Far visita a un grafo

corso di laurea in **Matematica**

Informatica Generale, **Ivano Salvo**

Lezione **22(a)** **[12/12/23]**

SAPIENZA
UNIVERSITÀ DI ROMA

Visite

Un gran numero di problemi su grafi si **risolve visitando tutti** (o una **porzione** de)**i nodi** del grafo. Ad esempio:

- determinazione delle **componenti connesse**
- trovare il **cammino più breve** tra due nodi
- verificare la presenza di **cicli**.

La visita di un grafo è un'operazione relativamente semplice. La difficoltà più importante è capire che occorre **evitare di rivisitare** gli **stessi nodi** e quindi di **ripercorrere** più volte **gli stessi cammini**.

Il problema è **analogo** trovare l'uscita in un **labirinto**: occorre evitare di ripercorrere le stesse stanze e corridoi: occorre marcare opportunamente i luoghi in cui si è già transitati (come il filo di Arianna o i sassolini di Pollicino nel bosco).

Questo problema è **meno rilevante nella visita degli alberi**, perché sono **aciclici** e c'è sempre un unico cammino tra due nodi.

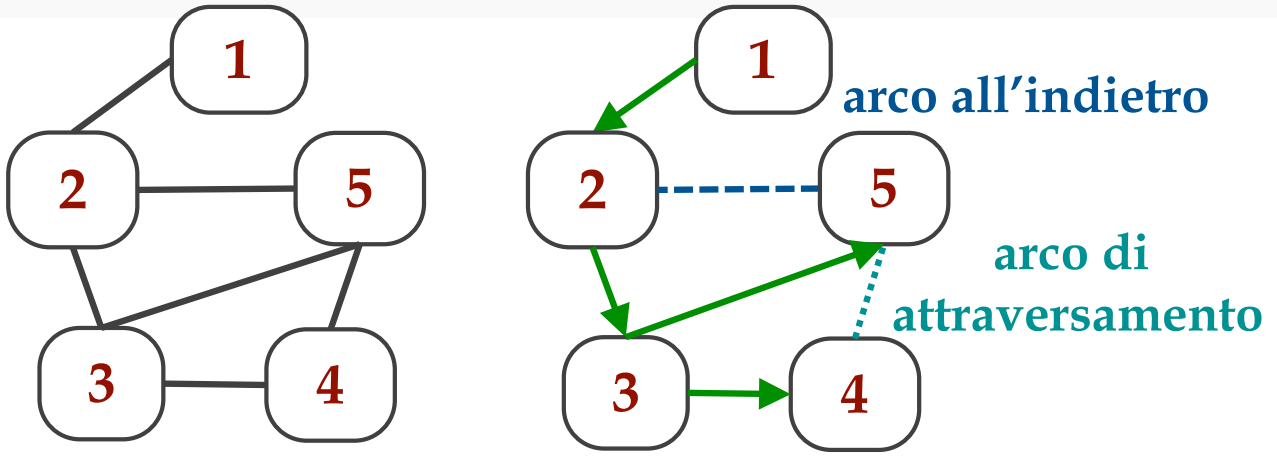
Risultati di una visita

Una visita comincia da un **nodo *s*** (**radice della visita**) e ha l'obiettivo di **visitare per tutti i nodi**.

Il **risultato** di una visita è **un albero ricoprente** di visita **radicato in *s*** che contiene tutti i nodi del grafo e gli **archi attraversati** per **scoprire** ciascun **nodo per la prima volta**.

Durante la visita si percorrono anche **gli altri archi**, ma questi non entrano dell'albero di visita, perché **conducono a nodi già visitati**. Sono classificati in:

- **archi all'indietro**: archi che congiungono un nodo u con un nodo v che è un antenato nell'albero di visita.
- **archi di attraversamento**: tutti gli altri archi (vanno da un nodo a nodi fratelli e cugini di vario grado nell'albero).

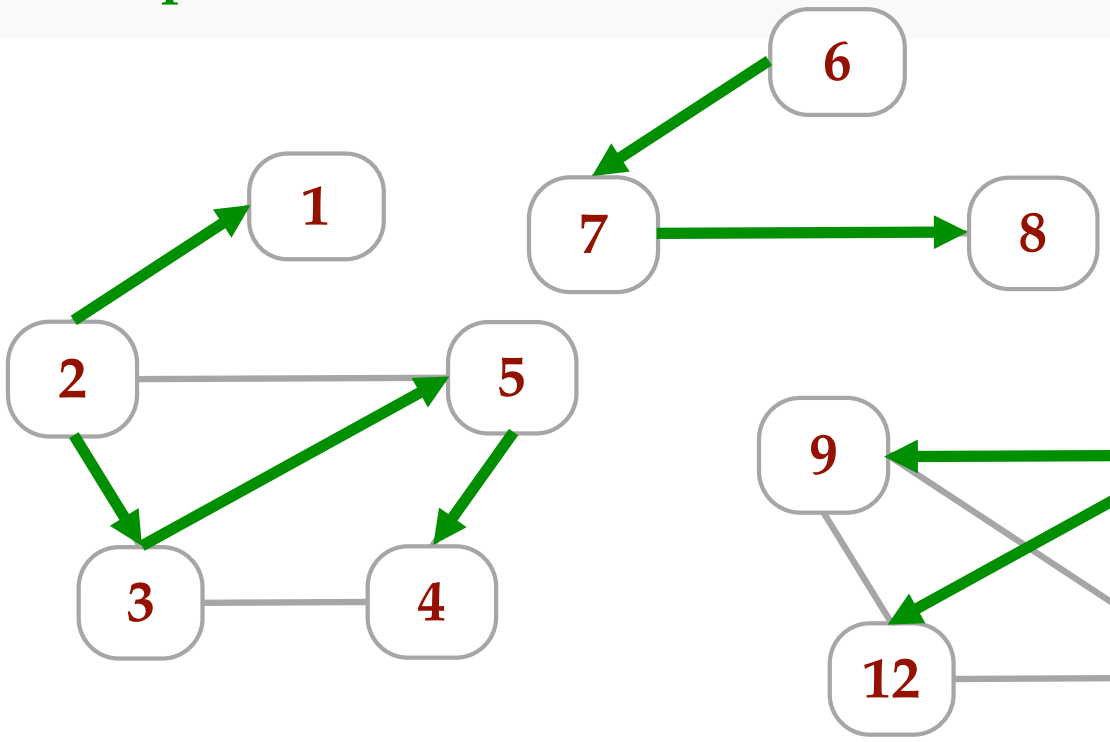


Visite e connessione

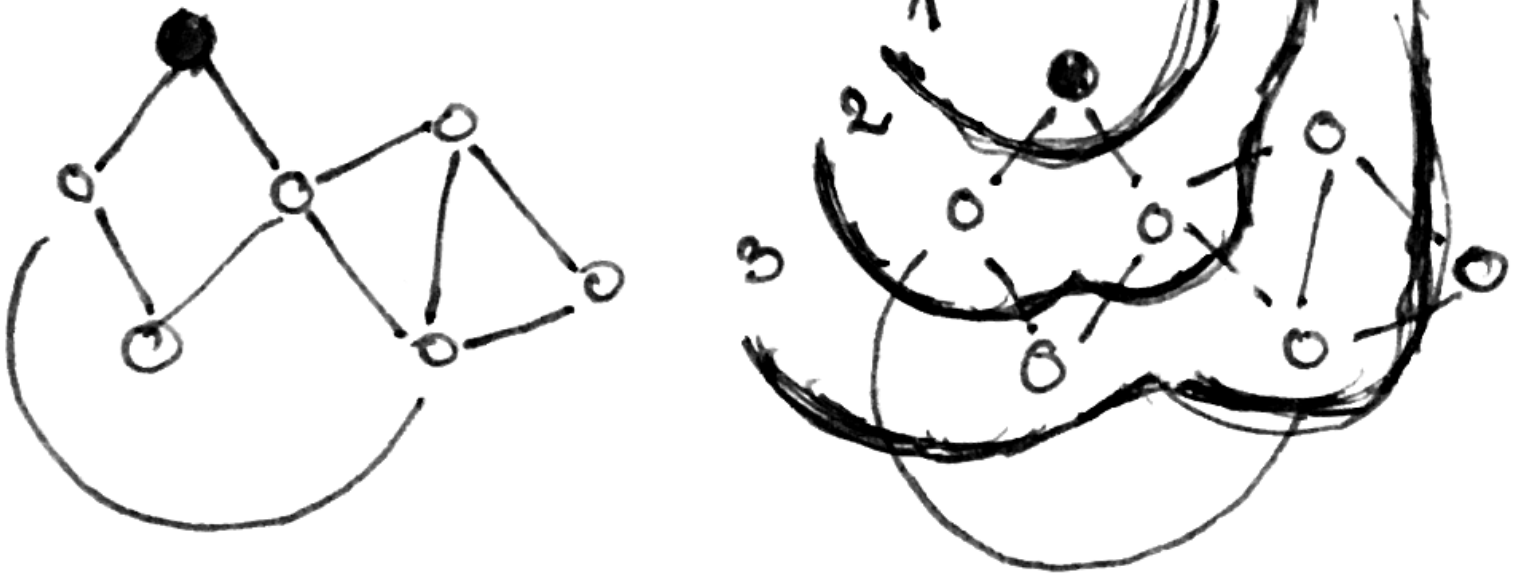
Cominciando una visita in un nodo *s* ed **esplorando** via via **nodi adiacenti** a quelli già scoperti, si **esplora** solo la **componente connessa di *s***.

A meno che non sia altrimenti specificato, **assumeremo il grafo** da visitare **connesso**.

Per **visitare** un grafo **non connesso**, **quando una visita termina**, si verifica che non ci siano nodi non visitati. Se ci sono nodi non visitati, **si fa partire una nuova visita**.



In questo caso la visita è formata da 3 alberi, tra loro non collegati.



Visita in ampiezza (BFS)

Visita in ampiezza

La **visita in ampiezza** (**breadth-first search**, **BFS**) è l'analogo della **visita per livelli** di un **albero**.

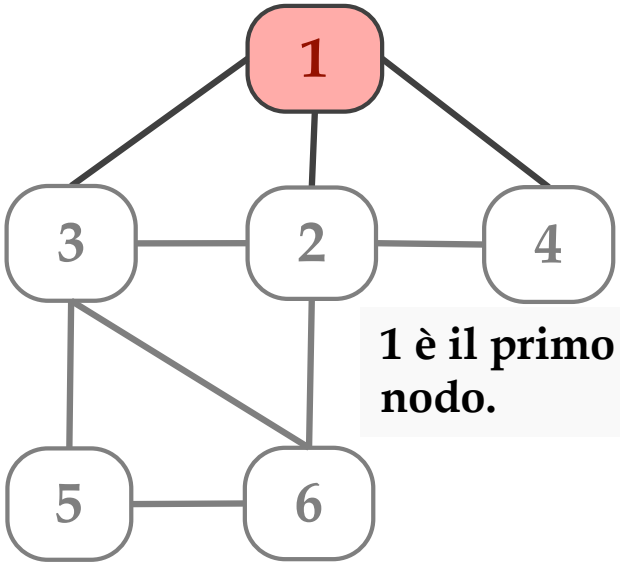
Partendo da un nodo *s*, l'idea è quella di **visitare prima tutti i vicini**, cioè i nodi adiacenti ad *s*, e poi riapplicare questa strategia ai nuovi nodi scoperti.

Per fare progressi, **occorre memorizzare i nodi già visitati** e quando si incontrano, **evitare di far partire nuove ricerche** da quei nodi.

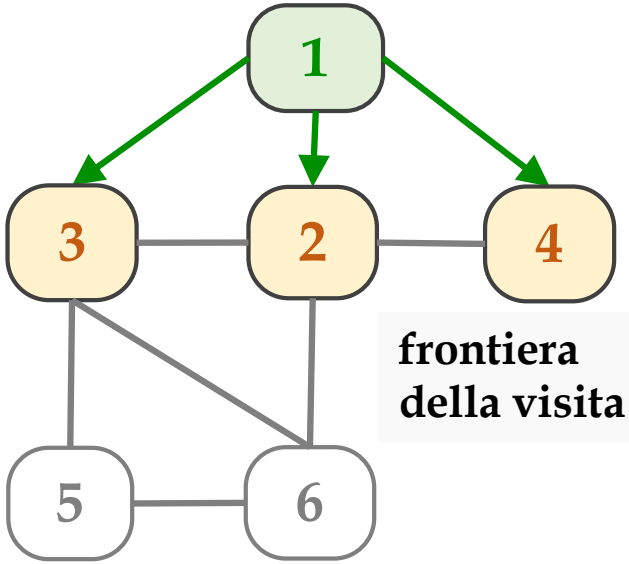
Tuttavia, verranno percorsi **tutti gli archi** perché non è possibile sapere che non conducano a nuovi nodi.

Quando **sono stati scoperti tutti nodi** da scoprire la **visita si arresta**.

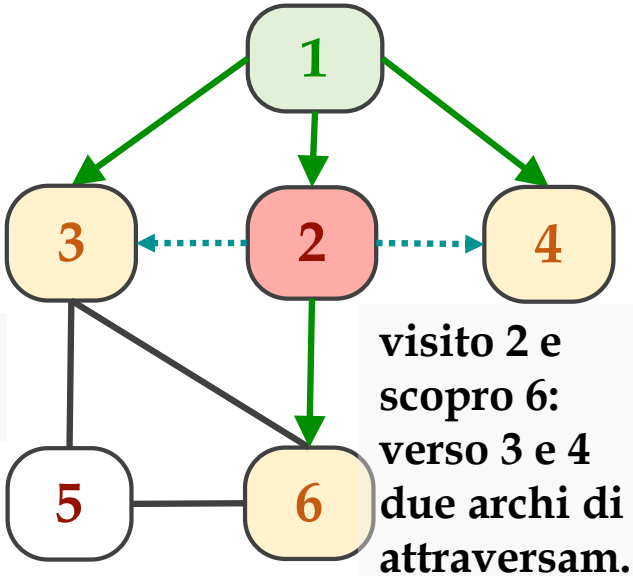
Visita in ampiezza: Esempio



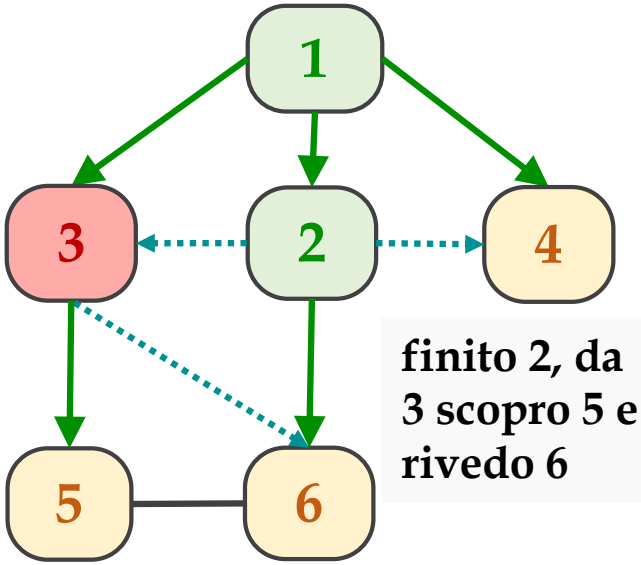
1 è il primo nodo.



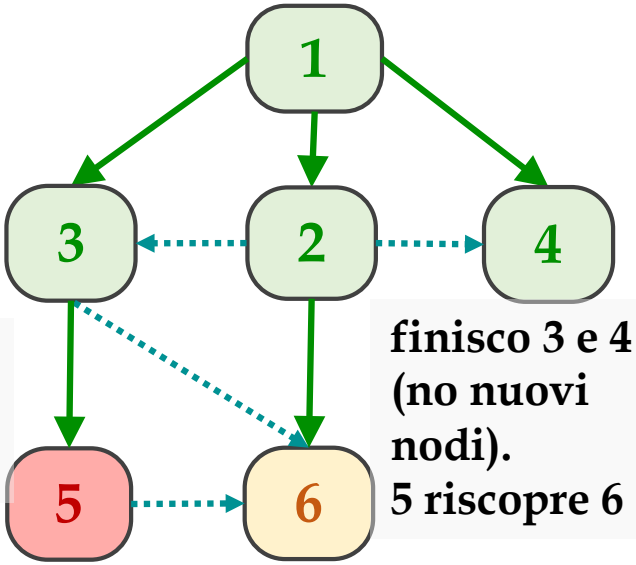
frontiera della visita



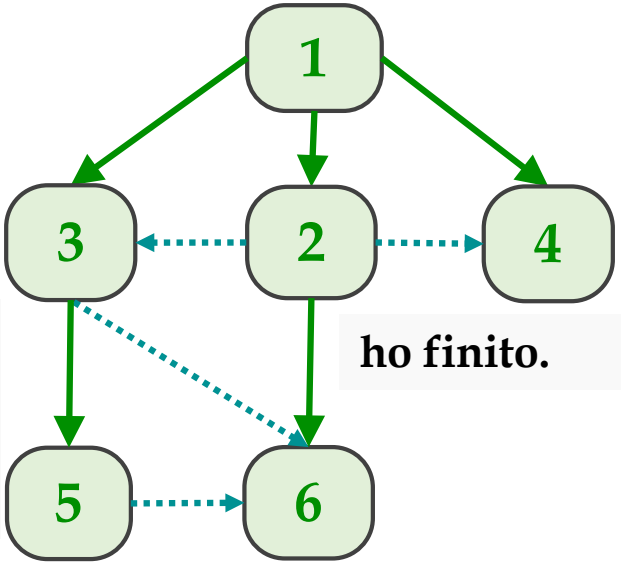
visito 2 e scopro 6: verso 3 e 4 due archi di attraversam.



finito 2, da 3 scopro 5 e rivedo 6



finisco 3 e 4 (no nuovi nodi).
5 riscopre 6



ho finito.

Visita in ampiezza: pseudocodice

Vediamo prima una versione astratta, basata su insiemi. In particolare, useremo i seguenti **insiemi di nodi**:

- **F** è la **frontiera** della visita, nodi incontrati ma di cui dobbiamo **ancora esplorare il vicinato**;
- **VIS** è l'insieme dei nodi **già visitati**
- **N** sono i **nuovi nodi scoperti**. Verranno **aggiunti alla frontiera, dopo che ho finito l' esplorazione di *F***.

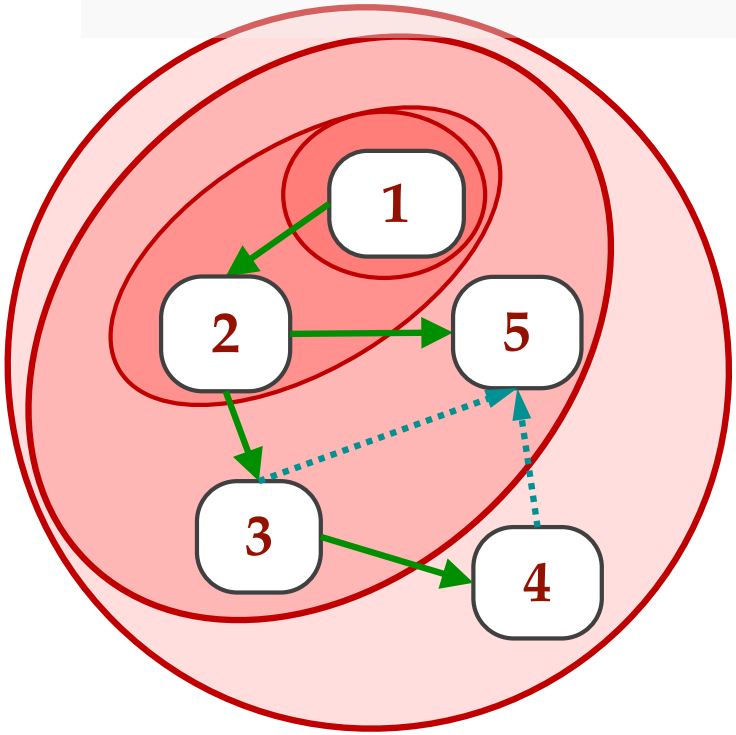
L'esplorazione termina quando $F = \varnothing$ (ciò è equivalente anche a $VIS = V$).

```
def bfs(G, s):
    F = {s} # frontiera della visita
    T = (G.V, ∅)   # albero di visita
    VIS = { s } # nodi visitati
    forall  u ∈ F:
        N = ∅
        forall v ∈ vicini(u):
            if v ∉ VIS: # nuovo nodo?
                N = N ∪ {v}
                T.E = T.E ∪ {(u, v)}
                VIS = VIS ∪ {v}
        F = N
    # end forall
    return T
```

Visita in ampiezza: proprietà

La visita in ampiezza ha **notevoli proprietà**:

- **non ci sono archi all'indietro**
- gli archi di **attraversamento** connettono **nodi sullo stesso livello** o su **livelli consecutivi**.
- l'**albero di visita** è un **albero di cammino minimi**: ogni nodo v è legato alla radice s da un **cammino minimo** da s a v.



Livelli della visita

Esempio: Gli archi **verdi** sono gli **archi dell'albero**. Il nodo 1 sta a livello 0, 2 al livello 1, 3 e 5 al livello 3 e infine 4 al livello 3.

Gli archi **tratteggiati azzurri** sono **archi di attraversamento**: l'arco (3, 5) connette due nodi allo stesso livello, mentre l'arco (5, 4) connette due nodi di due livelli successivi.

BFS: archi (im)possibili

Proposizione: **Non ci sono mai** in un grafo G **archi all'indietro** rispetto all'albero di visita prodotto da una **BFS**.

Dim: Supponiamo per assurdo ci sia un arco all'indietro $u \rightarrow v$.

Ciò significa che u è stato scoperto dopo v. Quindi u non era stato scoperto quando ho esplorato tutti i vicini di v. Ma u e v sono adiacenti perché esiste l'arco $u \rightarrow v$ nell'albero di visita. □

Proposizione: Rispetto all'albero di visita prodotto da una **BFS**, **non ci sono mai archi di attraversamento** in G tra due nodi che stanno nell'albero **in livelli distanti 2**.

Dim: Supponiamo per assurdo ci sia un arco di attraversamento $u \rightarrow v$ con u al livello l e v al livello almeno $l + 2$.

Ciò significa che v è stato scoperto dopo u. Quindi v non era stato scoperto quando ho esplorato tutti i vicini di u. Ma u e v sono adiacenti perché esiste l'arco $u \rightarrow v$ nell'albero di visita.

Quindi v è necessariamente al livello (al più) $l + 1$. □

BFS: cammini minimi

Proposizione: I **cammini nell'albero di** una **visita BFS** radicata in un nodo s sono **cammini minimi** tra s e v, per ogni $v \in V$.

Dim: Supponiamo per assurdo ci sia un cammino da s a v in G più corto di quello nell'albero di visita.

Questo cammino sarà costituito di archi dell'albero e archi di attraversamento. Andando da s a v immaginiamo di orientare gli archi verso livelli più grandi nell'albero.

Gli archi dell'albero fanno fare un progresso di 1 livello verso v, mentre quelli di attraversamento possono anch'essi far fare un progresso 1 oppure nessun progresso.

Di conseguenza questo ipotetico cammino dev'essere al meglio lungo quanto il cammino presente nell'albero. □

Visita in ampiezza: implementazione

L'implementazione della BFS necessita di specificare più in dettaglio le operazioni richieste dall'algoritmo astratto:

- **inserimento/estrazione** dagli insiemi ***VIS, F, N***.
- generazione dei **nodi adiacenti** a un nodo u: questo dipende da **come è memorizzato** il grafo.

Nella BFS, per visitare i nuovi nodi scoperti di N dopo quelli nella frontiera F, è sufficiente **inserire tutti i nodi in una coda** esattamente nell'**ordine in cui vengono scoperti**: infatti, così facendo, ogni nuovo nodo **entra dopo quelli già scoperti** (e inseriti nella coda) e uscirà dalla coda dopo che questi sono stati processati.

I **nodi già visitati** (insieme **VIS**) vengono rappresentati semplicemente con un vettore $marked[v]$ **indicizzato sui nodi** che viene messo a TRUE ogni volta che un nodo viene visitato.

Vedremo in seguito che potremo usare **vettori di questo tipo** anche per **raccogliere altre informazioni** durante una visita.

È comodo rappresentare l'**albero di visita** come **vettore dei padri**.

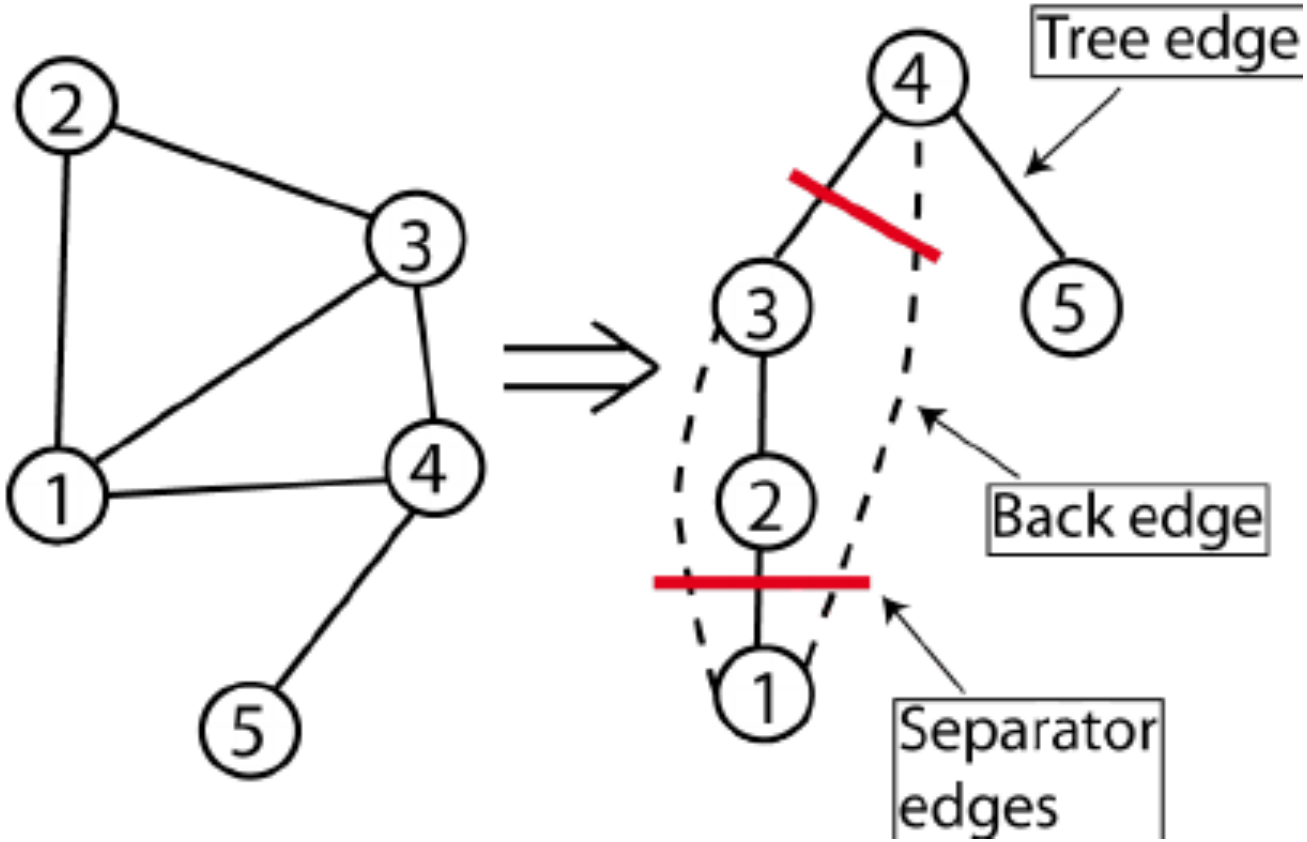
BFS: pseudocodice "specifico"

Vediamo lo pseudocodice con coda e grafo rappresentato con **liste di adiacenza**. Siccome **ogni nodo appare al più una volta in Q** (grazie al controllo su marked, che rappresenta VIS), e quindi si scorre **1 sola volta la lista di adiacenza di ogni nodo**, la complessità è $\Theta(n+m)$.

```
def bfs(G, s):
    Q, p = newQueue(), allocaV(|G.V|)
    marked = allFalse(n)
    enqueue(Q, s)
    p[s] = s, True # albero di visita
    marked[s] = True
    while not isEmpty(Q):
        u = dequeue(Q)
        # con liste adiacenza
        forall v ∈ G.adj(u):
            if not marked[v]:
                enqueue(Q, v)
                p[v], marked[v] = u, True
    # end while
    return p
```

Con la **matrice di adiacenza** il codice è molto simile, cambia solo la scansione degli adiacenti, ma la complessità diventa $\Theta(n^2)$

```
[...]
# matrice adiacenza
for v=1 to n:
    if G.a[u][v] == 1: # ∈ adj(u)
        if not marked[v]:
            [...]
```



Visita in profondità (DFS)

Visita in profondità

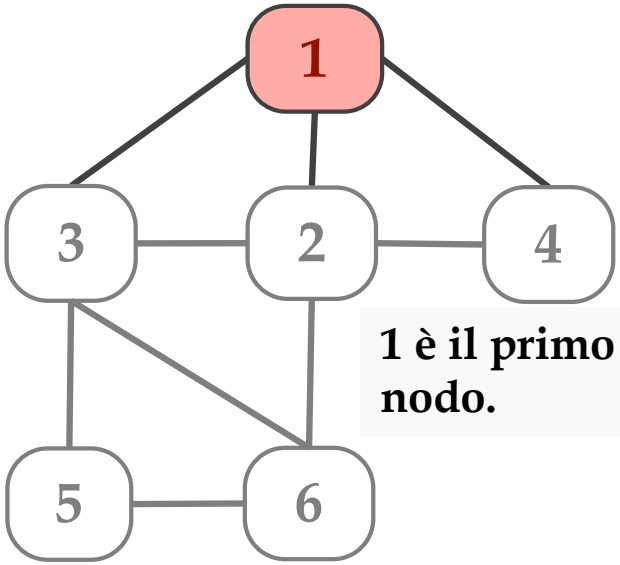
La **visita in profondità** (**depth-first search**, **DFS**) è l'analogo delle **visita in profondità** di un albero, anche se usualmente **non si distinguono diversi ordini** in cui visitare gli adiacenti.

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**.

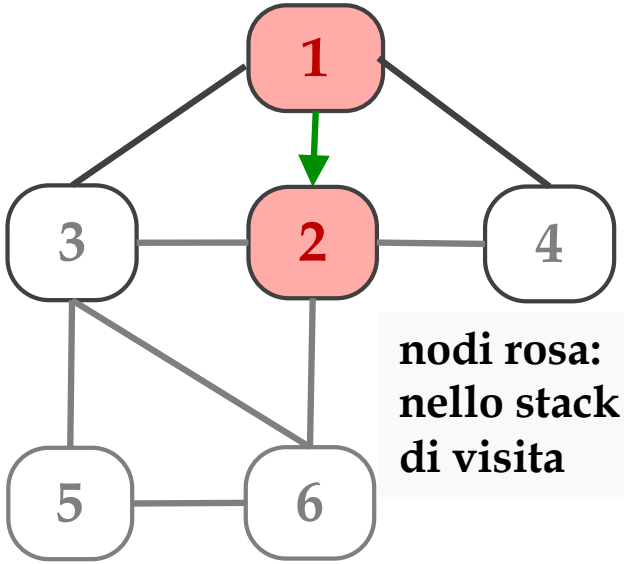
Anche nella DFS, per fare progressi, occorre memorizzare i nodi già visitati e quando si incontrano, evitare di far partire nuove ricerche da quei nodi.

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

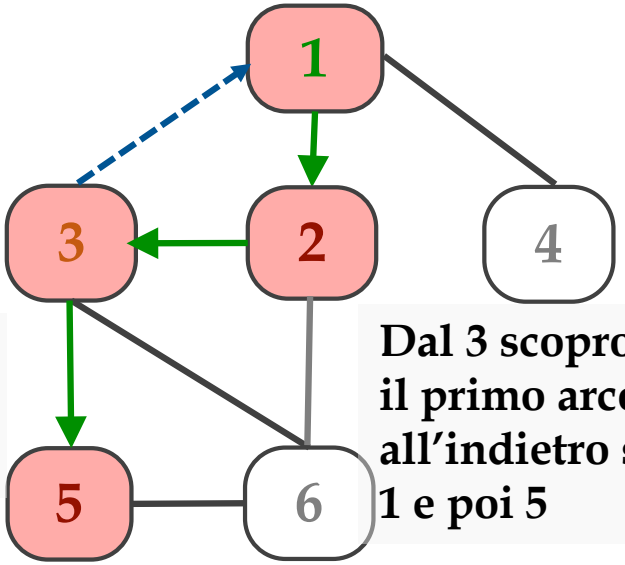
Visita in profondità: Esempio



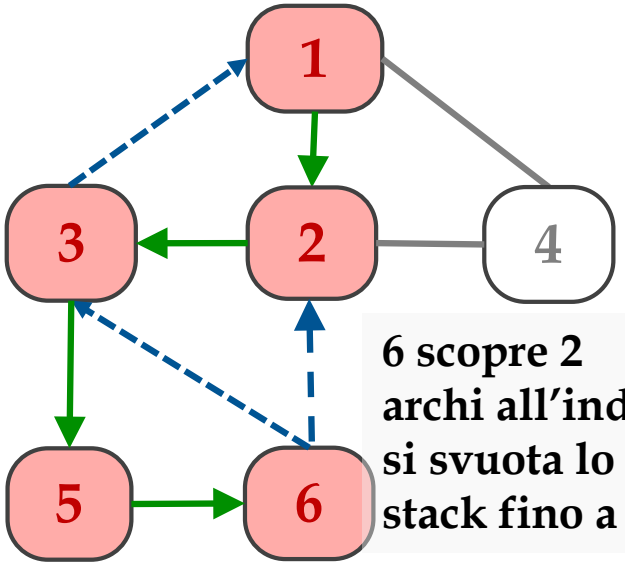
1 è il primo nodo.



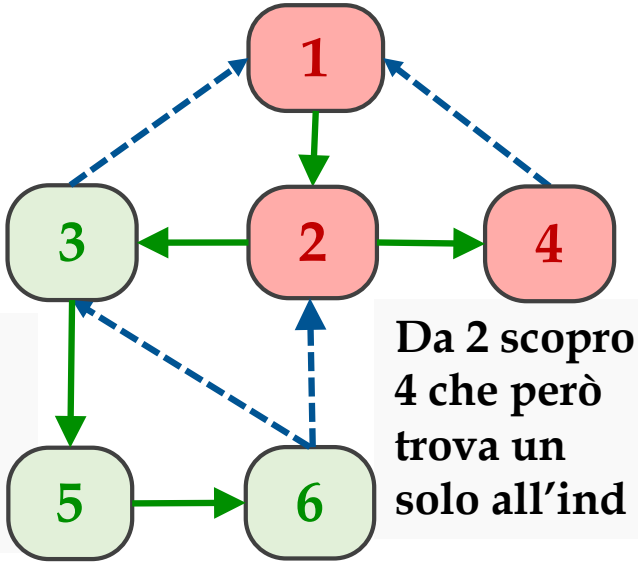
nodi rosa: nello stack di visita



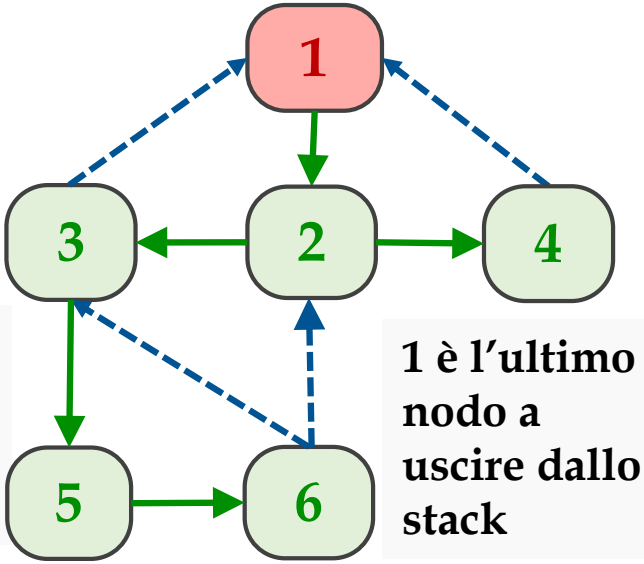
Dal 3 scopro il primo arco all'indietro su 1 e poi 5



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

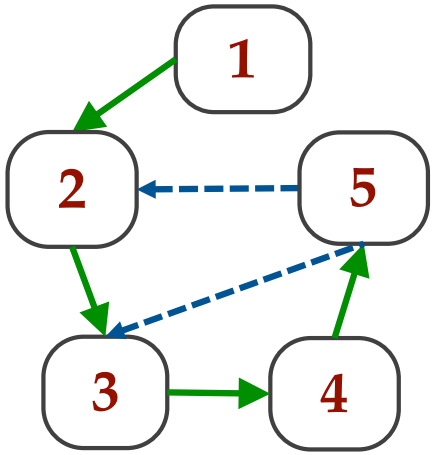
Per sostanziare una importante proprietà della dfs, **registriamo per ogni nodo il tempo di entrata** (vettore `in`) e il **tempo di uscita** (vettore `out`).

I tempi di ingresso/uscita dai nodi **sono tutti diversi** (`t` 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.



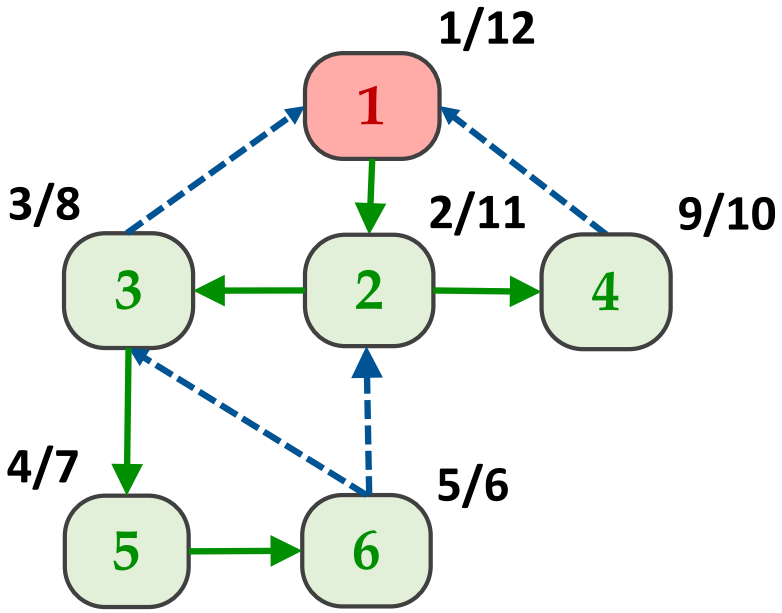
Esempio: Gli archi verdi sono gli archi dell'albero.

Gli archi **tratteggiati blu** sono **archi all'indietro**: gli archi (5, 3) e (5, 2) 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**.

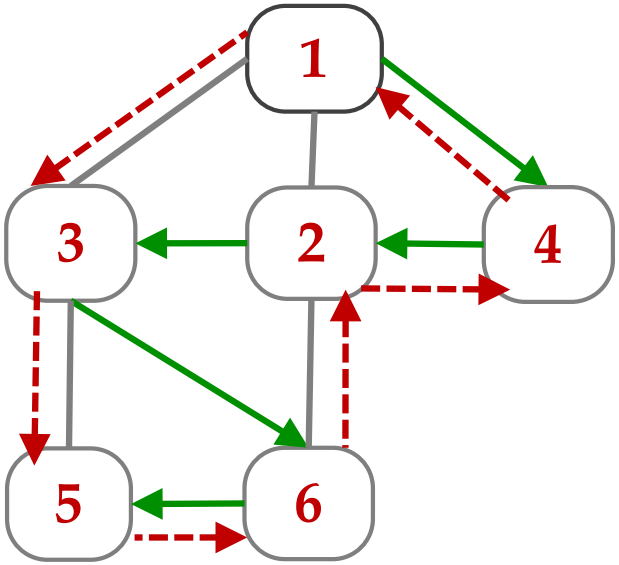
Quello di **6** è **contenuto** in **5**, che è contenuto in **3**, che è contenuto in **2**, che è contenuto in **1**.

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 trovato facendo **tutte le DFS** permutando l'ordine d'esplorazioni dei vicini, ad esempio con una procedura brute force (esaustiva) 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**.