Esercitazione1

June 11, 2023

1 Esercitazione 1

[1]:
```
import networkx as nx
import copy
```

[2]:
```
def drawGraph(graph):
    G = nx.Graph(graph)
    nx.draw_networkx(G, pos=nx.planar_layout(G))

def drawDiGraph(di_graph):
    G = nx.DiGraph(di_graph)
    nx.draw_networkx(G, pos=nx.planar_layout(G))
```

1.1 Es. 1. Componenti fortemente connesse archi esterni/interni in una DFS

Testo

Per un grafo diretto G, diciamo che un arco da u a v e' interno se u e v appartengono alla stessa componente fortemente connessa e altrimenti diciamo che e' esterno. In relazione ad una qualsiasi DFS, rispondere alle seguenti domande: 1. un arco in avanti puo' essere esterno? 2. un arco di attraversamento puo' essere interno? 3. un arco di attraversamento puo' essere esterno? 4. un arco all'indietro puo' essere esterno?

Per ognuna, in caso affermativo esibire un esempio e altrimenti dimostrare l'impossibilita'.

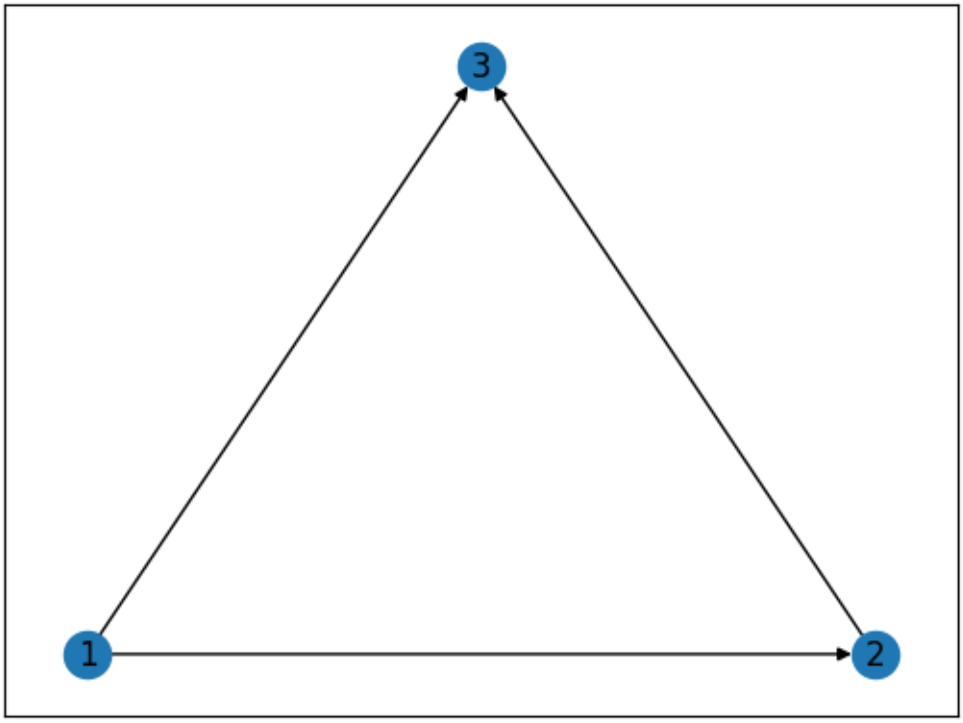
Ricorda che

Un arco in avanti (forward edge) in una DFS rappresenta un arco che collega un nodo a un suo discendente nell'albero DFS.
Un arco di attraversamento (cross edge) rappresenta un arco che collega due nodi non in relazione gerarchica tra loro, ovvero non uno dei due è discendente dell'altro.
Un arco all'indietro (back edge) rappresenta un arco che collega un nodo ad un suo antenato nell'albero DFS.

Soluzione

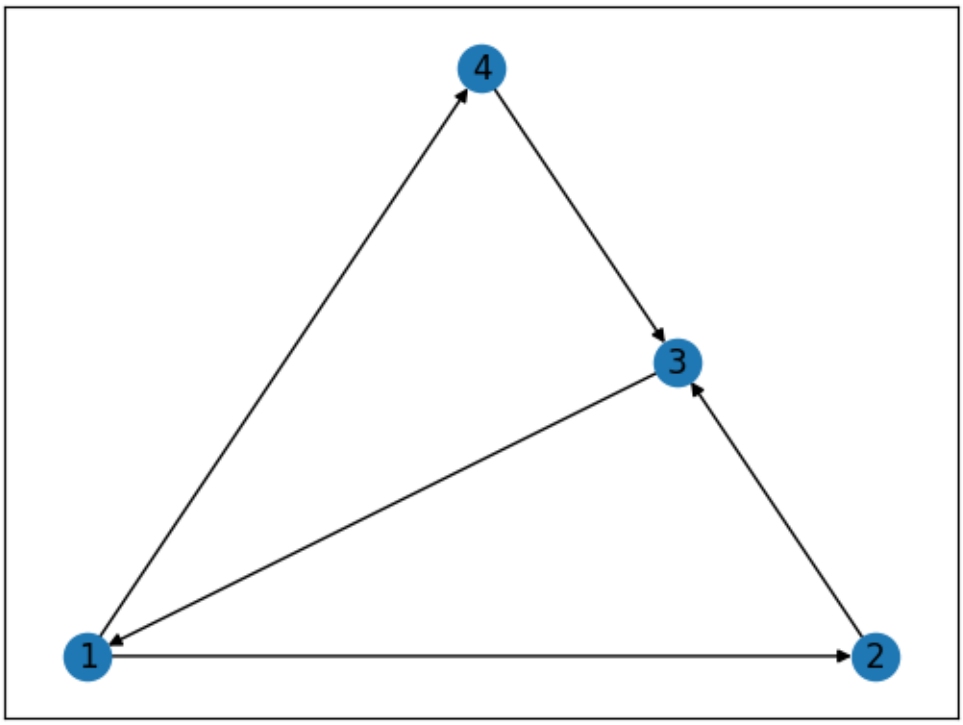
1. Sì, un arco in avanti può essere esterno. Nell'esempio in figura l'arco (1,3) e' un arco in avanti della visita DFS (1, 2, 3) che collega due nodi di due diverse componenti fortemente connesse.

[3]:
```
graph = {
    1 : [2, 3],
    2 : [3],
    3 : []
}
drawDiGraph(graph)
```



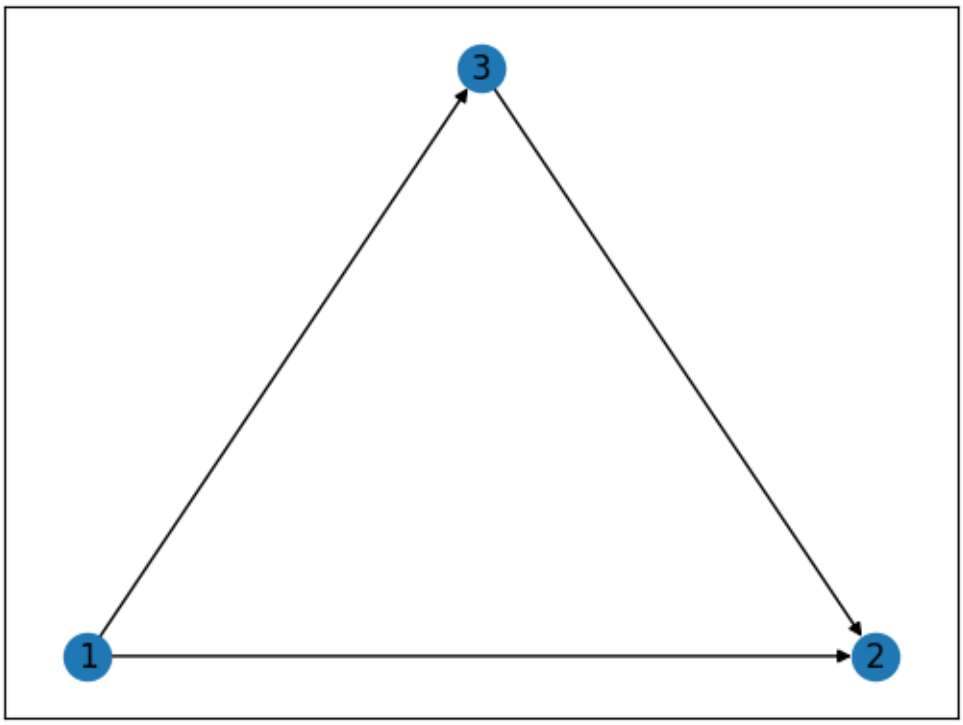
2. Sì, un arco di attraversamento può essere interno. Nell'esempio in figura l'arco (4,3) e' un arco di attraversamento della visita DFS (1, 2, 3, 4) che collega due nodi della stessa componente fortemente connessa.

[4]:
```
graph = {
    1 : [2, 4],
    2 : [3],
    3 : [1],
    4 : [3]
}
drawDiGraph(graph)
```



3. Sì, un arco di attraversamento può essere esterno. Nell'esempio in figura l'arco (3,2) e' un arco in avanti della visita DFS (1, 2, 3) che collega due nodi di due diverse componenti fortemente connesse.

[5]:
```
graph = {
    1 : [2, 3],
    2 : [],
    3 : [2]
}
drawDiGraph(graph)
```

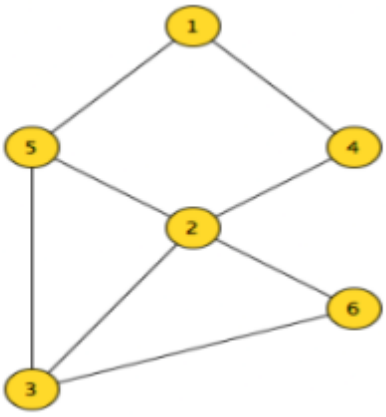


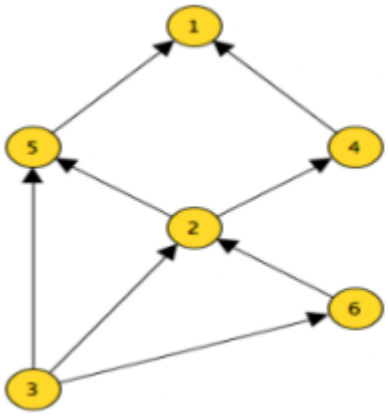
4. No, un arco all'indietro non può essere esterno, in quanto collega un nodo ad un suo antenato, e quindi entrambi i nodi appartengono alla stessa componente fortemente connessa.

1.2 Es. 2. Algoritmo per orientare gli archi in un grafo non orientato in modo da ottenere un DAG

Testo

Dato un grafo G, descrivere un algoritmo che ne orienta gli archi in modo da creare un grafo G' diretto e aciclico. L'algoritmo deve avere complessita' O(n + m).





Ad esempio per il grafo sopra a sinistra un orientamento degli archi lecito e' quello riportato sopra a destra.

Idea 1

Per ogni nodo u segniamo u come visitato e osserviamo tutti i suoi archi uscenti verso gli adiacenti v. Per ogni adiacente v, se v e' segnato come visitato, direzioniamo l'arco da v ad u, altrimenti da u a v. Per costruzione otterremo un DAG.

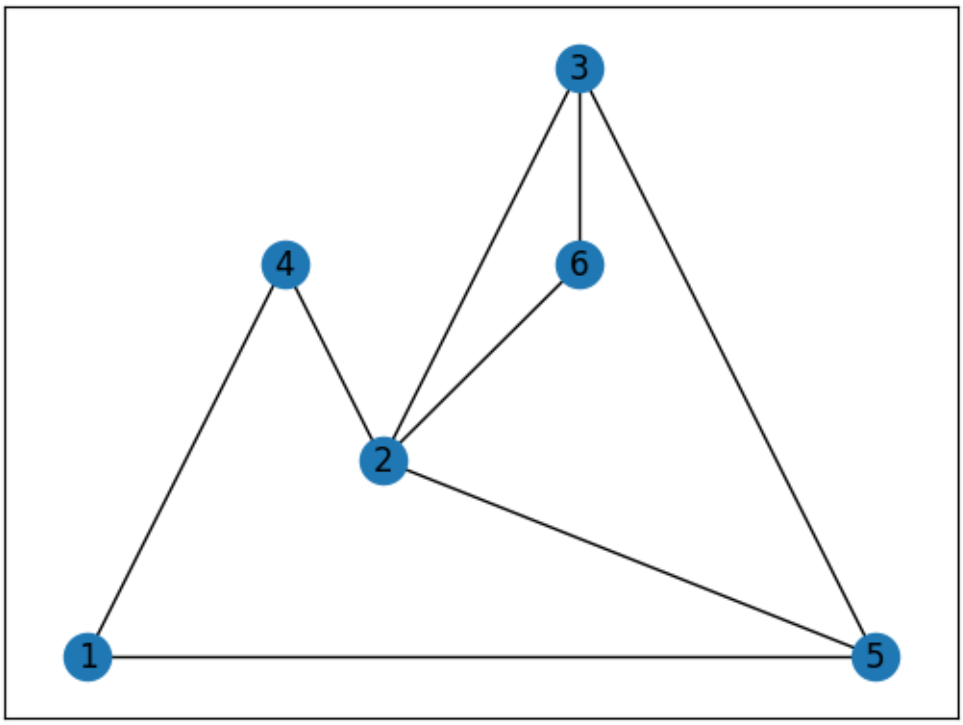
Soluzione 1

[6]:
```
def generate_DAG(graph):
    di_graph = {node: set() for node in graph}
    visited = {node: False for node in graph}
    for node in graph:
        visited[node] = True
        for adjacent in graph[node]:
            if visited[adjacent]:
                di_graph[adjacent].add(node)
            else:
                di_graph[node].add(adjacent)
    return di_graph
```

Esecuzione 1

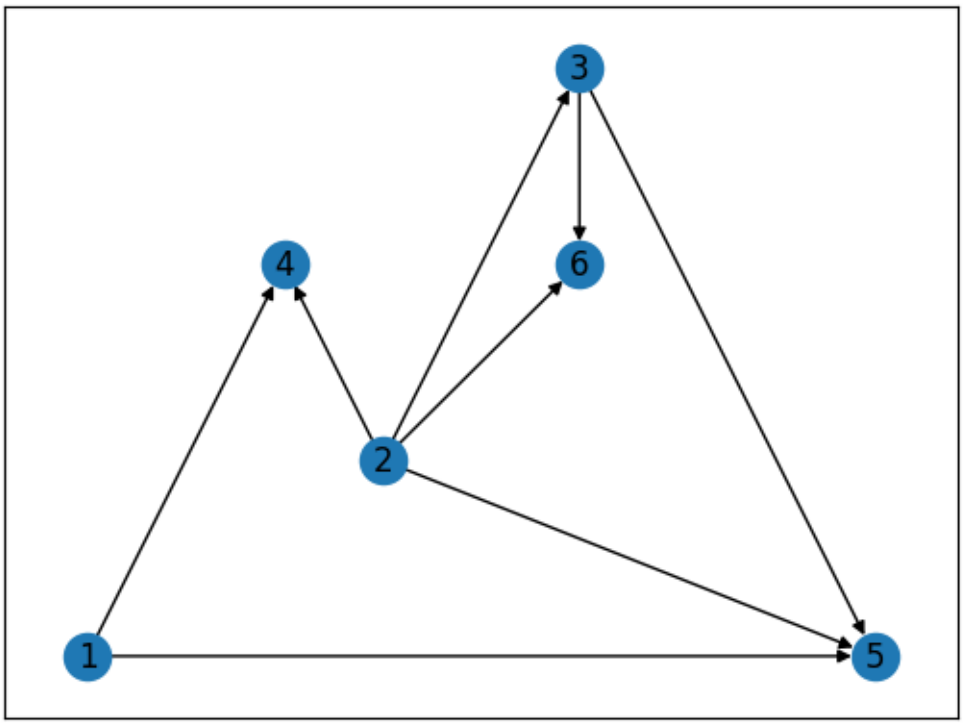
[7]:
```
graph = {
    1 : [4, 5],
    2 : [3, 4, 5, 6],
    3 : [2, 5, 6],
    4 : [1, 2],
    5 : [1, 2, 3],
    6 : [2, 3]
}
```

[8]:
```
drawGraph(graph)
```



[9]:
```
di_graph = generate_DAG(graph)
```

[10]:
```
drawDiGraph(di_graph)
```



Idea 2

Questo problema è più semplice di quanto non sembri.
Sappiamo che G è un DAG se e solo se è possibile ordinare i nodi in modo che u v implica che esiste un cammino da u a v in G. Per assicurare questa proprietà, è sufficiente ordinare in modo arbitrario i nodi e poi per ogni arco (u, v) nel grafo non orientato si mette un arco u→v se u v oppure v→u se v < u.
Più concretamente, avendo la matrice di adiacenza di G, e prendendo semplicemente l'ordine derivante dal posizionamento della matrice, è sufficiente ricopiare la matrice e cancellare tutti gli 1 nella parte inferiore della matrice (in cui i>j) [O(n2)].
Avendo le liste di adiacenza (e immaginando ancora di rappresentare i nodi con numeri interi) è sufficiente cancellare nella lista di adiacenza di u tutti i nodi v tali che v u [O(m+n)].

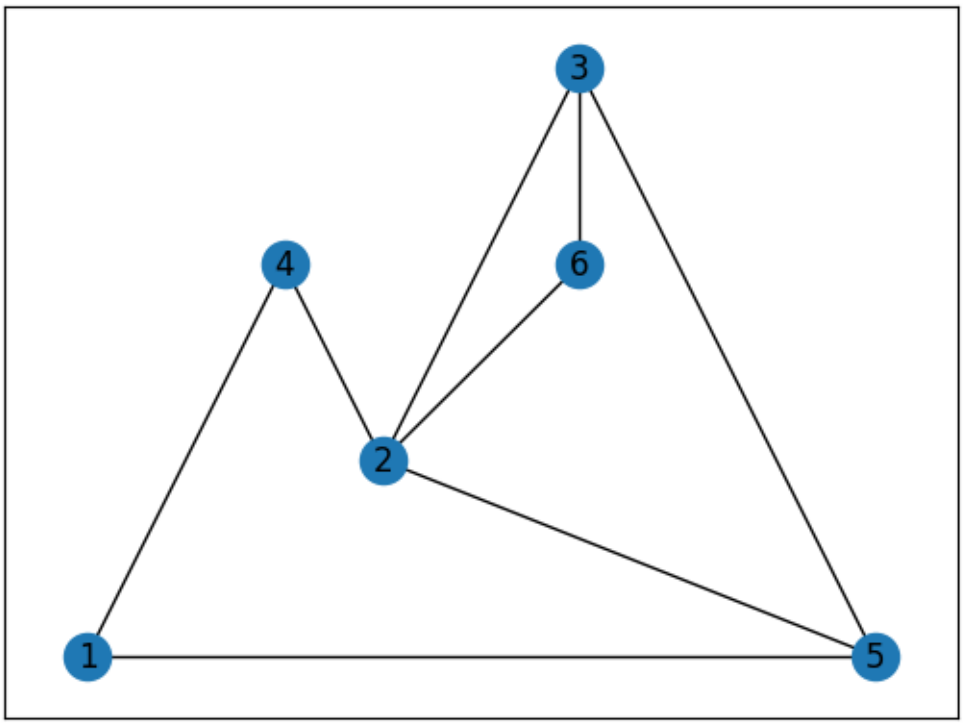
Soluzione 2

[11]:
```
def generate_DAG(graph):
    new_graph = copy.deepcopy(graph)
    for node in graph:
        for adjacent in graph[node]:
            if node < adjacent:
                new_graph[node].remove(adjacent)
    return new_graph
```

Esecuzione 2

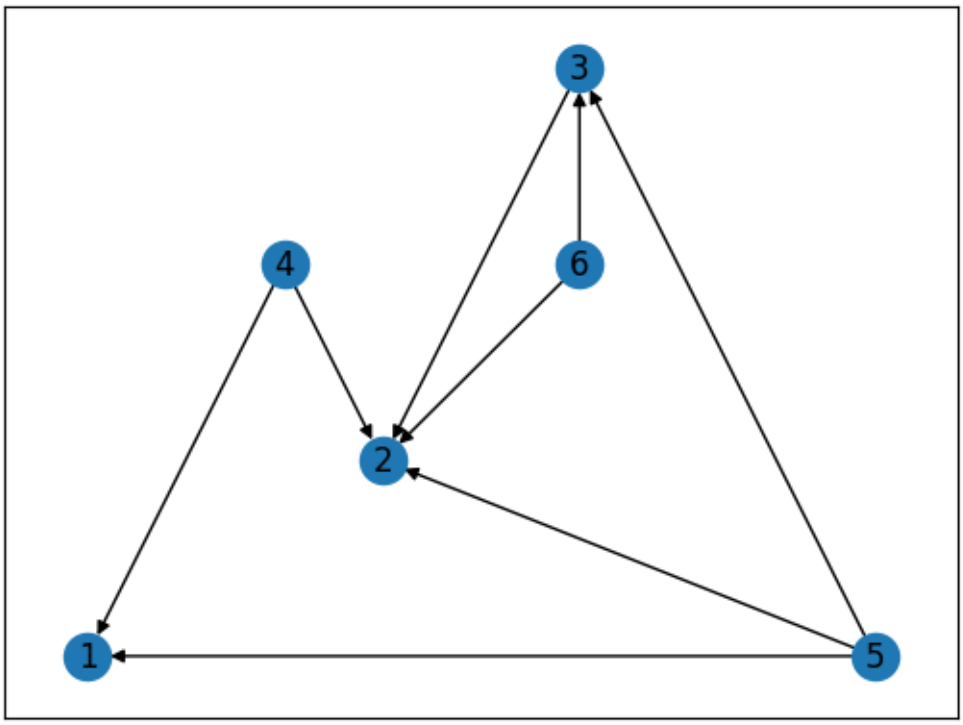
[12]:
```
graph = {
    1 : [4, 5],
    2 : [3, 4, 5, 6],
    3 : [2, 5, 6],
    4 : [1, 2],
    5 : [1, 2, 3],
    6 : [2, 3]
}
```

[13]:
```
drawGraph(graph)
```



[14]:
```
di_graph = generate_DAG(graph)
```

[15]:
```
drawDiGraph(di_graph)
```



1.3 Es. 3. Dire se v è un nodo principale in un grafo orientato

Testo

Un vertice v in un grafo diretto G, si dice principale se ogni altro vertice in G puo' essere raggiunto con un cammino diretto che parte da v.
Descrivere un algoritmo che dati un grafo G e un vertice v, determina se v e' un vertice principale in G. L'algoritmo deve avere complessita' O(n+m).

Idea

In un grafo orientato, un nodo è considerato un "nodo principale" se è raggiungibile da tutti gli altri nodi del grafo attraverso un percorso diretto. In altre parole, tutti i nodi del grafo raggiungono il "nodo principale".
Per verificare se v è un nodo con questa proprietà, possiamo notare che basta effettuare una visita DFS con v come radice della DFS e controllare se la visita raggiunge tutti i nodi.

Soluzione

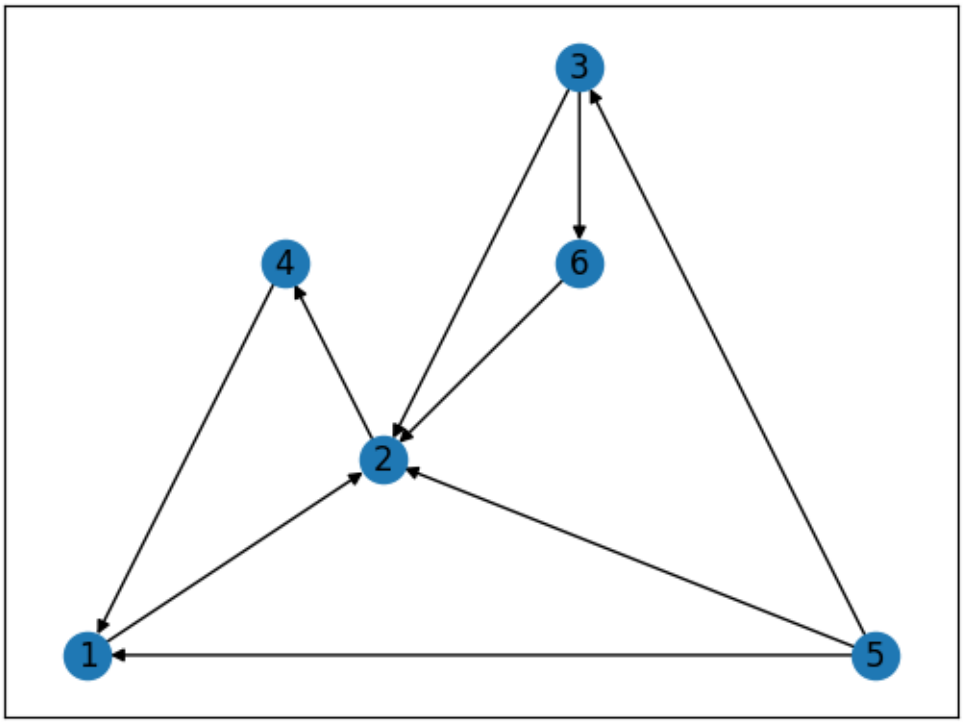
[16]:
```python
def is_main_node(graph, v):
    visited = set()
    stack = [v]
    while stack:
        current_node = stack.pop()
        visited.add(current_node)
        for neighbor in graph[current_node]:
```

```
            if neighbor not in visited:
                stack.append(neighbor)
    return len(visited) == len(graph)
```

Esecuzione

[17]:
```
di_graph = {
    1 : [2],
    2 : [4],
    3 : [2, 6],
    4 : [1],
    5 : [1, 2, 3],
    6 : [2]
}
```

[18]:
```
drawDiGraph(di_graph)
```



[19]:
```
is_main_node(di_graph, 2)
```

[19]: False

[20]:
```
is_main_node(di_graph, 5)
```

[20]: True

1.4 Es. 4. Trovare un insieme minimo di nodi principali

Testo

Descrivere un algoritmo che dato un grafo diretto G trova il minimo numero di vertici da cui e' possibile raggiungere tutti gli altri vertici del grafo. L'algoritmo deve avere complessita' O(n + m).

Idea

Per un grafo diretto, un insieme minimo di nodi principali si ottiene prendendo un rappresentante da ogni componente fortemente connessa che non ha archi entranti.
Una possibile soluzione è quindi quella di riconoscere tutte le componenti fortemente connesse per poi selezionare solo quelle che non sono raggiunte da altre componenti fortemente connesse. Infine basta selezionare un rappresentante delle componenti fortemente connesse rimanenti, da aggiungere all'insieme minimo di nodi principali.
Si noti che per un insieme minimale basta prendere un rappresentante per ogni componente fortemente connessa.

Soluzione

[21]:
```python
def DFSscc(node, c, nc, visited, components, stack):
    c += 1
    back = c
    visited[node] = c
    stack.append(node)
    for adj in di_graph[node]:
        if visited[adj] == 0:
            x, c, nc = DFSscc(adj, c, nc, visited, components, stack)
            back = min(back, x)
        else:
            if components[adj] == 0:
                back = min(back, visited[adj])
    if back == visited[node]:
        nc += 1
        while stack:
            w = stack.pop()
            components[w] = nc
            if w == node:
                break
    return back, c, nc

def get_scc(di_graph):
    visited = {node : 0 for node in di_graph}
    components = {node : 0 for node in di_graph}
    stack = []
    nc = 0
    c = 0
    for node in di_graph:
        if visited[node] == 0:
            _, c, nc = DFSscc(node, c, nc, visited, components, stack)
```

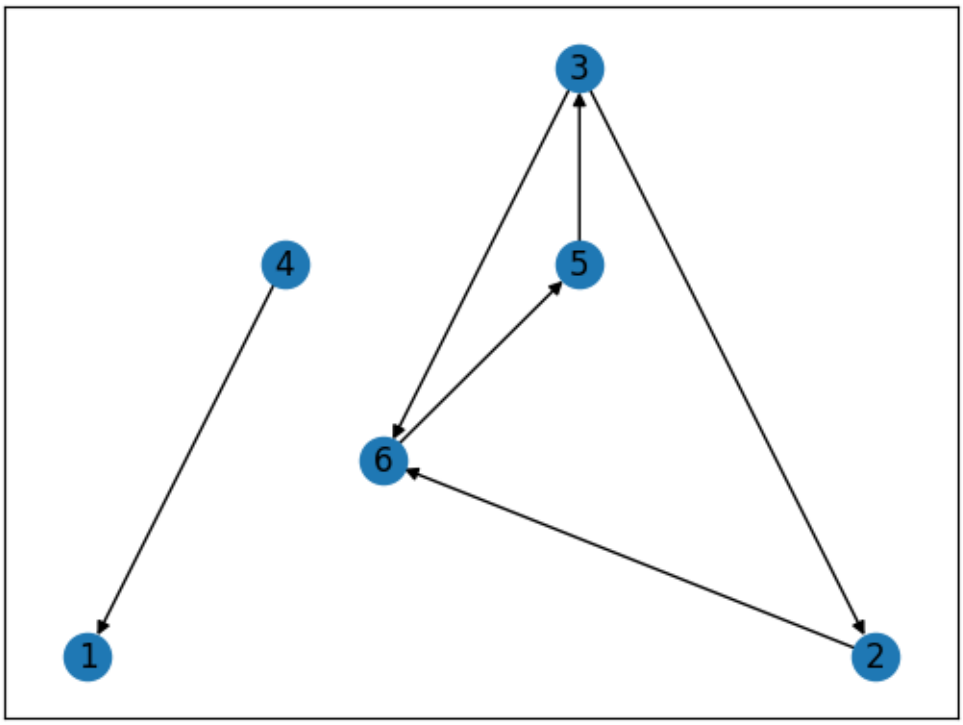
```
    return components, nc

def minimum_principal_node(di_graph):
    components, nc = get_scc(di_graph)
    scc = [[] for _ in range(nc)]
    for node in di_graph:
        scc[components[node]-1].append(node)
    scc_to_remove = set()
    for node in di_graph:
        for adj in di_graph[node]:
            if components[node] != components[adj]:
                scc_to_remove.add(components[adj])
    res = []
    for index in range(len(scc)):
        if index+1 not in scc_to_remove:
            res.append(scc[index][0])
    return res
```

Esecuzione

[22]:
```
di_graph = {
    1 : [],
    2 : [6],
    3 : [6, 2],
    4 : [1],
    5 : [3],
    6 : [5]
}
```

[23]:
```
drawDiGraph(di_graph)
```



[24]:
```
minimum_principal_node(di_graph)
```

[24]: [2, 4]