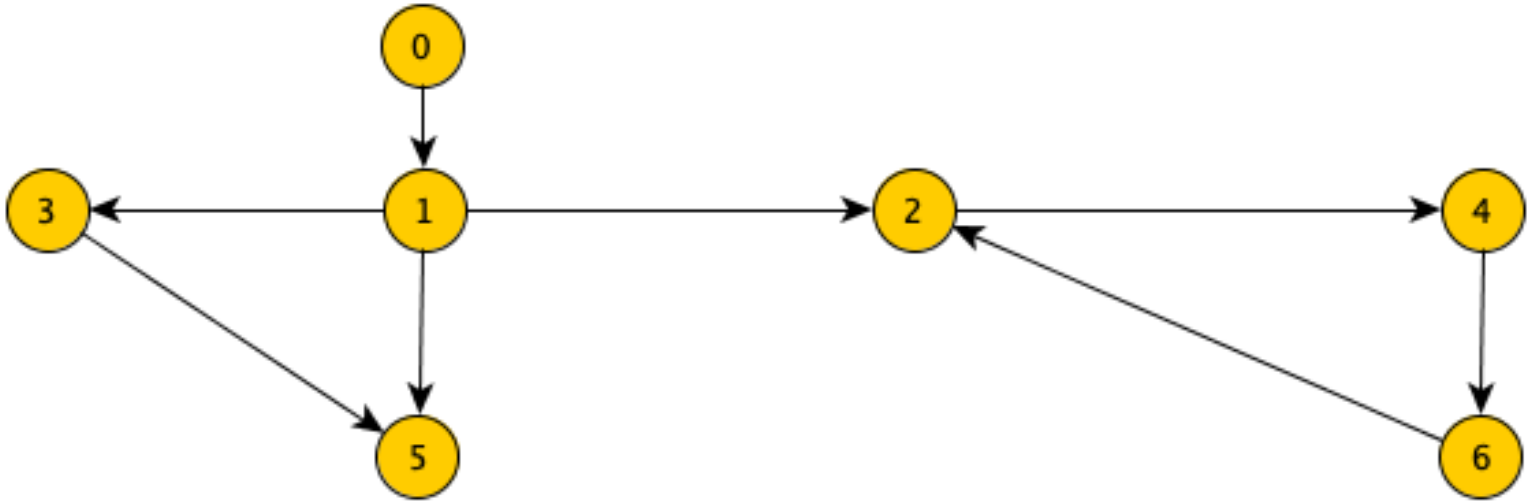
Corso di laurea in Informatica
Progettazione d'algoritmi

I grafi 2

Angelo Monti

SAPIENZA
UNIVERSITÀ DI ROMA

Dato un grafo G ed un suo nodo u vogliamo sapere quali nodi del grafo sono raggiungibili a partire dal nodo u:




```
>>> raggiungibili(0,G)
{0, 1, 2, 3, 4, 5, 6}
>>> raggiungibili(6,G)
{2, 4, 6}
>>> raggiungibili(5,G)
{5}
```

Visita in profondità (DFS) su grafo rappresentato tramite matrice di adiacenza

```
def DFS(u, M):
    '''esegue visita dei nodi di G raggiungibili
    a partire dal nodo u'''
    ####
    def DFSr(u, M, visitati):
        visitati[u] = 1
        for i in range(len(M)):
            if M[u][i] == 1 and not visitati[i]:
                DFSr(i, M, visitati)
    ####
    n = len(M)
    visitati = [0]*n
    DFSr(u, M, visitati)
    return [x for x in range(n) if visitati[x]]
```

- Al termine di *DFS(u, M)* si ha $visitati[i] = 1$ se e solo se i raggiungibile da u
- La complessità della procedura è $O(n) \times \Theta(n) = O(n^2)$

Visita in profondità (DFS) per grafo rappresentato tramite liste di adiacenza

```
def DFS(u, G):
    '''esegue visita dei nodi di G raggiungibili
    a partire dal nodo u'''
    ####
    def DFSr(u, G, visitati):
        visitati[u] = 1
        for v in G[u]:
            if not visitati[v]:
                DFSr(v, G, visitati)
    ####
    n = len(G)
    visitati = [0]*n
    DFSr(u, G, visitati)
    return [x for x in range(n) if visitati[x]]
```

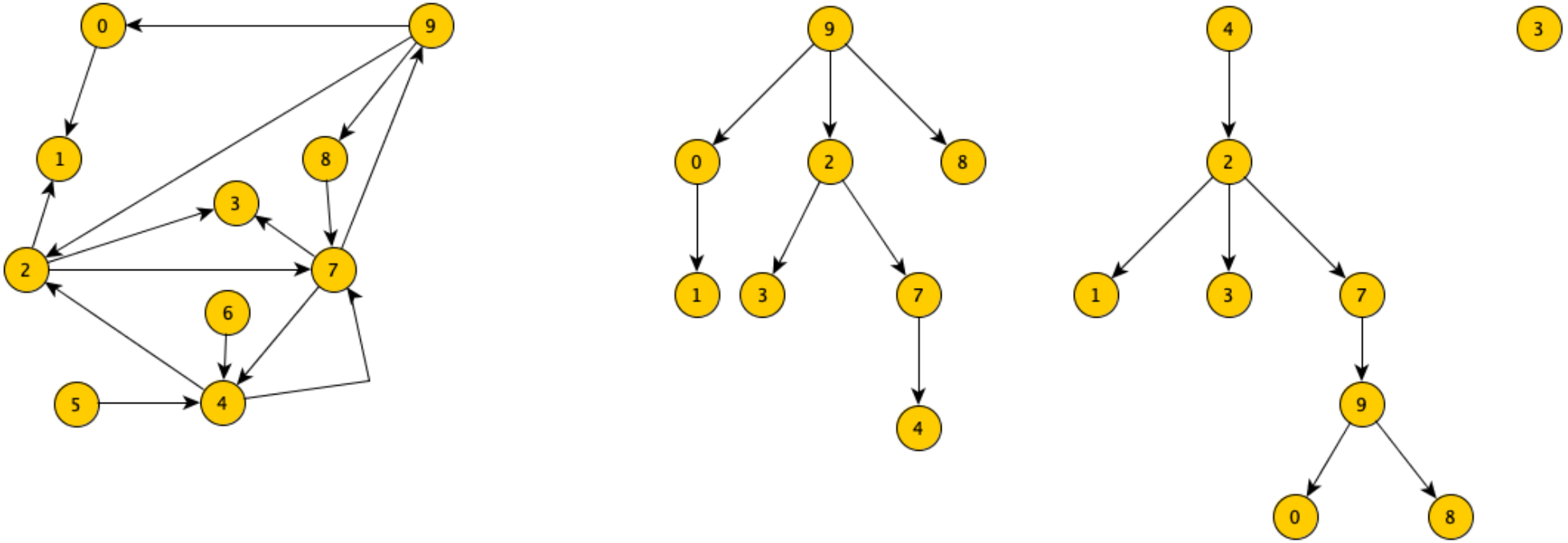
- Al termine di *DFS(u,G)* si ha *visitati*[i] = 1 se e solo se i è raggiungibile da u
- La complessità della procedura è $O(n+m)$
- La complessità di spazio della procedura è $O(n)$

Visita in profondità (DFS) per grafo rappresentato tramite liste di adiacenza VERSIONE ITERATIVA

```
def DFS_iterativo(u, G):
    '''esegue visita dei nodi di G raggiungibili a partire dal nodo u'''
    visitati = [0] * len(G)
    pila = [u]  # inizializza la pila con il nodo di partenza
    while pila:
        u = pila.pop()
        if not visitati[u]:
            visitati[u] = 1
            # Aggiungiamo i vicini non visitati;
            for v in G[u]:
                if not visitati[v]:
                    pila.append(v)
    return [x for x in range(len(G)) if visitati[x]]
```

- Al termine di *DFS_iterativo(u,G) si ha* $visitati[i] = 1$ se e solo se i è raggiungibile da u
- La complessità di tempo della procedura è $O(n+m)$
- La complessità di spazio della procedura è $O(n)$.

Con una visita DFS gli archi del grafo si bipartiscono in quelli che nel corso della visita sono stati attraversati (perché permettevano di raggiungere nuovi nodi) e gli altri:
I nodi visitati e gli archi effettivamente attraversati formano un albero detto albero DFS.



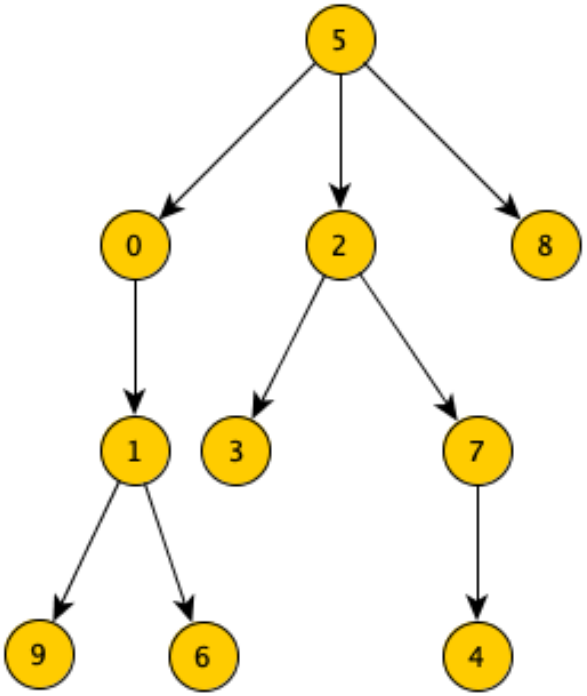
A sinistra un grafo G, a destra i tre alberi DFS che si ottengono facendo partire tre visite dai nodi $9, 4$ e 3 rispettivamente.

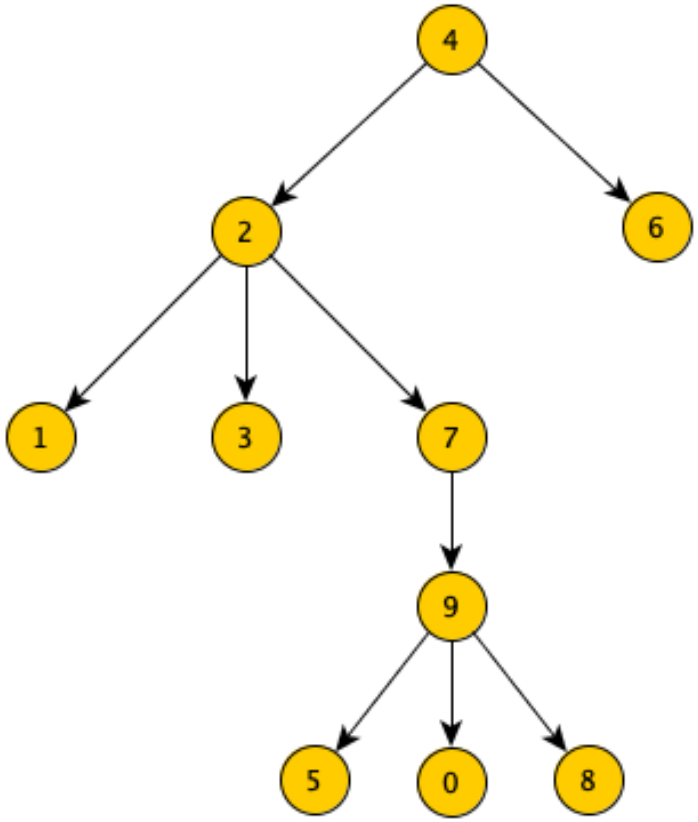
Un albero DFS può essere memorizzato tramite il vettore dei padri.

Il vettore dei padri P di un albero DFS di un grafo di n nodi ha n componenti:

- Se i è un nodo dell'albero DFS: $P[i]$ contiene il padre del nodo i (il padre della radice per convenzione è la radice stessa);
- Se i non è un nodo dell'albero: $P[i]$ per convenzione contiene -1

Esempi di alberi rappresentati tramite vettore dei padri





$P =$

5	0	5	2	7	5	1	2	5	1

$P =$

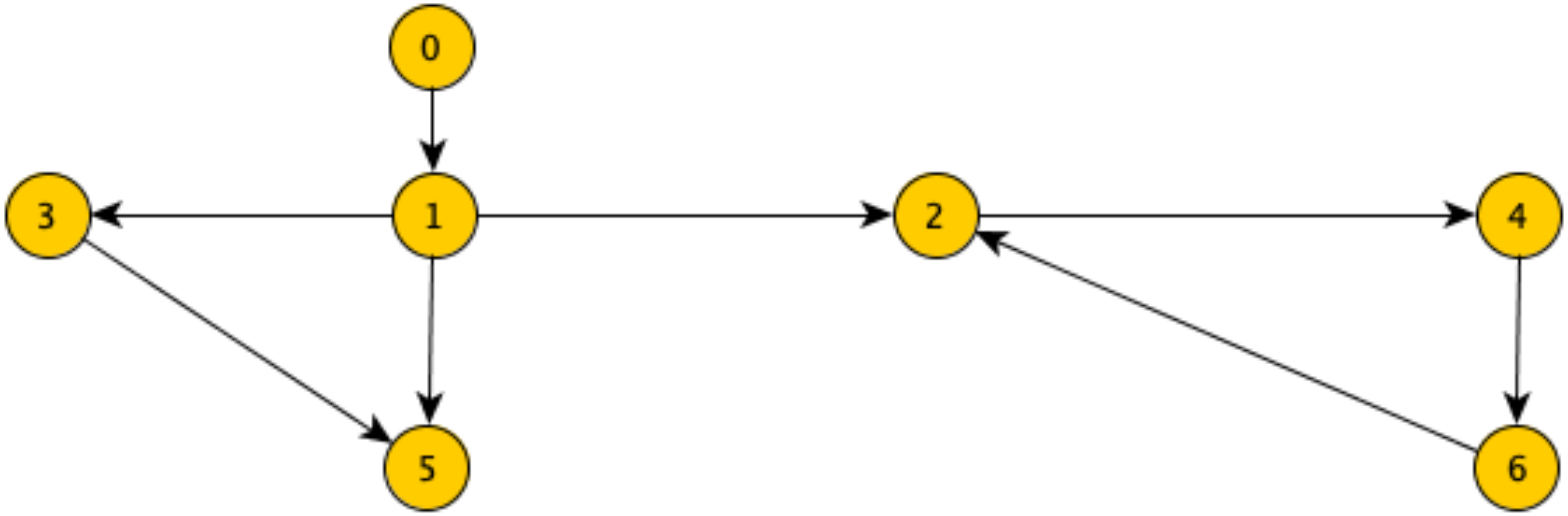
9	2	4	2	4	9	4	2	9	7

Modificando lievemente la procedura DFS è possibile fare in modo che restituisca il vettore dei padri P anzicché il vettore dei visitati.

```
def Padri(u, G):
    '''genera il vettore dei padri P radicato nel nodo u'''
    ####
    def DFSr(x, G, P):
        for y in G[x]:
            if P[y] == -1:
                P[y] = x
                DFSr(y, G, P)
    ####
    n = len(G)
    P = [-1]*n
    P[u] = u
    DFSr(u, G, P)
    return P
```

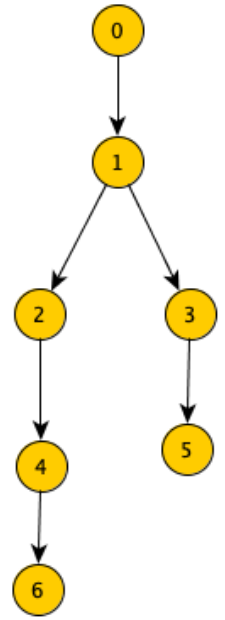
Al termine $P[v]$ contiene -1 se v non è stato visitato altrimenti contiene il padre di v nell'albero DFS.

Esempio di applicazione della procedura $Padri(u, G)$:



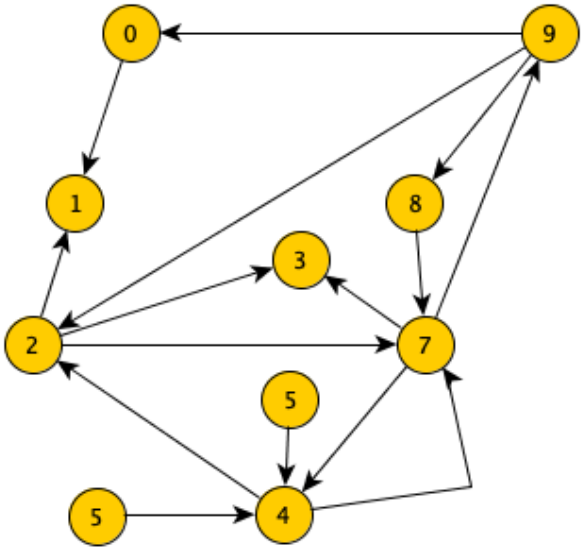
Al termine $P[v]$ contiene -1 se v non è stato visitato altrimenti contiene il padre di v nell'albero DFS.

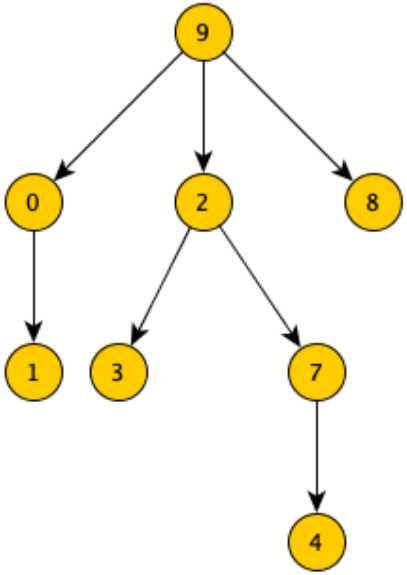
```
>>> G=[[1], [2, 3, 5], [4], [5], [6], [], [2]]

>>> Padri(0,G)
[0, 0, 1, 1, 2, 3, 4]

>>> Padri(5,G)
[-1, -1, -1, -1, -1, 5, -1]
```

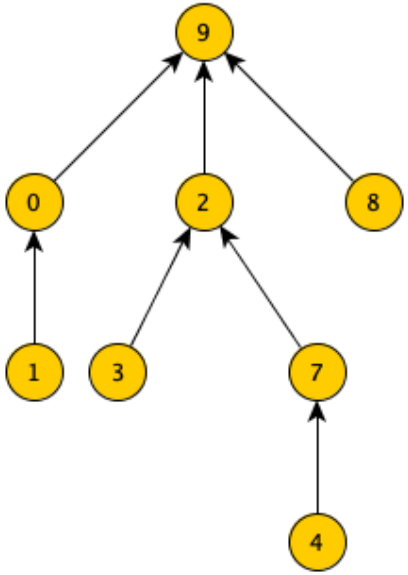


In molte applicazioni non basta sapere se un nodo y è raggiungibile a partire dal nodo x del grafo; se la risposta è positiva, vogliamo anche determinare un cammino che ci consenta di andare da x a y.
Il vettore dei padri dell'albero DFS, radicato in x, permette di ricavare facilmente tale cammino.

Basta controllare che il nodo y sia nell'albero e poi da y risalire alla radice ed effettuare il "reverse" dei nodi incontrati.







$$P = \begin{array}{|c|c|c|c|c|c|c|c|c|c|} \hline 9 & 0 & 9 & 2 & 7 & -1 & -1 & 2 & 9 & 9 \\ \hline \end{array}$$

```
>>>P=padri(9,G)
>>>cammino(7,P)
[9,2,7]
>>>cammino(1,P)
[9,0,1]
>>>cammino(5,P)
[]
```

Procedura iterativa per la ricerca del cammino

```
def Cammino(u, P):
    '''restituisce il cammino dal nodo
    radice dell'albero P  al nodo u'''
    if P[u] == -1: return []
    path = []
    while P[u] != u:
        path.append(u)
        u = P[u]
    path.append(u)
    path.reverse()
    return path
```

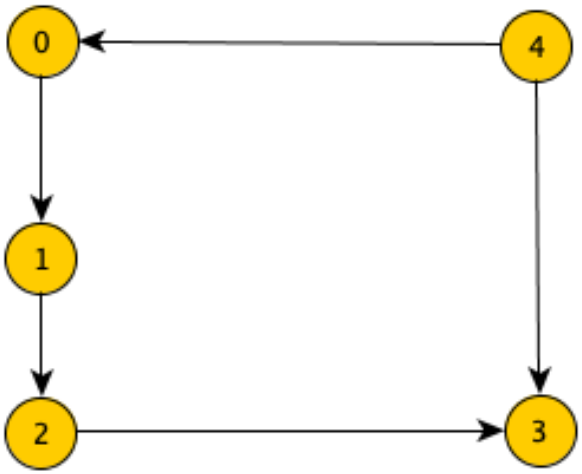
Disponendo del vettore dei padri, la complessità della procedura è $O(n)$

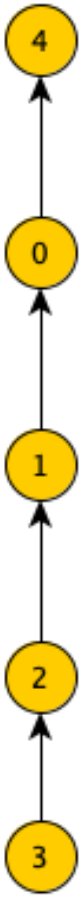
Procedura ricorsiva per la ricerca del cammino

```
def CamminoR(u, P):
    '''restituisce il cammino dal nodo
    radice dell'albero P  al nodo u'''
    if P[u] == -1: return []
    if P[u] == u: return [u]
    return CamminoR(P[u], P) + [u]
```

Disponendo del vettore dei padri, la complessità della procedura è $O(n)$.

ATTENZIONE: se esistono più cammini che dal nodo x portano al nodo y la procedura appena vista non garantisce di restituire il cammino minimo (vale a dire quello che attraversa il minor numero di archi)





Il cammino minimo da 4 a 3 è $[4, 3]$ mentre la procedura restituisce il cammino $[4, 0, 1, 2, 3]$.

Corso di laurea in Informatica
Introduzione agli Algoritmi

Esercizi per casa

SAPIENZA
UNIVERSITÀ DI ROMA

Sia G un grafo non orientato e connesso con n nodi. Rispondere ai seguenti quesiti:

1. Determinare condizioni necessarie e sufficienti affinché il grafo G possa avere tutti i suoi nodi di grado esattamente k.

2. Dimostrare che G contiene almeno un vertice la cui rimozione non rende il grafo sconnesso.

3. Sia m il numero di archi di G. Durante una visita in profondità del grafo, quante volte si incontrano nodi già visitati? In altre parole, quante volte, nel corso della visita, scorrendo le liste di adiacenza dei nodi si incontrano nodi che erano già stati esplorati?

4. Per ogni nodo x in G, sia D_x la distanza massima da x nel grafo G (cioè il numero massimo di archi che bisogna attraversare per raggiungere, a partire da x, un qualunque altro nodo di G). Rispondere ai seguenti quesiti:

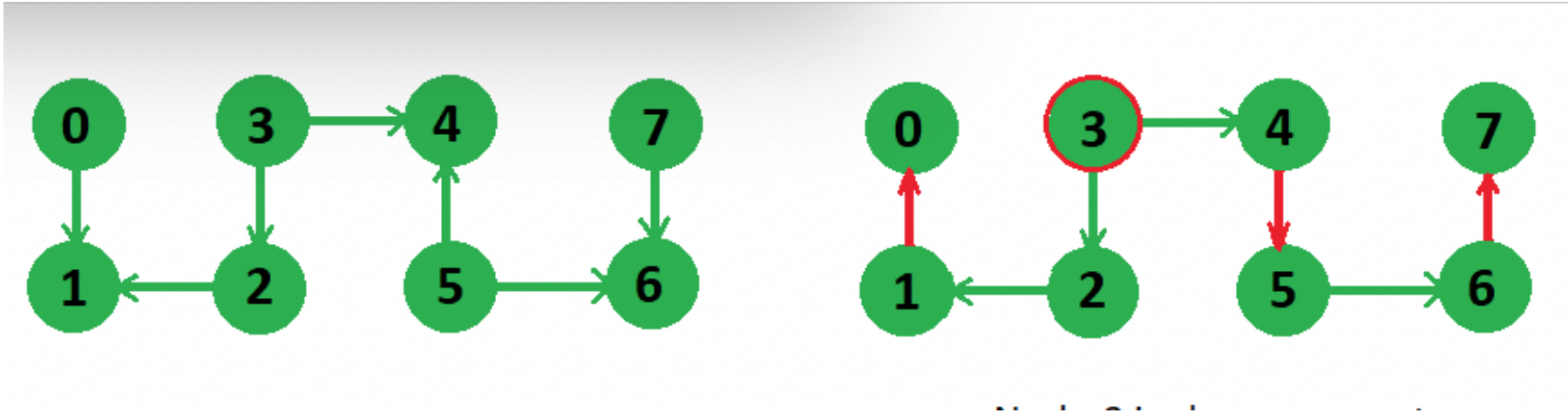
   (a) Possono esistere due nodi u e v in G tali che $D_u = 4$ e $D_v = 8$?

   (b) Possono esistere due nodi u e v in G tali che $D_u = 4$ e $D_v = 9$?

   In caso affermativo, fornire un esempio di un grafo G con questa proprietà; in caso negativo, dimostrare l'impossibilità di tale configurazione.

ESERCIZIO:

In un grafo aciclico T con n nodi $(0, 1, 2, \ldots, n-1)$ gli archi sono stati orientati a caso. Vogliamo sapere in quali nodi radicare il grafo in modo tale che risulti minimo il numero di archi la cui direzione va invertita per far si che tutti i nodi siano raggiungibili dalla radice.



Ad esempio in figura è riportato a sinistra il grafo aciclico T con gli archi orientati ed a destra si mostra che radicandolo nel nodo 3 bisogna poi invertire 3 archi.

Progettare un algoritmo che prende come parametro l'albero orientato e in tempo $O(n)$ risolve il problema restituendo l'insieme di nodi da scegliere come radici.

Esempi:

- Per l'albero diretto T=[[1], [ ], [1], [2, 4], [ ], [4, 6], [ ], [6]] l'algoritmo deve restituire l'insieme $\{3, 5\}$.
- Per l'albero diretto T=[[1, 2], [ ], [6], [0, 7, 8], [1], [1], [ ], [ ], [ ]] l'algoritmo deve restituire l'insieme $\{3\}$.