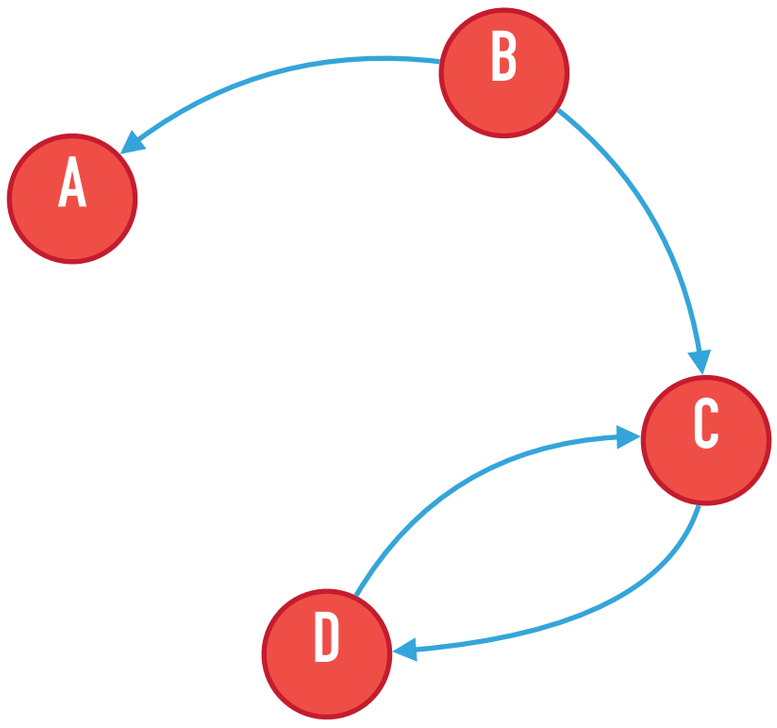
GRAFI
RICERCA IN AMPIEZZA

INFORMATICA

COSA È UN GRAFO?



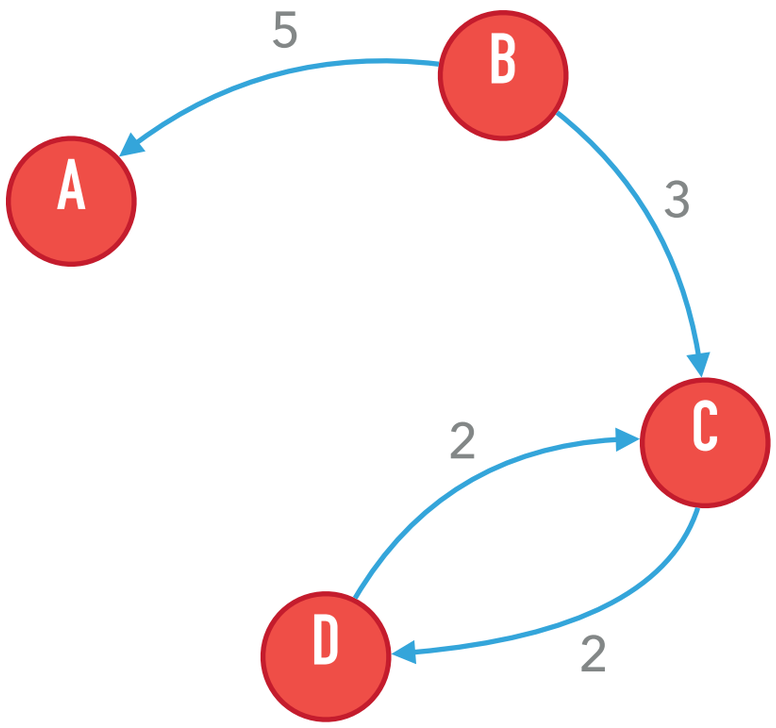
Insieme di nodi:

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

Insieme di archi:

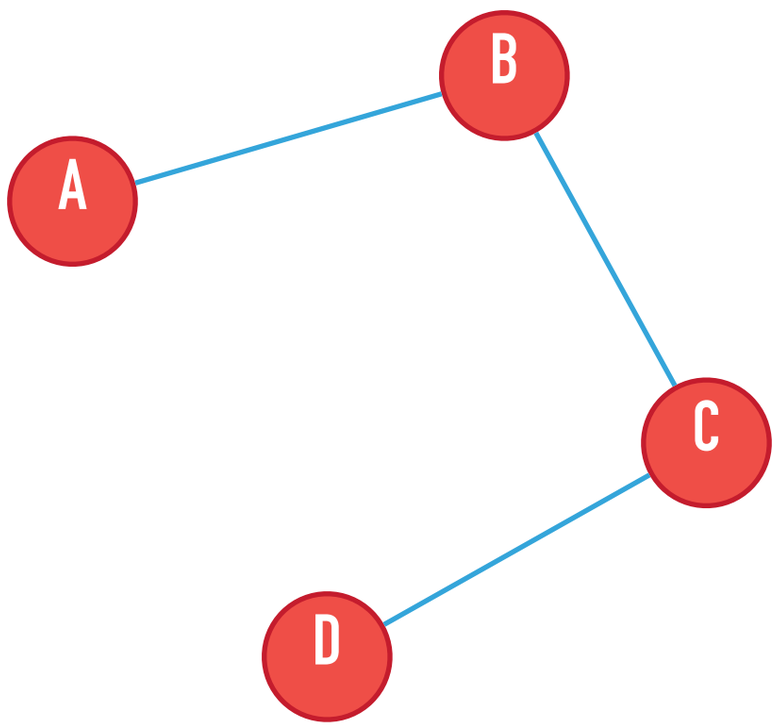
$$E = \{(b, a), (b, c), (c, d), (d, c)\}$$

COSA È UN GRAFO (VARIANTI)?



Archi pesati

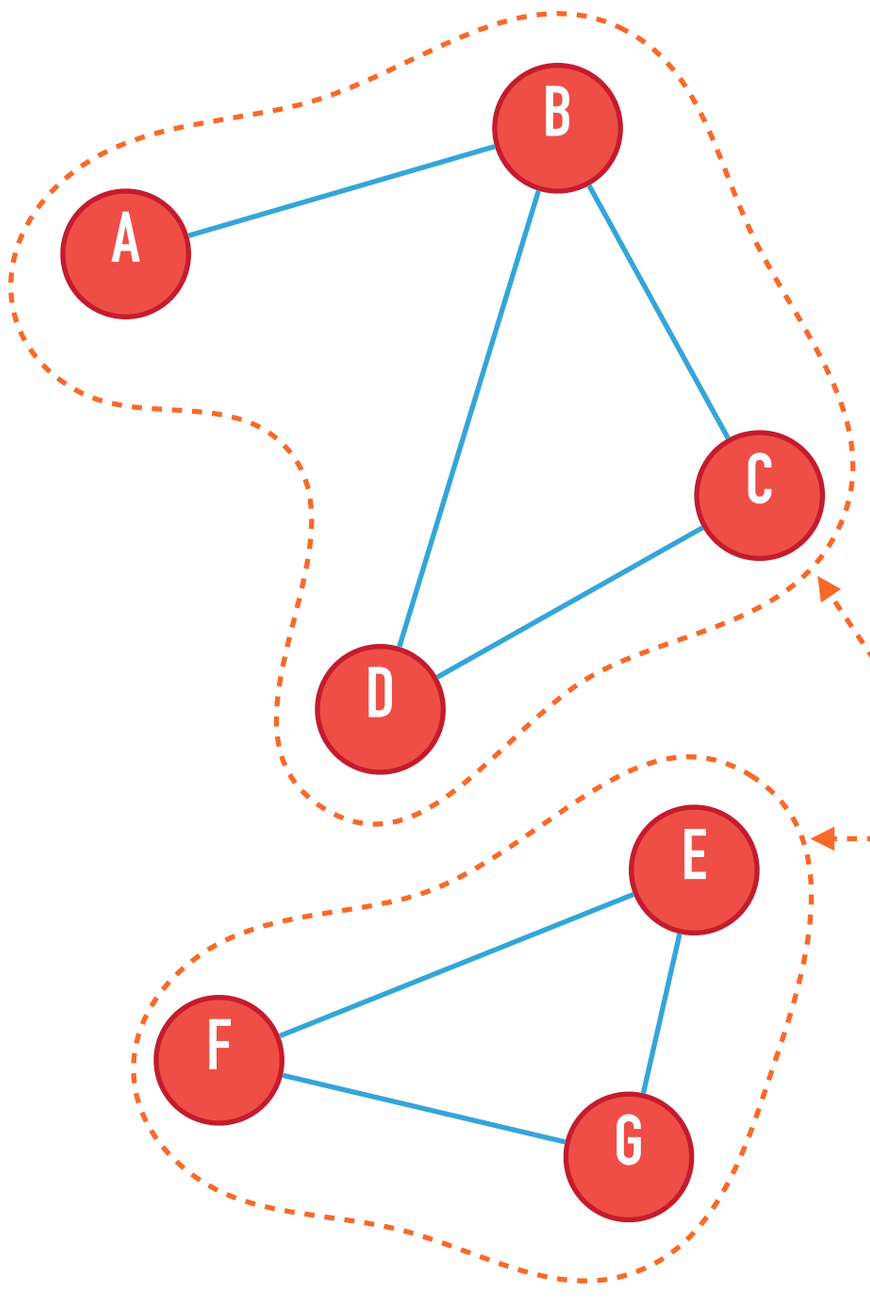
Ovvero esiste una funzione $w : E \to \mathbb{R}$
e indichiamo $w((i, j))$ come $w_{i,j}$



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 (i.e., 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 al 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 viene misurato in modi dipendente da $|V|$ e $|E|$, quindi due parametri e non uno
- 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 $O(|V| + |E|)$

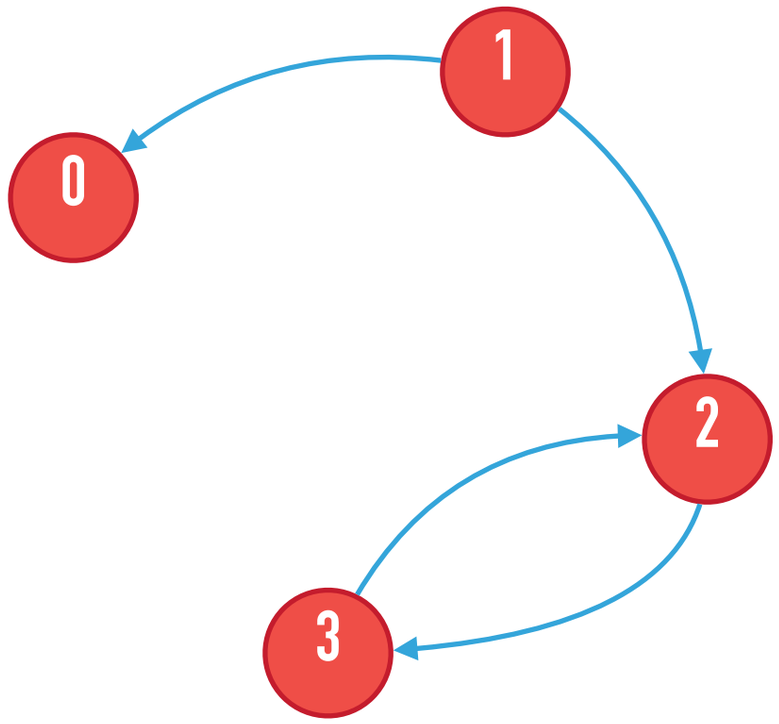
NOTAZIONE

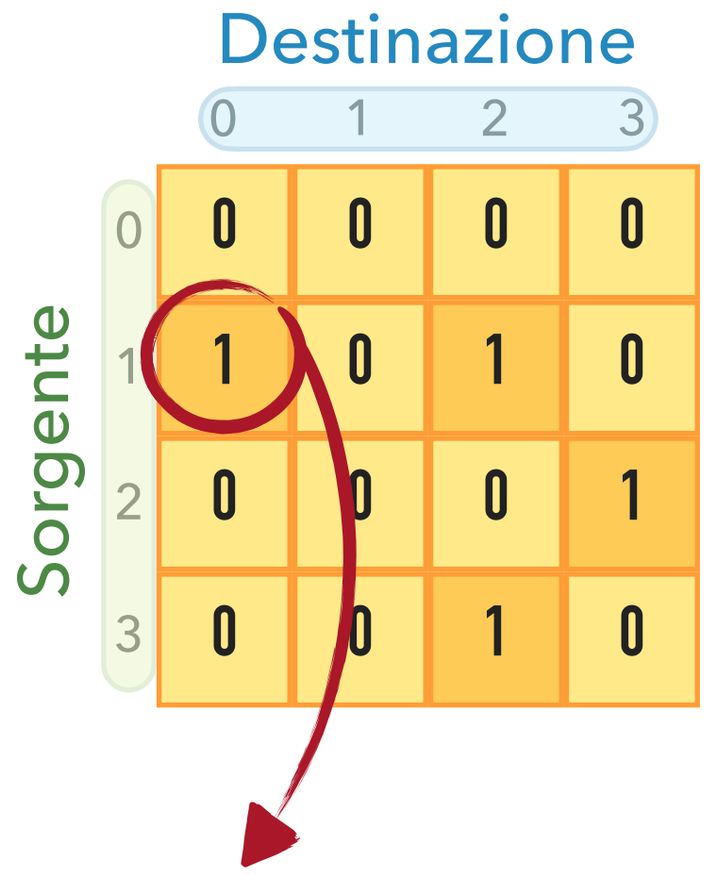
- Un grafo può avere al più $O(V^2)$ archi, con il valore esatto che dipende dal fatto che il grafo sia non diretto o diretto
- Solitamente un grafo con un numero di archi vicino al massimo possibile viene detto **denso** mentre uno con un pochi archi viene detto **sparso**.
- Cosa effettivamente sia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

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 (i.e., non cambi nel tempo)

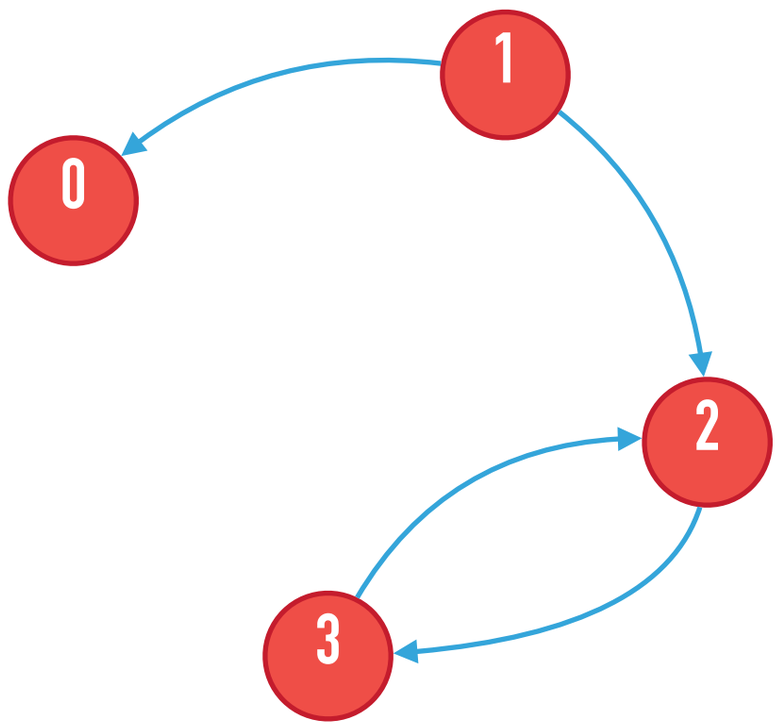
MATRICI DI ADIACENZA

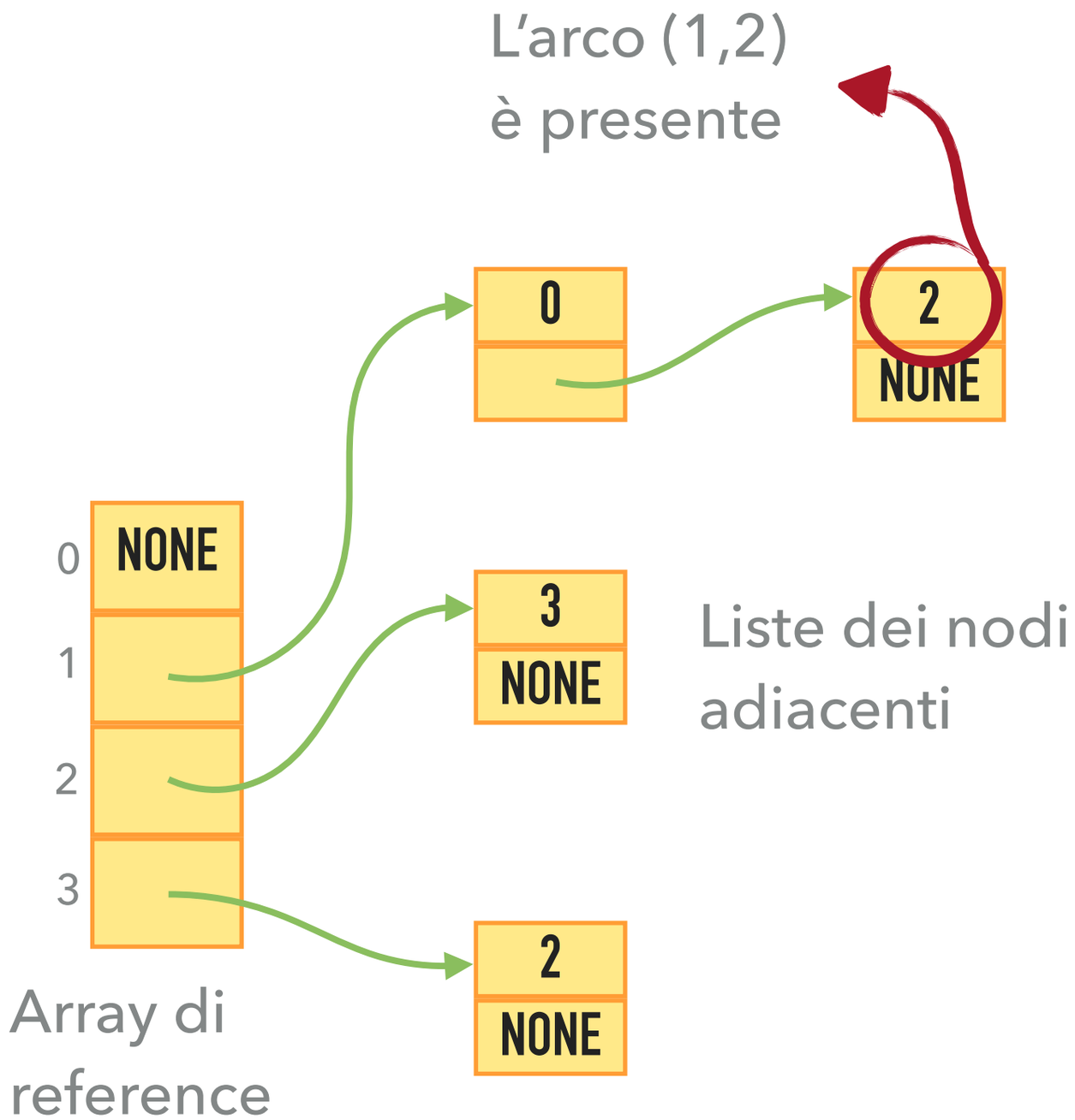




L'arco (1,0) è presente

LISTE DI ADIACENZA





QUALE RAPPRESENTAZIONE USARE

Matrici di adiacenza

Veloce (tempo costante)
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)
stabilire se un arco esiste

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

Funziona bene per grafi sparsi

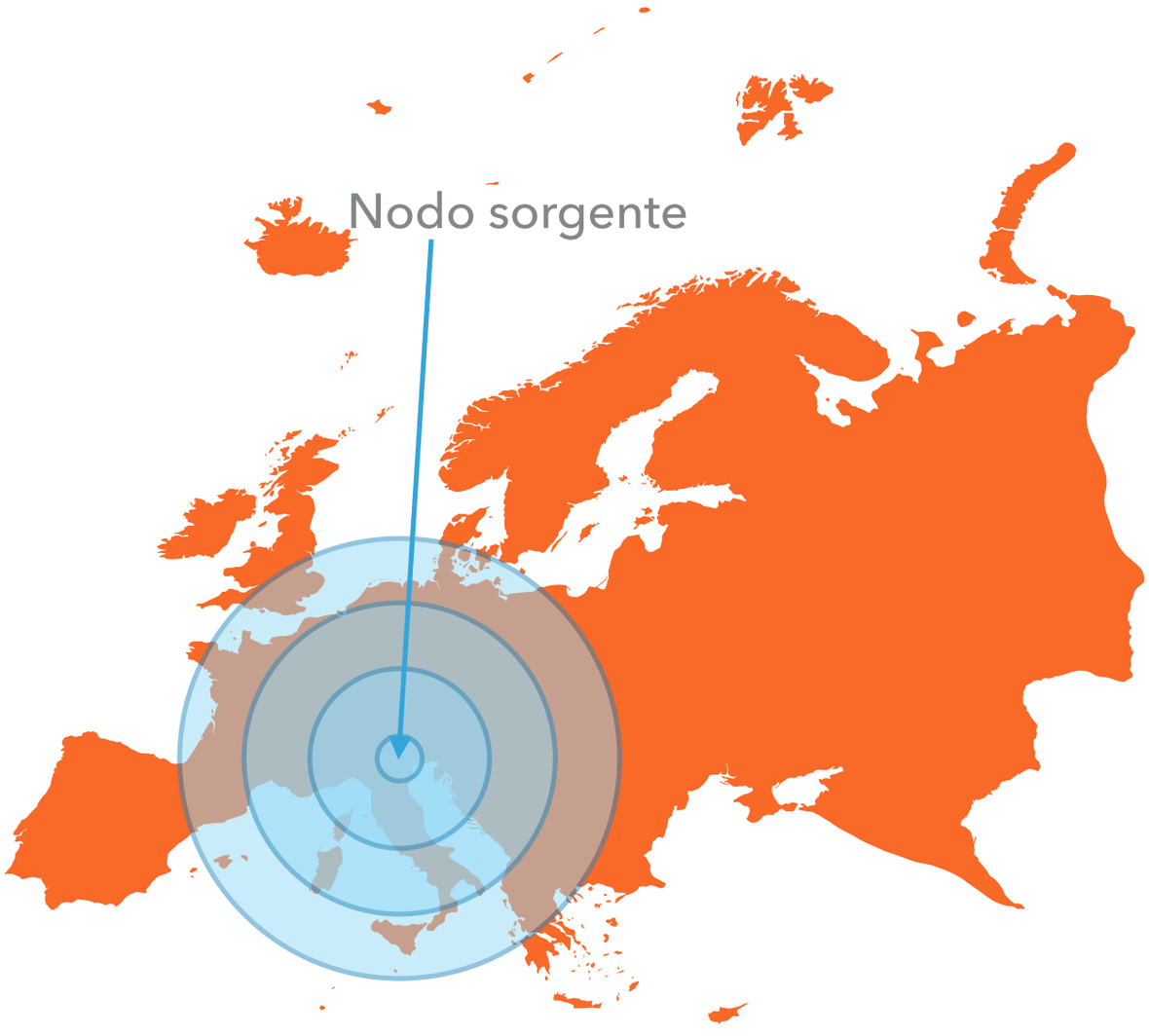
RICERCA IN AMPIEZZA

- 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), etc.
- Quindi prima di allontanarci dal nodo sorgente esploriamo tutti quelli vicini, poi i lori vicini, etc.

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

IDEA DELL'ALGORITMO

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

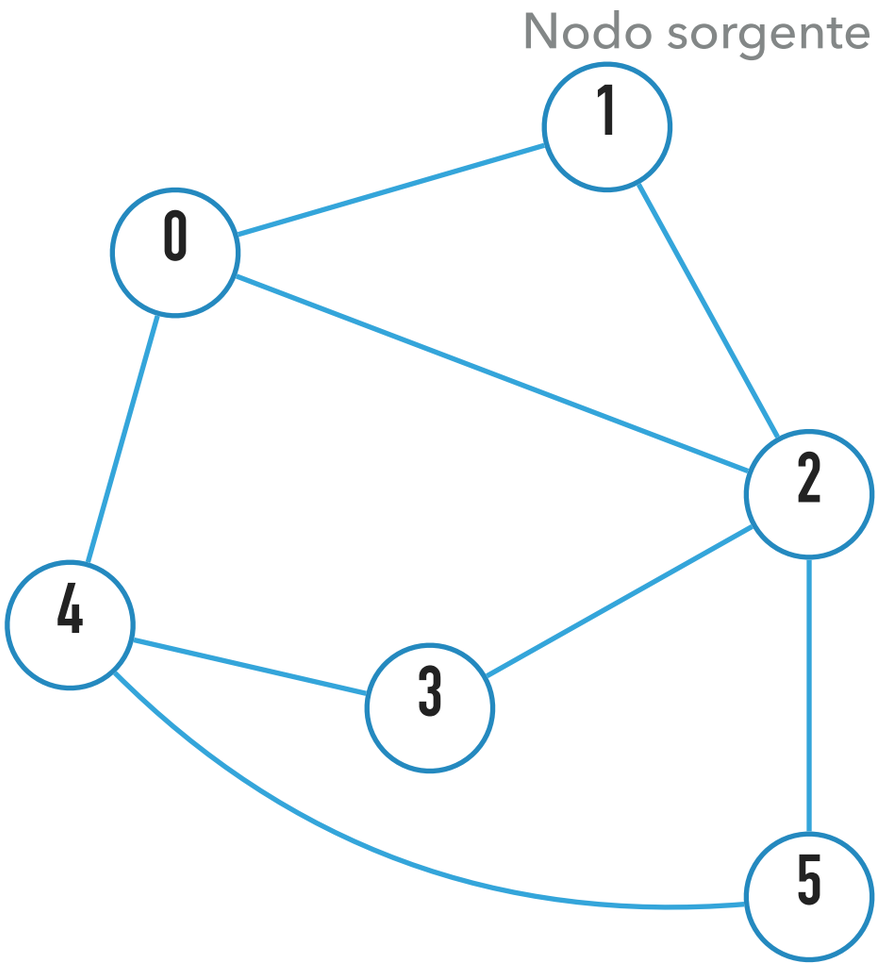
PSEUDOCODICE: INIZIALIZZAZIONE

```
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)
```

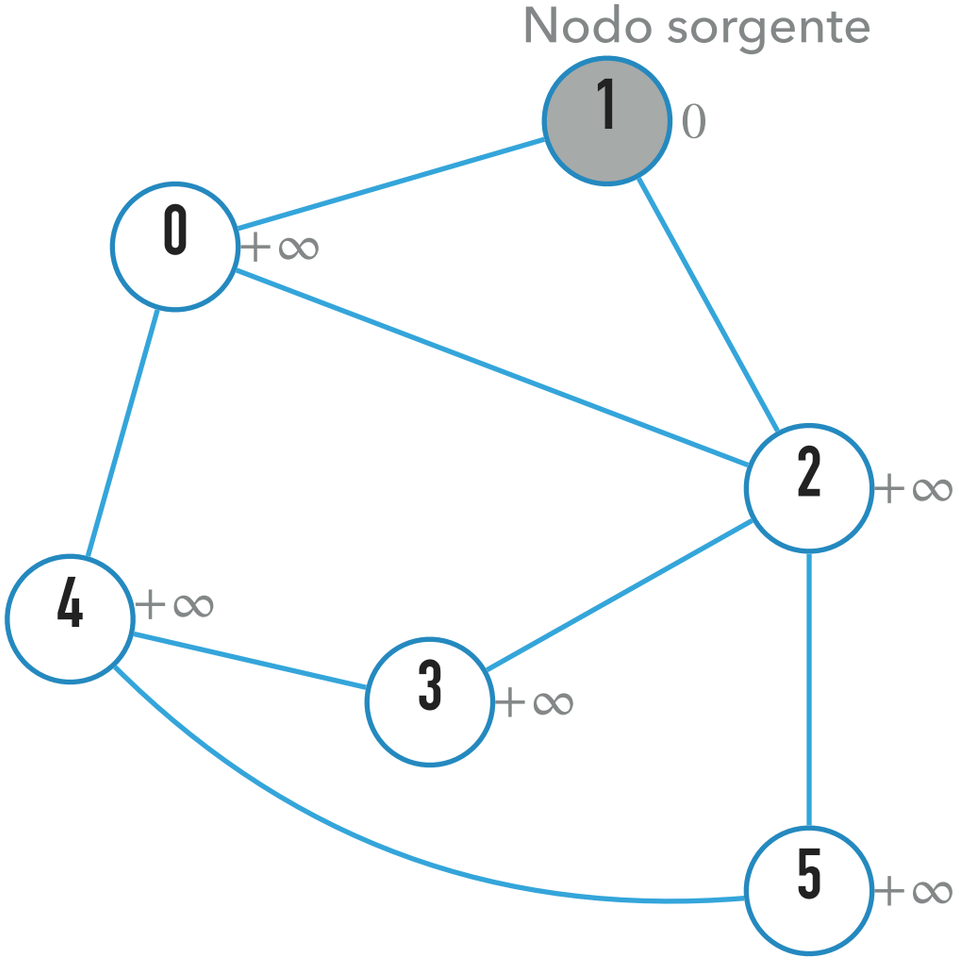
PSEUDOCODICE: CICLO PRINCIPALE

```
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
```

ESEMPIO DI ESECUZIONE



ESEMPIO DI ESECUZIONE



Coda dei nodi da visitare

1

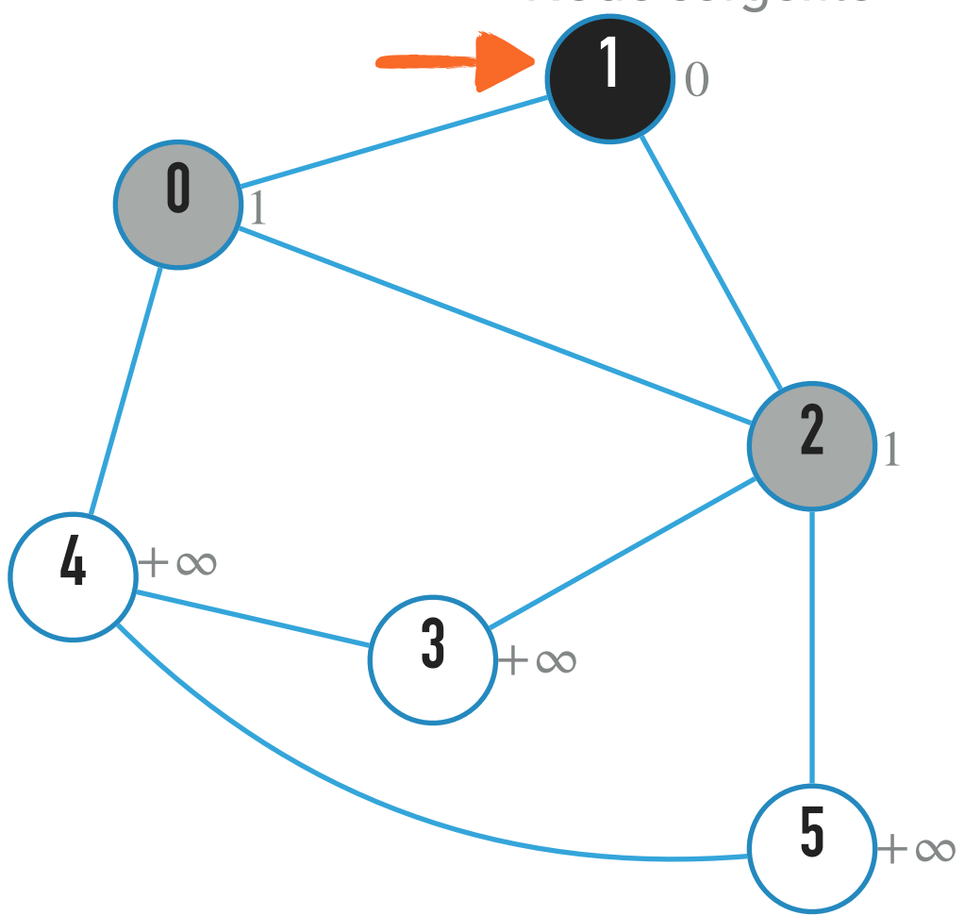
Inizialmente la distanza conosciuta
di tutti i nodi dal nodo di partenza è $+\infty$

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

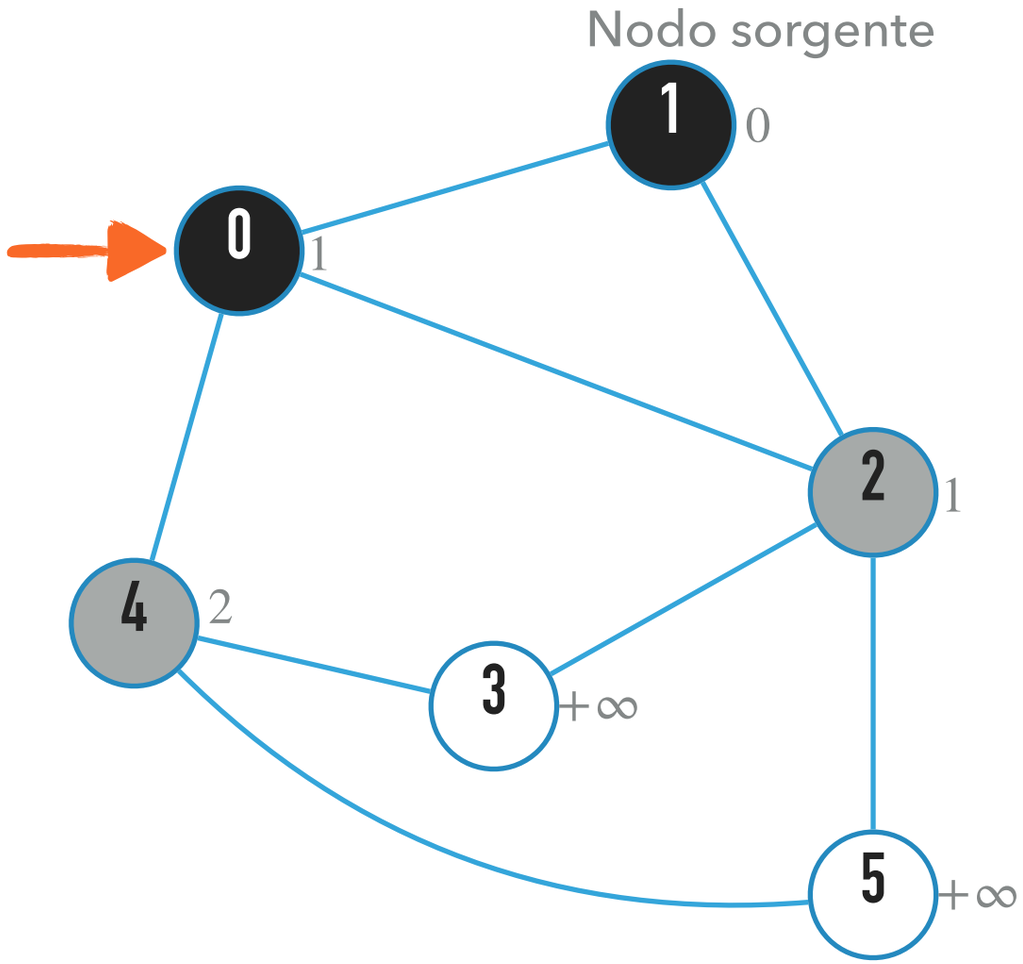
Per ogni vicino, se è bianco lo coloriamo di grigio, aggiorniamo distanza (come distanza$[u] + 1$) e precedessore (u) e lo accodiamo

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

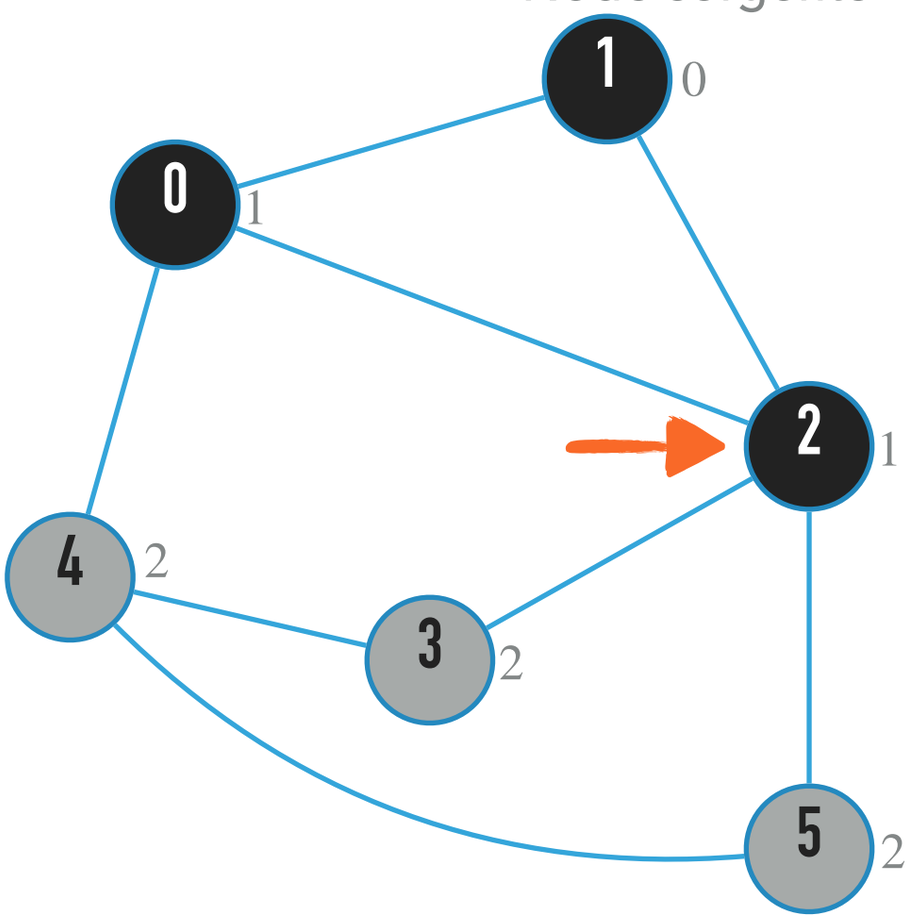
Per ogni vicino, se è bianco lo coloriamo di grigio, aggiorniamo distanza (come distanza$[u] + 1$) e precedessore (u) e lo accodiamo

Coloriamo u di nero

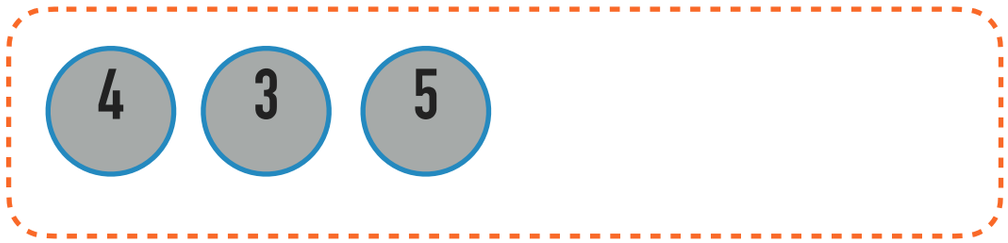
	0	1	2	3	4	5
Distanza	1	0	1	∞	2	∞

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

ESEMPIO DI ESECUZIONE



Coda dei nodi da visitare



Estraiamo il nodo u dalla coda

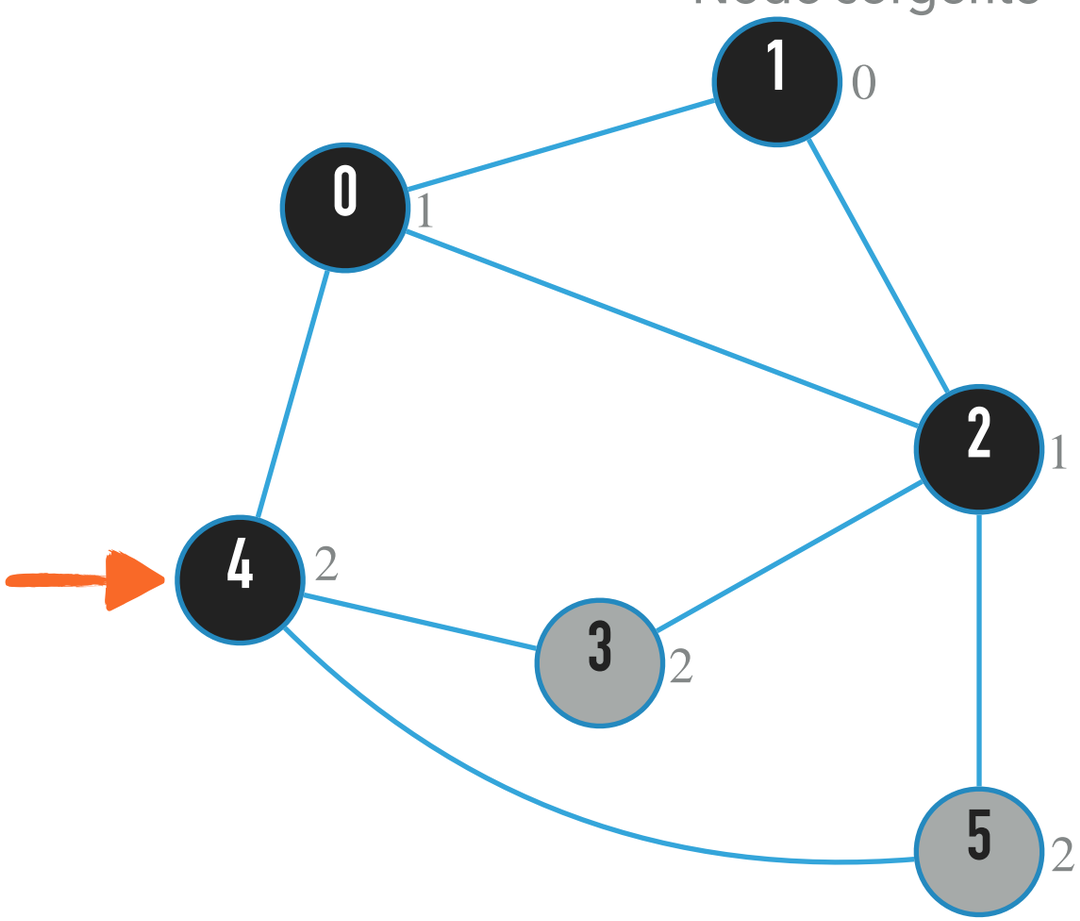
Per ogni vicino, se è bianco lo coloriamo di grigio, aggiorniamo distanza (come distanza$[u] + 1$) e precedessore (u) e lo accodiamo

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

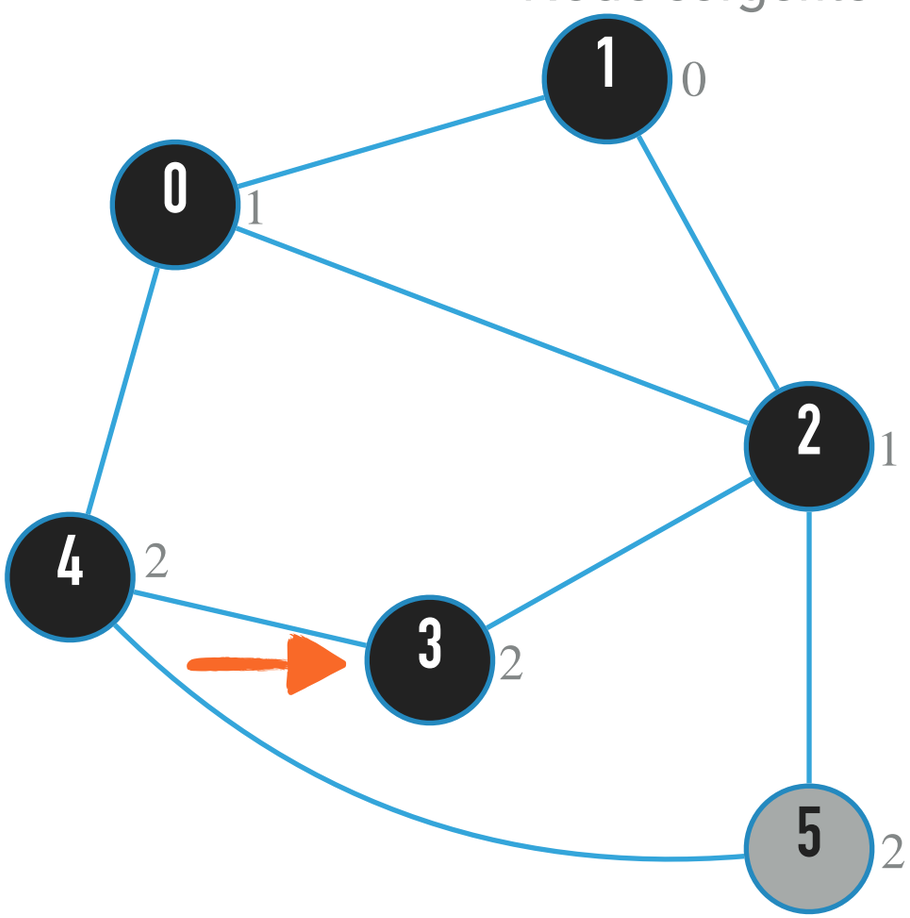
Per ogni vicino, se è bianco lo coloriamo di grigio, aggiorniamo distanza (come distanza$[u] + 1$) e precedessore (u) e lo accodiamo

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

5

Estraiamo il nodo u dalla coda

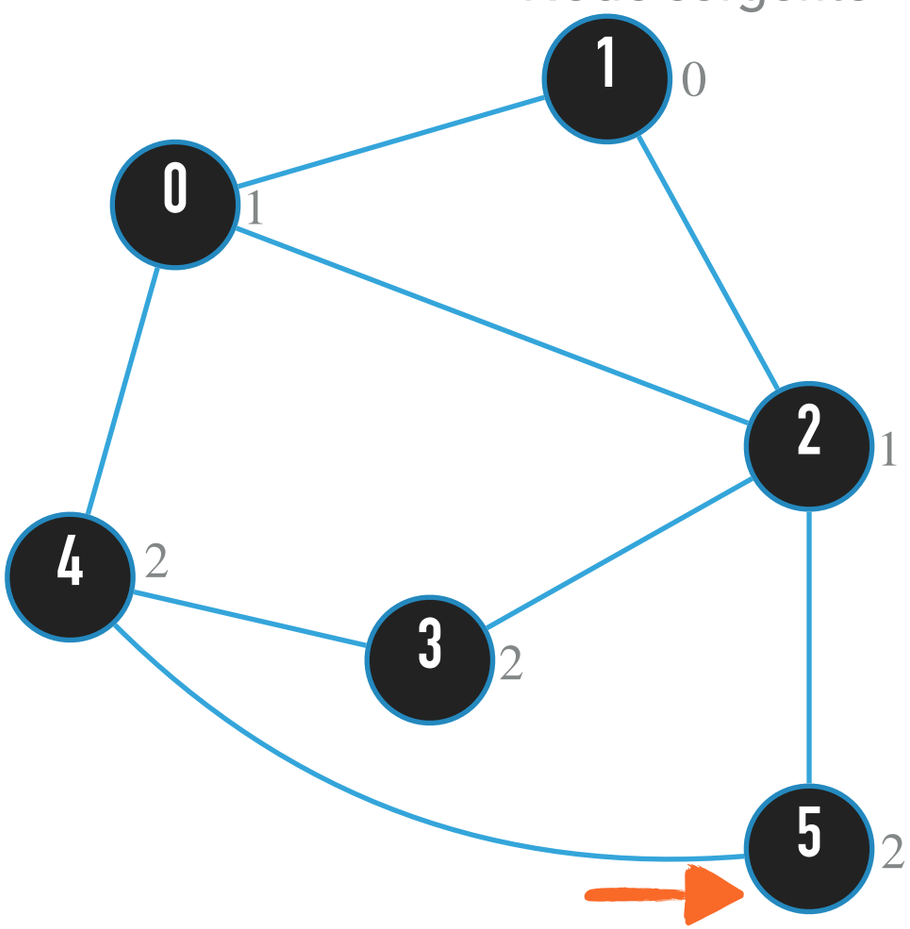
Per ogni vicino, se è bianco lo coloriamo di grigio, aggiorniamo distanza (come distanza$[u] + 1$) e precedessore (u) e lo accodiamo

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

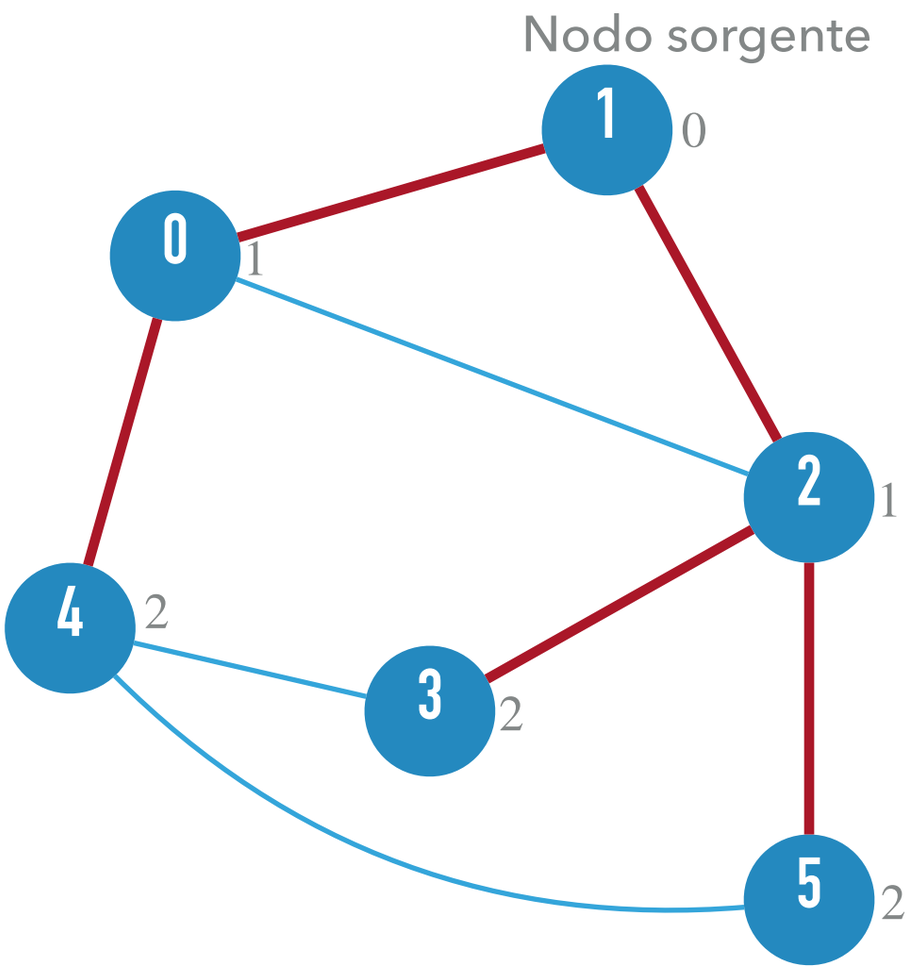
Per ogni vicino, se è bianco lo coloriamo di grigio, aggiorniamo distanza (come distanza$[u] + 1$) e precedessore (u) e lo accodiamo

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

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 esegue $O(V)$ volte

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 $O(E)$
- Il tempo totale di esecuzione è quindi $O(V + E)$