b._parent = a
            a._size += b._size
        else:
            a._parent = b
            b._size += a._size
```