

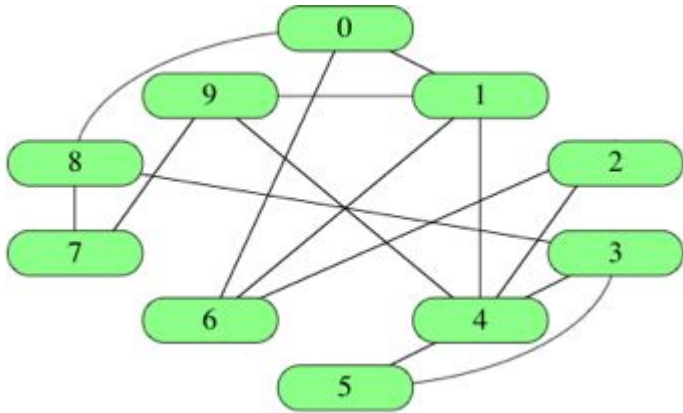
# Diskretna matematika 2

# Alati i biblioteke

- Python
- networkx
- matplotlib

# Predstavljanje grafa u računar

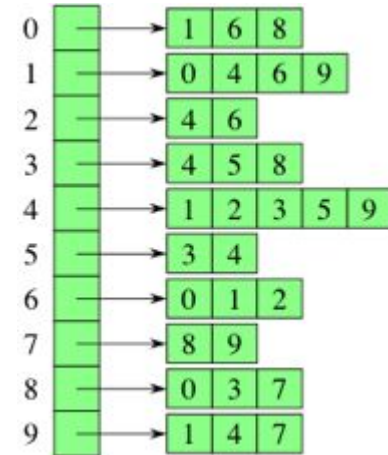
Graf



Matrica susjedstva

	0	1	2	3	4	5	6	7	8	9
0	0	1	0	0	0	0	1	0	1	0
1	1	0	0	0	1	0	1	0	0	1
2	0	0	0	0	1	0	1	0	0	0
3	0	0	0	0	1	1	0	0	1	0
4	0	1	1	1	0	1	0	0	0	1
5	0	0	0	1	1	0	0	0	0	0
6	1	1	1	0	0	0	0	0	0	0
7	0	0	0	0	0	0	0	0	1	1
8	1	0	0	1	0	0	0	1	0	0
9	0	1	0	0	1	0	0	1	0	0

Lista susjedstva



# Biblioteka networkx

Python biblioteka za rad sa grafovima

- Strukture za predstavljanje
  - Grafova - neusmjerenih grafova bez višestrukih grana
  - Digrafa - usmjerenih grafova
  - Multigrafova - grafova sa višestrukim grandma i petljama
- Standardne algoritme nad grafovima
- Algoritme za analiz grafa
- Poznate grafove, slučajno generisanje grafova
- Crtanje grafa
- ...

# Instalacija biblioteke

Networkx možete instalirati na svom računaru izvršavanjem naredbe

```
pip3 install networkx
```

Ili možete koristiti google colab za vježbu

<https://colab.research.google.com/>

Napomena:

Kolokvijum će se realizovati na univerzitetskim računarima bez pristupa internetu, studenti će imati instaliran python, visual studio code i sve potrebne biblioteke.

# Networkx biblioteka



```
import networkx as nx
import matplotlib.pyplot as plt
```

Uključujemo biblioteku networkx za rad sa grafovima i biblioteku pyplot iz paketa matplotlib koji služi za vizualizaciju



```
G = nx.Graph()
G.add_node(1)
G.add_nodes_from([2, 3])
G.add_edge(1, 2)
G.add_edges_from([(1, 3), (2, 3)])
G.add_edge(5, 6)
```

--	--	--	--	--	--	--	--	--	--

```
#kreiramo prazan neusmjereni graf
#dodajemo čvor 1
#dodajemo čvorove 2 i 3
#dodajemo granu između čvorova 1 i 2
#dodajemo grane (1, 3) i (2, 3)
#dodajemo granu između čvorova 5 i 6,
#čvorovi 5 i 6 se automatski dodaju u graf
```

# Networkx biblioteka

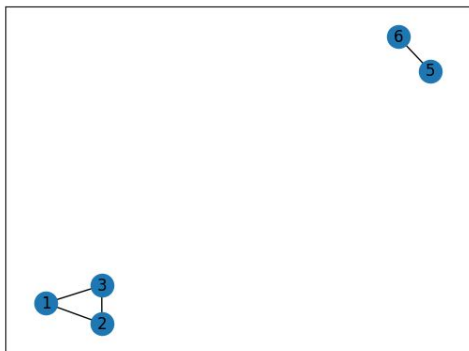
```
▶ ~ print(f'Graf G sadrži {G.number_of_nodes()} čvorova i {G.number_of_edges()} grana.')
```

[16] ✓ 0.0s

... Graf G sadrži 5 čvorova i 4 grana.

```
▶ ~ nx.draw_networkx(G) #crtamo graf  
plt.show() #prikazujemo graf
```

[17] ✓ 0.0s



# Networkx biblioteka

```
Gradovi = nx.Graph()

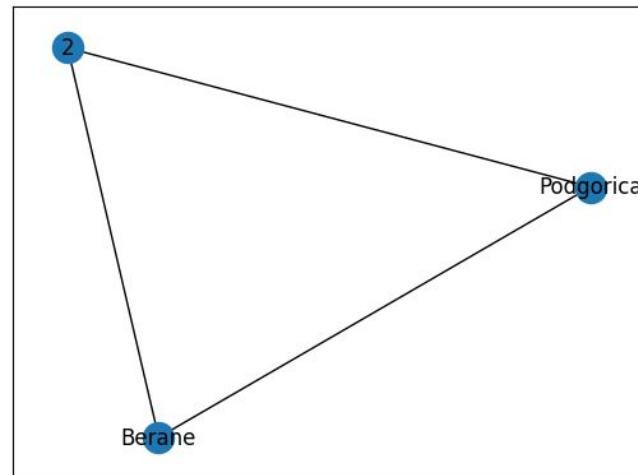
Gradovi.add_node("Berane")
Gradovi.add_node("Podgorica")
Gradovi.add_node(2)

Gradovi.add_edge("Berane", 2)
Gradovi.add_edge("Berane", "Podgorica")
Gradovi.add_edge("Podgorica", 2)

nx.draw_networkx(Gradovi)
plt.show()
```

[38]

✓ 0.0s



Čvorovi grafa mogu da budu bilo šta, brojevi, riječi, json dokumenti ...

Svakoj grani se mogu dodijeliti proizvoljni atributi

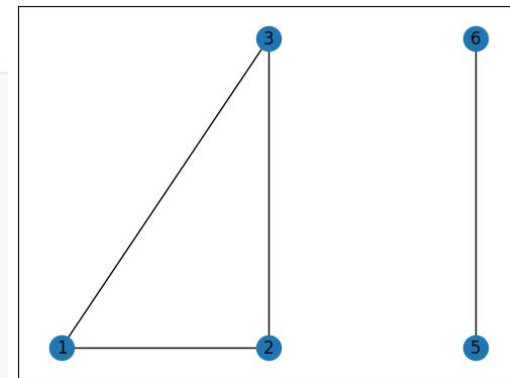


# Networkx biblioteka

```
pos = {  
    1: (0, 0),  
    2: (1, 0),  
    3: (1, 1),  
    5: (2, 0),  
    6: (2, 1)  
}  
  
nx.draw_networkx(G, pos)  
plt.show()
```

[18] ✓ 0.0s

#crtamo graf s pozicijama čvorova  
#prikazujemo graf



Za specificiranje pozicija koristimo dictionary koji za svaki čvor sadrži koordinate na kojima će se on prikazati.

# Networkx biblioteka

```
G = nx.complete_graph(5)          #generiše kompletan graf sa 5 čvorova  
pos = nx.spring_layout(G)
```

Position nodes using Fruchterman-Reingold force-directed algorithm.

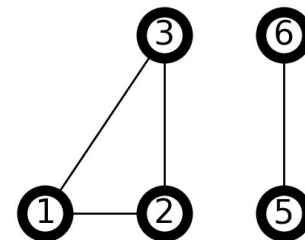
The algorithm simulates a force-directed representation of the network treating edges as springs holding nodes close, while treating nodes as repelling objects, sometimes called an anti-gravity force. Simulation continues until the positions are close to an equilibrium.

```
nx.draw_networkx_nodes(G, pos, nodelist=G.nodes, node_size=700, node_color='skyblue') #crtā čvorove  
nx.draw_networkx_edges(G, pos, edgelist=G.edges, edge_color='gray', alpha=0.3)         #crtā listu grana  
nx.draw_networkx_labels(G, pos, font_color='black')                                 #crtā labele
```

# Networkx biblioteka

```
options = {  
    "font_size": 32,  
    "node_size": 2000,  
    "node_color": "white",  
    "edgecolors": "black",  
    "linewidths": 10, #debljina ivica čvorova  
    "width": 2 #debljina linija  
}  
  
nx.draw_networkx(G, pos, **options)    #crtamo graf s pozicijama čvorova  
  
ax = plt.gca()                        #uzimamo referencu na objekat koji p  
ax.margins(0.2)                       #povećavamo margine grafika  
plt.axis("off")                       #isključujemo prikaz osa  
plt.show()                           #prikazujemo graf
```

[43] ✓ 0.0s



**\*\*options**, zamjenjuje vrijednosti iz dictionary u argumente funkcije

# Čvorovi i grane grafa

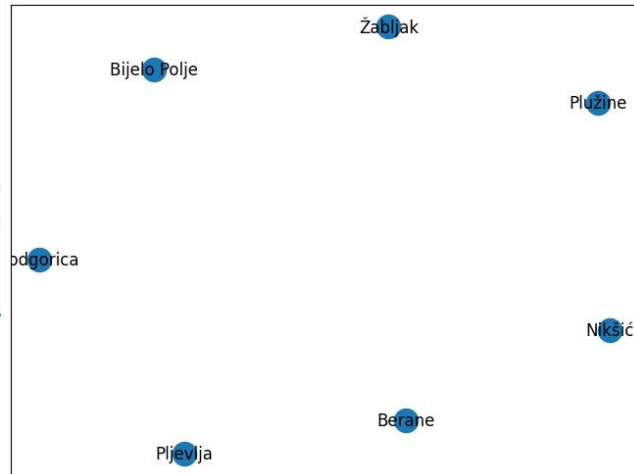


```
G = nx.Graph()

G.add_node('Berane')
G.add_nodes_from(['Podgorica', 'Nikšić', 'Pljevlja', 'Bijelo Polje'])
G.add_nodes_from(
    [('Plužine', {'broj_stanovnika': 5000}),
     ('Žabljak', {'broj_stanovnika': 4000})])

nx.draw(G, with_labels=True)
plt.axis("on")
plt.show()
```

[16] ✓ 0.1s



# Čvorovi i grane grafa

```
▶ print(G.nodes['Plužine'])
```

```
[22] ✓ 0.0s
```

```
... {'broj_stanovnika': 5000}
```

```
▶ G.nodes['Podgorica']['broj_stanovnika'] = 200000  
G.nodes.data()
```

```
[27] ✓ 0.0s
```

```
... NodeDataView({'Berane': {}, 'Podgorica': {'broj_stanovnika': 200000}, 'Nikšić': {}, 'Pljevlja': {}, 'Bijelo Polje': {}, 'Plužine': {'broj_stanovnika': 5000},
```

# Čvorovi i grane

```
PMF = nx.Graph(opis='Prirodno-matematički fakultet')
PMF.add_node("Andrijana", tip="profesor")
PMF.add_nodes_from([
    ("Mina", {"tip": "student", "godina_upisa": 2019}),
    ("Janko", {"tip": "student", "godina_upisa": 2018})
])

PMF.add_node("Diskretna matematika", broj_casova=3)
PMF.add_nodes_from([
    ("Algebra", {"broj_casova": 4}),
    ("Analiza", {"broj_casova": 4})
])
PMF.add_edges_from([
    ("Andrijana", "Diskretna matematika"),
    ("Mina", "Algebra", {"ocjena": 10}),
    ("Janko", "Diskretna matematika", {"ocjena": 9}),
    ("Janko", "Analiza", {"ocjena": 8})
])
```

```
PMF.nodes.data()
```

```
[35] ✓ 0.0s
```

```
Python
```

```
NodeDataView({'Andrijana': {'tip': 'profesor'}, 'Mina': {'tip': 'student', 'godina_upisa': 2019}, 'Janko': {'tip': 'student', 'godina_upisa': 2018}, 'Diskretna matematika': {'broj_casova': 3}, 'Algebra': {'broj_casova': 4}, 'Analiza': {'broj_casova': 4}})
```

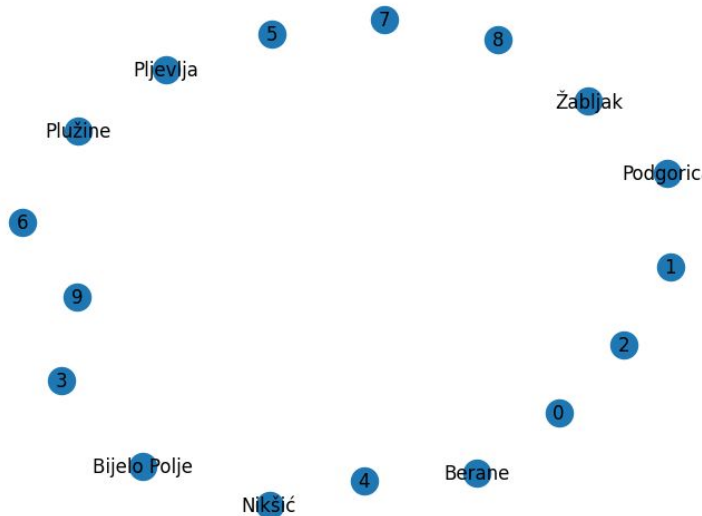
Možemo da dodijelimo atribute grafu, čvoru ili grani.

# Čvorovi i grane

```
▶ ~  
T = nx.path_graph(10)  
G.add_nodes_from(T)  
nx.draw(G, with_labels=True)
```

[42] ✓ 0.0s

Možemo da dodamo čvorove jednog grafa u drugi graf.



# Čvorovi i grane

```
▷ ▾  
nx.draw(G, with_labels=True)
```

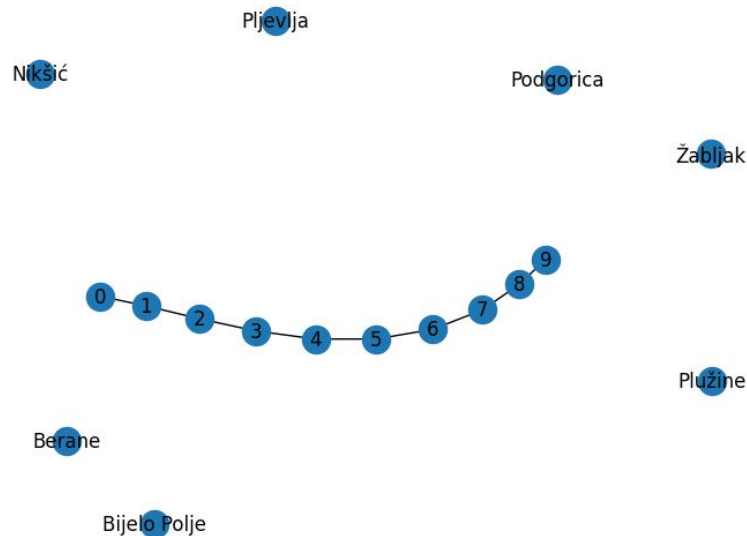
[47] ✓ 0.0s

Možemo da dodamo grane jednog grafa u drugi graf.

```
▷ ▾  
G.clear()
```

[48] ✓ 0.0s

Briše sve što se nalazi u grafu.





# Svojstva grafa

Iz grafa možemo da dobijemo **spisak čvorova**, **grana**, **stepen čvorova** i **listu susjedstva**.

Dobijamo objekte slične dictionaryu koje možemo da kastujemo u listu ili dictionary.

# Svojstva grafa

```
▷ ▾  
G = nx.Graph()  
G.add_node(1)  
G.add_nodes_from([2, 3])  
G.add_edge(1, 2)  
G.add_nodes_from('abc') # add nodes 'a', 'b', 'c'  
G.add_node('efg') # add node 'efg'  
G.add_edges_from([('a', 'b'), 'ac', 'bc'])
```

[19] ✓ 0.0s

```
▷ ▾  
list(G.nodes) #dobijemo neki iterable container, i onda da bi dobili listu, moramo da ga castujemo u listu
```

[23] ✓ 0.0s

... [1, 2, 3, 'a', 'b', 'c', 'efg']

```
list(G.edges)
```

[24] ✓ 0.0s

... [(1, 2), ('a', 'b'), ('a', 'c'), ('b', 'c')]

Gen

Start Chat to Gener

# Svojstva grafa



```
dict(G.adj)
```

[28]

✓ 0.0s

```
... {1: AtlasView({2: {}}),  
     2: AtlasView({1: {}}),  
     3: AtlasView({}),  
     'a': AtlasView({'b': {}, 'c': {}}),  
     'b': AtlasView({'a': {}, 'c': {}}),  
     'c': AtlasView({'a': {}, 'b': {}}),  
     'efg': AtlasView({})}
```



```
list(G.adj['a'])
```

[32]

✓ 0.0s

```
... ['b', 'c']
```

# Svojstva grafa



```
dict(G.degree)
```

[35] ✓ 0.0s

... {1: 1, 2: 1, 3: 0, 'a': 2, 'b': 2, 'c': 2, 'efg': 0}



```
G.degree['a']
```

[36] ✓ 0.0s

... 2

# Uklanjanje čvorova i grana

```
▶ G.remove_node('a')  
G.remove_nodes_from([1, 2])  
G.remove_edge('b', 'c')  
G.remove_edges_from([('a', 'b'), ('b', 'c')])
```

[47] ✓ 0.0s

Uklanjanjem čvora uklanjanju se i sve njegove grane. Ako čvor/grana nije u grafu dobićemo exception prilikom poziva metode `remove_node` / `remove_edge`. Metode `remove_nodes_from` i `remove_edges_from` ne bacaju exception.

# Vizualizacija grafa

```
# Perform DFS and construct the DFS tree
dfs_edges = list(nx.dfs_edges(graph, source=start_node))
dfs_tree = nx.DiGraph(dfs_edges)

# Plot settings
fig, axes = plt.subplots(1, 2, figsize=(12, 6))

# Draw original graph
ax = axes[0]
pos = nx.spring_layout(graph) # Layout for node positioning
nx.draw(graph, pos, ax=ax, with_labels=True, node_color='lightblue', edge_color='gray')
ax.set_title("Original Graph")

# Draw DFS tree
ax = axes[1]
nx.draw(dfs_tree, pos, ax=ax, with_labels=True, node_color='lightgreen', edge_color='red', arrows=True)
ax.set_title("DFS Tree")

plt.show()
```

# Vizualizacija grafa

```
G = nx.Graph()
G.add_edges_from([
    ('a', 'b', {'weight': 5}),
    ('c', 'b', {'weight': 2}),
    ('e', 'b', {'weight': 5}),
    ('a', 'e', {'weight': 3})
])

pos = nx.spring_layout(G)
nx.draw_networkx_nodes(G, pos, nodelist=G.nodes)
nx.draw_networkx_labels(G, pos, font_color='black')
nx.draw_networkx_edges(G, pos, edgelist=G.edges)
nx.draw_networkx_edge_labels(G, pos, {e: v['weight'] for e, v in G.edges.items()}, label_pos=0.7,
verticalalignment='bottom')
```

# DFS algoritam

```
dict(nx.traversal.dfs_edges(G, 'a'))
```

1. Napisati program koji u prvom redu učitava broj čvorova ( $n$ ) i broj grana ( $m$ ) neusmjerenog grafa  $G$ .  
U narednih  $m$  redova se učitavaju po dva broja koji predstavljaju indekse čvorova jedne grane.  
U  $m+2$  liniji se učitava indeks još jednog čvora  $u$ .  
Potrebno je ispitati da li je:
  - a. Sve čvorove koji se nalaze u istoj komponenti povezanosti kao čvor  $u$ .
  - b. Koliko komponenti povezanosti sadrži graf  $G$ .
  - c. Štampati najveću komponentu povezanosti.
  - d. Da li je graf  $G$  bipartitan.
  - e. Da li  $G$  sadrži cikluse.
  - f. Prikazati graf  $G$  i stablo DFS obilaska grafa  $G$ .



# Isctavanje grafa

```
def visualize_graph_and_dfs_tree(graph, start_node):  
    # Perform DFS and construct the DFS tree  
    dfs_edges = list(nx.dfs_edges(graph, source=start_node))  
    dfs_tree = nx.DiGraph(dfs_edges)  
  
    # Plot settings  
    fig, axes = plt.subplots(1, 2, figsize=(12, 6))  
  
    # Draw original graph  
    ax = axes[0]  
    pos = nx.spring_layout(graph) # Layout for node positioning  
    nx.draw(graph, pos, ax=ax, with_labels=True, node_color='lightblue', edge_color='gray')  
    ax.set_title("Original Graph")  
  
    # Draw DFS tree  
    ax = axes[1]  
    nx.draw(dfs_tree, pos, ax=ax, with_labels=True, node_color='lightgreen', edge_color='red', arrows=True)  
    ax.set_title("DFS Tree")  
  
    plt.show()
```

# Iscrtavanje grafa

```
G = nx.Graph()
edges = [('A', 'B'), ('A', 'C'), ('B', 'D'), ('C', 'D'),
         ('D', 'E'), ('E', 'F'), ('E', 'G'), ('G', 'H'), ('F', 'H')]
G.add_edges_from(edges)
d_edges = list(nx.dfs_edges(G, source='A')) # Grane koje DFS obidje

plt.figure(figsize=(10, 10))
pos = nx.spring_layout(G)

nx.draw_networkx_nodes(G, pos, nodelist=G.nodes, node_size=700, node_color='skyblue')
nx.draw_networkx_edges(G, pos, edgelist=G.edges, edge_color='gray', alpha=0.3)
nx.draw_networkx_labels(G, pos, font_color='black')
nx.draw_networkx_edges(G, pos, edgelist=d_edges, edge_color='red')

plt.title('Iscrtavanje v2')
plt.show()
```

# Grafički nizovi

2. Napisati program koji provjerava da li je niz čvorova  $\{d_1, d_2, d_3, \dots, d_{n-1}, d_n\}$ , grafički.

Teorema: Niz  $\{d_1, d_2, d_3, \dots, d_{n-1}, d_n\}$  je grafički akko je grafički niz  $\{d_2-1, d_3-1, \dots, d_{d_1-1}-1, d_{d_1}-1, d_{d_1+1}-1, \dots, d_{n-1}, d_n\}$ .

# DFS algoritam

In Depth First Search (or DFS) for a graph, we traverse all adjacent vertices one by one. When we traverse an adjacent vertex, we completely finish the traversal of all vertices reachable through that adjacent vertex

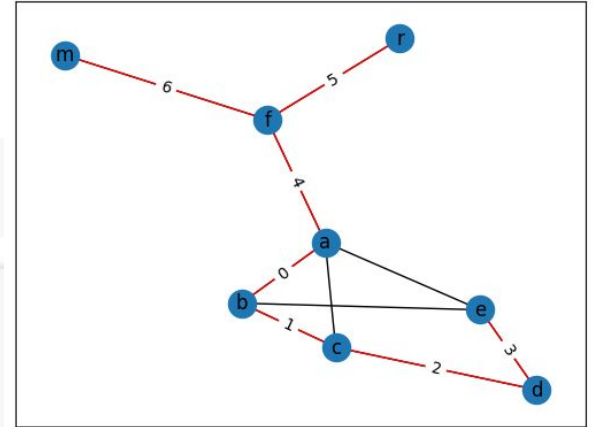
- Koristi se za provjeru da li su dva čvora u istoj komponenti povezanosti
- Koliko imamo komponenti povezanosti
- Nalaženje artikulacionih čvorova
- Nalaženje mostova
- Nalaženje ciklusa u grafu
- ...

# DFS algoritam (dfs\_edges)

Vraća listu grana redosljedom kako su obiđene DFS algoritmom.

```
[21] dfs_edges = list(nx.traversal.dfs_edges(g, 'a'))
```

```
▶ pos = nx.spring_layout(g)
  nx.draw_networkx_nodes(g.nodes, pos)
  nx.draw_networkx_edges(g, pos, g.edges)
  nx.draw_networkx_edges(g, pos, dfs_edges, edge_color='red')
  nx.draw_networkx_labels(g, pos)
  nx.draw_networkx_edge_labels(g, pos, {e: i for i, e in enumerate(dfs_edges)})
```



# Nalazenje komponenti povezanosti

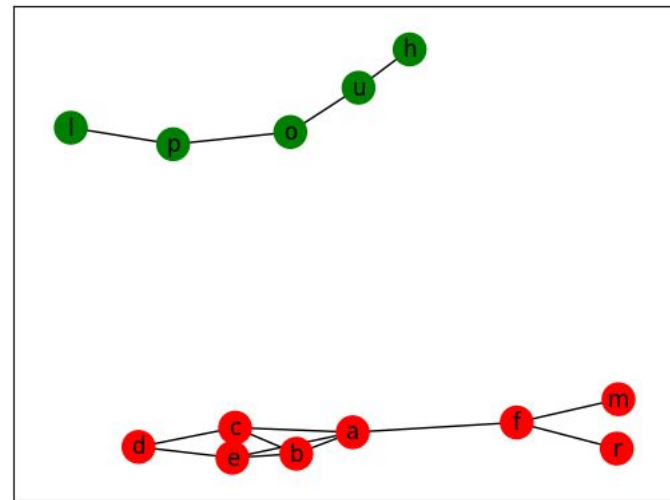
```
[29] components = {n:-1 for n in g.nodes}

cnt = 0
for n, c in components.items():
    if c == -1:
        dfs_edges = list(nx.dfs_edges(g, n))

        for u, v in dfs_edges:
            components[u] = components[v] = cnt

        cnt += 1

print(components)
```



⇒ {'a': 0, 'b': 0, 'c': 0, 'd': 0, 'e': 0, 'f': 0, 'r': 0, 'm': 0, 'h': 1, 'u': 1, 'o': 1, 'p': 1, 'l': 1}

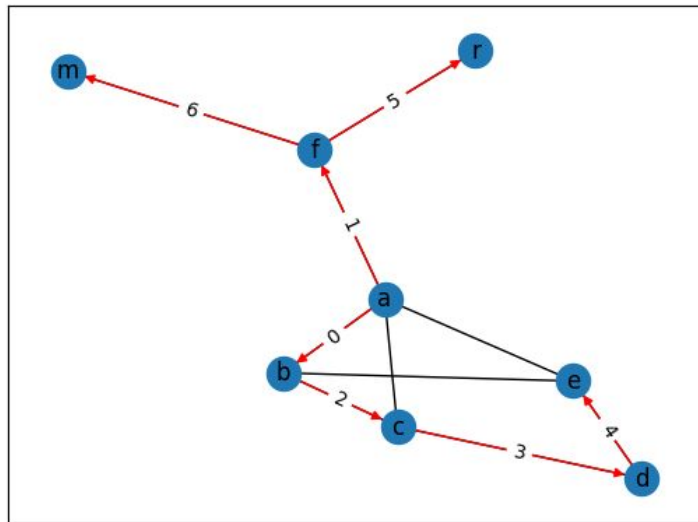
```
▶ colors = ['red', 'green']
colors = [colors[components[n]] for n in g.nodes]
nx.draw_networkx_nodes(g, pos, node_color=colors, cmap=plt.cm.rainbow)
nx.draw_networkx_edges(g, pos, edgelist=g.edges)
nx.draw_networkx_labels(g, pos)
plt.show()
```

# DFS algoritam (dfs\_tree)

Vraća stablo sa čvorovima usmjerenim grandma u smjeru kako se je DFS algoritmom kretao.

```
[26] dfs_tree = nx.dfs_tree(g, 'a')
```

```
[36] nx.draw_networkx_nodes(g.nodes, pos)
      nx.draw_networkx_edges(g, pos, g.edges)
      nx.draw_networkx_labels(g, pos)
      nx.draw_networkx_edges(dfs_tree, pos, dfs_tree.edges, edge_color='red')
      nx.draw_networkx_edge_labels(g, pos, {e: i for i, e in enumerate(dfs_tree.edges)})
```

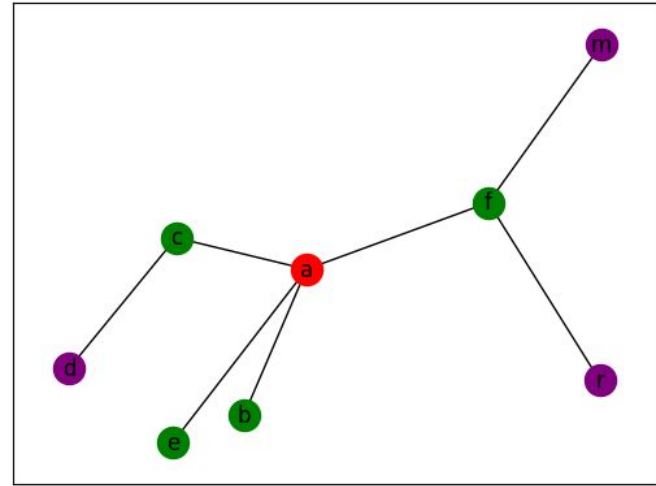


# BFS algoritam

bfs\_edges - vraća grane redosljedom kojim su obiđene u BFS algoritmu

bfs\_layers - vraća slojeve u bfs obilasku

```
▶ bfs_edges = list(nx.bfs_edges(g, 'a'))  
bfs_layers = list(nx.bfs_layers(g, 'a'))  
  
colors = ['red', 'green', 'purple']  
  
for i, layer in enumerate(bfs_layers):  
    nx.draw_networkx_nodes(g, pos, layer, node_color=colors[i])  
    nx.draw_networkx_labels(g, pos, {l: l for l in layer})  
  
nx.draw_networkx_edges(g, pos, bfs_edges)
```





# BFS algoritam

Napisati program koji provjerava da li je graf bipartitan

```
▶ def bipartite(g):  
    bfs_layers = list(nx.bfs_layers(g, 'a'))  
    parity = {n: -1 for n in g.nodes}  
  
    for n in g.nodes:  
        if parity[n] == -1:  
            bfs_layers = list(nx.bfs_layers(g, 'a'))  
  
            for i, nodes in enumerate(bfs_layers):  
                for node in nodes:  
                    parity[node] = i % 2  
  
    for u, v in g.edges:  
        if parity[u] == parity[v]:  
            return False  
  
    return True  
  
bipartite(g)
```

# DFS algoritam

```
print(nx.is_connected(G))  
print(nx.is_bipartite(G))  
print(nx.cycle_basis(G))  
print(list(nx.simple_cycles(G)))
```

# Čvorna povezanost grafa

```
print(nx.all_pairs_node_connectivity(G))
```

Za svaki par čvorova  $(u, v)$  dobijamo minimalan broj čvorova koje treba ukloniti iz grafa  $G$  da bi čvorovi  $u$  i  $v$  bili u različitim komponentama.



TO BE CONTINUED