UNIVERSITÀ
DEGLI STUDI
DI TRIESTE

MODULO 2: Grafi

Prof.ssa Giulia Cisotto

giulia.cisotto@units.it

Trieste, 29 aprile 2026

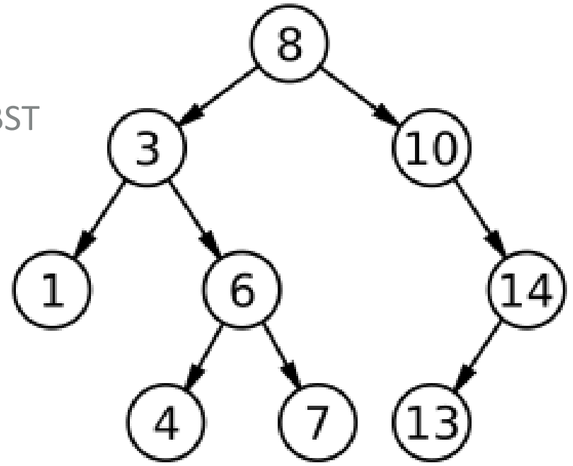
UNIVERSITÀ
DEGLI STUDI
DI TRIESTE

- Definizioni
- Rappresentazioni
- Ricerca/visita in ampiezza
- Ricerca/visita in profondità

DAGLI ALBERI AI GRAFI

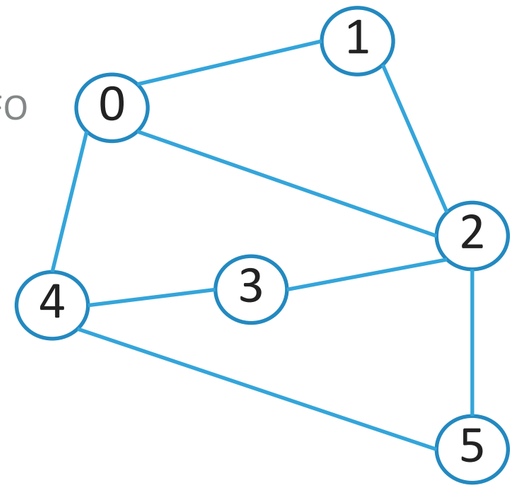
A seconda dell'applicazione posso imporre nuovi vincoli e «costringere» l'inserimento di nuovi elementi in determinate posizioni e «vietarne» l'inserimento in altre.

BST



Se rilassiamo i vincoli imposti per avere un albero, permettiamo i cicli e più di un genitore per nodo, otteniamo un **GRAFO**.

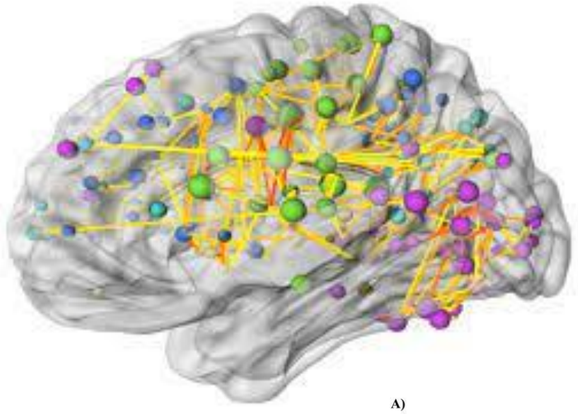
GRAFO



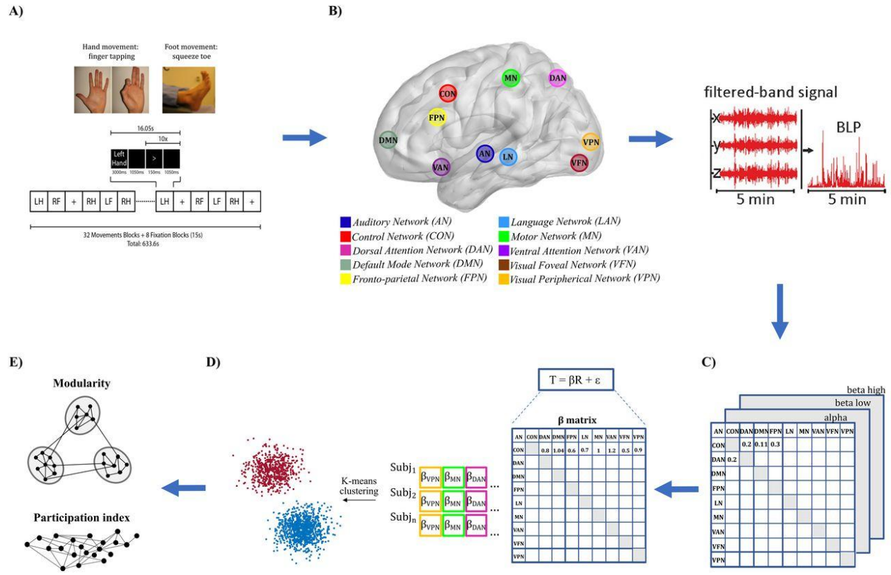
Intuitivamente: rilassare i vincoli, quindi permettendo **maggior libertà** nell'organizzazione dei dati, ha la conseguenza di poter **modellare con un grafo più scenari** ma al tempo stesso molte delle operazioni che si possono effettuare in modo efficiente su un BTS avranno un **costo maggiore** su un grafo.

ESEMPI DI APPLICAZIONE

Air traffic in Europe (in/out)

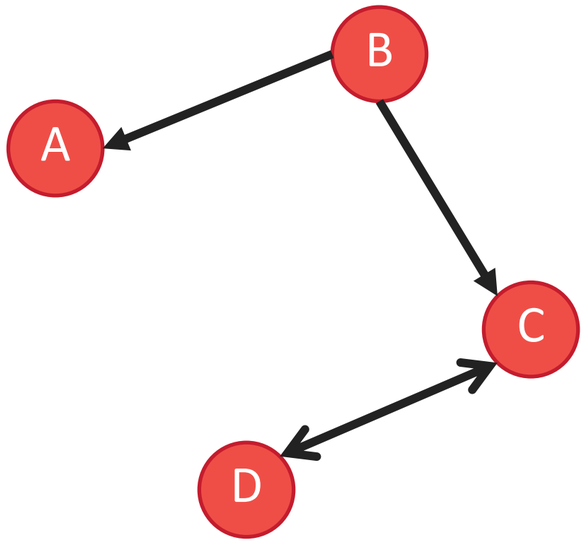


Brain connectome (image from DTW)



Maddaluno, O., Della Penna, S., Pizzuti, A., Spezialetti, M., Corbetta, M., de Pasquale, F., & Betti, V. (2024). Encoding Manual Dexterity through Modulation of Intrinsic α Band Connectivity. *Journal of Neuroscience*, *44*(20).

COSA È UN GRAFO?



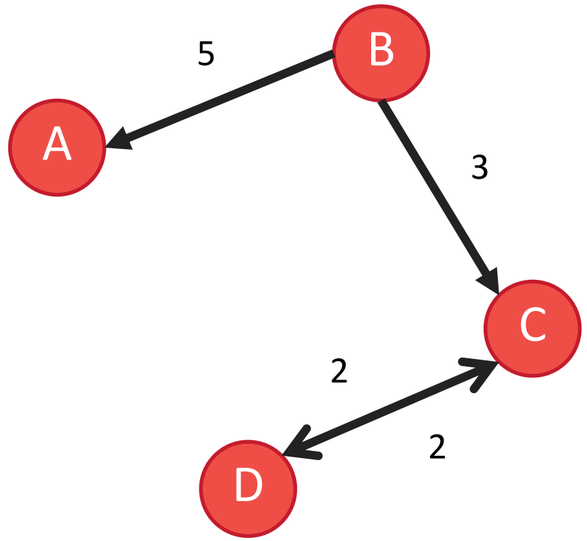
Insieme di nodi:

$$V = \{a, b, c, d\}$$

Insieme di archi:

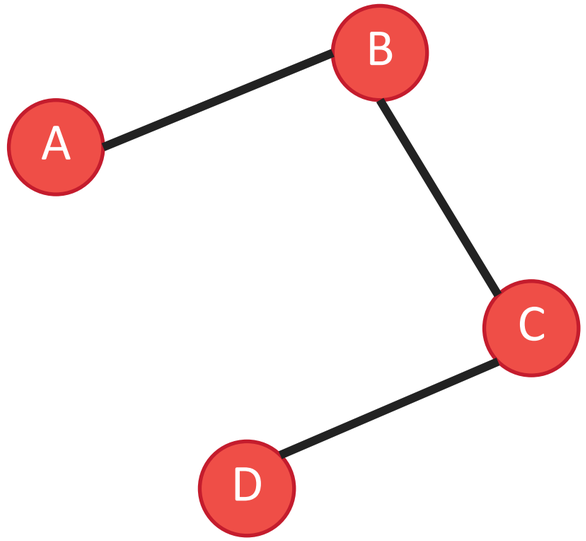
$$E = \{ \quad (b, a), (b, c), \\ (c, d), (d, c)\}$$

COSA È UN GRAFO (VARIANTI)?



Archi pesati e grafo orientato

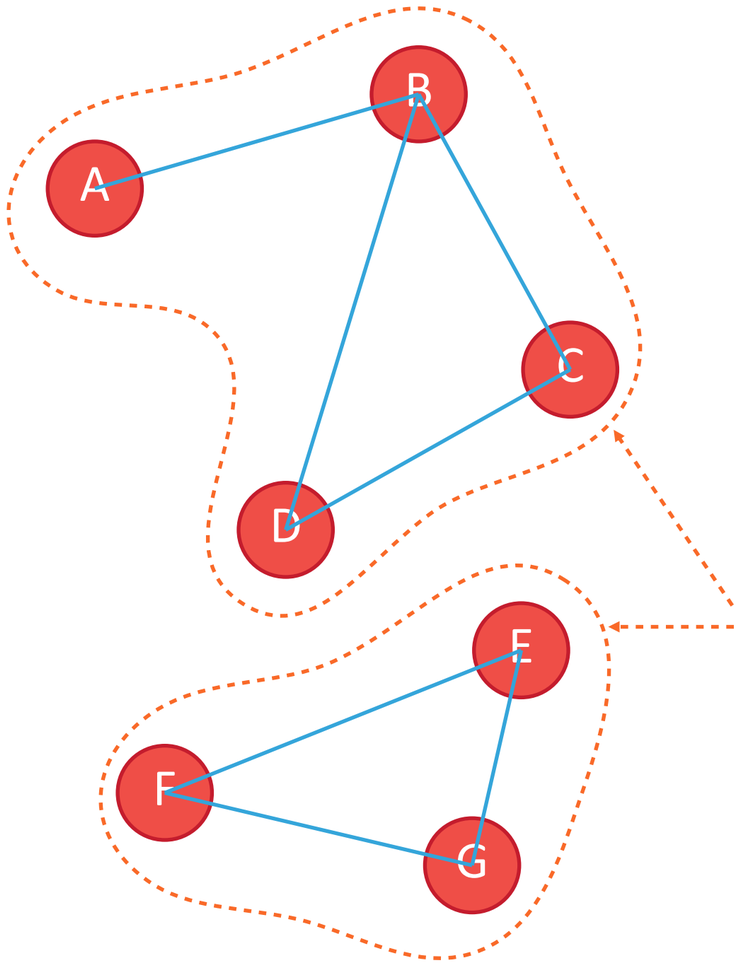
Ovvero esiste una funzione $w: E \to \mathbb{R}$
e indichiamo $w((i,j))$ come $w_{i,j}$



Non orientato

Ovvero $(a,b) \in E \iff (b,a) \in E$
per ogni $a, b \in V$

GRAFI: NOZIONI DI BASE



Percorso: sequenza di archi $(v_0, v_1), (v_1, v_2), \ldots$
dove due archi consecutivi nella sequenza sono adiacenti nel grafo (cioè sono nella forma $(a, b)(b, c)$)

Lunghezza del percorso: numero di archi che il percorso contiene

Distanza tra due nodi a e b: lunghezza del percorso più corto che inizia dal nodo a e termina al nodo b.
Se non esiste un percorso diremo che la distanza è $+\infty$

NOTAZIONE

- Dato un grafo $G = (V, E)$ solitamente l'input alla struttura dati consiste in due parametri (non più solo *n*), ovvero $|V|$ e $|E|$.
- All'interno della notazione asintotica (e solo in quel caso) faremo la semplificazione di utilizzare V e E per indicare $|V|$ e $|E|$.
- Quindi un algoritmo che richiede tempo $O(V + E)$ è da leggersi come $O(|V| + |E|)$

NOTAZIONE

- **Un grafo può avere al più $O(V^2)$ archi**, con il valore esatto che dipende dal fatto che il grafo sia diretto (orientato) o meno.
- Solitamente un grafo con un numero di archi vicino al massimo possibile viene detto **denso** mentre uno con un pochi archi viene detto **sparso**.
- Cosa effettivamente sia considerato un grafo denso o sparso dipende molto dall'applicazione

Per un grafo **semplice non orientato**, **senza loop**, il massimo è: $\frac{V(V-1)}{2}$

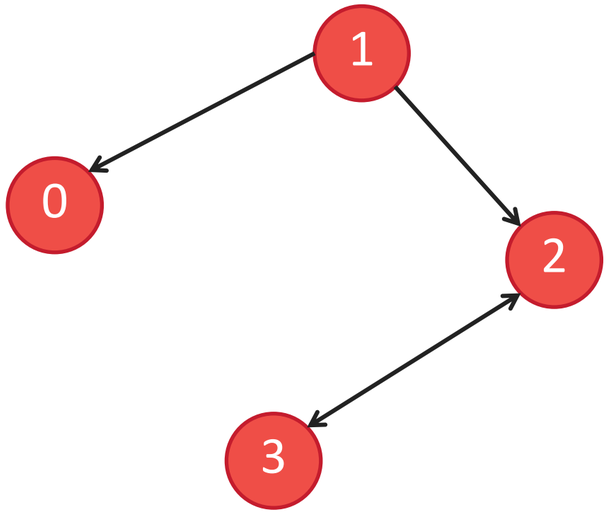
Se si ammettono anche i loop, allora il numero massimo di archi è: $\frac{V(V-1)}{2} + V$

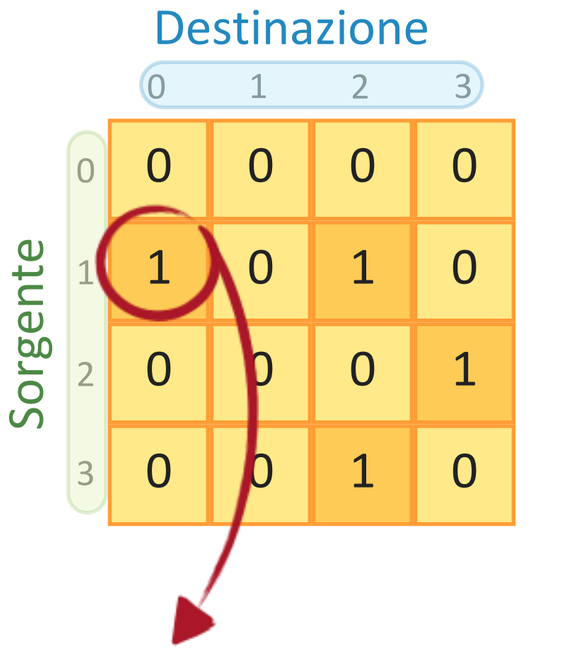
Nel grafo **orientato con loop ammessi**, ogni vertice può avere un arco diretto verso qualunque vertice, incluso sé stesso. Ogni vertice ha V possibili archi uscenti e ci sono V vertici. Il totale è: $V \cdot V = V^2$
Se invece i loop non sono ammessi, vanno tolti i V archi del tipo $A \to A$, $B \to B$, ecc., quindi: $V^2 - V = V(V-1)$

COSA DOBBIAMO GESTIRE?

- Dobbiamo essere in grado di salvare i nodi…
- …e gli archi (eventualmente con un peso).
- Assumiamo che i nodi abbiano nomi {0,…,n-1}
- Ci sono due modi “standard” di rappresentare un grafo:
  - Matrici di adiacenza
  - Liste di adiacenza
- Assumiamo che il grafo sia statico (cioè non cambi nel tempo)

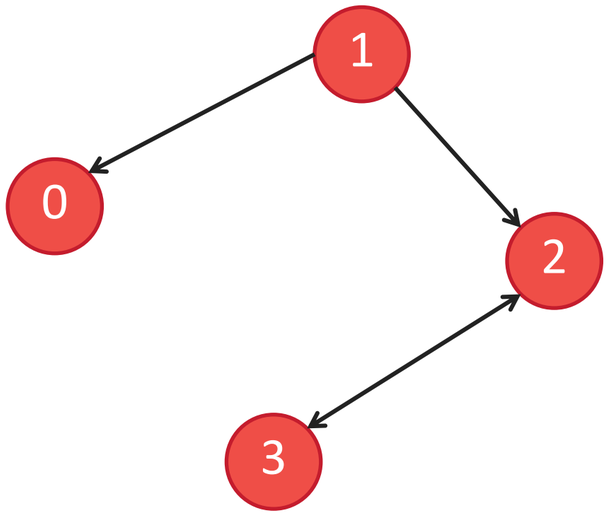
MATRICI DI ADIACENZA

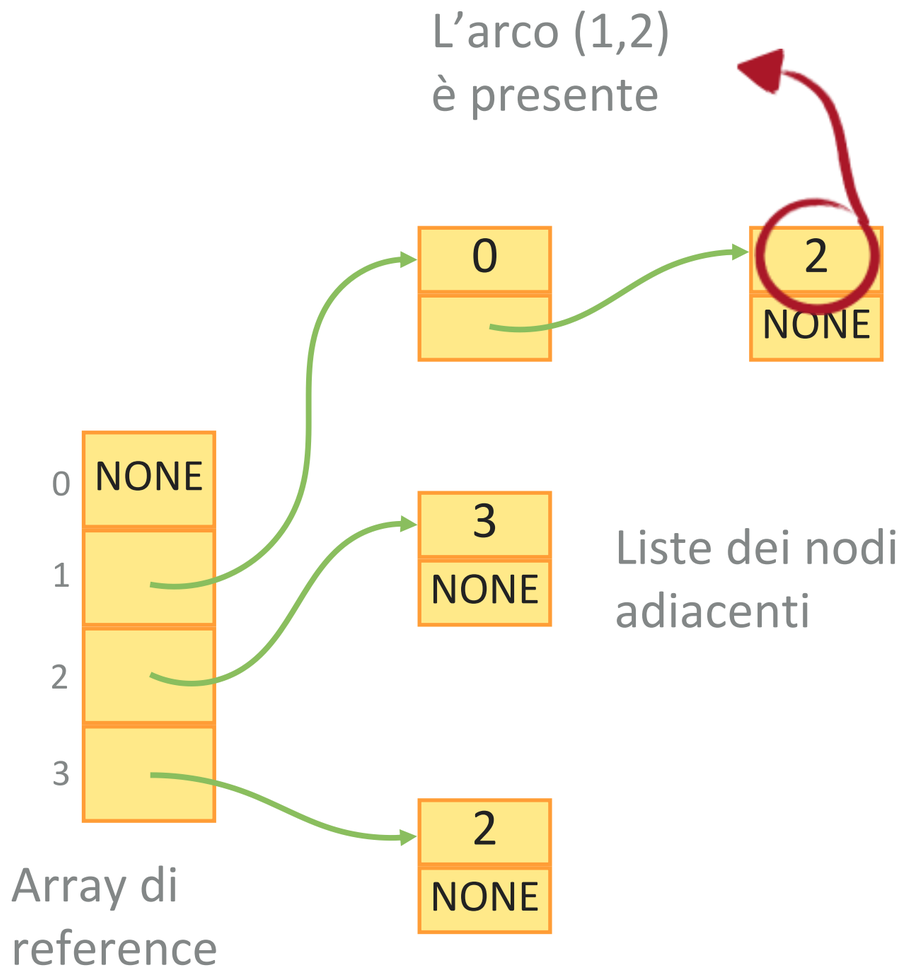




L’arco (1,0) è presente

LISTE DI ADIACENZA





Nota. La lista di adiacenza mette in fila tutti i figli di ogni vertice (non rappresenta connessioni tra questi ultimi)

QUALE RAPPRESENTAZIONE USARE

Matrici di adiacenza

Veloce (tempo costante) a stabilire se un arco esiste

Occupazione quadratica di memoria* rispetto al numero di vertici $O(V^2)$

Funziona bene per grafi densi

Liste di adiacenza

Lento (serve scandire una lista) per stabilire se un arco esiste

Occupazione lineare di memoria rispetto al numero di vertici e archi $O(V + E)$

Funziona bene per grafi sparsi

implementata come **array bidimensionale (n × n, con n = numero di nodi). Data una coppia di indici, l'accesso è immediato O(1).*

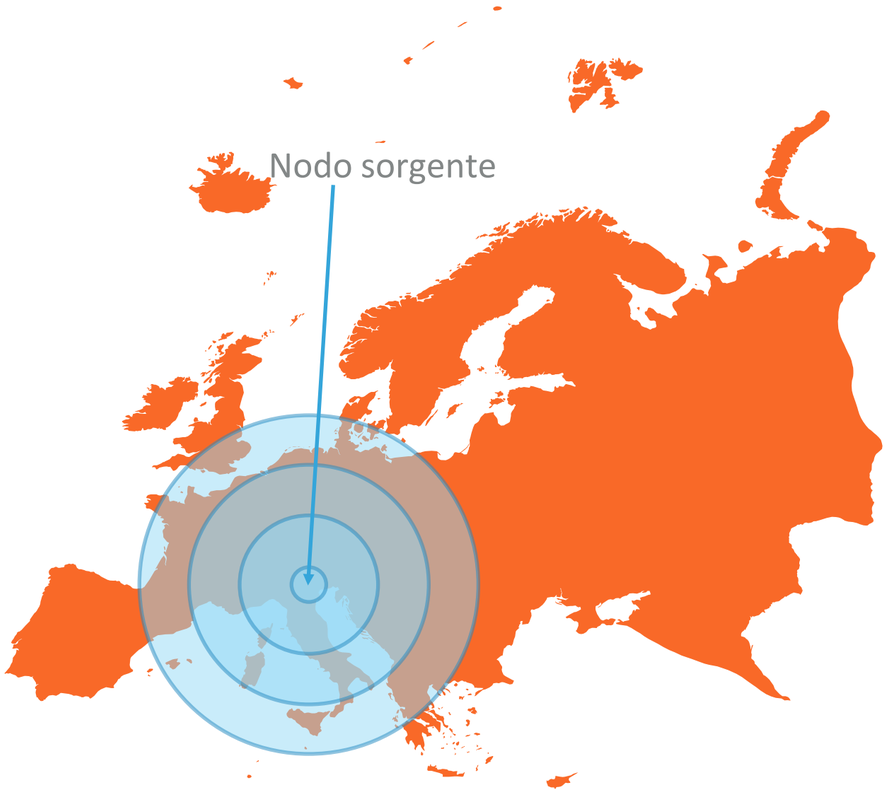
RICERCA IN AMPIEZZA

- La ricerca in ampiezza (**breadth-first search o BFS**) è uno degli algoritmi di base per la ricerca su grafi
- Dato un grafo $G = (V, E)$ e un nodo $s \in V$ detto ***nodo sorgente*** la ricerca in ampiezza esplora tutti i nodi raggiungibili a partire da s individuando:
  - La distanza da s a ognuno dei vertici raggiungibili
  - Un albero (detto albero BFS) che contiene tutti i vertici raggiungibili

RICERCA IN AMPIEZZA

- Perché ricerca in ampiezza?
- A partire dal nodo s si esplorano prima tutti i nodi direttamente raggiungibili da s (quelli a distanza 1)
- Poi tutti i nodi raggiungibili in due passi (distanza 2), ecc.
- Quindi prima di allontanarci dal nodo sorgente esploriamo tutti quelli vicini, poi i loro vicini, ecc.

AMPIEZZA VS PROFONDITÀ



Ricerca in ampiezza:
esploriamo tutti nodi vicini prima di allontanarci

Ricerca in profondità:
seguiamo un singolo percorso il più possibile
prima di cambiare strada

COLORARE I NODI

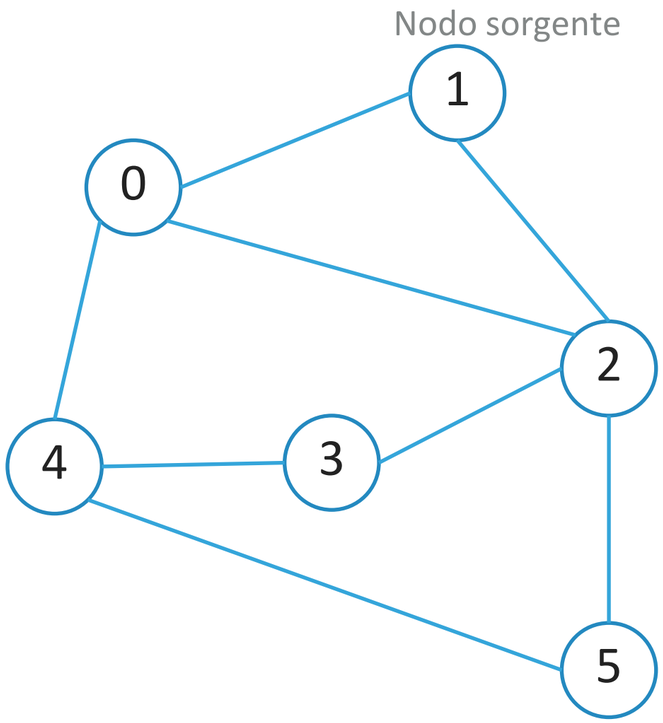
- Durante la ricerca in ampiezza (e in generale per le ricerche nei grafi) dobbiamo evitare di visitare un nodo più volte. Per questo assegnamo ad ogni nodo un colore:
- **Bianco**: il nodo non è ancora stato visitato
- **Grigio**: il nodo è stato visitato ma potrebbe avere dei vicini non visitati
- **Nero**: il nodo è stato visitato ed anche tutti i suoi vicini

COME VISITARE TUTTI I NODI DEL GRAFO SENZA RIPETIZIONI?

Effettuiamo queste «azioni» nell'ordine:

1. Teniamo una coda di nodi grigi (dei quali potrebbero mancarci dei vicini da esplorare)
2. Inizialmente solo il nodo sorgente è grigio e viene accodato
3. Estraiamo un nodo dalla coda, coloriamo di grigio tutti i vicini bianchi e li aggiungiamo in coda
4. Ripetiamo finché la coda non è vuota

ESEMPIO DI ESECUZIONE

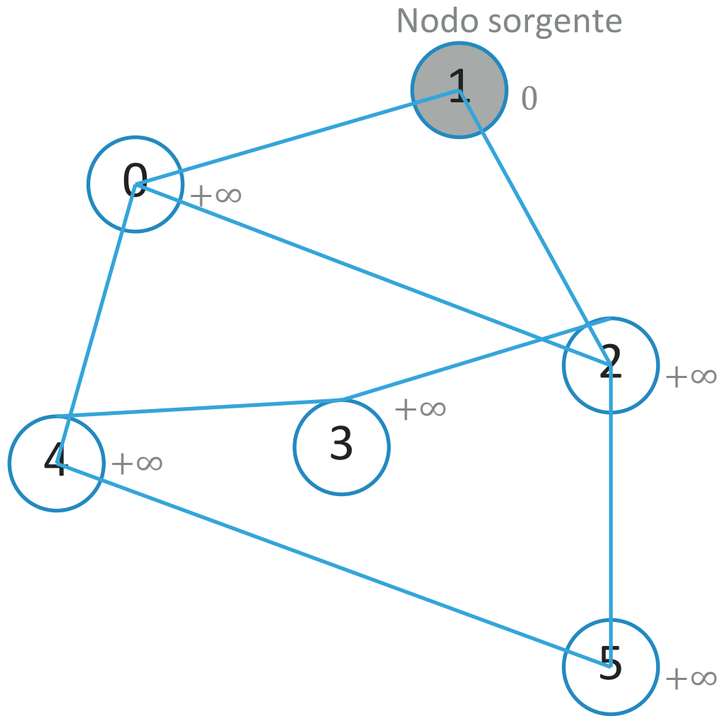


Nota. Come scelgo il nodo sorgente?

- **in modo arbitrario** se si vuole esplorare tutto il grafo.
- i**n base al problema** se si cerca la distanza minima da un nodo specifico verso tutti gli altri.

ESEMPIO DI ESECUZIONE

Coda dei nodi da visitare



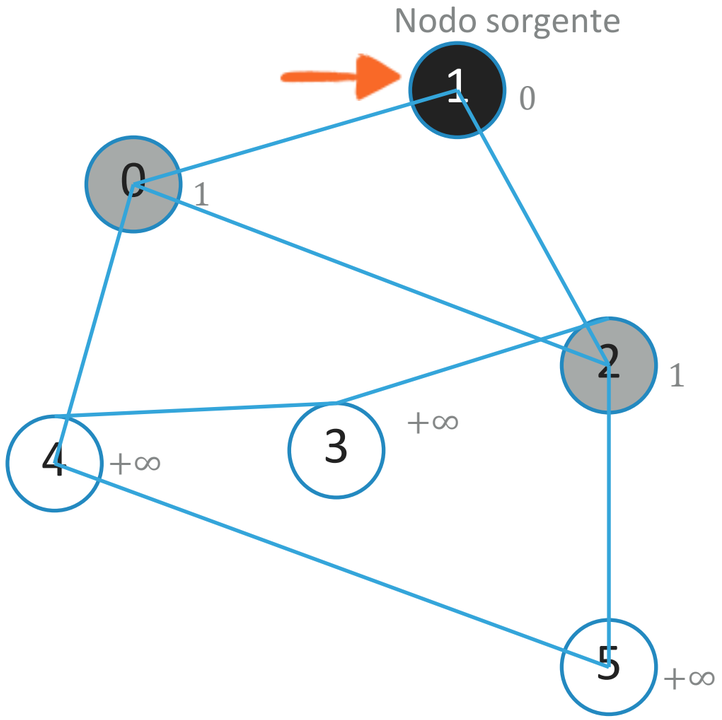
Inizialmente la distanza conosciuta
di tutti i nodi dal nodo di partenza è $+\infty$

Solo il nodo iniziale ha distanza 0 da se stesso
e colore grigio

	0	1	2	3	4	5
Distanza	∞	0	∞	∞	∞	∞

	0	1	2	3	4	5
Predecessore	-	-	-	-	-	-

ESEMPIO DI ESECUZIONE



Coda dei nodi da visitare

Estraiamo il nodo u dalla coda (all'inizio c'è solo l'1)

Per ogni vicino, se è bianco lo coloriamo di grigio, aggiorniamo distanza (come $distanza[u] + 1$) e precedessore (u) e lo accodiamo

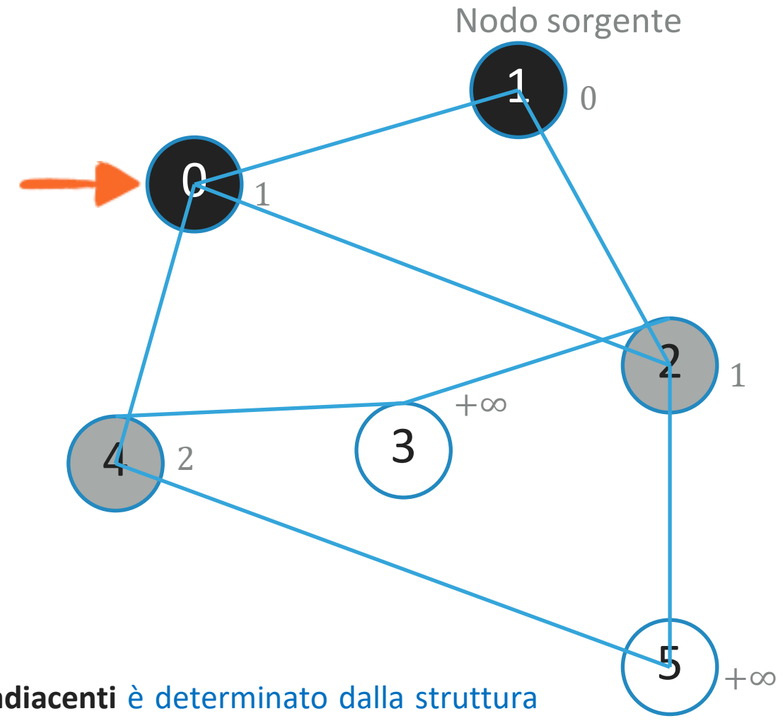
Coloriamo u di nero

	0	1	2	3	4	5
Distanza	1	0	1	∞	∞	∞

	0	1	2	3	4	5
Predecessore	1	-	1	-	-	-

ESEMPIO DI ESECUZIONE

Coda dei nodi da visitare



Estraiamo il nodo u dalla coda

Per ogni vicino, se è bianco lo coloriamo di grigio, aggiorniamo distanza (come $distanza[u] + 1$) e precedessore (u) e lo accodiamo

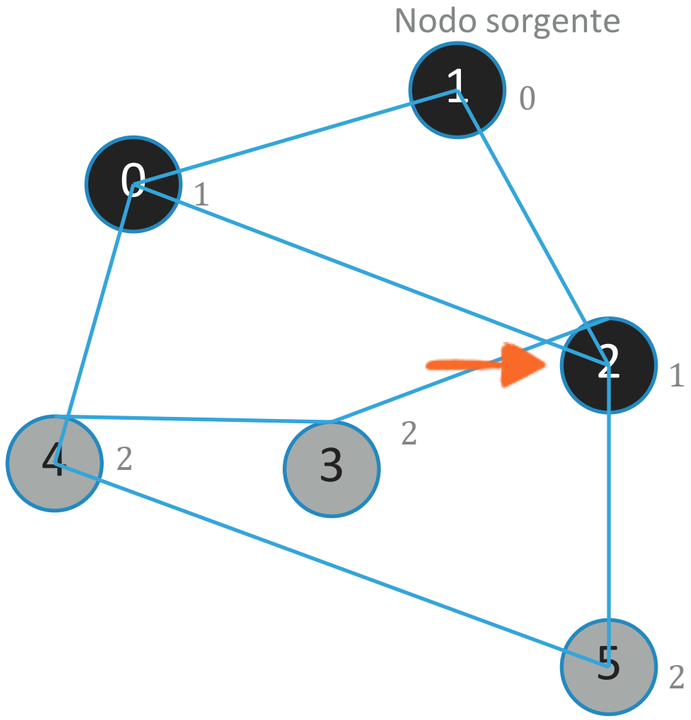
Coloriamo u di nero

	0	1	2	3	4	5
Distanza	1	0	1	∞	2	∞

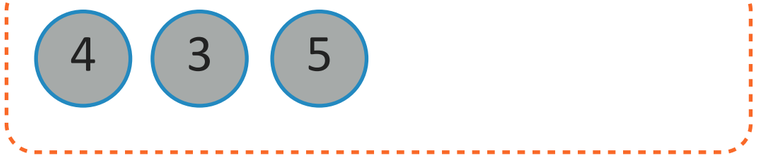
	0	1	2	3	4	5
Predecessore	1	-	1	-	0	-

L'ordine di visita dei nodi adiacenti è determinato dalla struttura dati utilizzata per rappresentare il grafo. ***Se si utilizza una lista di adiacenza***, l'ordine dei vicini dipende dall'ordine in cui sono memorizzati nella lista. ***Se si utilizza una matrice di adiacenza***, l'ordine è determinato dall'indice dei nodi. In assenza di una specifica esigenza, l'ordine di visita dei nodi adiacenti può essere considerato arbitrario

ESEMPIO DI ESECUZIONE



Coda dei nodi da visitare



Estraiamo il nodo u dalla coda

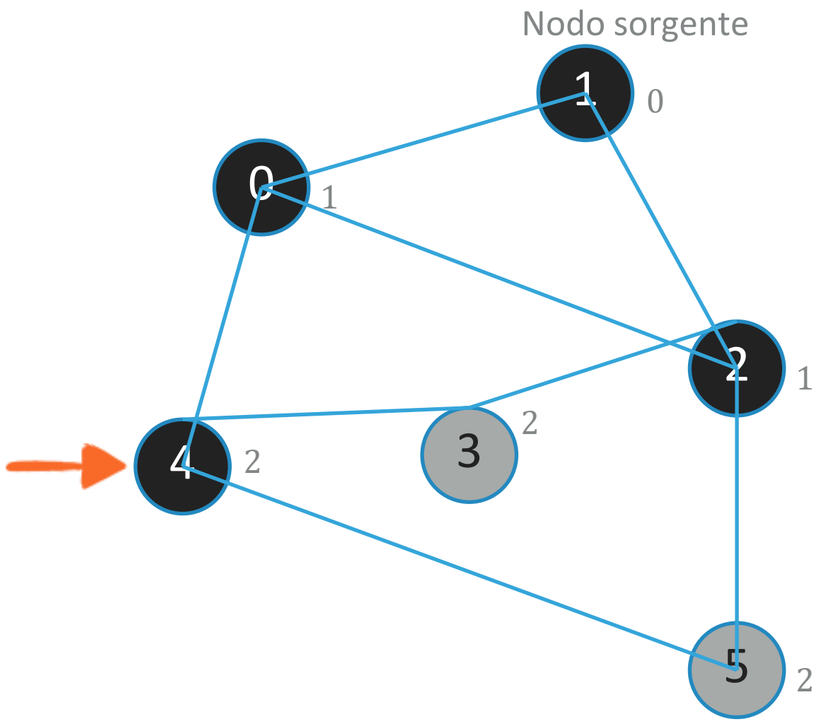
Per ogni vicino, se è bianco lo coloriamo di grigio, aggiorniamo distanza (come $distanza[u] + 1$) e precedessore (u) e lo accodiamo

Coloriamo u di nero

	0	1	2	3	4	5
Distanza	1	0	1	2	2	2

	0	1	2	3	4	5
Predecessore	1	-	1	2	0	2

ESEMPIO DI ESECUZIONE



Coda dei nodi da visitare

Estraiamo il nodo u dalla coda

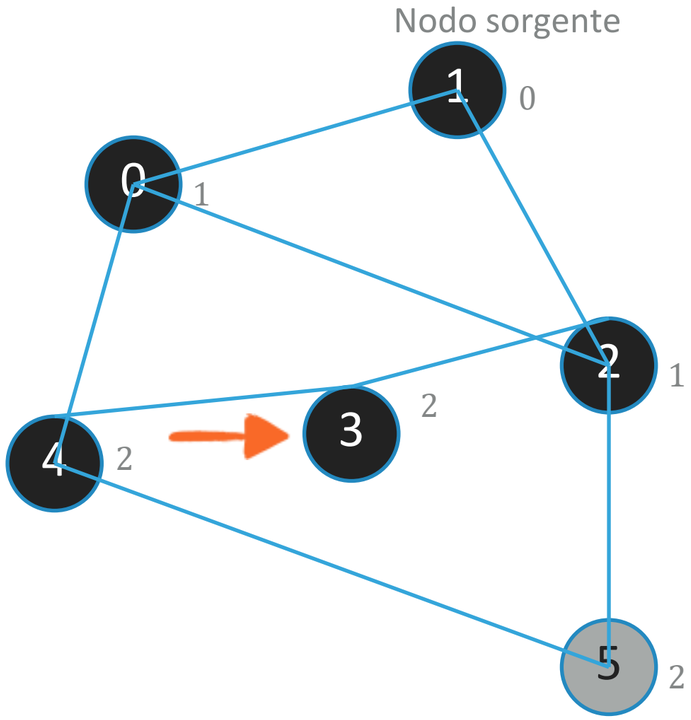
Per ogni vicino, se è bianco lo coloriamo di grigio, aggiorniamo distanza (come $distanza[u] + 1$) e precedessore (u) e lo accodiamo

Coloriamo u di nero

	0	1	2	3	4	5
Distanza	1	0	1	2	2	2

	0	1	2	3	4	5
Predecessore	1	-	1	2	0	2

ESEMPIO DI ESECUZIONE



Coda dei nodi da visitare

Estraiamo il nodo u dalla coda

Per ogni vicino, se è bianco lo coloriamo di grigio, aggiorniamo distanza (come $distanza[u] + 1$) e precedessore (u) e lo accodiamo

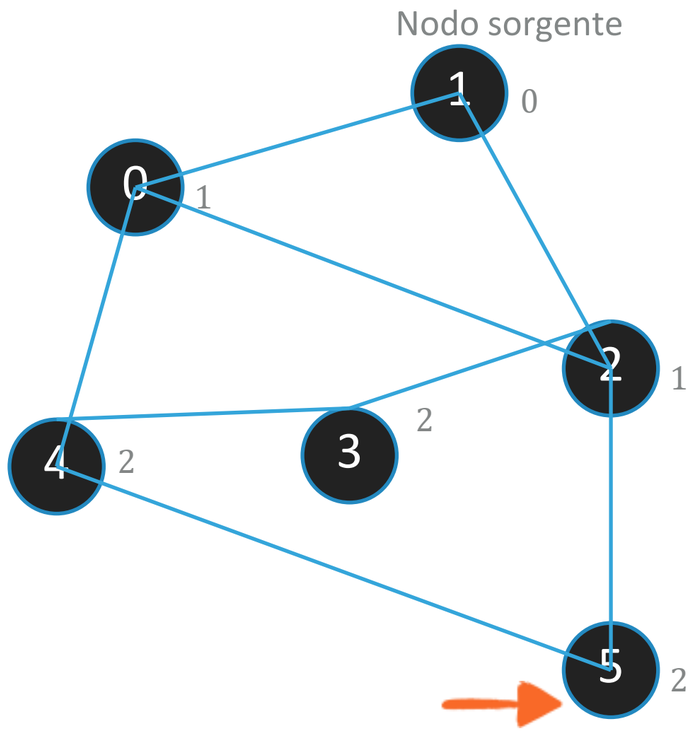
Coloriamo u di nero

	0	1	2	3	4	5
Distanza	1	0	1	2	2	2

	0	1	2	3	4	5
Predecessore	1	-	1	2	0	2

ESEMPIO DI ESECUZIONE

Coda dei nodi da visitare



Estraiamo il nodo u dalla coda

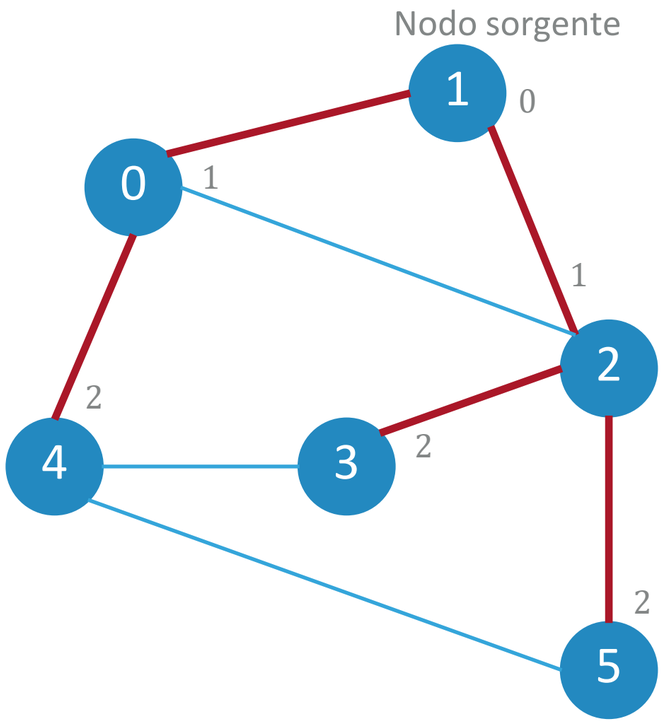
Per ogni vicino, se è bianco lo coloriamo di grigio, aggiorniamo distanza (come $distanza[u] + 1$) e precedessore (u) e lo accodiamo

Coloriamo u di nero

	0	1	2	3	4	5
Distanza	1	0	1	2	2	2

	0	1	2	3	4	5
Predecessore	1	-	1	2	0	2

COSA ABBIAMO OTTENUTO?



	0	1	2	3	4	5
Distanza	1	0	1	2	2	2

	0	1	2	3	4	5
Predecessore	1	-	1	2	0	2

In aggiunta alla distanza, **l'array dei predecessori ci permette di ottenere** un albero (non necessariamente binario) che è un sottografo del grafo di partenza, detto ***albero BFS* (*BFS tree*)**

Per ogni nodo v, l'albero ha un unico percorso dal nodo sorgente a v e questo è un percorso di lunghezza minima

PSEUDOCODICE: INIZIALIZZAZIONE

```
Parametri: grafo G, nodo sorgente s
# inizialmente impostiamo distanza, colore e predecessore di tutti i nodi
for all v ∈ V:
    colore[v] = bianco
    distanza[v] = +∞
    predecessore[v] = None
# il nodo sorgente è il primo che visitiamo e quindi
colore[s] = grigio # assume colore grigio
distanza[s] = 0 # e distanza 0 da sé stesso
Q = Coda()
# nella coda dei nodi che potrebbero avere ancora vicini da visitare
# viene aggiunto s
enqueue(Q, s)
```

PSEUDOCODICE: CICLO PRINCIPALE

```
while Q is not empty:
    # questo ciclo deve continuare finché ci rimangono dei nodi da visitare
    u = dequeue(Q) # prendiamo il primo nodo dalla coda
    for all v adiacenti a u # e ne esploriamo tutti i vicini
        if color[v] == bianco # consideriamo solo i vicini mai visitati prima
            colore[v] = grigio
            # possiamo arrivare a v con un passo partendo da u
            distanza[v] = distanza[u] + 1
            predecessore[v] = u
            enqueue(Q, v) # accodiamo perché potrebbe avere dei vicini bianchi
    colore[u] = nero # abbiamo finito di visitare tutti i vicini di u
# una volta usciti dal ciclo abbiamo visitato tutti i nodi raggiungibili da s
```

ANALISI DELLA COMPLESSITÀ

- **Assumiamo di utilizzare liste di adiacenza** per rappresentare il grafo $G = (V, E)$
- Le operazioni di *inserimento* e *rimozione* dalla coda richiedono $O(1)$
- Notiamo che ogni nodo può venire accodato al più una volta, perché deve passare da bianco a grigio prima di venire accodato e non tornerà mai più bianco, quindi **il ciclo while si esegue $\Theta(V)$ volte**

ERRATA CORRIGE

ANALISI DELLA COMPLESSITÀ

- **Il ciclo for** che itera su tutti i vicini di un nodo è trattabile in un modo un poco particolare. Invece di vedere una singola esecuzione, vediamo il numero totale di volte che viene eseguito
- Questo numero **dipende dal numero di archi del grafo** (per andare da un nodo al suo vicino ci serve un arco), quindi è **limitato da** $\Theta(E)$
- Visito ogni nodo e ogni arco una sola volta, quindi: **il tempo totale di esecuzione è quindi** $\Theta(V + E)$

Nota: per ogni vertice non si controllano tutti gli archi del grafo, ma solo la sua lista di adiacenza. La somma delle lunghezze di tutte le liste di adiacenza è proporzionale a E.

RICERCA IN PROFONDITÀ

- La ricerca in profondità (**depth-first search o DFS**) è l'altro algoritmo standard di visita dei grafi
- Dato un grafo $G = (V, E)$ e un nodo $s \in V$ detto nodo sorgente la ricerca in profondità esplora tutti i nodi raggiungibili a partire da s.
- Se rimangono nodi non esplorati si ripete la ricerca in profondità su di essi*

* questo è fattibile anche con BFS, ma generalmente BFS si usa per trovare la distanza minima da un nodo sorgente, DFS si usa per altri scopi.

RICERCA IN PROFONDITÀ

- Solitamente DFS salva due tempi:
  - Il **tempo di scoperta o inizio visita**, quando il nodo è stato visitato per la prima volta (quando *diventa grigio*)
  - Il **tempo di fine visita**, quando tutti i suoi vicini sono stati visitati (quando *diventa nero*)
  - Entrambi i tempi fanno riferimento ad un «tempo globale» dato da un **timer**. Il timer è resettato all'avvio della procedura.
- Questi due tempi sono poi utilizzati da altri algoritmi che hanno DFS come subroutine

RICERCA IN PROFONDITÀ

La ricerca in profondità può essere espressa in due modi diversi: ricorsivo e iterativo.

- **Ricorsivo** è come viene solitamente presentata, ed il caso che vedremo.
- Una versione **iterativa** può essere ottenuta sostituendo nella BFS la coda con uno stack.

RICERCA IN PROFONDITÀ: IDEA

- Dato un nodo $u \in V$
- Colora il nodo di grigio
- Se esiste, scegli un nodo bianco adiacente e richiama ricorsivamente la visita in profondità su quel nodo
- Ripeti il punto precedente finché rimangono nodi bianchi
- Colora il nodo di nero

PSEUDOCODICE: INIZIALIZZAZIONE

```
Parametri: grafo G
# inizialmente impostiamo colore e predecessore per tutti i nodi
for all v ∈ V:
    colore[v] = bianco
    predecessore[v] = None
tempo = 0  # un contatore globale per il tempo di visita dei nodi
stack = [] # stack delle chiamate ricorsive
           # (in realtà è implicito nel fatto che stiamo chiamando delle procedure in modo ricorsivo)
for all u ∈ V
    if colore[u] == bianco
        DFS-VISIT(G, u) # chiamiamo la procedura di ricorsiva visita
```

Aggiunto rispetto alla lezione

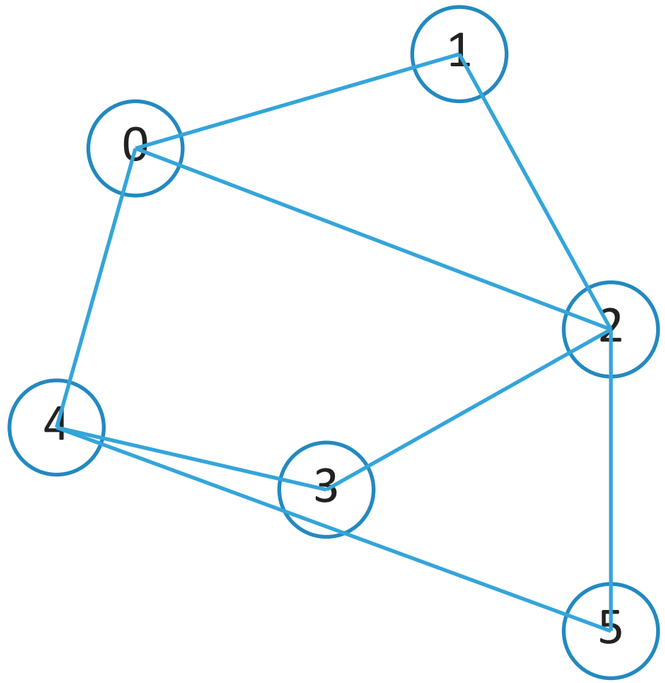
PSEUDOCODICE: PROCEDURA DFS-VISIT

```
DFS-VISIT(G, u)
Parametri: grafo G, nodo u
push(stack, u)     # aggiungiamo u nello stack
tempo = tempo + 1 # incrementiamo il tempo globale
tempo_inizio[u] = tempo
colore[u] = grigio
for all v adiacenti a u
    if colore[v] == bianco # se è la prima volta che vediamo v
        precedessore[v] = u # ci siamo arrivati da u
        DFS-VISIT(G, v) # e ricorsivamente iniziamo la procedura di visita
colore[u] = nero # arrivati qui abbiamo visitato tutti i nodi adiacenti a u
tempo = tempo + 1
tempo_fine[u] = tempo
pop(stack) # estraiamo u dallo stack: la sua visita è conclusa
```

Aggiunto rispetto alla lezione

Aggiunto rispetto alla lezione

ESEMPIO DI ESECUZIONE

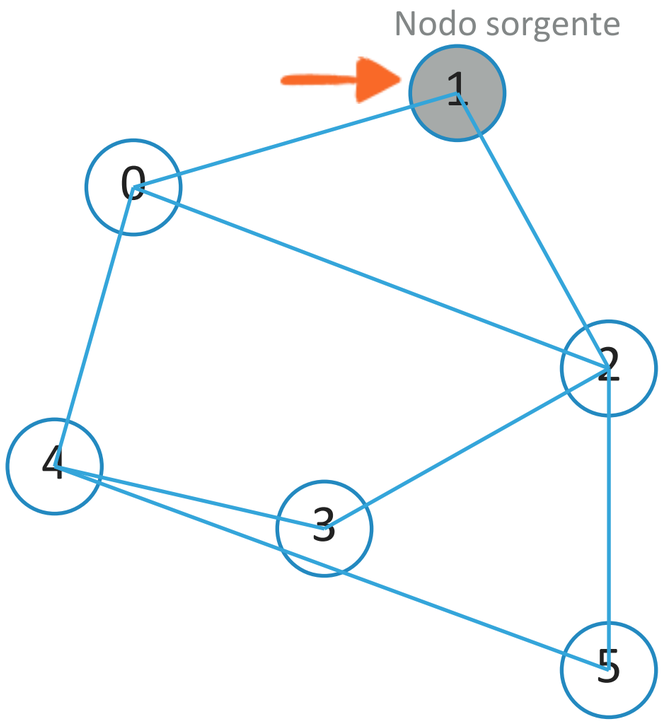


Tempo 0

ESEMPIO DI ESECUZIONE

Stack di chiamate

NODO: 1

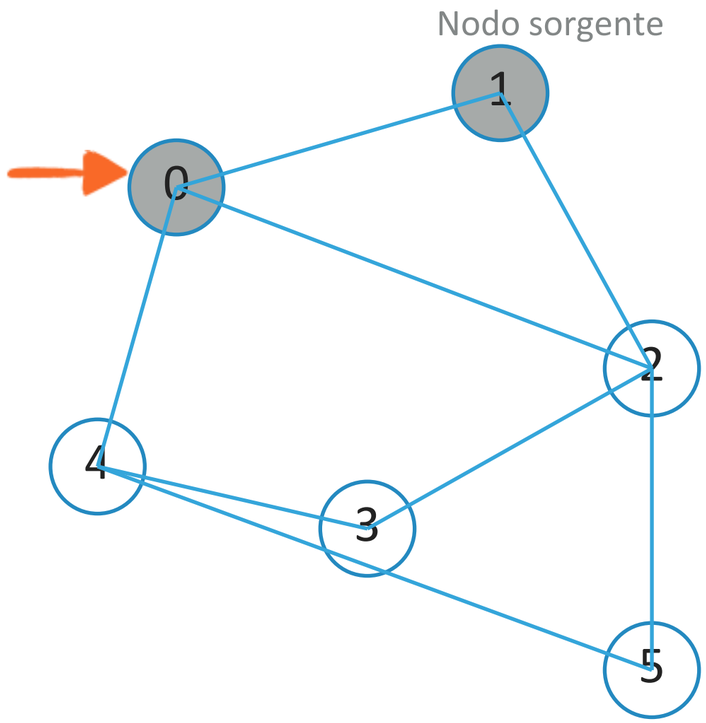


Iniziamo chiamando la procedura di visita sul primo nodo bianco che troviamo e lo coloriamo di grigio

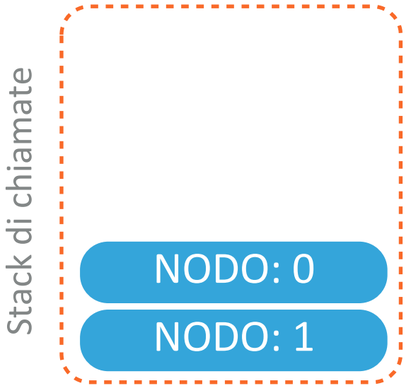
	0	1	2	3	4	5
Tempo_inizio		1				
Tempo_fine						
Predecessore	-	-	-	-	-	-

Tempo 1

ESEMPIO DI ESECUZIONE



Tempo 2



Chiamiamo la procedura di visita in profondità ricorsivamente su ciascuno dei nodi bianchi adiacenti quello corrente

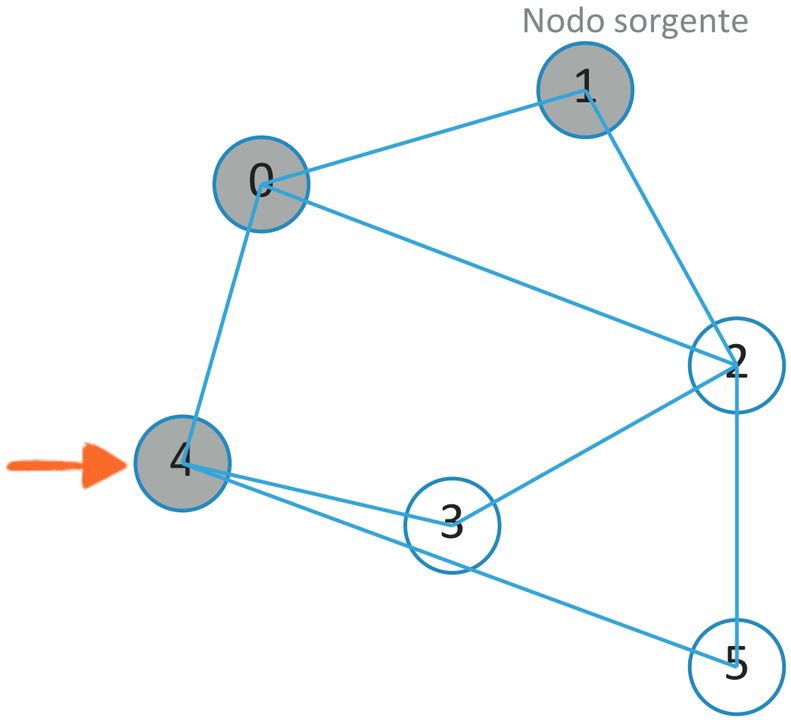
	0	1	2	3	4	5
Tempo_inizio	2	1				

	0	1	2	3	4	5
Tempo_fine						

	0	1	2	3	4	5
Predecessore	1	-	-	-	-	-

ESEMPIO DI ESECUZIONE

Stack di chiamate



Chiamiamo la procedura di visita in profondità ricorsivamente su ciascuno dei nodi bianchi adiacenti quello corrente

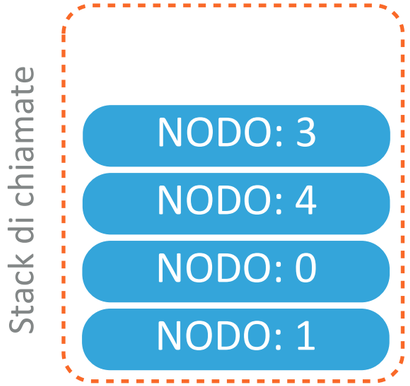
	0	1	2	3	4	5
Tempo_inizio	2	1			3	

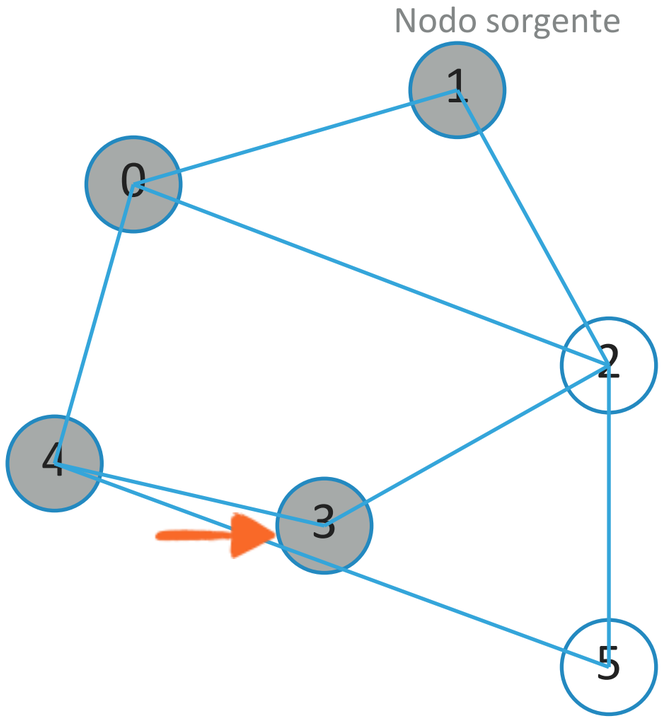
	0	1	2	3	4	5
Tempo_fine						

	0	1	2	3	4	5
Predecessore	1	-	-	-	0	-

Tempo	3

ESEMPIO DI ESECUZIONE





Chiamiamo la procedura di visita in profondità ricorsivamente su ciascuno dei nodi bianchi adiacenti quello corrente

	0	1	2	3	4	5
Tempo_inizio	2	1		4	3	

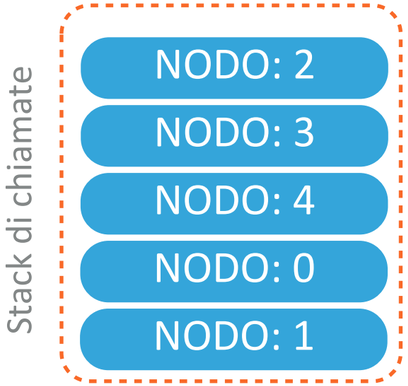
	0	1	2	3	4	5
Tempo_fine						

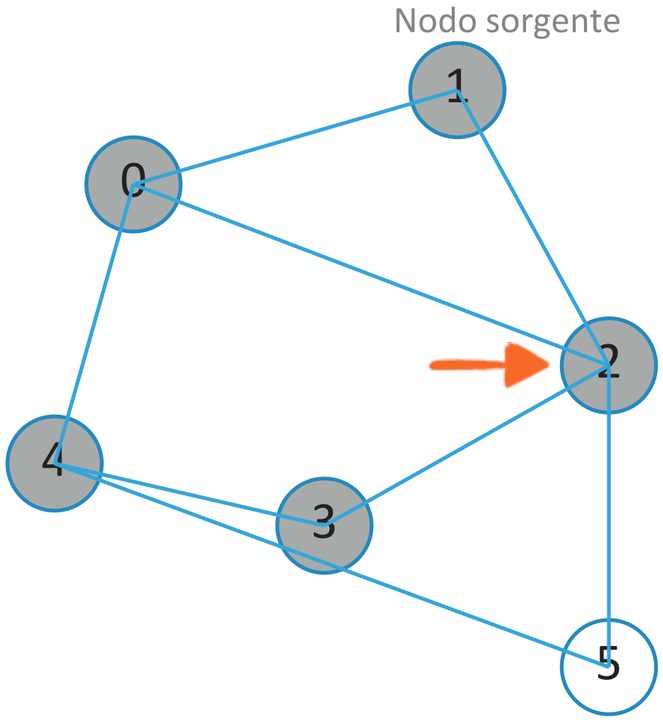
	0	1	2	3	4	5
Predecessore	1	-	-	4	0	-

Tempo	4

ESEMPIO DI ESECUZIONE

Stack di chiamate





Chiamiamo la procedura di visita in profondità ricorsivamente su ciascuno dei nodi bianchi adiacenti quello corrente

	0	1	2	3	4	5
Tempo_inizio	2	1	5	4	3	

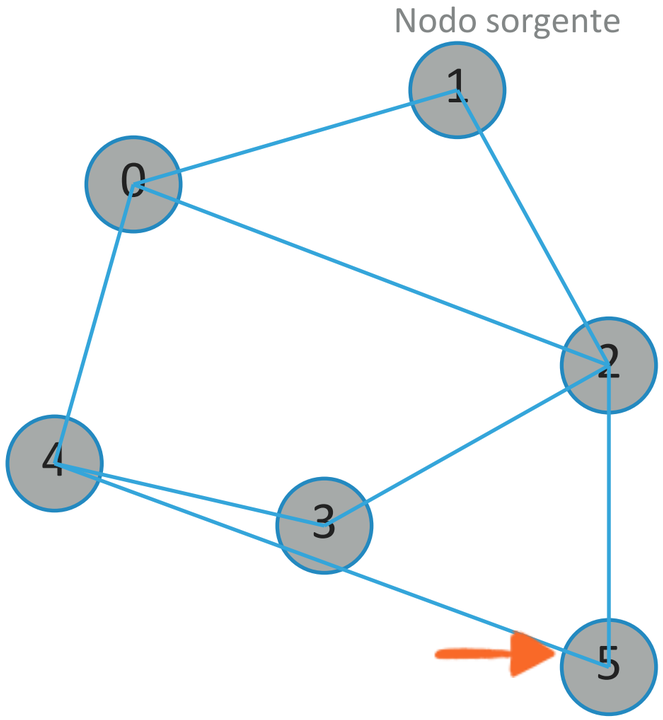
	0	1	2	3	4	5
Tempo_fine						

	0	1	2	3	4	5
Predecessore	1	-	3	4	0	-

Tempo: 5

ESEMPIO DI ESECUZIONE

Stack di chiamate



Chiamiamo la procedura di visita in profondità ricorsivamente su ciascuno dei nodi bianchi adiacenti quello corrente

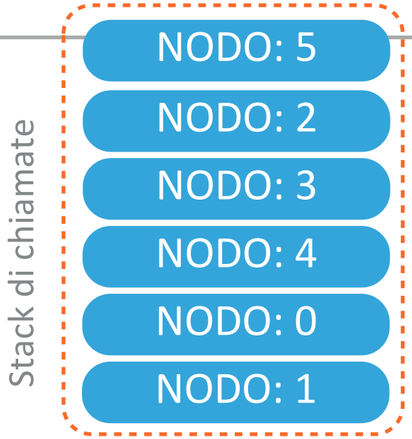
	0	1	2	3	4	5
Tempo_inizio	2	1	5	4	3	6

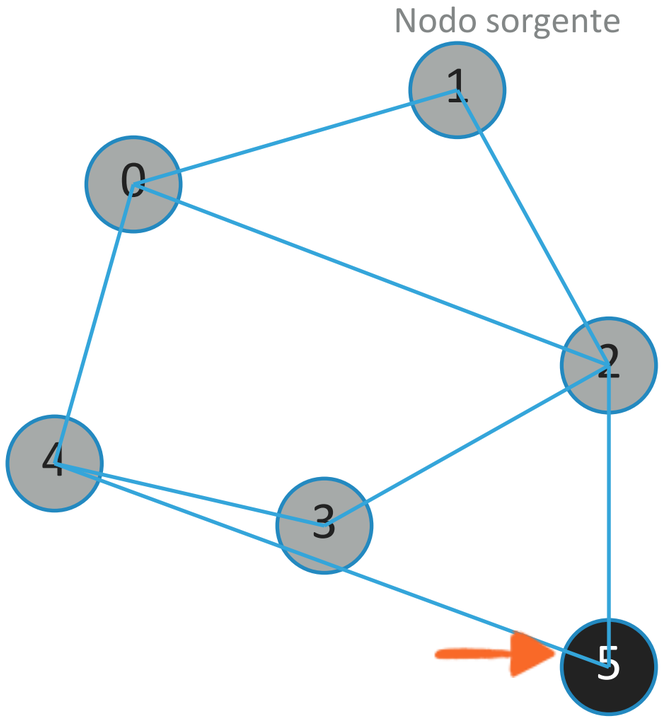
	0	1	2	3	4	5
Tempo_fine						

	0	1	2	3	4	5
Predecessore	1	-	3	4	0	2

Tempo	6

ESEMPIO DI ESECUZIONE





Dato che non possiamo più effettuare chiamate ricorsive, abbiamo finito di visitare il nodo corrente. aggiorniamo colore, tempo di fine e ritorniamo dalla chiamata ricorsiva

	0	1	2	3	4	5
Tempo_inizio	2	1	5	4	3	6

	0	1	2	3	4	5
Tempo_fine						7

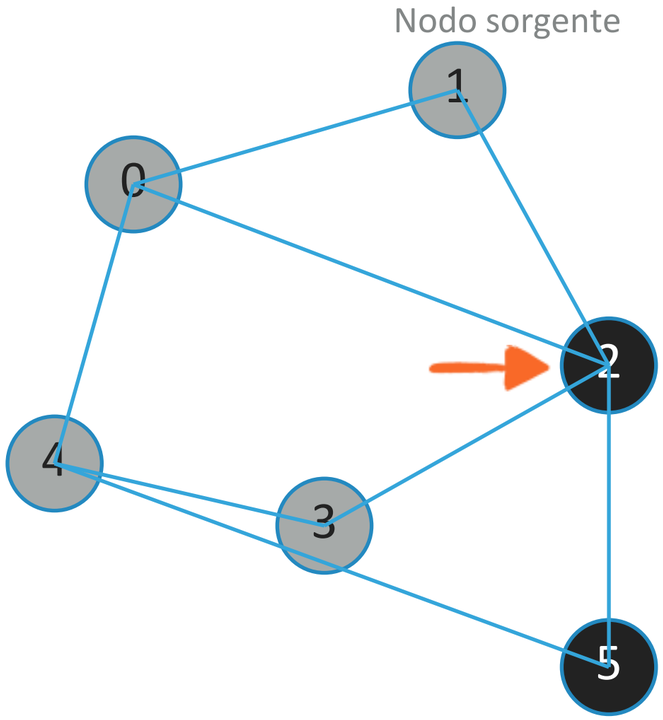
	0	1	2	3	4	5
Predecessore	1	-	3	4	0	2

Tempo	7

ESEMPIO DI ESECUZIONE

Stack di chiamate

- NODO: 2
- NODO: 3
- NODO: 4
- NODO: 0
- NODO: 1

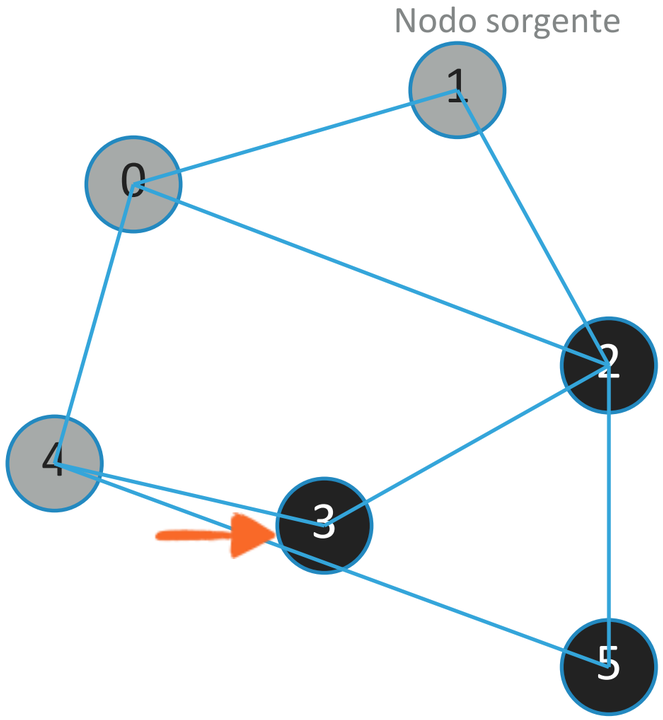


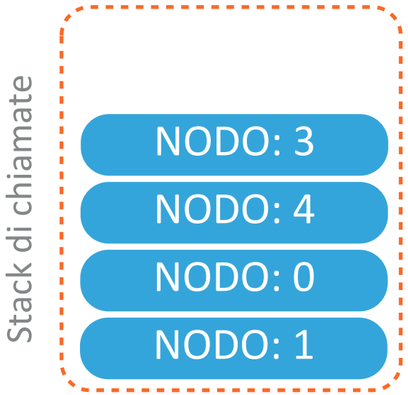
Dato che non possiamo più effettuare chiamate ricorsive, abbiamo finito di visitare il nodo corrente. aggiorniamo colore, tempo di fine e ritorniamo dalla chiamata ricorsiva

	0	1	2	3	4	5
Tempo_inizio	2	1	5	4	3	6
Tempo_fine			8			7
Predecessore	1	-	3	4	0	2

Tempo: 8

ESEMPIO DI ESECUZIONE





Dato che non possiamo più effettuare chiamate ricorsive, abbiamo finito di visitare il nodo corrente. aggiorniamo colore, tempo di fine e ritorniamo dalla chiamata ricorsiva

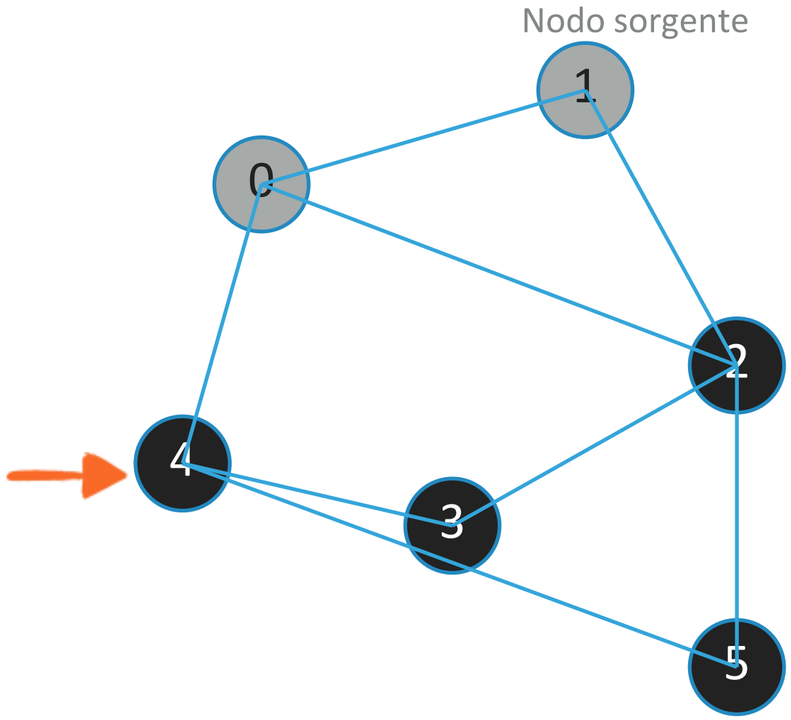
	0	1	2	3	4	5
Tempo_inizio	2	1	5	4	3	6
Tempo_fine			8	9		7
Predecessore	1	-	3	4	0	2

Tempo: 9

ESEMPIO DI ESECUZIONE

Stack di chiamate

NODO: 4
NODO: 0
NODO: 1



Dato che non possiamo più effettuare chiamate ricorsive, abbiamo finito di visitare il nodo corrente. aggiorniamo colore, tempo di fine e ritorniamo dalla chiamata ricorsiva

	0	1	2	3	4	5
Tempo_inizio	2	1	5	4	3	6
Tempo_fine			8	9	10	7
Predecessore	1	-	3	4	0	2

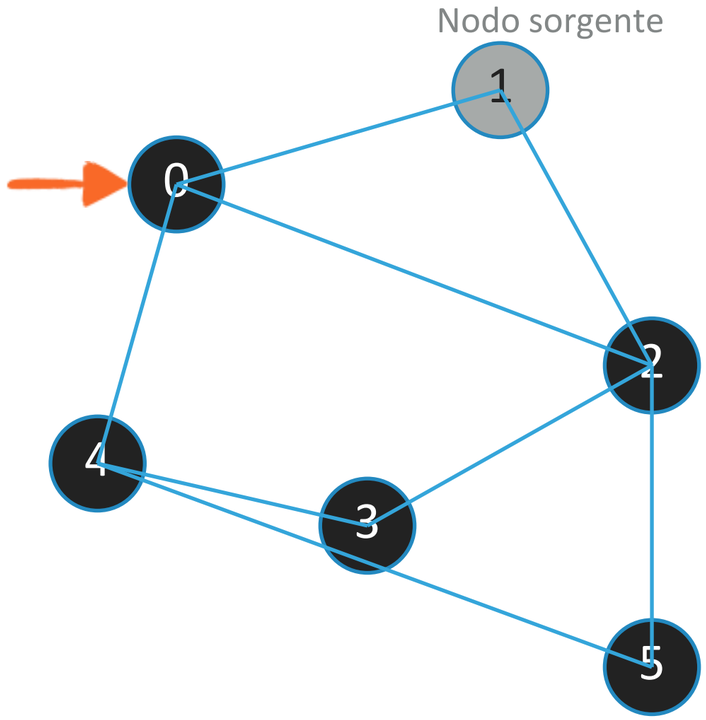
Tempo 10

ESEMPIO DI ESECUZIONE

Stack di chiamate

NODO: 0

NODO: 1



Dato che non possiamo più effettuare chiamate ricorsive, abbiamo finito di visitare il nodo corrente. aggiorniamo colore, tempo di fine e ritorniamo dalla chiamata ricorsiva

	0	1	2	3	4	5
Tempo_inizio	2	1	5	4	3	6

	0	1	2	3	4	5
Tempo_fine	11		8	9	10	7

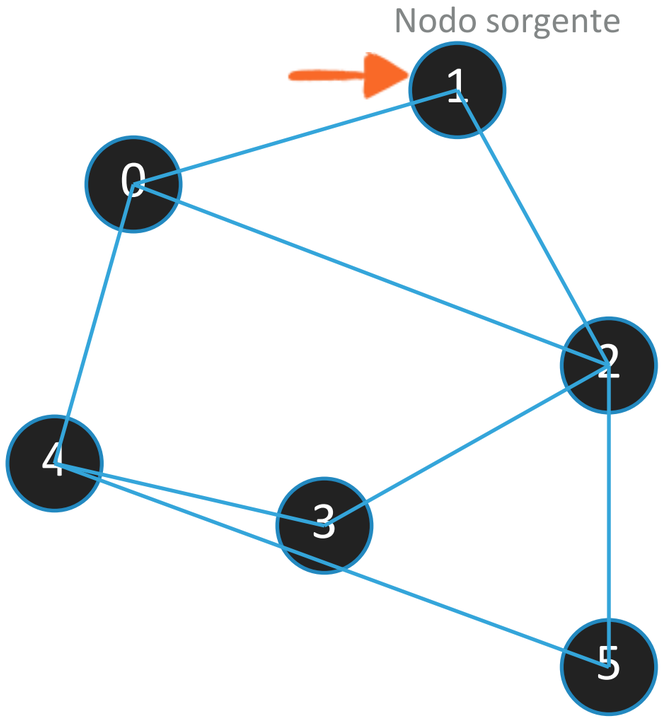
	0	1	2	3	4	5
Predecessore	1	-	3	4	0	2

Tempo	11

ESEMPIO DI ESECUZIONE

Stack di chiamate

NODO: 1



Dato che non possiamo più effettuare chiamate ricorsive, abbiamo finito di visitare il nodo corrente. aggiorniamo colore, tempo di fine e ritorniamo dalla chiamata ricorsiva

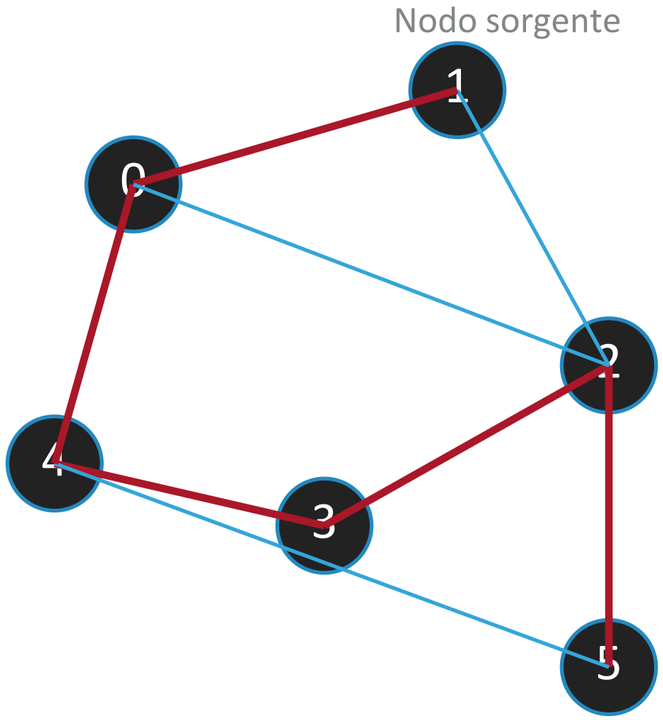
	0	1	2	3	4	5
Tempo_inizio	2	1	5	4	3	6

	0	1	2	3	4	5
Tempo_fine	11	12	8	9	10	7

	0	1	2	3	4	5
Predecessore	1	-	3	4	0	2

Tempo	12

RISULTATI



Abbiamo ottenuto un albero DFS. Notate come sia diverso dall'albero BFS e come i percorsi su di esso non rappresentino, in generale, il percorso di lunghezza minima dal nodo di partenza

	0	1	2	3	4	5
Tempo_inizio	2	1	5	4	3	6

	0	1	2	3	4	5
Tempo_fine	11	12	8	9	10	7

	0	1	2	3	4	5
Predecessore	1	-	3	4	0	2

ANALISI DELLA COMPLESSITÀ

- Notiamo che DFS-VISIT viene chiamata al più una volta per ogni nodo, dato che la chiamata avviene solo se il nodo viene visitato per la prima volta (era bianco) e viene immediatamente ricolorato di grigio, quindi $\Theta(V)$ chiamate a DFS-VISIT
- Su **tutte** le chiamate a DFS-VISIT, il ciclo che itera sui nodi adiacenti viene eseguito in totale $\Theta(E)$ volte
- Otteniamo quindi una complessità di $\Theta(V + E)$

Nota. Per ogni nodo visitato, si devono controllare tutti i **suoi** archi (per vedere che i vicini non siano bianchi), *non tutti gli archi del grafo*. Per ogni vertice u, quando lo si visita, si scorre solo la sua lista di adiacenza, cioè solo gli archi che partono da u. Quindi ogni arco viene esaminato una volta. Il totale numero di archi è E (o 2E per grafo non orientato).

AMPIEZZA VS PROFONDITÀ

- Gli algoritmi di ricerca in ampiezza e profondità si possono applicare *sia quando i grafi sono orientati sia quando sono non orientati.*
- **Se i grafi sono pesati, per questi algoritmi i pesi vengono ignorati**. Andrà valutato se sia meglio usare altri algoritmi (che vedremo in seguito)
- La scelta di una o dell'altra tipologia di visita dipende dalle specifiche dell'algoritmo
- Per trovare il **percorso di lunghezza minima** si usa BFS. Ma ATTENZIONE: se il grafo è pesato (e i pesi sono diversi), allora BFS non restituisce il percorso minimo. Vanno usati altri algoritmi (che vedremo in seguito)
- DFS è più utile su grafi orientati e non pesati, quando interessa la struttura delle dipendenze, dei cicli e dell'ordinamento.
- Notate come in BFS la coda sia esplicita, mentre in DFS lo stack è implicitamente fornito dalle chiamate ricorsive (anche se nello pseudocodice è stato aggiunto esplicitamente)

Materiale per la lezione

- Cormen et al. **CAP. 20**

*Prossima lezione: 30 aprile**, h.14:00**, aula 4C*