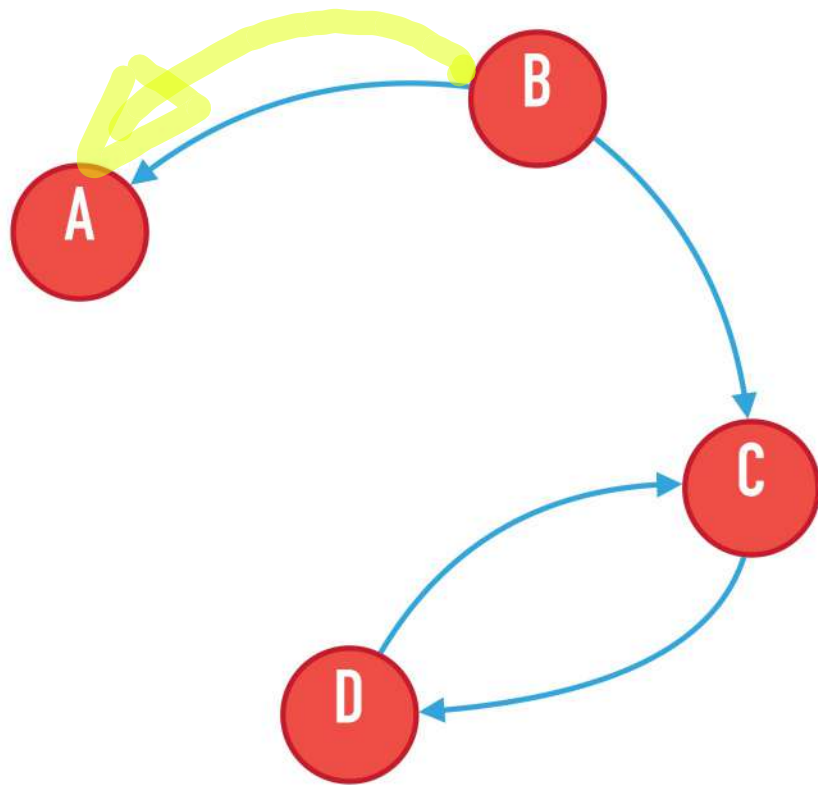


GRAFI
RICERCA IN AMPIEZZA

INFORMATICA

COSA È UN GRAFO?



Insieme di nodi:

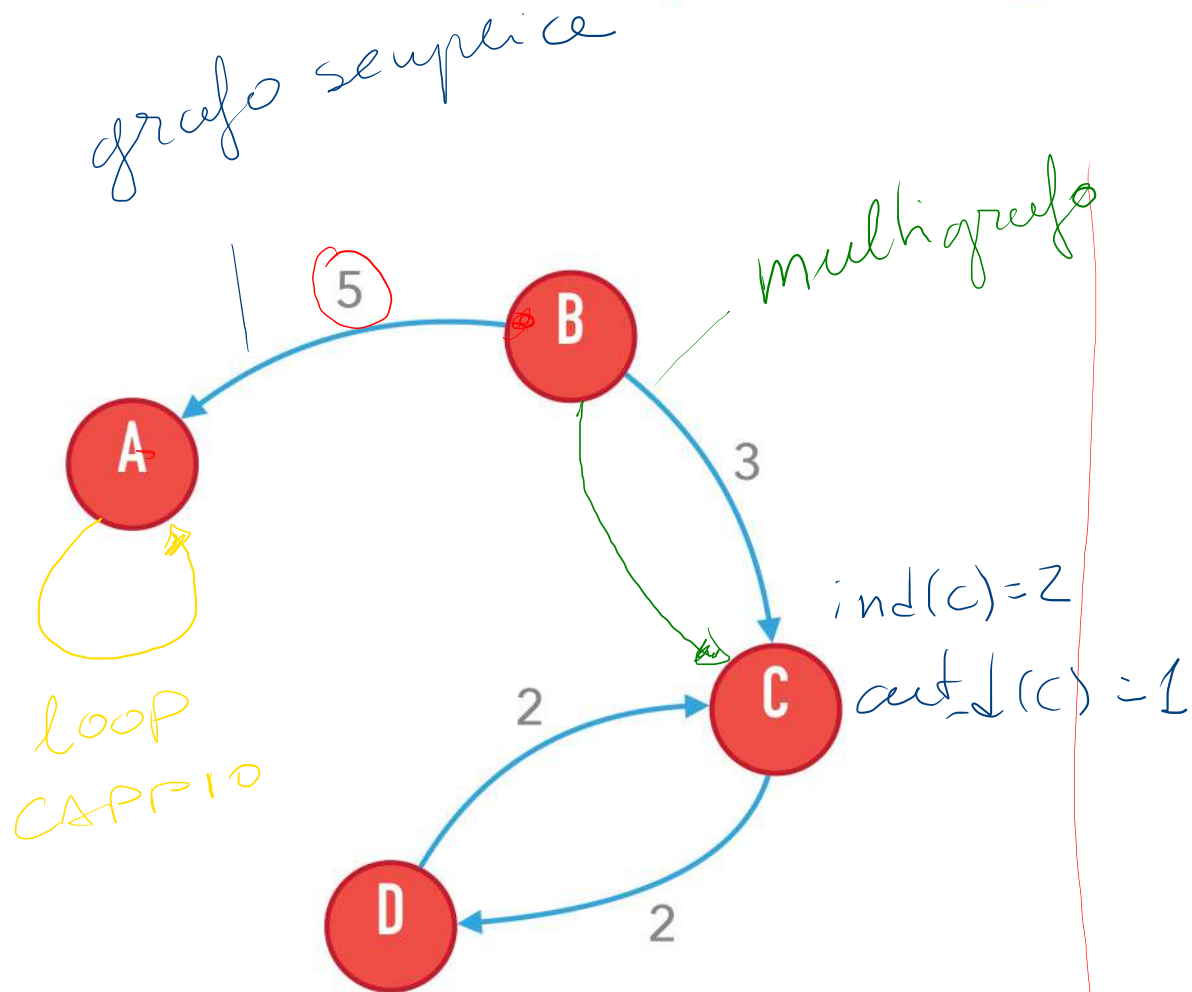
$$V = \{a, b, c, d\}$$

Insieme di archi:

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

Handwritten annotations: A red arrow labeled "SOURCE" points to the first element of the pair (b, a). A red arrow labeled "TARGET" points to the second element of the pair (b, a). The entire set E is underlined in yellow.

COSA È UN GRAFO (VARIANTI)?



$in\ degree(v) = \# \text{ nodi in ingresso}$

$out\ degree(v) = \# \text{ nodi in uscita}$

Archi pesati

Ovvero esiste una funzione $w : E \rightarrow \mathbb{R}$

e indichiamo $w((i, j))$ come $w_{i,j}$

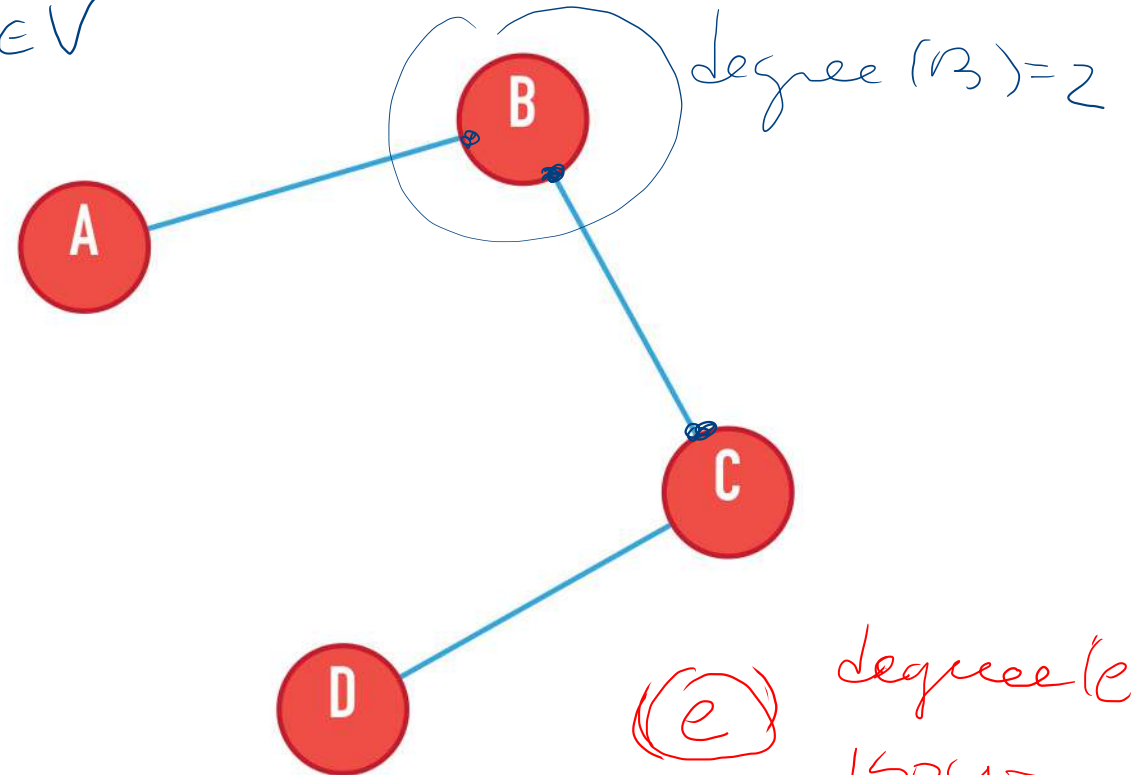
IPERGRAFI $(E \subseteq \mathcal{P}(V) \setminus \emptyset, V^* = V \cup V^2 \cup V^3 \cup \dots \cup V^k \cup \dots)$

$GRADO(v)$ degree

"

ARCHI INCIDENTI IN V

$$\sum_{v \in V} degree(v) = 2|E|$$



Non orientato

Ovvero $(a, b) \in E \iff (b, a) \in E$

per ogni $a, b \in V$

Se un grafo non ha cicli \Leftrightarrow è detto
ACICLICO

GRAFI: NOZIONI DI BASE

$$\pi, \gamma : \langle v_0, v_1, \dots, v_k \rangle$$

$$(v_i, v_{i+1}) \in E$$

PATH

\rightarrow CAMMINO α

(Percorso): sequenza di archi $(v_0, v_1), (v_1, v_2), \dots$
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 $+\infty$

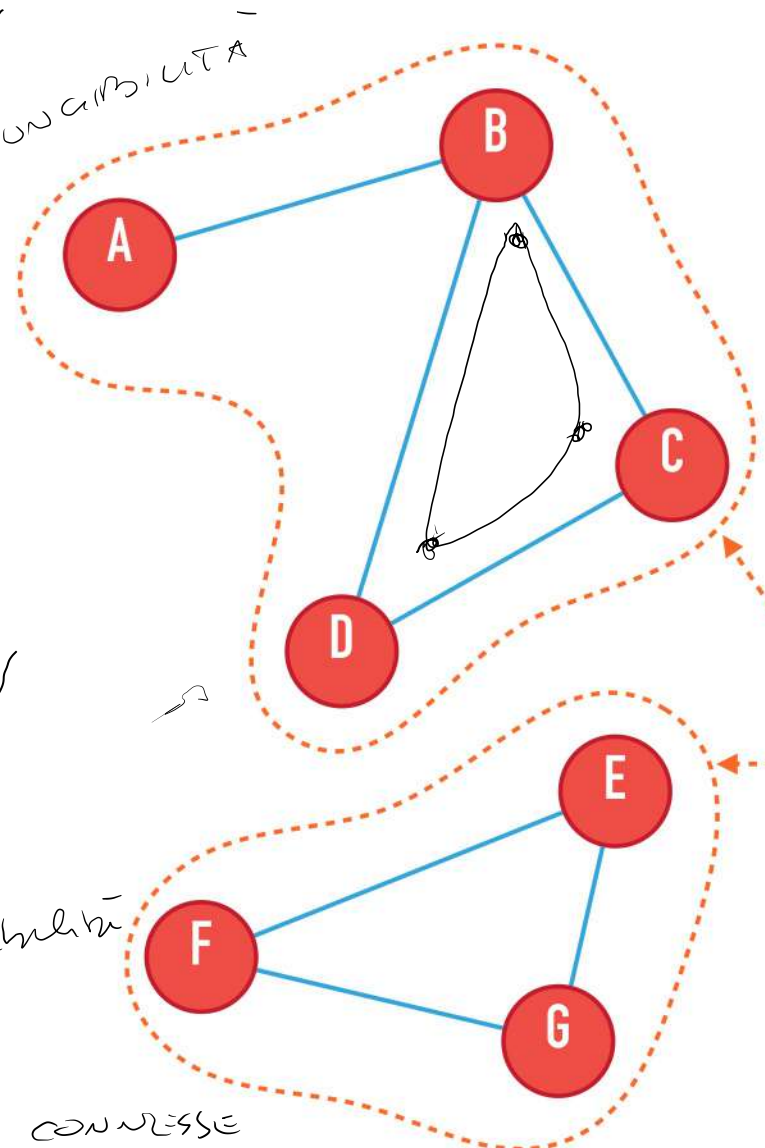
Componenti connesse

$$\langle v_0, \dots, v_k \rangle$$

$$\langle v_i, \dots, v_j \rangle \text{ sottocammino}$$

• CAMMINO SEMPLICE: ogni vertice è distinto dagli altri

• CICLO: $\langle v_0, v_1, \dots, v_k, v_0 \rangle$, $v_0 \neq v_k$ distinti
lunghezza k



$$u \rightsquigarrow v$$

$$\pi: u \rightsquigarrow v$$

RAGGIUNGIBILITÀ

$$\exists \pi: u \rightsquigarrow v$$

$$u \rightsquigarrow v$$

RAGGIUNGIBILITÀ

simmetrica in non oriented graphs.

relazioni di equivalenza su V

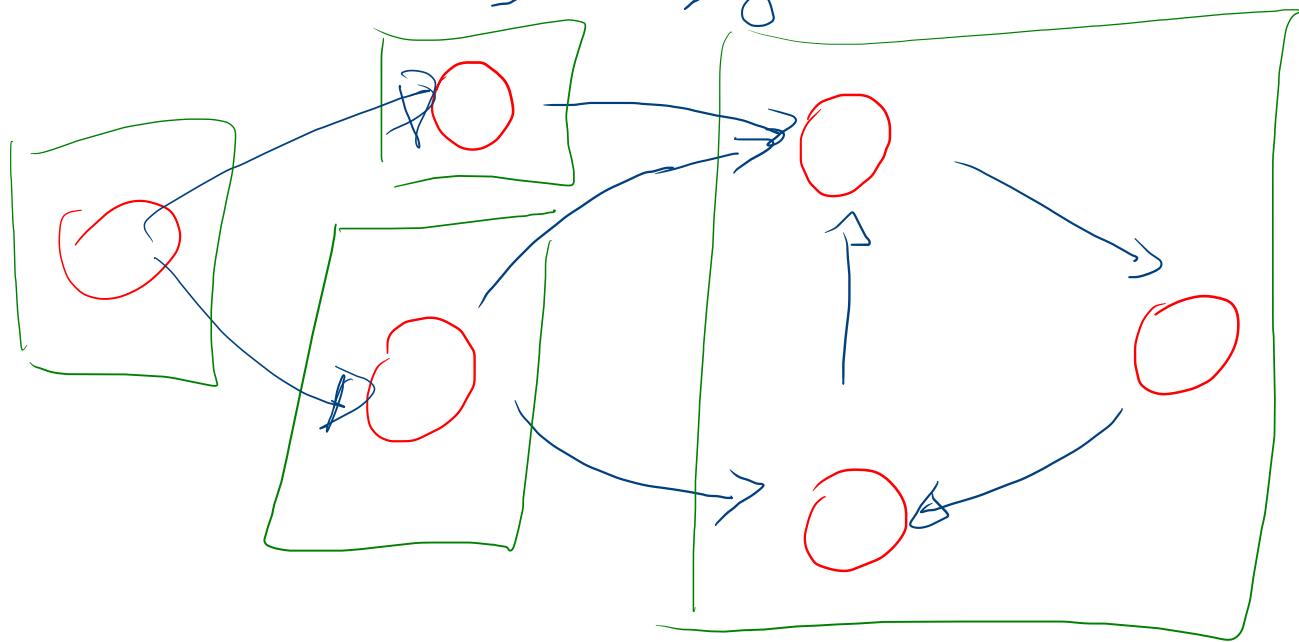
COMPONENTE CONNESSA

$$u \rightsquigarrow v \text{ mutua raggiungibilità}$$

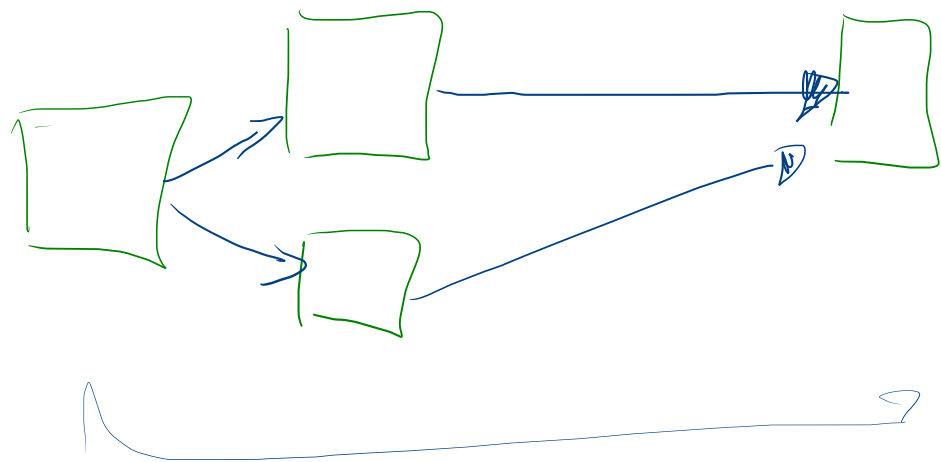
$$u \rightsquigarrow v \text{ e } v \rightsquigarrow u$$

COMPONENTI FORTEMENTE CONNESSE

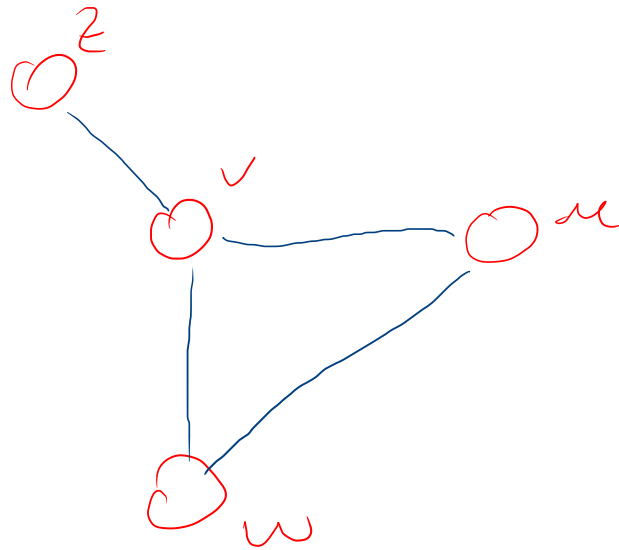
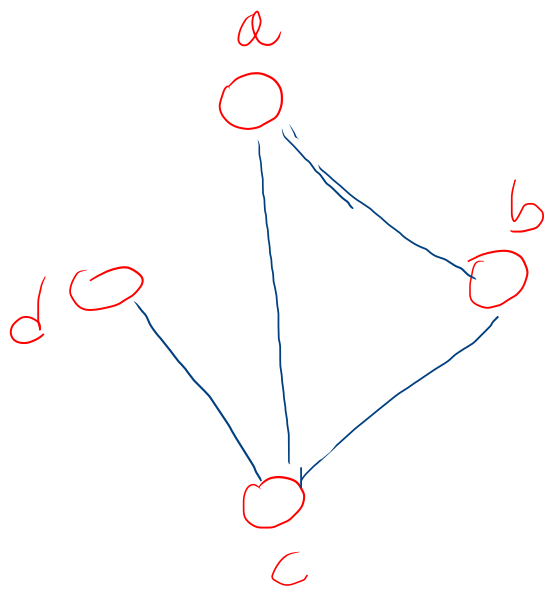
strongly connected components



SCC graph is acyclic.



Sottografo $V' \subseteq V$
 $(V', E') \subseteq (V, E)$ $E' = \{ (u, v) \in E \mid u, v \in V' \}$



$f: G \rightarrow G'$ \Leftrightarrow f ISOMORFISMO

$f: V \rightarrow V'$ $(u, v) \in E \Leftrightarrow (f(u), f(v)) \in E'$

MULTIGRATO ✓

ALBERO \equiv GRAFO NON ORIENTATO, ACICLICO, CONNESSO G .

FORESTA \equiv GRAFO NON ORIENTATO, ACICLICO

DAG \equiv GRAFO ORIENTATO ACICLICO

GRAFO BIPARTITO $\equiv G = (V, E)$ s.c. $V = V_1 \cup V_2$, $V_1 \cap V_2 = \emptyset$, $(u, v) \in E$,
 $u \in V_1$, $v \in V_2$ oppure $u \in V_2$ o $v \in V_1$

NOTAZIONE

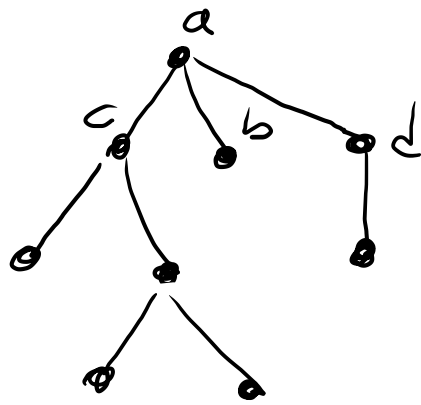
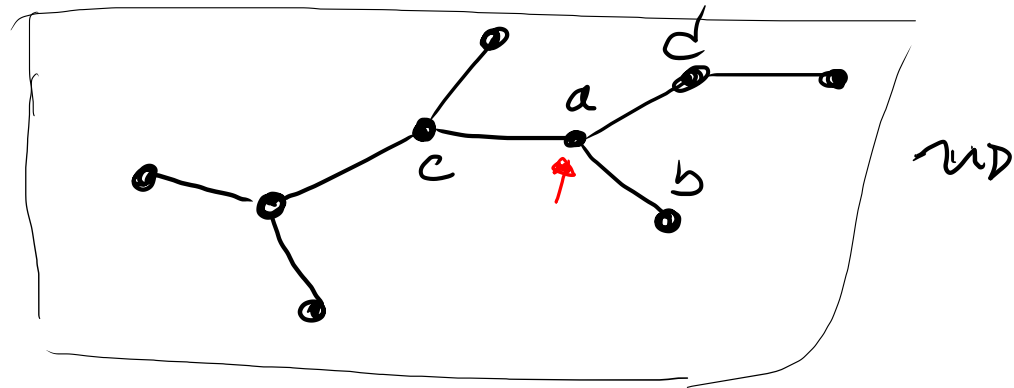
- ▶ Dato un grafo $G = (V, E)$ solitamente l'input viene misurato in modi dipendente da $|V|$ e $|E|$, quindi due parametri e non uno
- ▶ All'interno della notazione asintotica (e solo in quel caso) faremo la semplificazione di utilizzare V e E per indicare $|V|$ e $|E|$.
- ▶ Quindi un algoritmo che 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

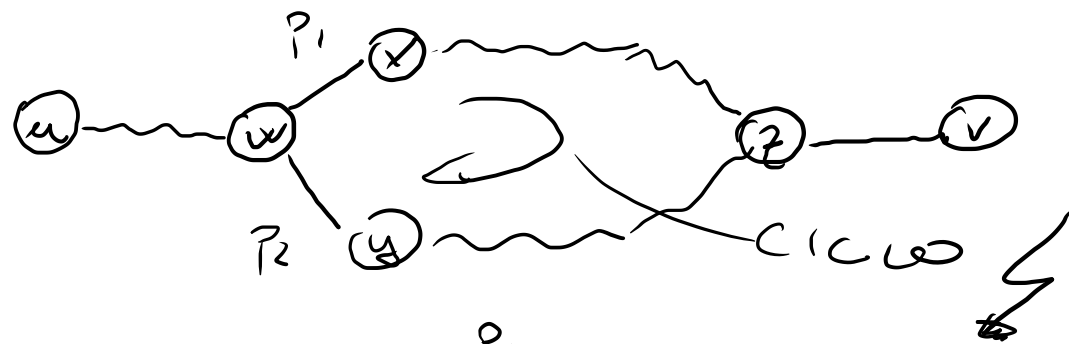
ALBERI

GRAFII NON ORIENTATI, CONNESSI, ACICLICI

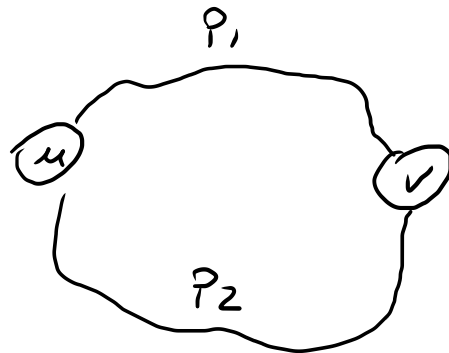


1) G è un albero \Leftrightarrow Due qualsiasi vertici di G sono connessi da un unico cammino semplice.

" \Rightarrow " P.A. $u \xrightarrow{P_1} v$, $u \xrightarrow{P_2} v$, $P_1 \neq P_2$



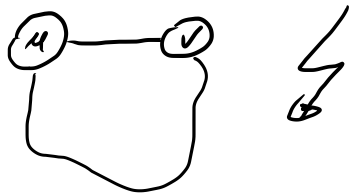
" \Leftarrow " Se G è aciclico



ho P_1, P_2 , $P_1 \neq P_2$ da u a v .

2) G è un albero $\Leftrightarrow G$ è connesso, ma rimuovendo un arco diventa sconnesso

\Rightarrow



\Leftarrow



3) G è un albero $\Leftrightarrow G$ è aciclico e se aggiungo un arco a G , ottengo un ciclo.

4) Se G è un albero

$$|E| \geq |V| - 1$$

$$|E| = |V| - 1$$

$$|E| \leq |V| - 1$$

tolgo un arco, V_1, V_2 $V_1 \cap V_2 = \emptyset$, $V_1 \cup V_2 = V$

$|V_j| < |V|$, vale ipotesi induttiva

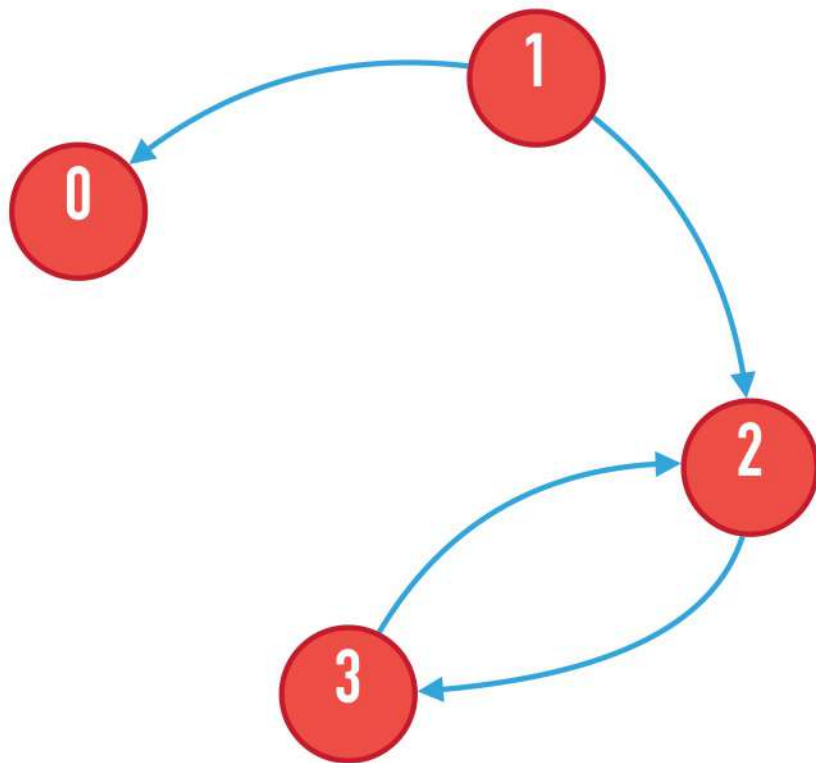
$$\underbrace{1 + |E_1| + |E_2|}_{|E|} \leq |V_1| - 1 + |V_2| - 1 + 1 = |V| - 1$$

$$|E|$$

COSA DOBBIAMO GESTIRE?

- ▶ Dobbiamo essere in grado di salvare i nodi...
- ▶ ...e gli archi (eventualmente con un peso).
- ▶ Assumiamo che i nodi abbiano nomi $\{0, \dots, n-1\}$
- ▶ 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

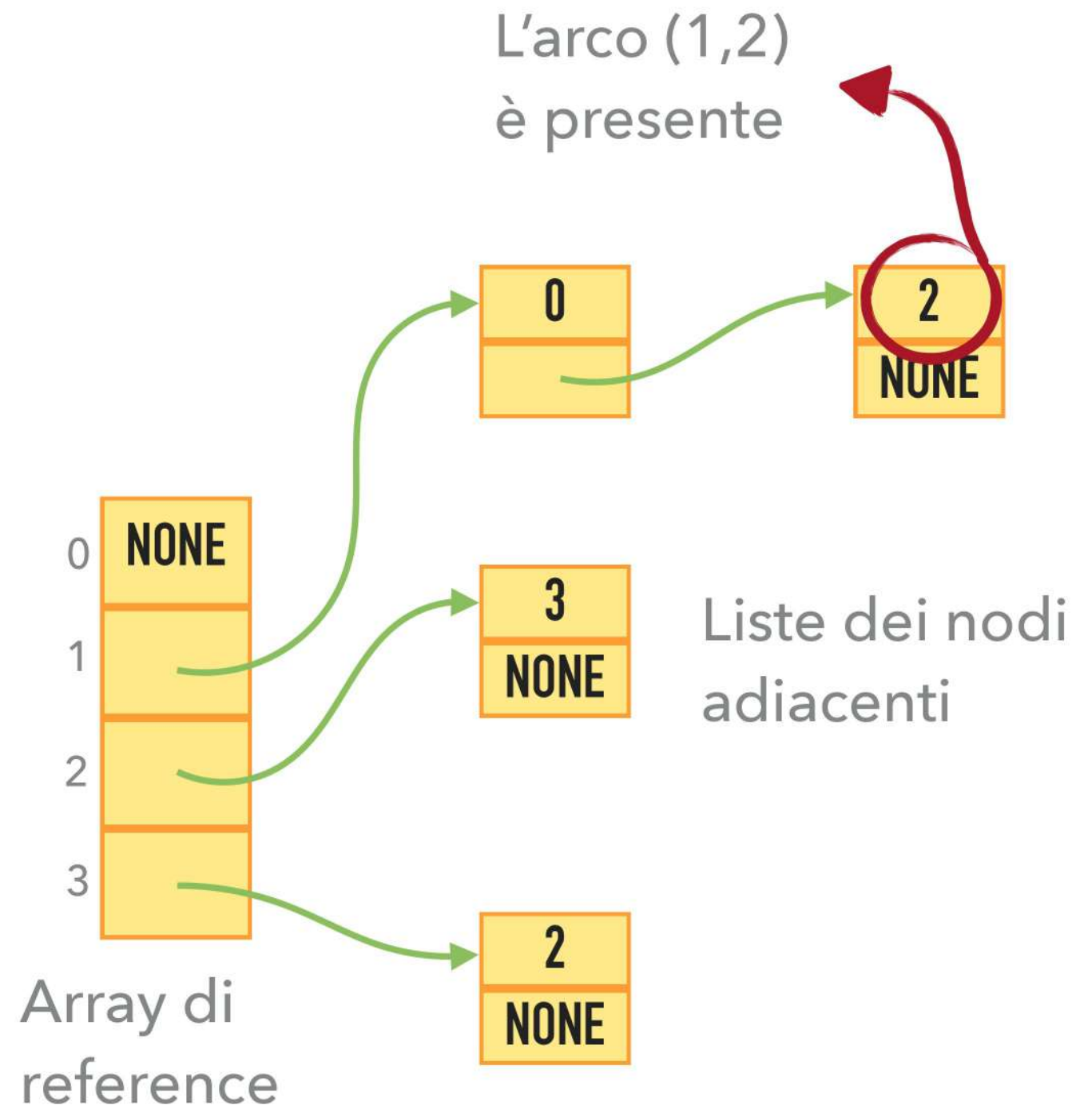
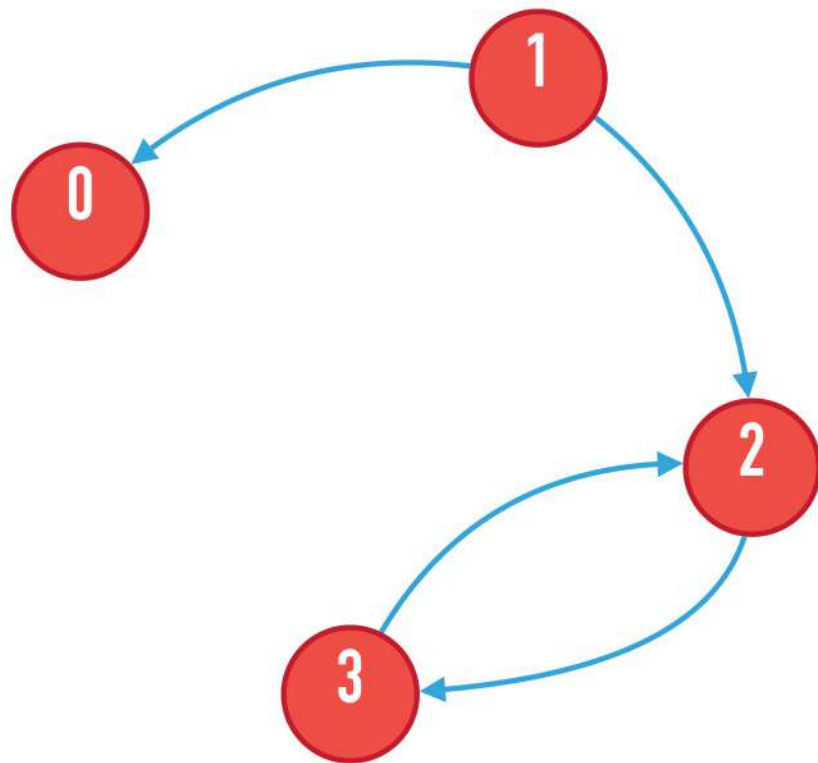


Destinazione

	0	1	2	3
Sorgente 0	0	0	0	0
Sorgente 1	1	0	1	0
Sorgente 2	0	0	0	1
Sorgente 3	0	0	1	0

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 loro vicini, etc.

AMPIEZZA VS PROFONDITÀ



Ricerca in ampiezza:
esploriamo tutti nodi vicini prima di allontanarci



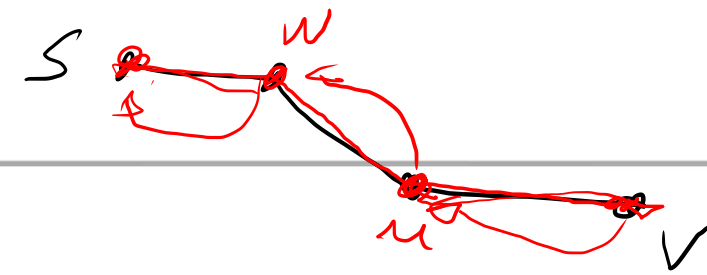
Ricerca in profondità:
seguiamo un singolo percorso il più possibile
prima di cambiare strada

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: INIZIALIZAZIONE

Parametri: grafo G , nodo sorgente s

inizialmente impostiamo distanza, colore e predecessore di tutti i nodi

for all $v \in V$:

 colore[v] = bianco

 distanza[v] = $+\infty$

 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

$Q = \text{Coda}()$

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
```

```
     $u = \text{dequeue}(Q)$  # prendiamo il primo nodo dalla coda
```

```
    for all  $v$  adiacenti a  $u$  # e ne esploriamo tutti i vicini
```

```
        if  $\text{color}[v] == \text{bianco}$  # consideriamo solo i vicini mai visitati prima
```

```
             $\text{colore}[v] = \text{grigio}$ 
```

```
            # possiamo arrivare a  $v$  con un passo partendo da  $u$ 
```

```
             $\text{distanza}[v] = \text{distanza}[u] + 1$ 
```

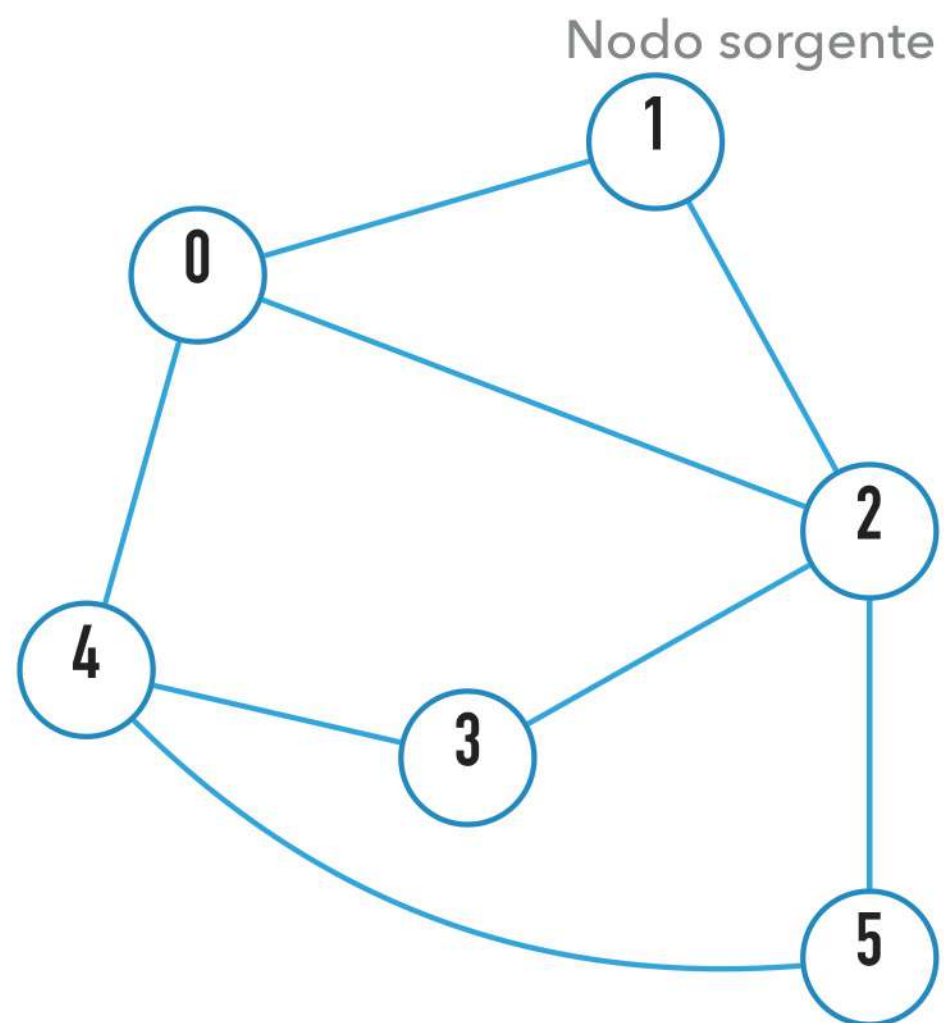
```
             $\text{predecessore}[v] = u$ 
```

```
             $\text{enqueue}(Q, v)$  # accodiamo perché potrebbe avere dei vicini bianchi
```

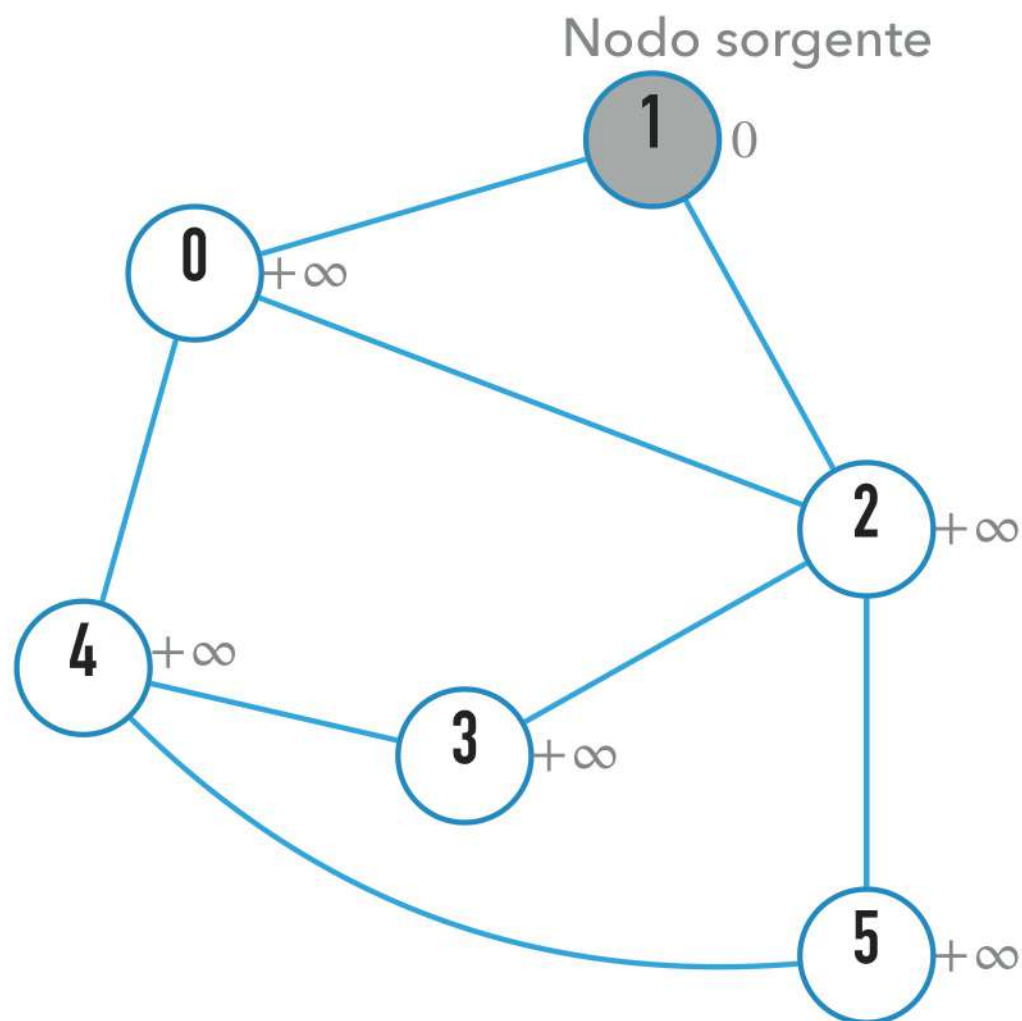
```
     $\text{colore}[u] = \text{nero}$  # abbiamo finito di visitare tutti i vicini di  $u$ 
```

```
# una volta usciti dal ciclo abbiamo visitato tutti i nodi raggiungibili da  $s$ 
```

ESEMPIO DI ESECUZIONE



ESEMPIO DI ESECUZIONE



Coda dei nodi da visitare



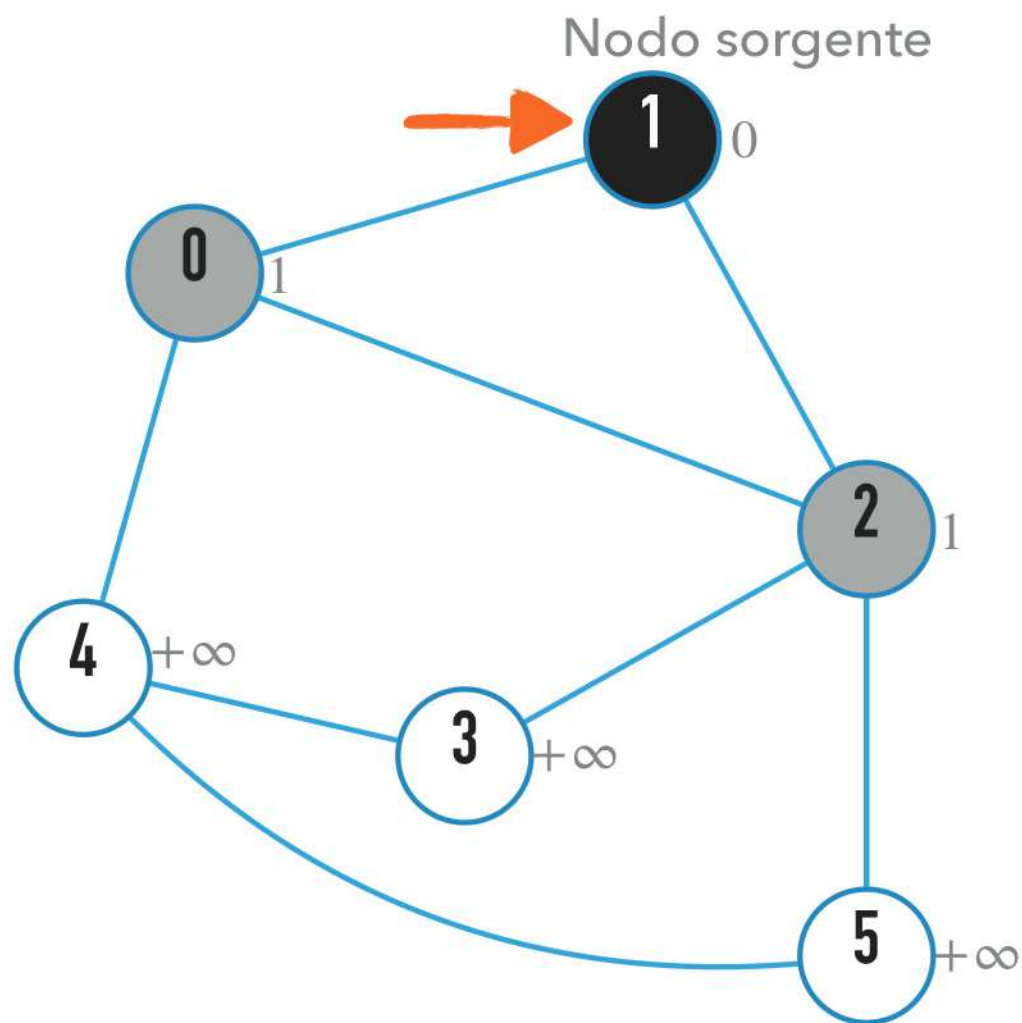
Inizialmente la distanza conosciuta di tutti i nodi dal nodo di partenza è $+\infty$

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

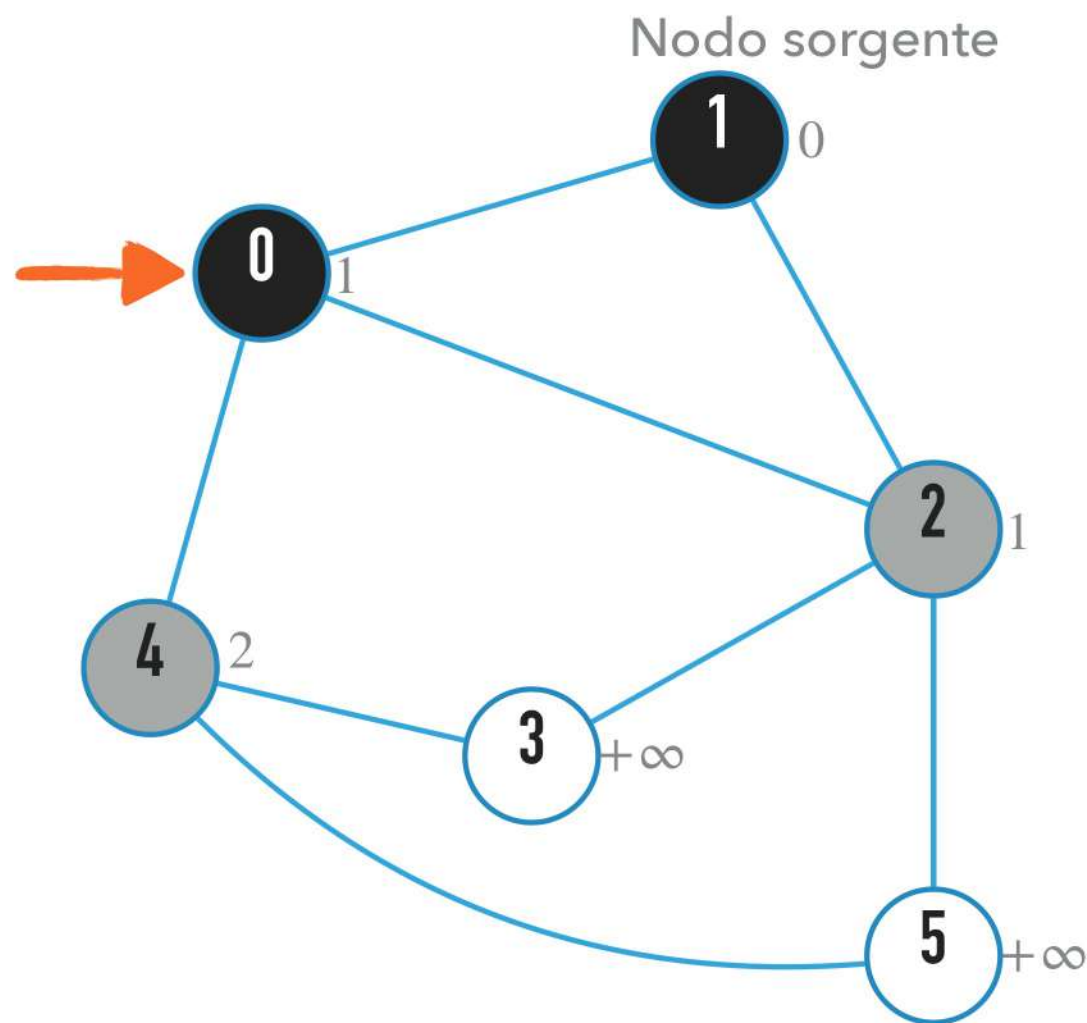
Per ogni vicino, se è bianco lo coloriamo di grigio, aggiorniamo distanza (come $distanza[u] + 1$) e predecessore (u) e lo accodiamo

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

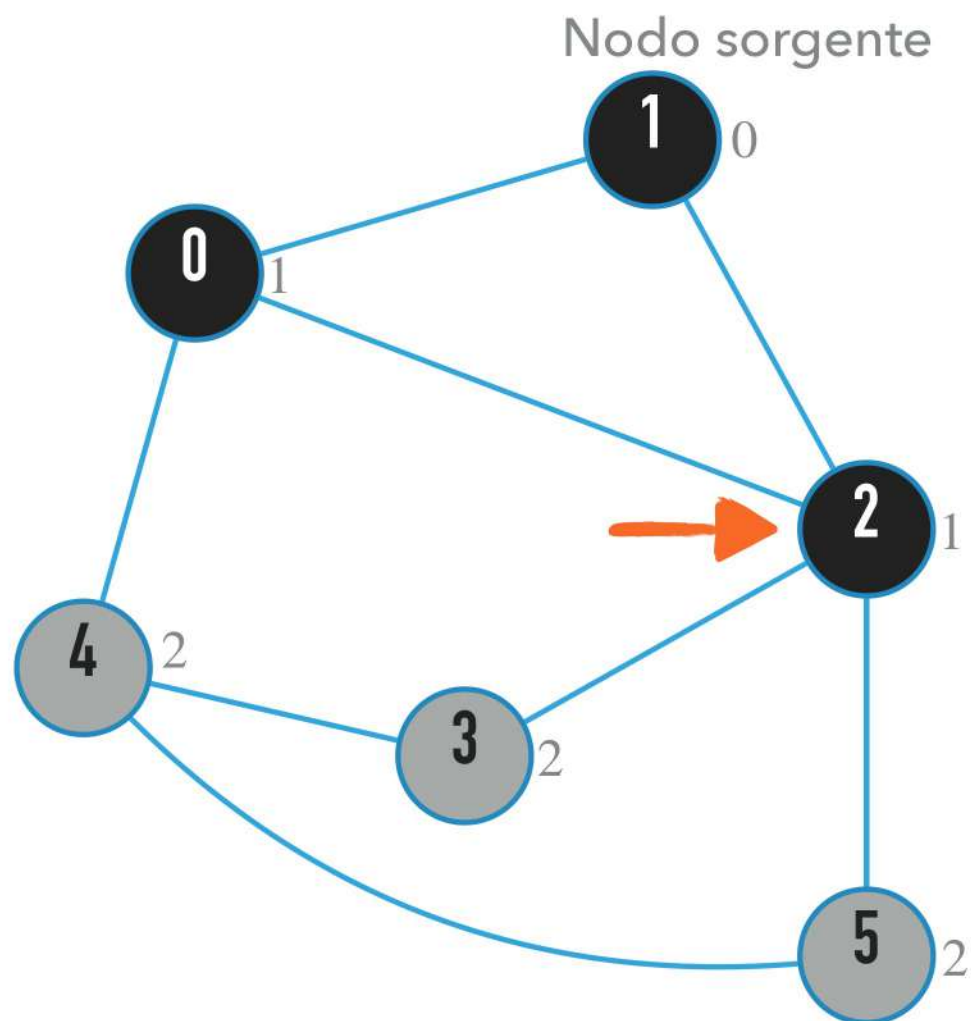
Per ogni vicino, se è bianco lo coloriamo di grigio, aggiorniamo distanza (come $distanza[u] + 1$) e predecessore (u) e lo accodiamo

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



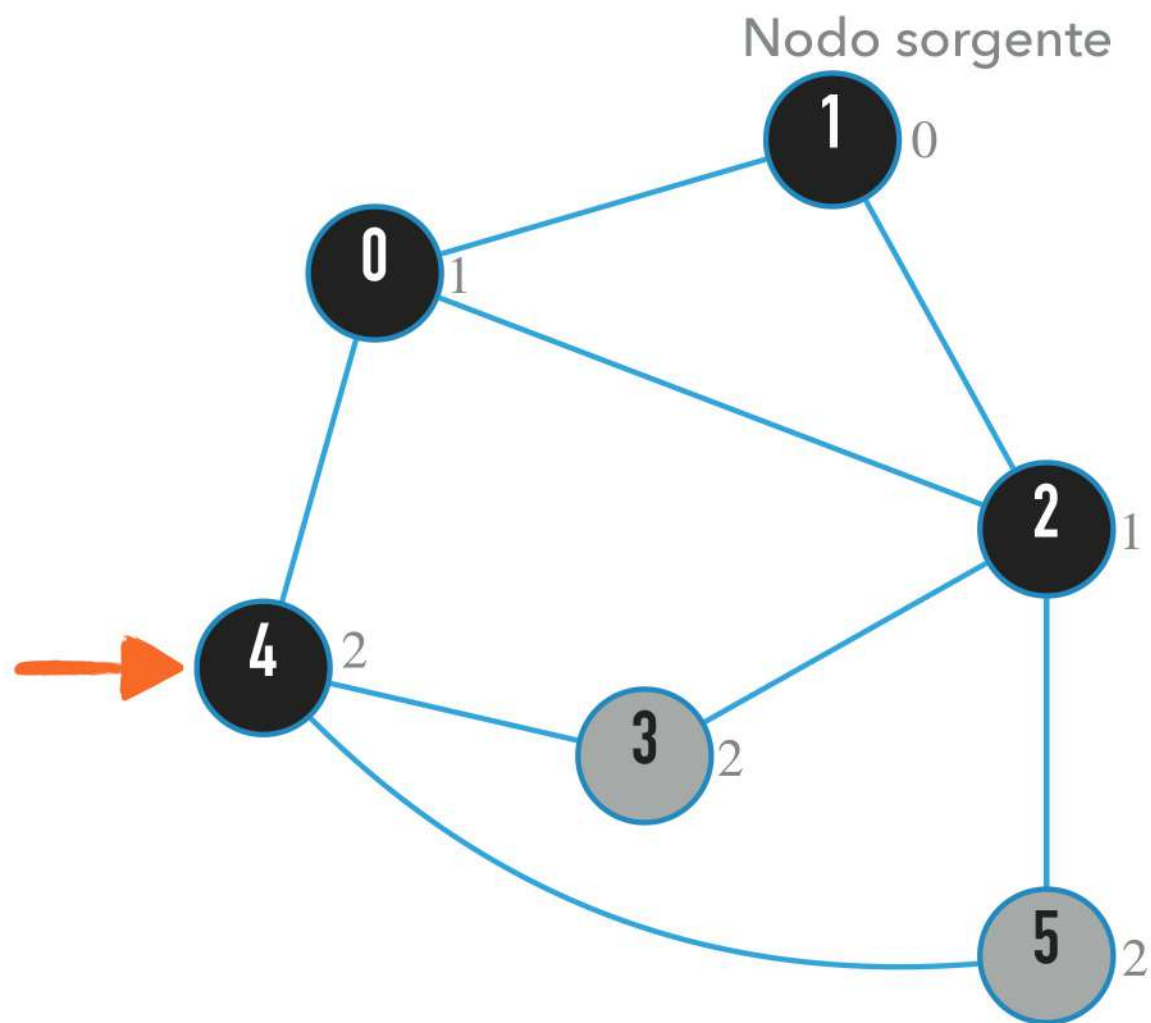
Estraiamo il nodo u dalla coda

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

Coloriamo u di nero

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

ESEMPIO DI ESECUZIONE



Coda dei nodi da visitare



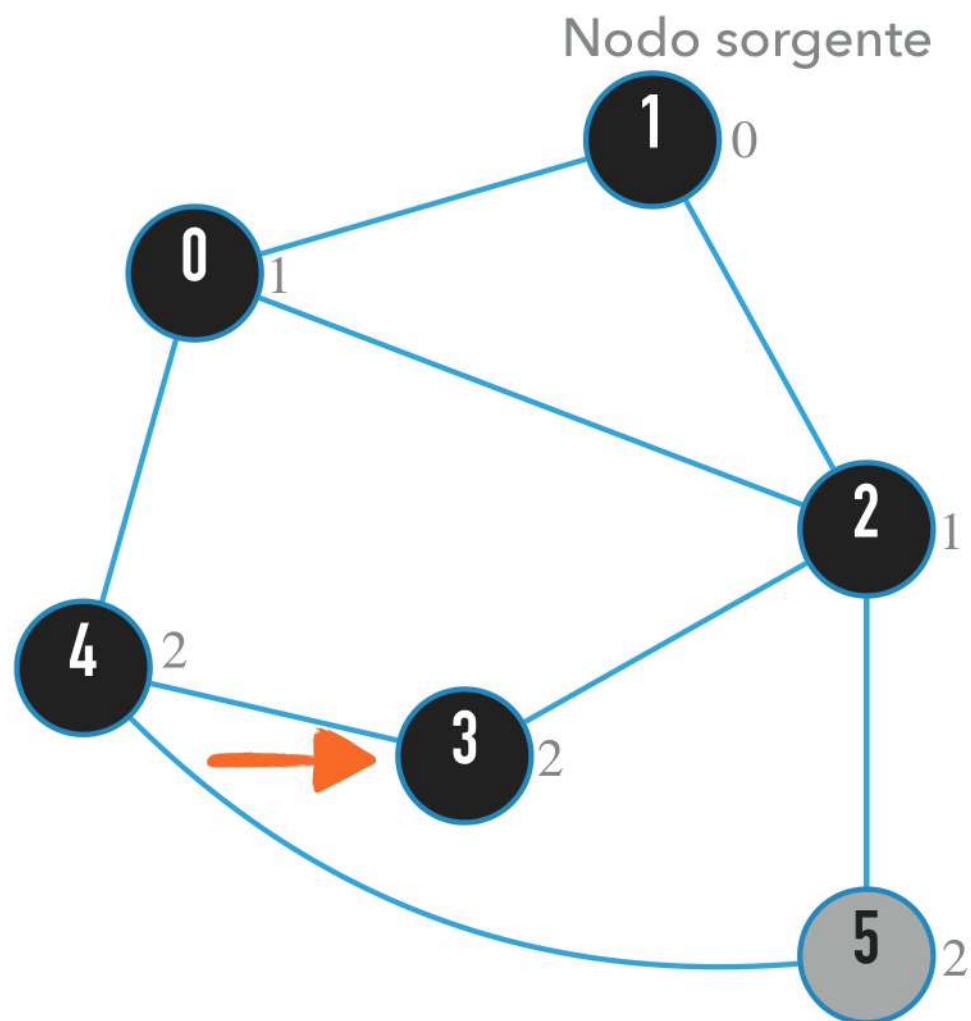
Estraiamo il nodo u dalla coda

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

Coloriamo u di nero

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

ESEMPIO DI ESECUZIONE



Coda dei nodi da visitare



Estraiamo il nodo u dalla coda

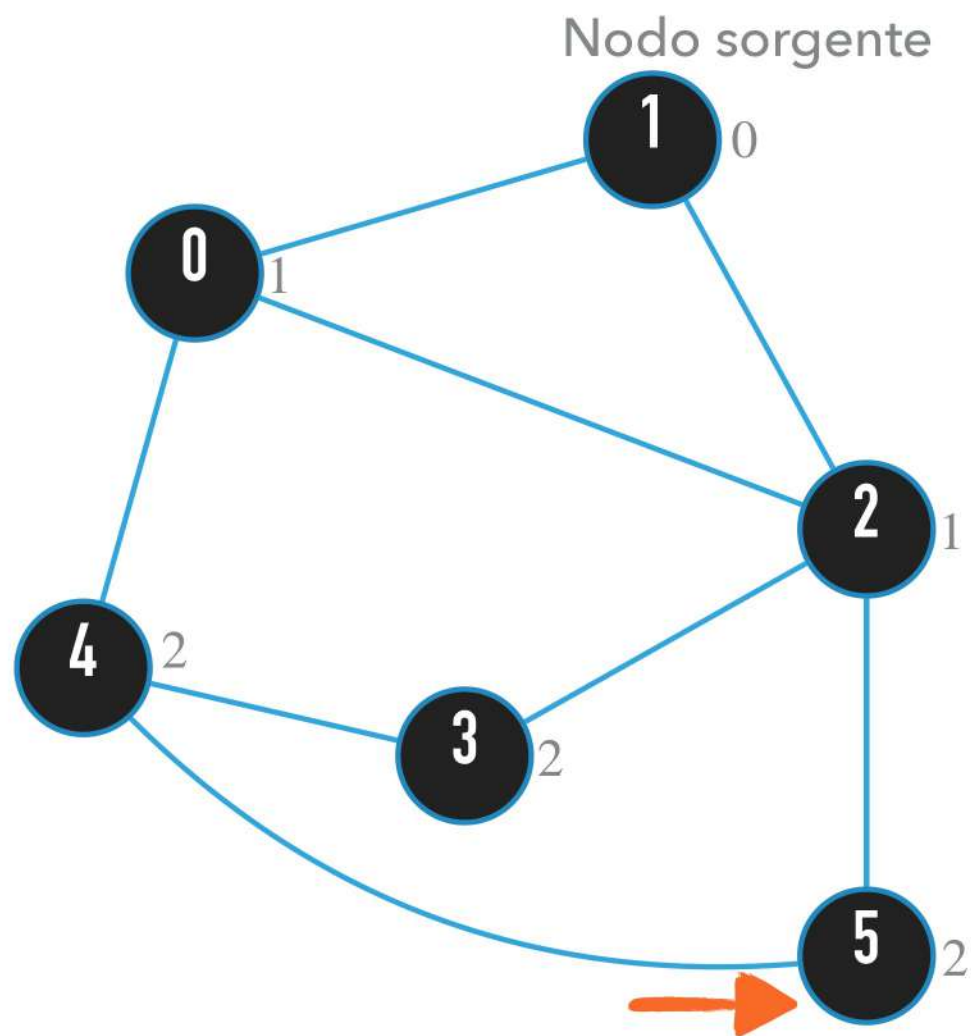
Per ogni vicino, se è bianco lo coloriamo di grigio, aggiorniamo distanza (come $distanza[u] + 1$) e predecessore (u) e lo accodiamo

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

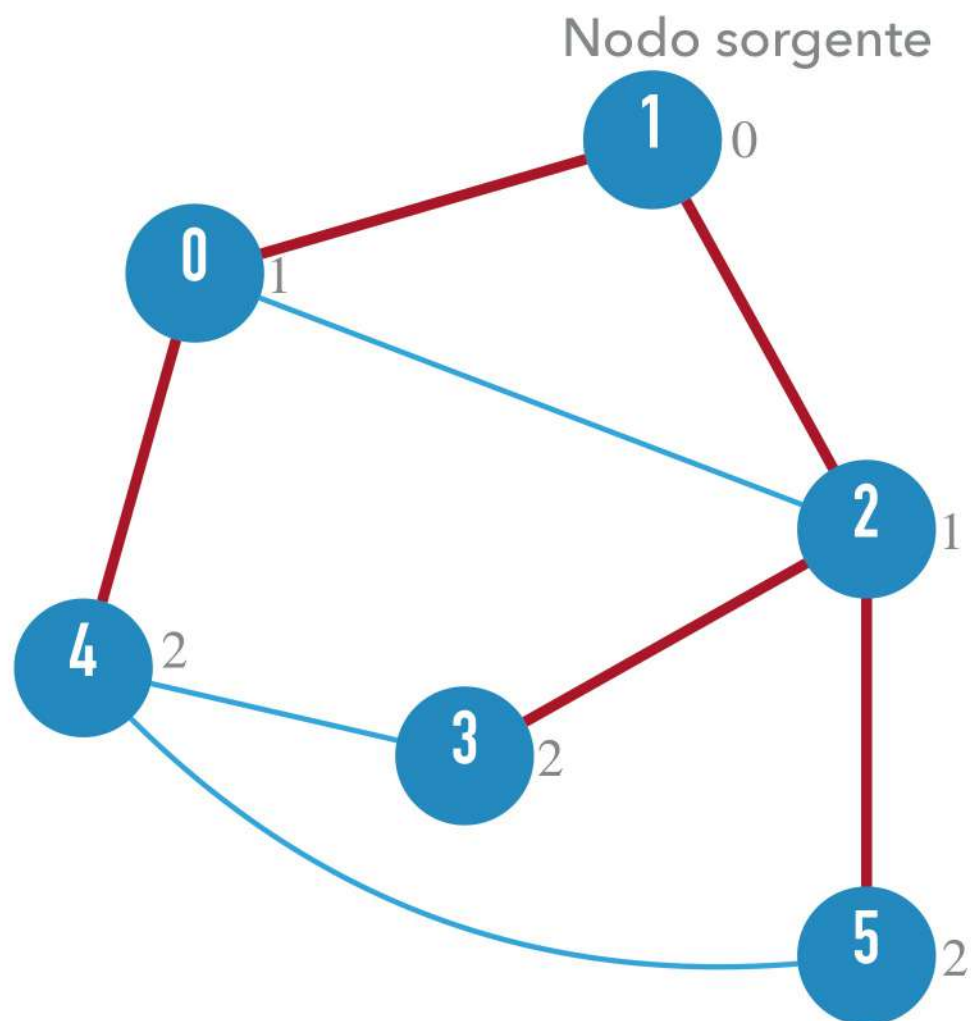
Per ogni vicino, se è bianco lo coloriamo di grigio, aggiorniamo distanza (come $distanza[u] + 1$) e predecessore (u) e lo accodiamo

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 $O(V + E)$