Visita in profondità (DFS)

corso di laurea in **Matematica**

Informatica Generale, Lezione **22(b)**

Ivano Salvo

SAPIENZA
UNIVERSITÀ DI ROMA

Visita in profondità

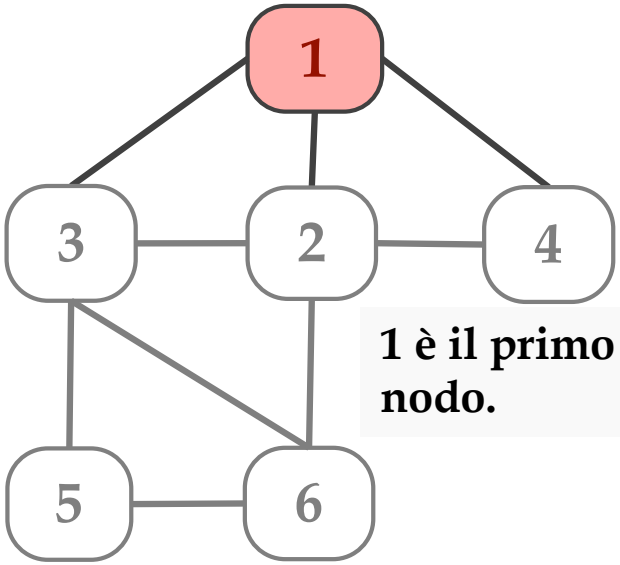
La **visita in profondità** (**depth-first search**, **DFS**) è l'analogo delle **visita in profondità** di un albero, anche se usualmente non si distinguono diversi ordini in cui visitare gli adiacenti.

Partendo da s, l'idea è quella di **visitare prima un nodo v adiacente e tutti i suoi discendenti**, e solo dopo gli eventuali nodi adiacenti (i "fratelli di v") **non** incontrati come **discendenti di v**.

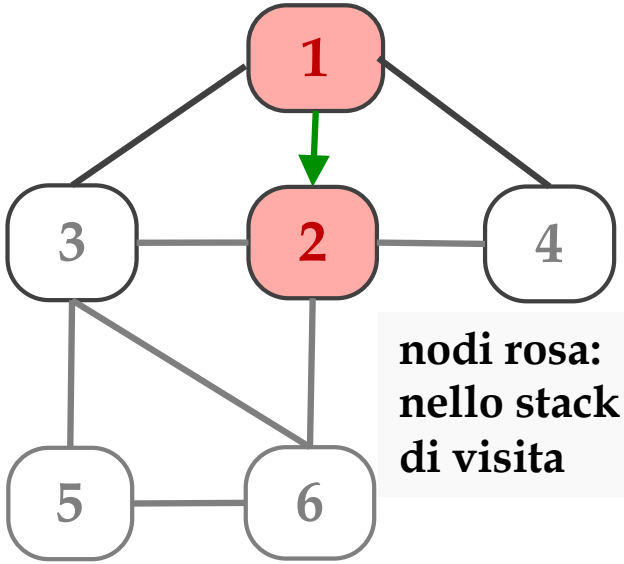
Anche nella DFS, per fare progressi, occorre memorizzare i nodi già visitati e quando si incontrano, evitare di far partire nuove ricerche da quei nodi.

Quando non ci sono più nodi da scoprire la visita si arresta.

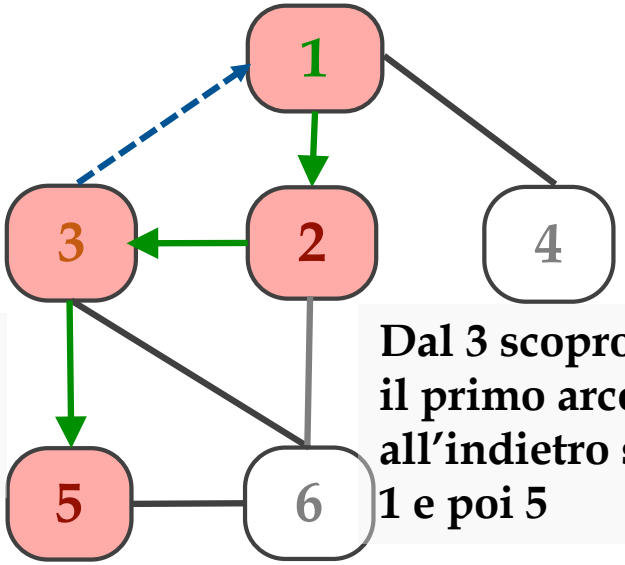
Visita in profondità: Esempio



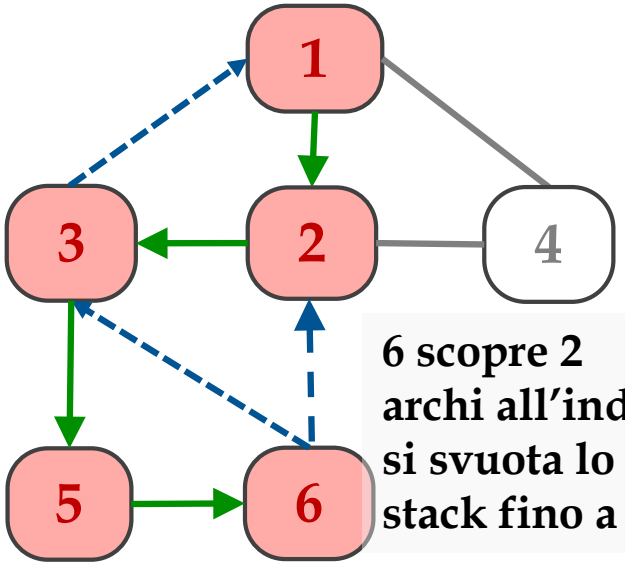
1 è il primo nodo.



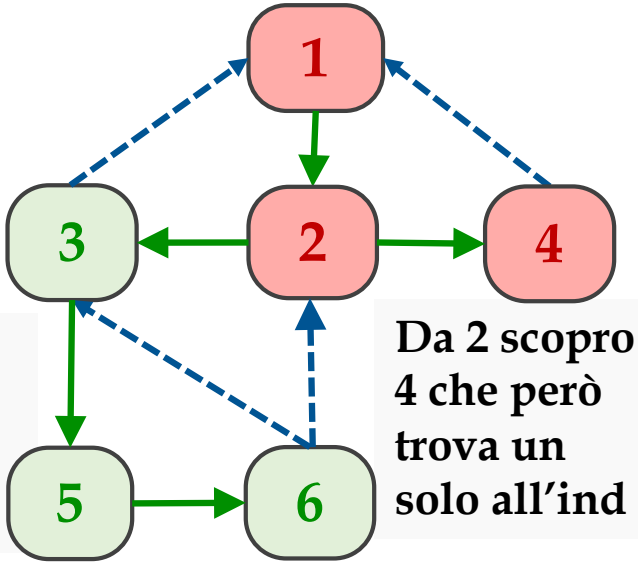
nodi rosa: nello stack di visita



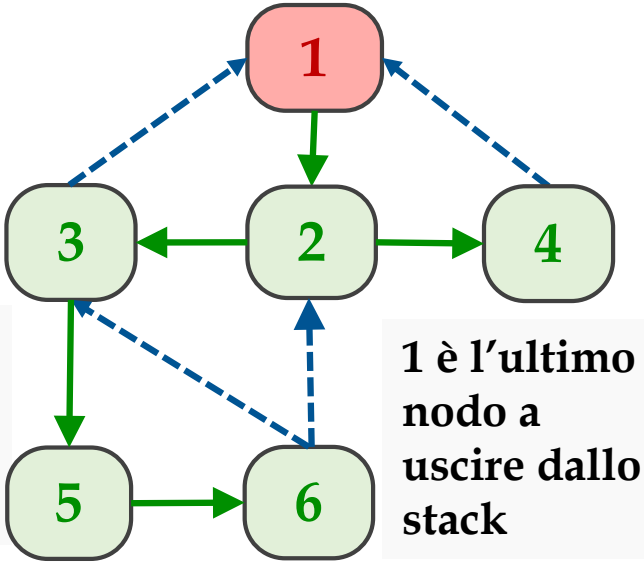
Dal 3 scopro il primo arco all'indietro su 1 e poi 5



6 scopre 2 archi all'ind si svuota lo stack fino a 2



Da 2 scopro 4 che però trova un solo all'ind



1 è l'ultimo nodo a uscire dallo stack

Visita in profondità: pseudocodice

A differenza della BFS, la DFS si esprime naturalmente in **forma ricorsiva**. In questo caso è sufficiente mantenere solo l'insieme **VIS** dei nodi **già visitati**.

Nella DFS, ogni volta che si scopre un nodo **si continuano a visitare i suoi discendenti**, prima di tornare a visitare eventuali "fratelli".

Osservazione importante: se gli altri adiacenti di u fossero anche discendenti di v, **sarebbero incontrati** durante la **DFS** radicata in v **prima di tornare** dalle chiamate ricorsive **a** u.

```
def dfs(G, u, VIS, t, in, out):
    VIS, t = VIS ∪ {u}, t + 1
    in[u] = t
    forall v ∈ adj(u):
        if v ∉ VIS: # in[v] > 0
          p[v] = u
          t = dfs(G, v, VIS, t, in, out)
    out[u] = t+1
    return t+1
```

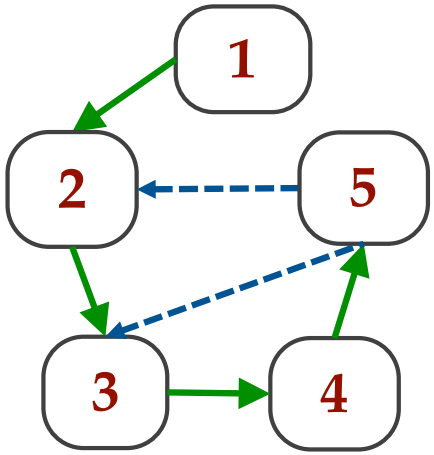
Per sostanziare l'ultima proprietà della dfs, **registriamo per ogni nodo il tempo di entrata** (vettore **in**) e il **tempo di uscita** (vettore **out**).

I tempi di ingresso/uscita dai nodi **sono tutti diversi** (time si incrementa ogni volta che entro e lascio un nodo).

Visita in profondità: proprietà

Anche la visita in profondità ha **notevoli proprietà**:

- **non ci sono archi di attraversamento**
- gli archi all'**indietro** connettono un **nodo con suoi antenati** nell'albero di visita.



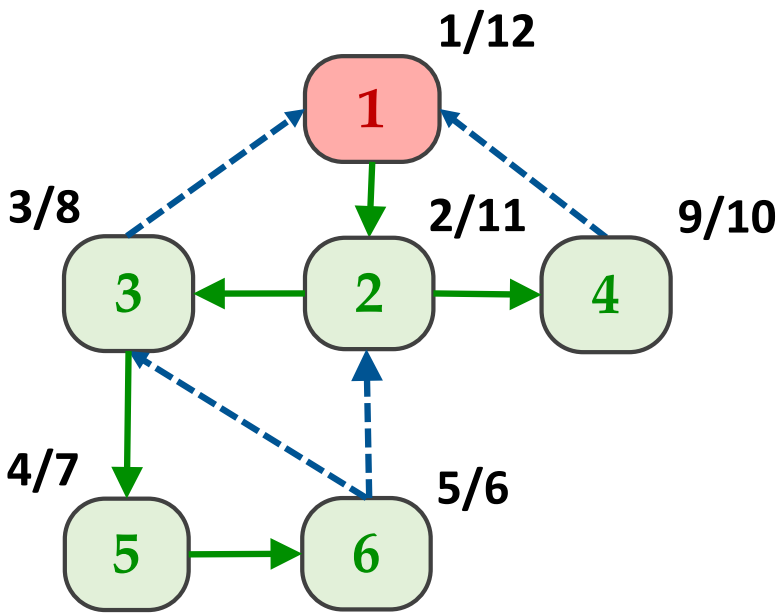
Esempio: Gli archi verdi sono gli archi dell'albero.

Gli archi **tratteggiati blu** sono **archi all'indietro**: gli archi (3, 5) e (2, 5) vengono provati **dal nodo 5**, quando i nodi 2 e 3 sono già stati scoperti e quindi sono suoi antenati nell'albero di visita.

DFS: struttura di "parentesi"

Proposizione: In una visita DFS, presi due nodi $u, v \in V$, può verificarsi, relativamente ai tempi di visita, una sola tra le seguenti situazioni :

- $[in[u], out[u]] \subset [in[v], out[v]]$
- $[in[v], out[v]] \subset [in[u], out[u]]$
- $[in[u], out[u]]$ e $[in[v], out[v]]$ sono disgiunti



Esempio: Vediamo i tempi di ingresso/uscita di tutti i nodi nel nostro esempio principale.

I tempi di **4** sono **disgiunti** da quelli di **3**, **5** e **6**.

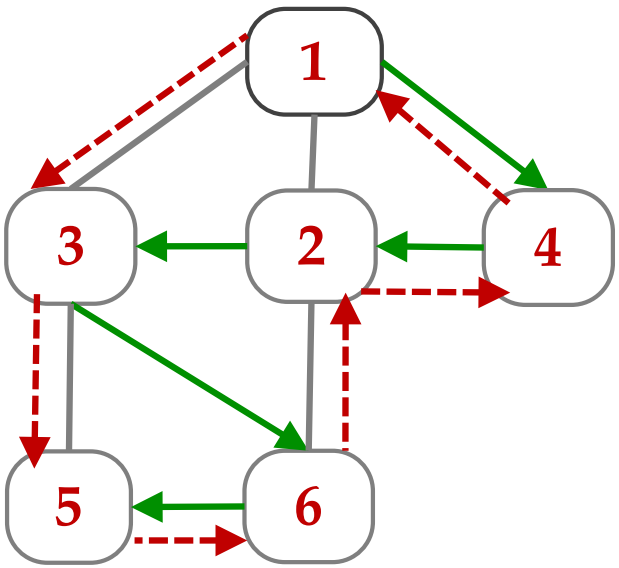
Quello di **6** è **contenuto** in **5**, che è contenuto in **3**, che è contenuto in **2**, che è contenuto in **1**.

DFS: cammino più lungo

Si potrebbe credere che la DFS allontanandosi sempre dal nodo iniziale **dia sempre il cammino più lungo**.

Il cammino più lungo **è nel risultato di una qualche DFS**, ma il fatto che esso sia uno dei cammini dell'albero di visita dipende **dall'ordine di visita**.

Il cammino più lungo potrebbe essere trovanto facendo **tutte le DFS** permutando l'ordine d'esplorazioni dei vicini, ad esempio con una procedura brute force che fa backtrack.



Nel nostro esempio, il cammino più lungo è anche un **cammino Hamiltoniano**, cioè un cammino che attraversa 1 sola volta tutti i nodi: ce ne sono diversi, ad esempio **1 4 2 3 6 5** oppure **1 3 5 6 2 4** che può anche essere chiuso in **circuito Hamiltoniano**.

Non si conoscono algoritmi polinomiali per il problema del **cammino più lungo** o del **cammino hamiltoniano**.

Alcune applicazioni delle visite

corso di laurea in **Matematica**

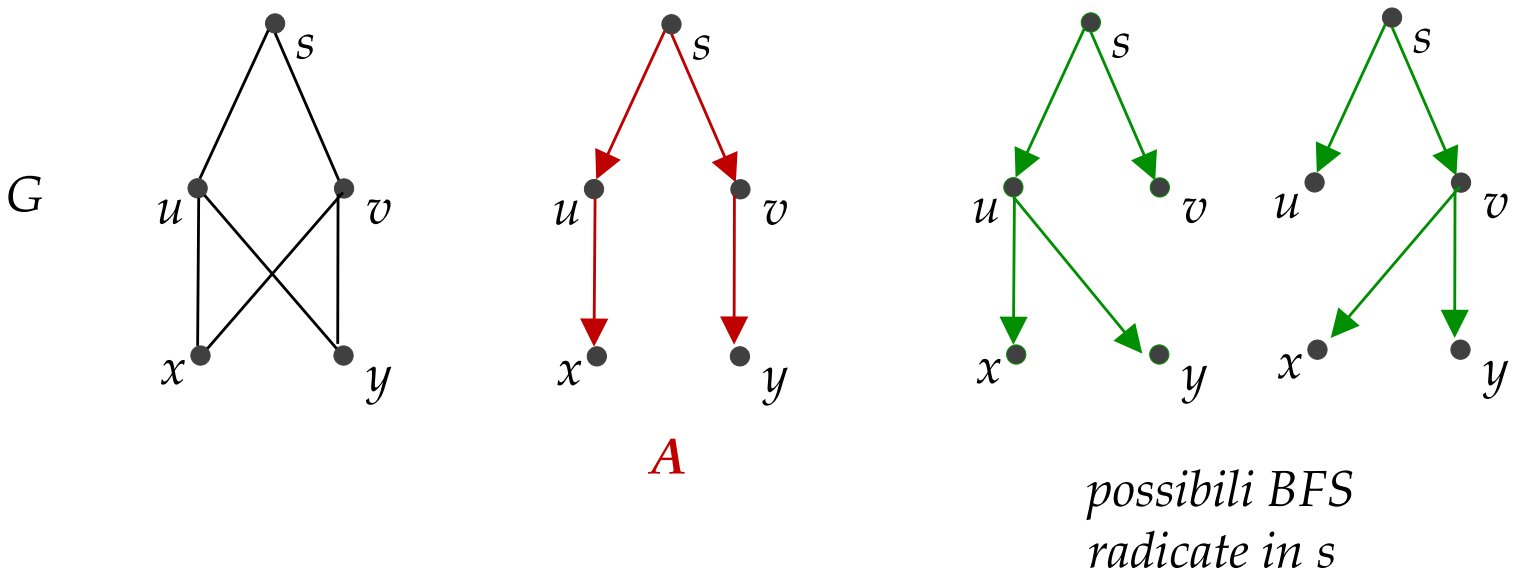
Informatica Generale, Lezione **22(b²)**

Ivano Salvo

SAPIENZA
UNIVERSITÀ DI ROMA

Capire la BFS

Fare un esempio di un grafo $G = (V, E)$, una sorgente $s \in V$ e di un albero ricoprente A di G tale che per ogni vertice in $v \in V$ l'unico cammino semplice da s a v sia minimo in A, ma A non sia il risultato di nessuna BFS.



Capire la BFS & DFS

Come deve essere fatto un grafo **non** orientato **connesso** G affinché la BFS e la DFS (radicate nello stesso nodo) producano lo stesso albero di visita?

Soluzione:

In una visita DFS ho **solo** archi dell'albero e archi all'indietro.
In una visita BFS ho **solo** archi dell'albero e archi trasversali.

Quindi DFS(G)=BFS(G) solo se contengono solo archi dell'albero di visita e quindi G è esso stesso un albero.

D'altro canto è ovvio che se G fosse un albero, allora:

$$\mathrm{DFS}(G) = \mathrm{BFS}(G) = G.$$

Possiamo quindi dire che **DFS(G)=BFS(G) se e solo se G è un albero**.

Visite: raccogliere informazioni

Durante una visita si possono **raccogliere informazioni** sui **nodi** (o più raramente sugli archi) attraverso **vettori indicizzati sui nodi** (o archi) come abbiamo fatto con i **tempi di ingresso/uscita**.

Queste informazioni permettono spesso di **risolvere un problema sui grafi con una** (o più) **visita**, ma senza dover ricalcolare più volte le stesse cose.

Vediamo nel seguito diversi esempi di questo fatto.

Ovviamente, a **seconda del problema**, occorre capire quale sia l'**informazione giusta** da considerare.

Cammino minimo da s a t

Il problema di trovare un (singolo) **cammino minimo** in **numero di archi** da s a t può **sembrare** un problema **più semplice** della **visita** di un intero grafo.

Tuttavia **non si può sapere** per **dove passi** tale cammino.

Quindi, **è necessario** fare una **BFS radicata in s** che si può arrestare non appena venga trovato t, tenendo **traccia delle distanze**.

Il modo più semplice è tenere un vettore d e assegnare $d[z] = d[v] + 1$ a tutti i vicini z di v.

Il nodo s da cui si inizia la visita ha distanza 0.

L'informazione nel vettore d **sussume l'informazione di marked**, se inizializziamo d a un valore distanza impossibile, al solito -1.

In tal modo, avranno **distanza negativa solo i nodi non ancora scoperti**, e quindi vale la proprietà:

$$marked[u] = \text{TRUE} \quad \text{se e solo se} \quad d[u] \geq 0$$

Cammino minimo da s a t

Nel codice sottostante, la visita si arresta **non appena viene trovato** il nodo di **arrivo** *t*.

Ovviamente ***t* potrebbe essere anche l'ultimo nodo da visitare**, quindi la complessità è la stessa di una visita, cioè $\Theta(n + m)$.

Ma cosa calcoliamo se lasciamo andare questa visita fino alla fine?

```
def dist(G, s, t):
    Q = newQueue()
    d = init(nunNodi[G], -1)
    d[s] = 0
    enqueue(Q, s)
    while not isEmpty(Q):
        u = dequeue(Q)
        forall v ∈ G.adj(u):
            if d[v] < 0:
                enqueue(Q, v)
                d[v] = d[u] + 1
                if v == t: return d[v]
    return d[t] # -1 se t non ragg.
```

Distanza minima da s a tutti i nodi

L'algoritmo precedente ci suggerisce che durante una **BFS** di un grafo G possiamo calcolare **le distanze dalla radice s** della visita a **tutti gli altri nodi** di G.

Infatti, se lasciamo completare la visita, il vettore d conterrà tutte **le distanze minime** da s a v, **per ogni v**, sempre al costo di $\Theta(n + m)$.

```
def dist(G, s):
    Q = newQueue()
    d = init(numNodi(G), -1)
    d[s] = 0
    enqueue(Q, s)
    while not isEmpty(Q):
        u = dequeue(Q)
        forall v ∈ G.adj(u):
            if d[v] < 0:
                enqueue(Q, v)
                d[v] = d[u] + 1
    return d
```

Componenti Connesse

Le componenti connesse si calcolano **decorando ciascun nodo** con un **numero** che **rappresenta la componente connessa** a cui appartiene.

Teniamo un **vettore *cc* indicizzato sui nodi**, inizializzato a tutti 0. Per tutti i nodi u scoperti nella visita della prima componente connessa, assegneremo 1 a $cc[u]$. Poi troveremo il primo nodo tale che $cc[v]=0$ e faremo partire una nuova visita da v.

È **indifferente in questo problema** usare una BFS oppure una DFS. Tutte le operazione necessarie hanno un costo dominato dal costo $\Theta(n + m)$ della visita.

```
def componentiConnesse(G):
    cc = allocZ(numNodi[G], 0)
    c = 1
    for v = 1 to numNodi[G]:
        if cc[v] == 0:
            dfsCC(G, v, cc, c)
            c = c + 1
    return cc, c
```

```
def dfsCC(G, s, cc, c):
    cc[u] = c
    forall v ∈ G.adj(u):
        if cc[v] == 0:
            cc[v] = c
            dfsCC(G, v, cc, c)
```

Liste di nodi delle CC

Osservate che **per trovare il prossimo nodo** da cui cominciare la visita di una nuova componente connessa, durante la visita **si scorre il vettore *cc* una sola volta da sinistra a destra**, con un costo complessivo (**analisi aggregata**) di $\Theta(n)$ per questa operazione.

Possiamo generare un vettore di liste *lcc* tale che *lcc*[1] è la lista dei nodi della prima componente connessa, *lcc*[2] la lista dei nodi della seconda e così via…

Il vettore *lcc* può essere anche caricato scorrendo una sola volta *cc* alla fine della visita, quindi con un costo $\Theta(n)$.

```
def listaCC(G):
    c, cc = componentiConnesse(G)
    lcc = allocaV(c)
    for i=1 to n:
        lcc[cc[i]] = cons(i, lcc[cc[i]])
    return lcc
```

Il nodo s appartiene a un ciclo?

La presenza di un **arco di attraversamento** (nella BFS) o di un **arco all'indietro** (in una DFS) **manifestano sempre la presenza di un ciclo** (del resto si tratta di un arco da aggiungere a un albero…).

Questo fatto può essere usato facilmente per **determinare se un grafo sia aciclico o meno** in un grafo non orientato.

Nei grafi orientati, la questione è leggermente più complessa.

O se sia un albero (in questo caso dobbiamo verificare anche che sia connesso).

Tuttavia, **non è banale** (anche se non difficile) **ricostruire il ciclo** da **un arco di attraversamento di una BFS** (esercizio).

Di conseguenza, se ci chiediamo se uno specifico nodo s appartenga a un ciclo, qual è la cosa più comoda?

Fare una DFS radicata in s e verificare se ci sia **un arco all'indietro che arriva in s**.

Ciclo contenente s: pseudocodice

Scriviamo il codice che è ancora una volta una variazione della DFS. È necessario trasportare tra i parametri il nodo *s* per riconoscerlo, eventualmente, quando lo trovo con un arco all'indietro.

Ma se volessi il ciclo come lista di nodi?

La costruisco "al ritorno".

```
# prima chiamata
def sInCiclo(G, s):
    VIS = ∅
    return dfsCiclo(G, s, s, VIS)
```

```
def dfsCiclo(G, s, u, VIS):
    forall v ∈ adj(u):
        if v == s: return True, NULL
        if v ∉ VIS
            VIS = VIS ∪ {v}
            b, L = dfsCiclo(G, s, v, VIS)
            if b: return True, cons(v, L)
            # sospendo la ricerca
            # costruisco il ciclo all'indietro
    return False, NULL
```