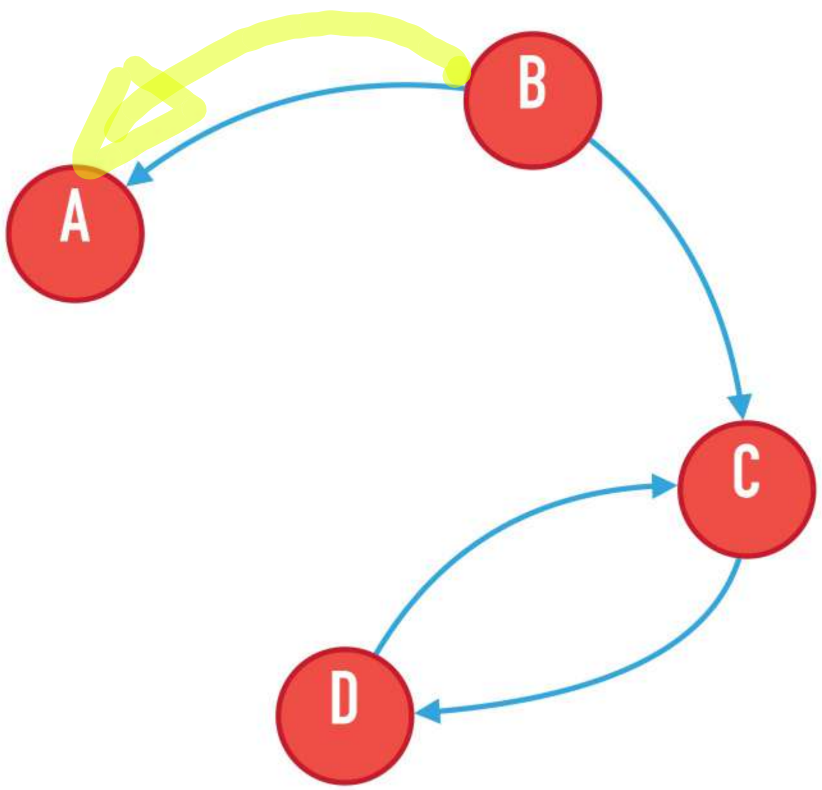
GRAFI
RICERCA IN AMPIEZZA

INFORMATICA

COSA È UN GRAFO?



Insieme di nodi:

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

Insieme di archi:

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

TARGET

SOURCE

COSA È UN GRAFO (VARIANTI)?

GRADO (v) degree

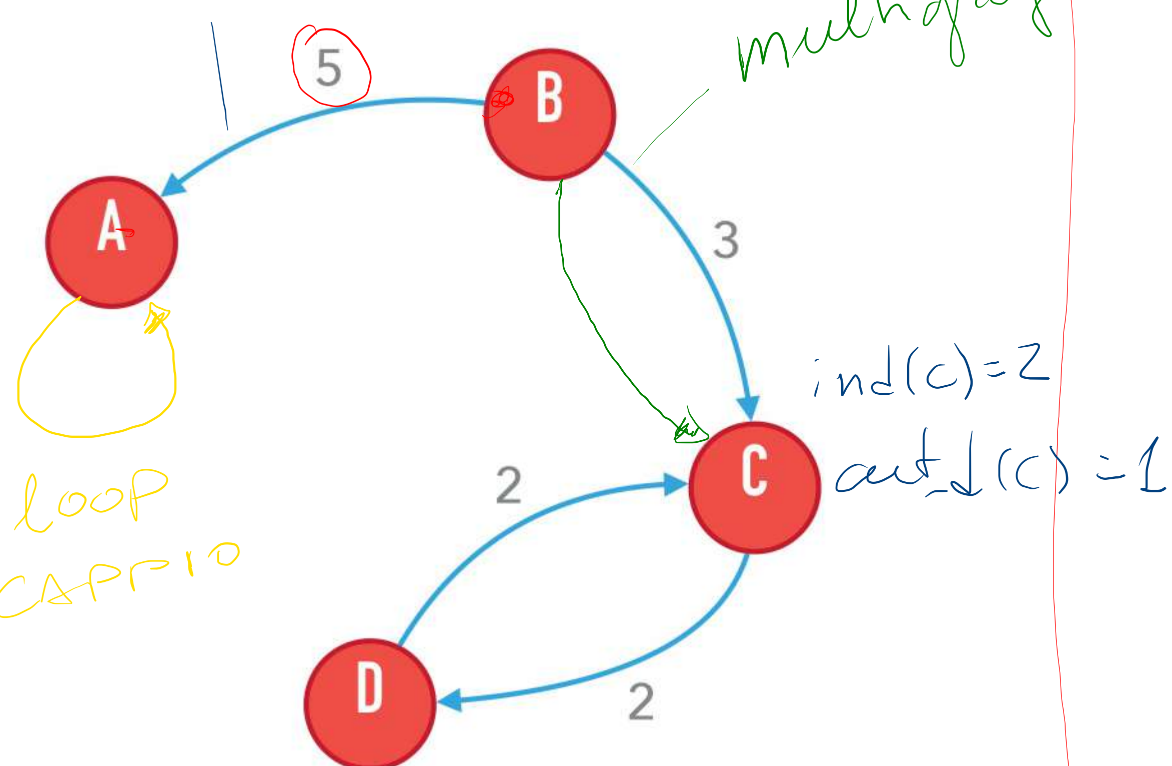
=

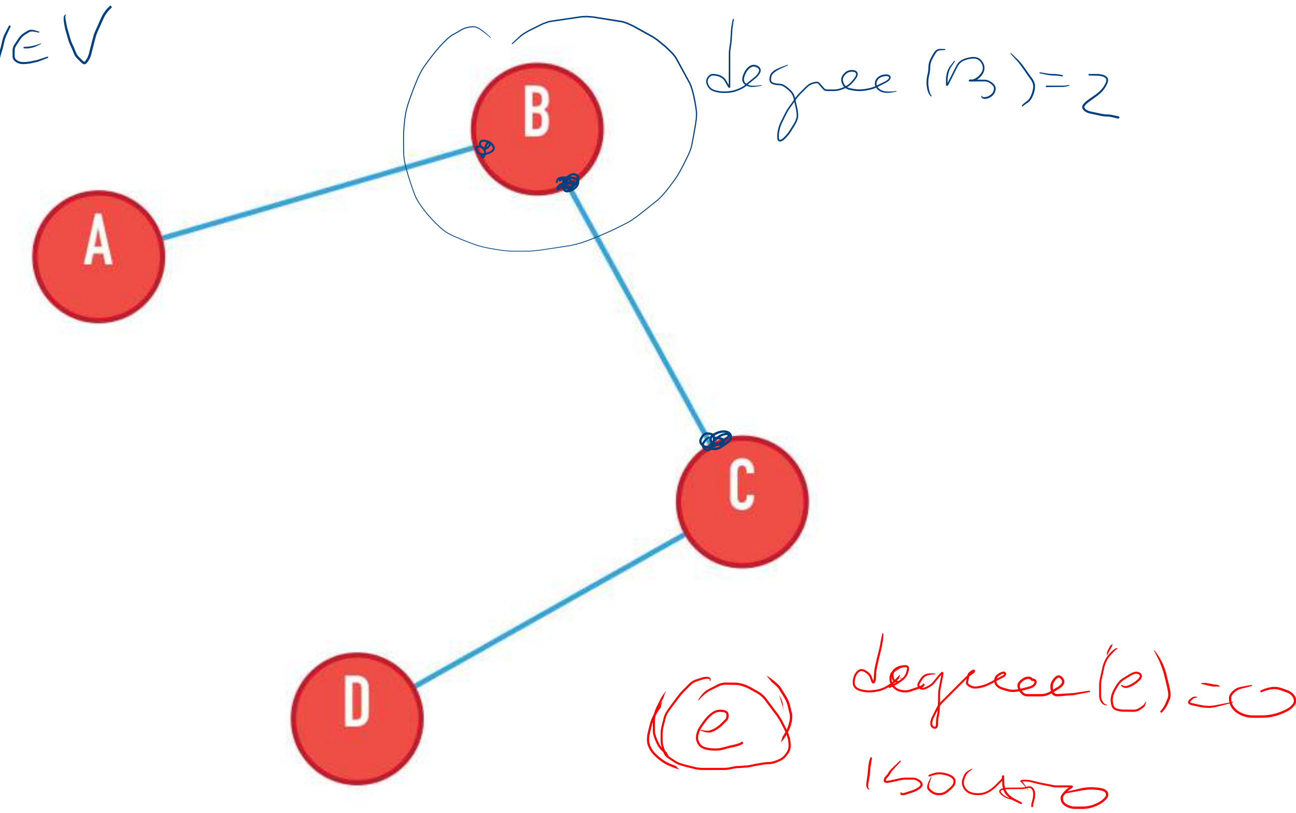
ARCHI INCIDENTI IN v

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

grafo semplice

multigrafo





indegree (v) = # nodi in ingresso

outdegree (v) = # nodi in uscita

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 $a, b \in V$

IPERGRAFI ($E \subseteq V^k$ o $V^* = V \cup V^2 \cup V^3 \cup \dots \cup V^4 \cup \dots$)

Se un grafo NON HA CICLI si dice ACICLICO

GRAFI: NOZIONI DI BASE

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

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

$u \overset{\pi}{\rightsquigarrow} v$

$\pi : u, \ldots \to v$

RAGGIUNGIBILITÀ

$\exists \pi : u \overset{\pi}{\rightsquigarrow} v$

$u \rightsquigarrow v$

RAGGIUNGIBILITÀ simmetrica in non oriented graph

⇓

relazione di equivalenza su V

⇓

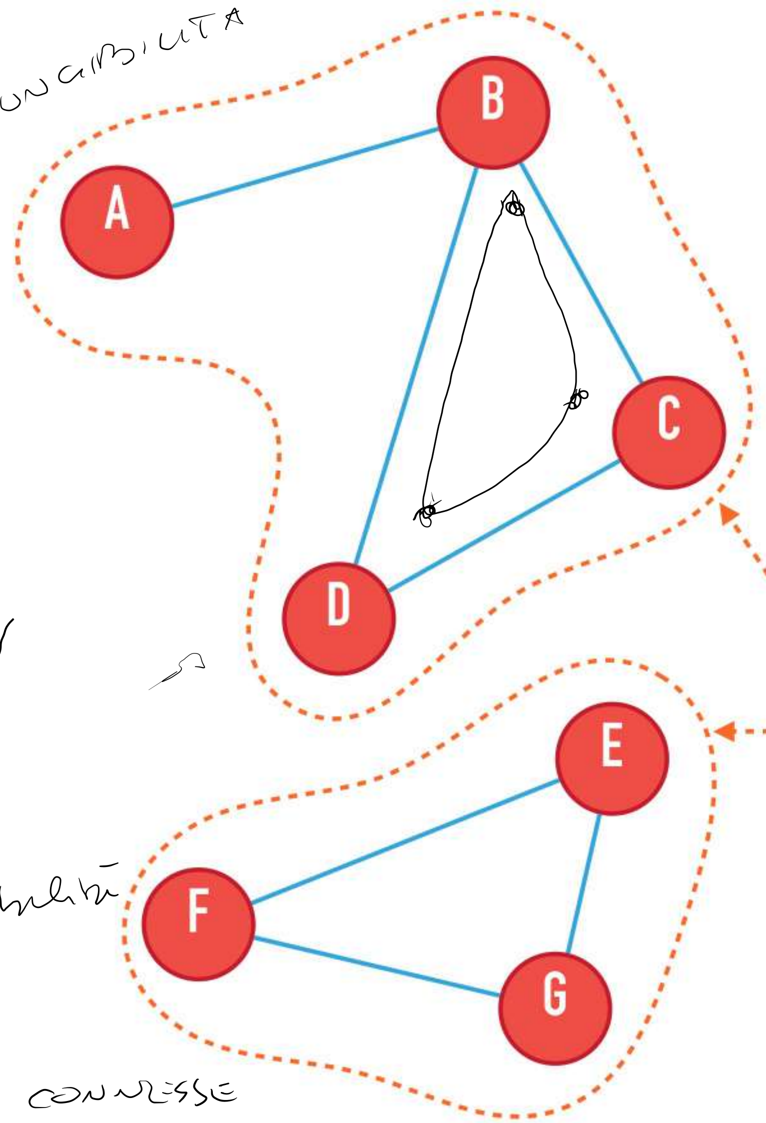
COMPONENTE CONNESSA

$u \leftrightsquigarrow v$ mutua raggiungibilità

⇕

$u \rightsquigarrow v$ e $v \rightsquigarrow u$

COMPONENTI FORTEMENTE CONNESSE



Componenti connesse

PATH

→ CAMMINO o (**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 $+\infty$

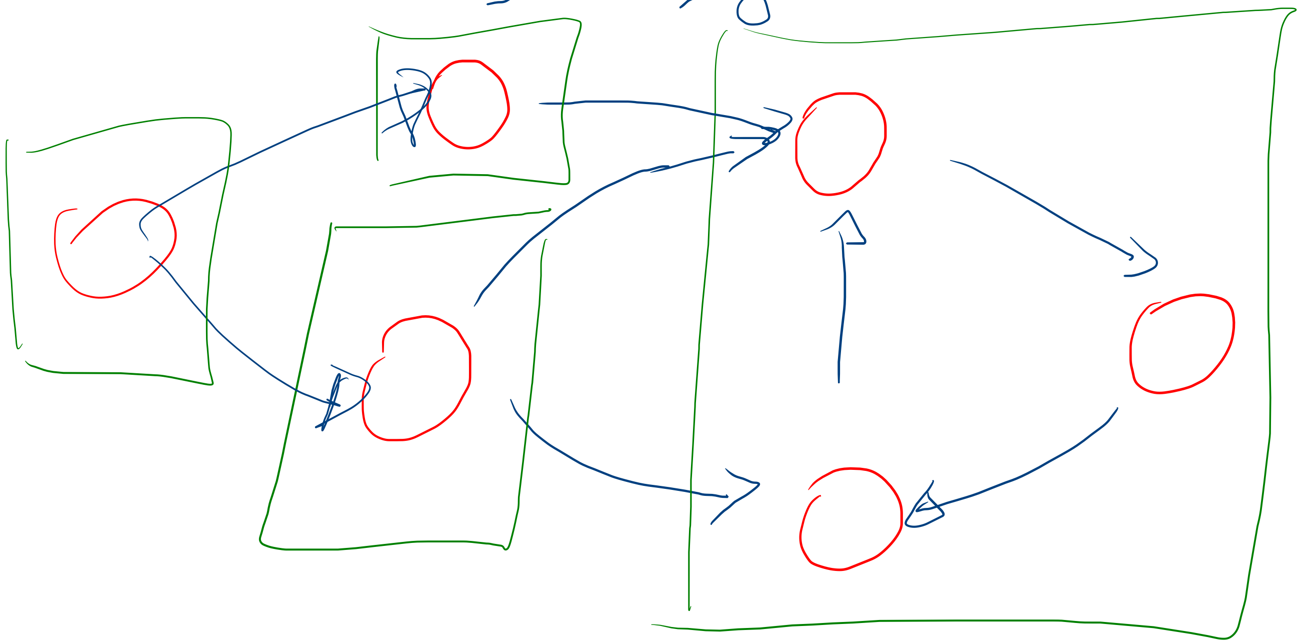
$\langle v_0 \ldots v_k \rangle$

$\cup$|

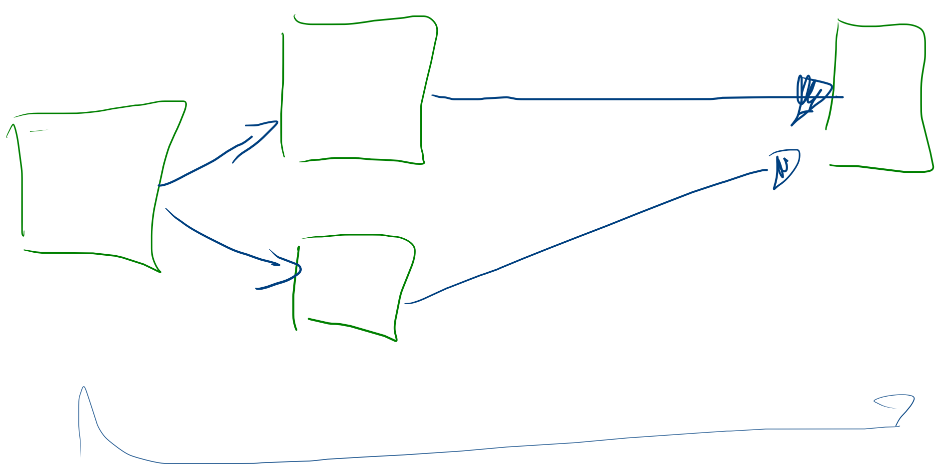
$\langle v_i \ldots v_j \rangle$ SOTTOCAMMINO

- CAMMINO SEMPLICE: ogni vertice è distinto dagli altri
- CICLO: $\langle \underbrace{v_0, v_1, \ldots v_k}_{\text{lunghezza } k}, v_0 \rangle$, $v_0 \ldots v_k$ distinti

strongly connected components

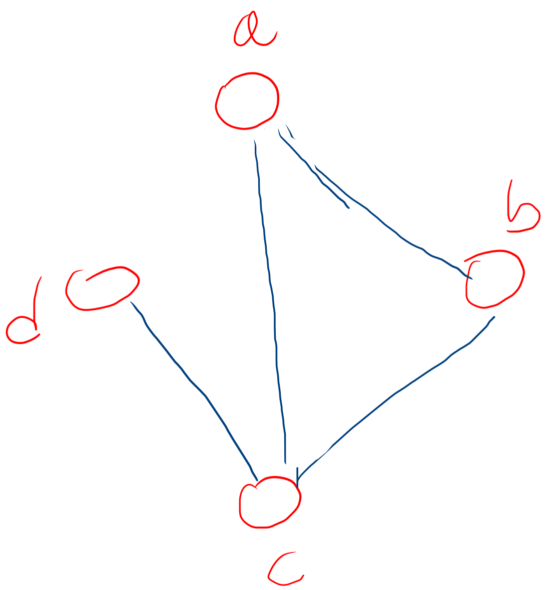


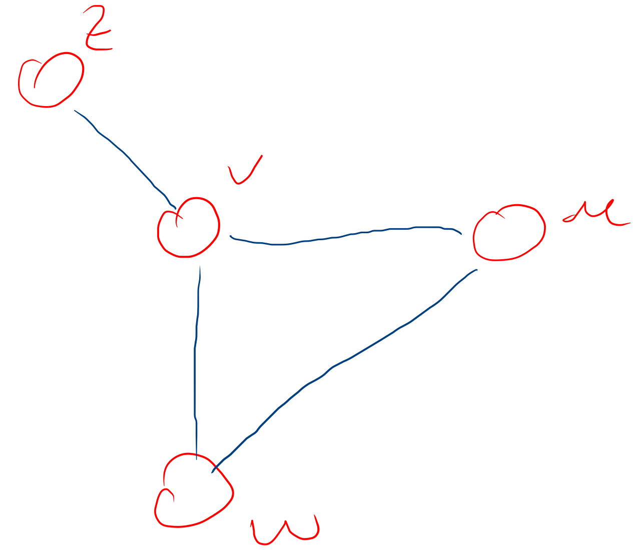
SCC GRAPH is acyclic.



SOTTOGRAFO $\quad V' \subseteq V$

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





$f : G \longrightarrow G'$, $\leftarrow$ f isomorfismo

$f : V \longrightarrow V'$ $\quad (u,v) \in E \iff (f(u), f(v)) \in E'$

MULTIGRAFO ✓

ALBERO = GRAFO NON ORIENTATO, ACICLICO, CONNESSO

FORESTA = GRAFO NON ORIENTATO, ACICLICO

DAG = GRAFO ORIENTATO ACICLICO

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

NOTAZIONE

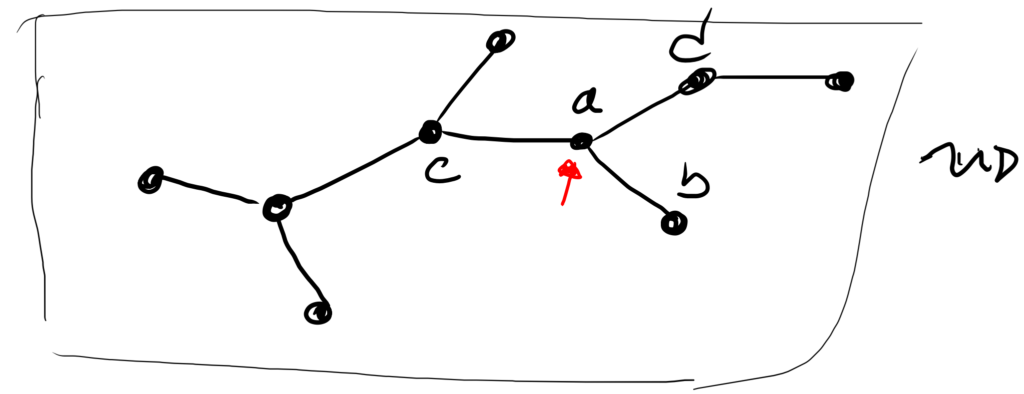
- Dato un grafo $G = (V, E)$ solitamente l'input viene misurato in modi dipendente da $|V|$ e $|E|$, quindi due parametri e non uno
- All'interno della notazione asintotica (e solo in quel caso) faremo la semplificazione di utilizzare V e E per indicare $|V|$ e $|E|$.
- Quindi un algoritmo che richiede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

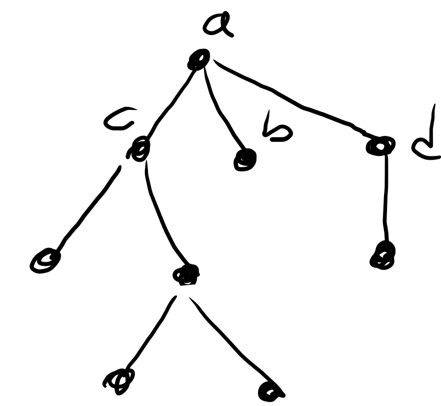
NOTAZIONE

- 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

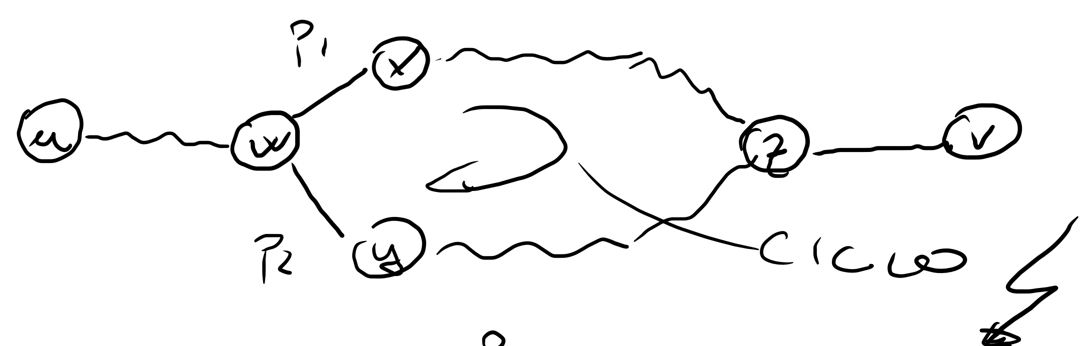
GRAFI NON ORIENTATI, CONNESSI, ACICLICI



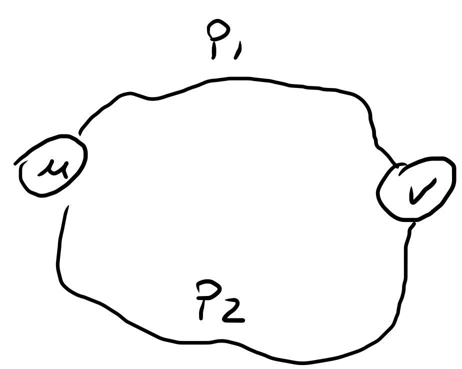


1) G è un albero $\Longleftrightarrow$ Due qualsiasi vertici di G sono connessi da un unico cammino semplice.

"$\Rightarrow$" P.A. $u \overset{p_1}{\leadsto} v$, $u \overset{p_2}{\leadsto} v$, $p_1 \neq p_2$

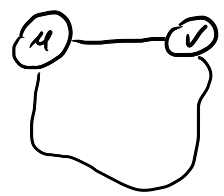
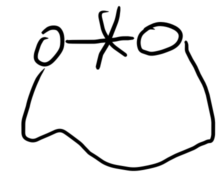


"$\Leftarrow$" Se G è ciclico



↯ ho p_1, p_2, $p_1 \neq p_2$ da u a v.

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

$\Rightarrow$  ↯ "$\Leftarrow$"  ↯

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

4) Se G è un albero $|E| = |V| - 1$

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

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

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

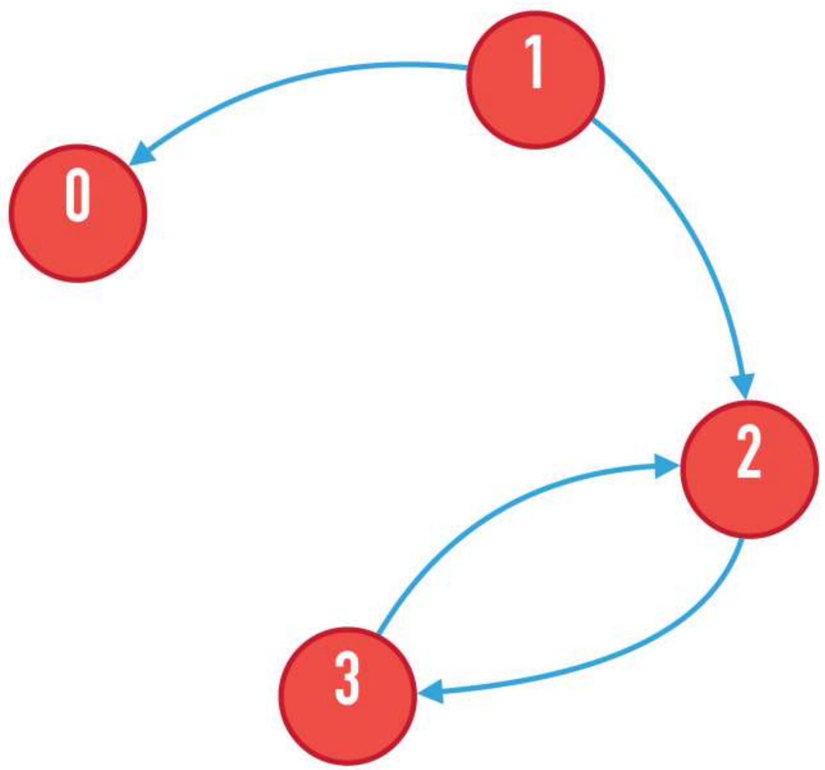
$|V_j| < |V|$, vale ipotesi induttiva

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

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

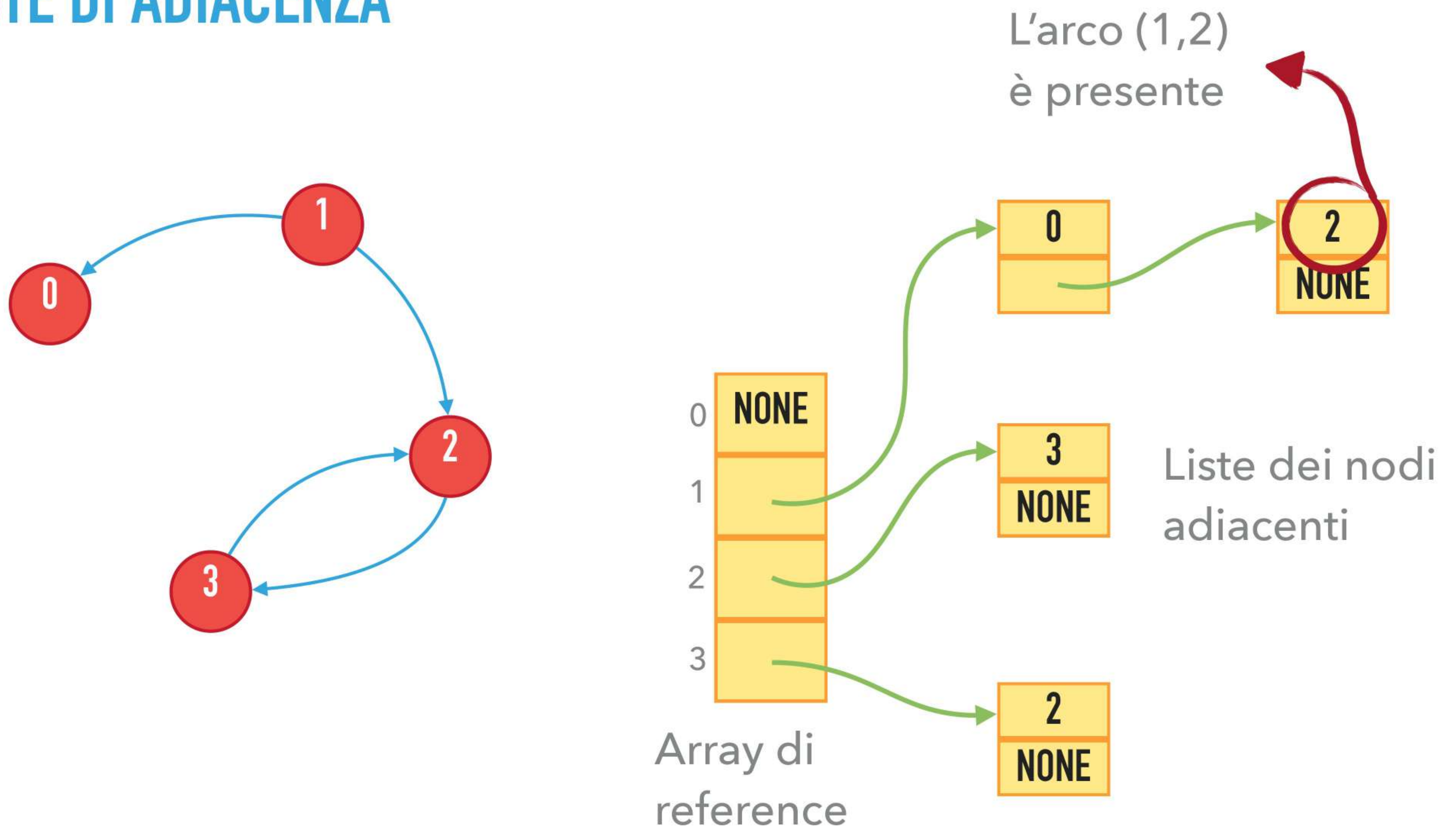


Destinazione

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

L'arco (1,0) è presente

LISTE DI ADIACENZA



QUALE RAPPRESENTAZIONE USARE

Matrici di adiacenza	**Liste di adiacenza**
Veloce (tempo costante) stabilire se un arco esiste	Lento (serve scandire una lista) stabilire se un arco esiste
Occupazione quadratica di memoria rispetto al numero di vertici $O(V^2)$	Occupazione lineare di memoria rispetto al numero di vertici e archi $O(V+E)$
Funziona bene per grafi densi	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

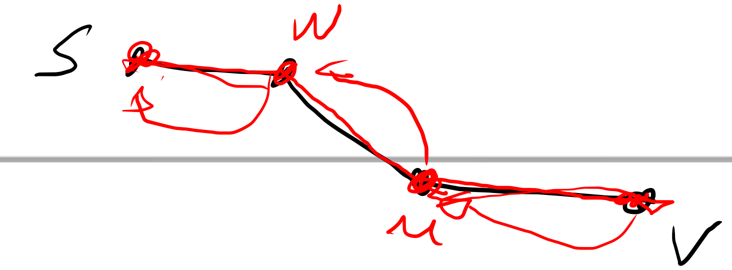
Ricerca in profondità:
seguiamo un singolo percorso il più possibile prima di cambiare strada

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



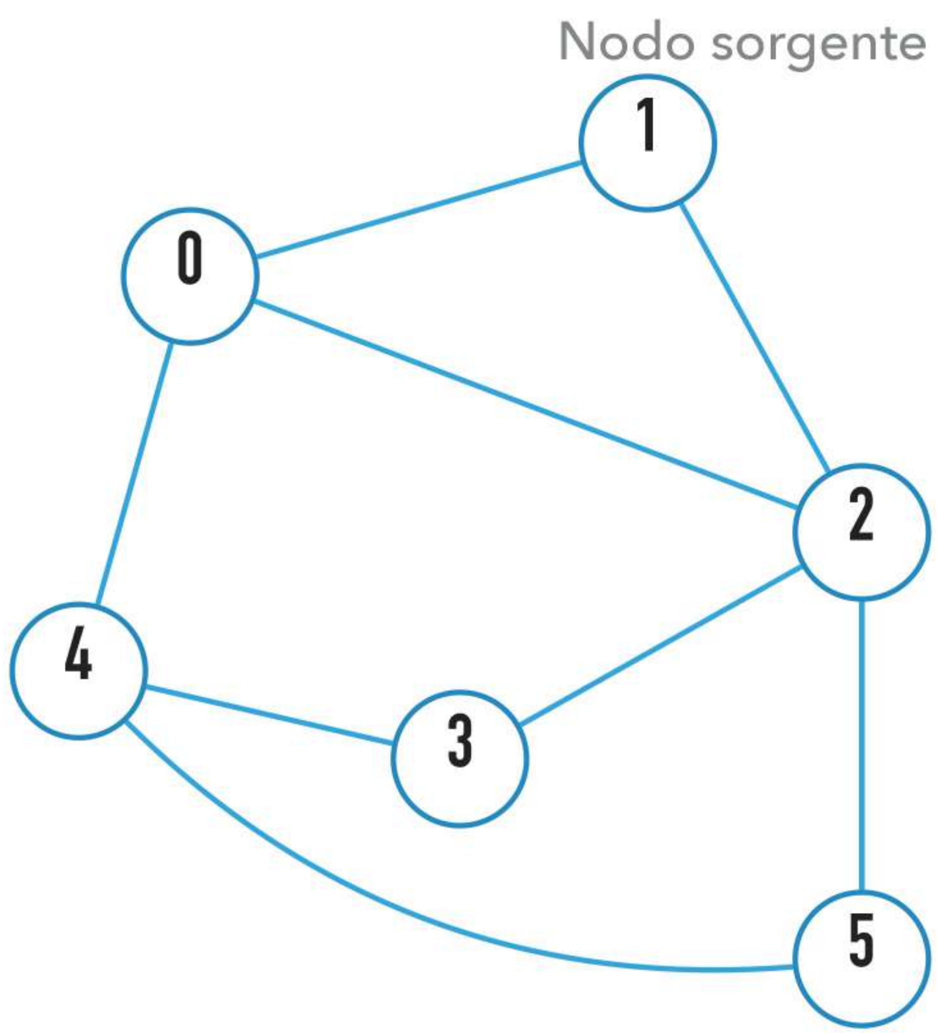
PSEUDOCODICE: INIZIALIZZAZIONE

```
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

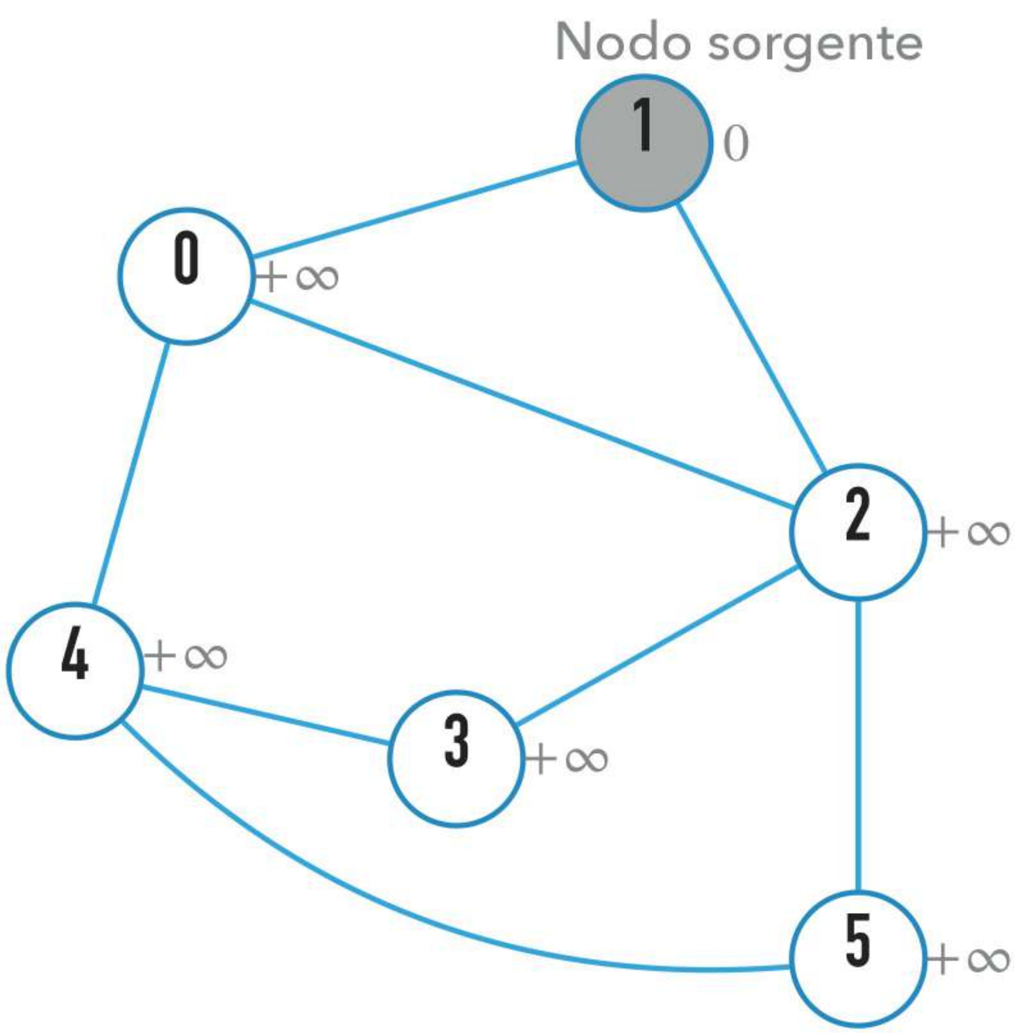
```
while Q is not empty:
    # questo ciclo deve continuare finché ci rimangono dei nodi da visitare
    u = dequeue(Q) # prendiamo il primo nodo dalla coda
    for all v adiacenti a u # e ne esploriamo tutti i vicini
        if color[v] == bianco # consideriamo solo i vicini mai visitati prima
            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



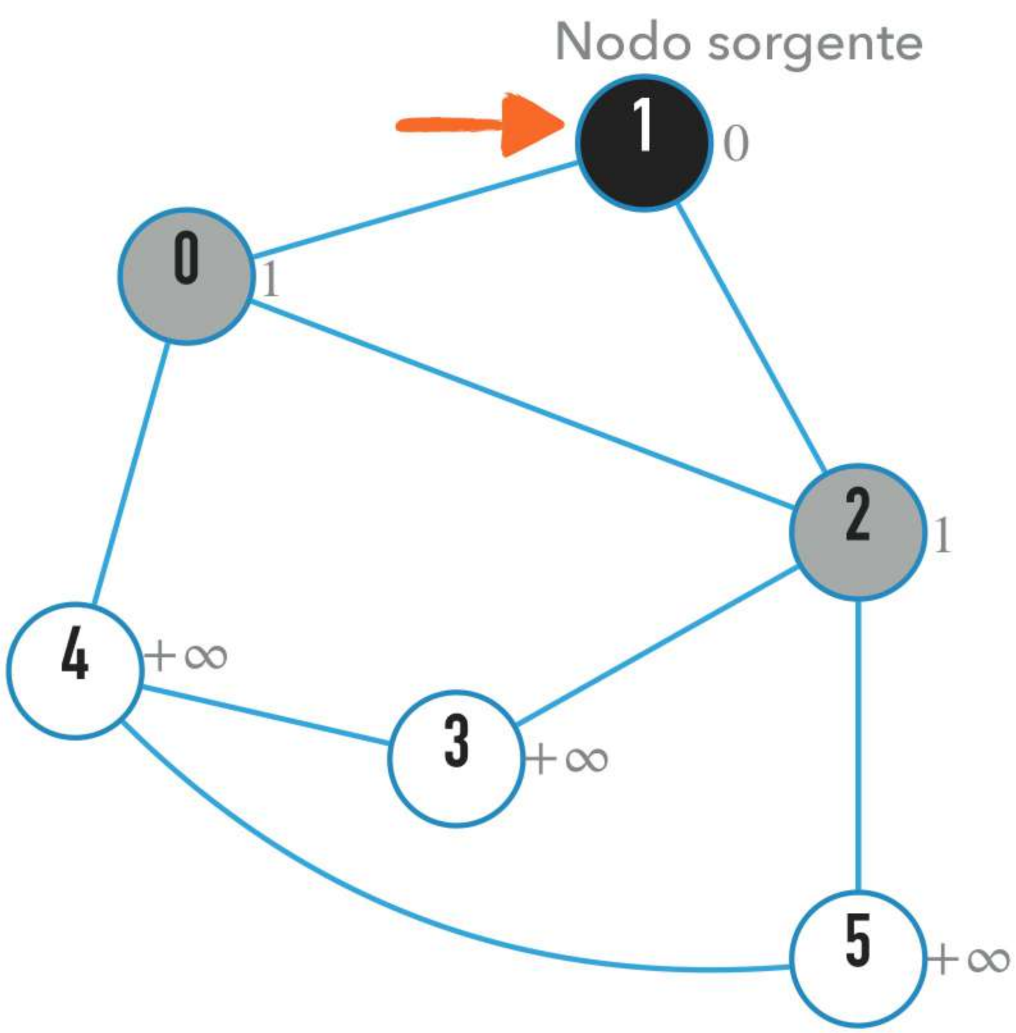
Inizialmente la distanza conosciuta
di tutti i nodi dal nodo di partenza è $+\infty$

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

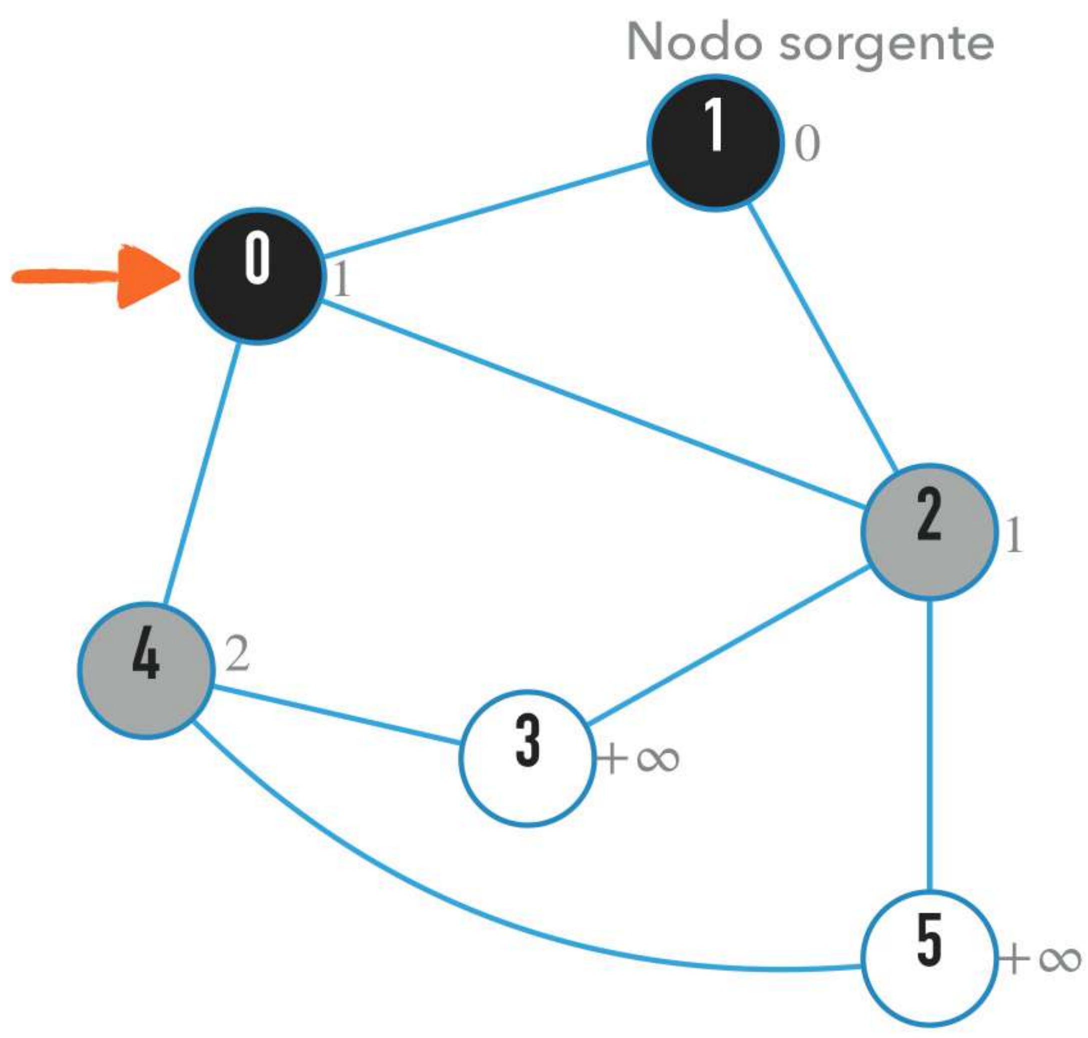
Per ogni vicino, se è bianco lo coloriamo di grigio, aggiorniamo distanza (come distanza$[u] + 1$) e precedessore (u) e lo accodiamo

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

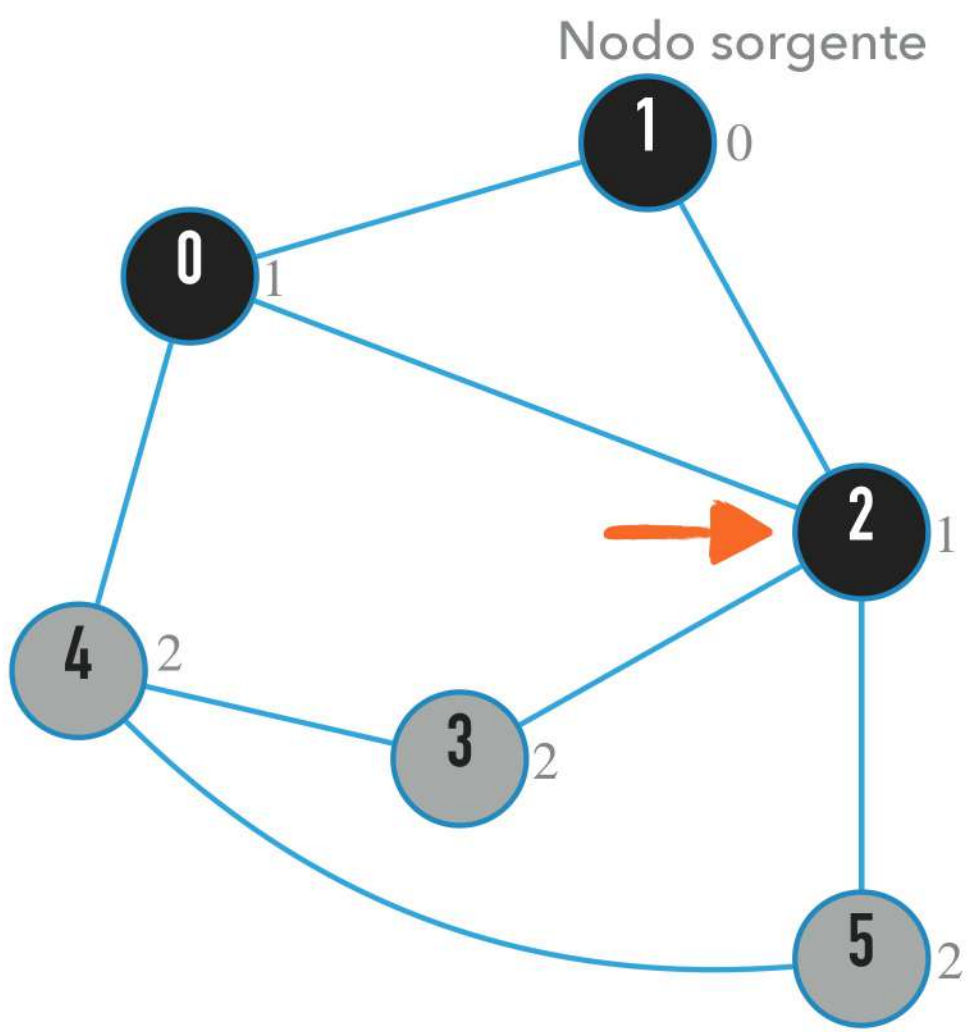
Per ogni vicino, se è bianco lo coloriamo di grigio, aggiorniamo distanza (come distanza$[u] + 1$) e precedessore (u) e lo accodiamo

Coloriamo u di nero

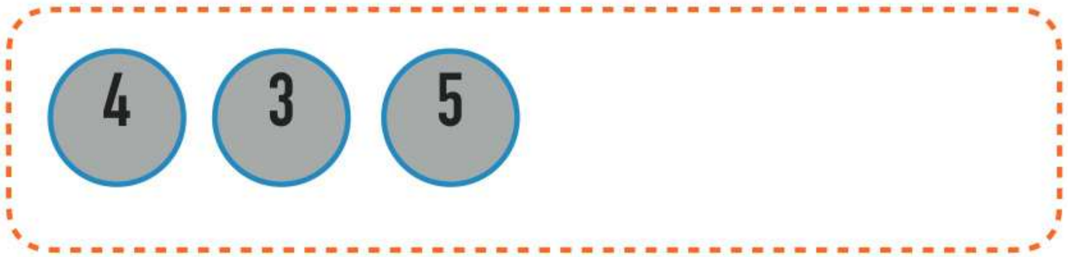
	0	1	2	3	4	5
Distanza	1	0	1	∞	2	∞

	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

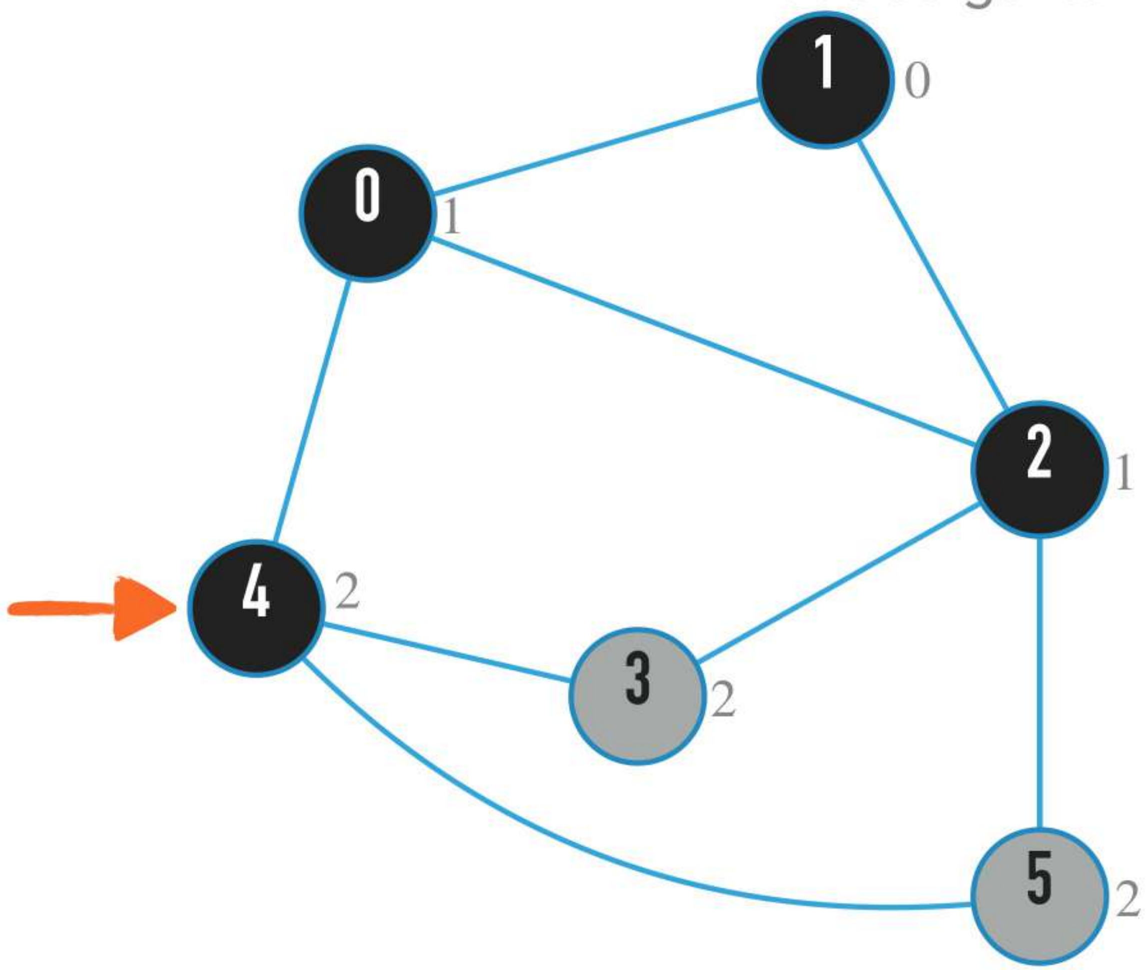
Coloriamo u di nero

	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

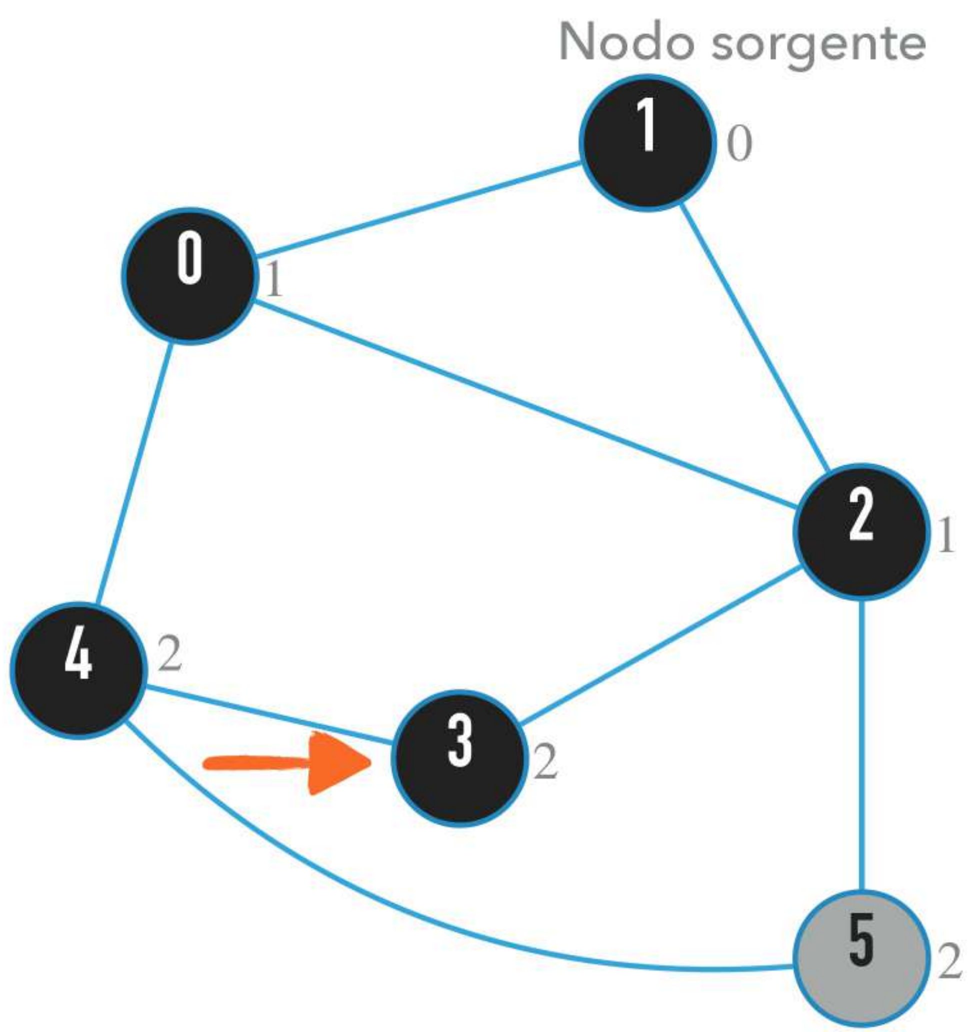
Per ogni vicino, se è bianco lo coloriamo di grigio, aggiorniamo distanza (come distanza$[u] + 1$) e precedessore (u) e lo accodiamo

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

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

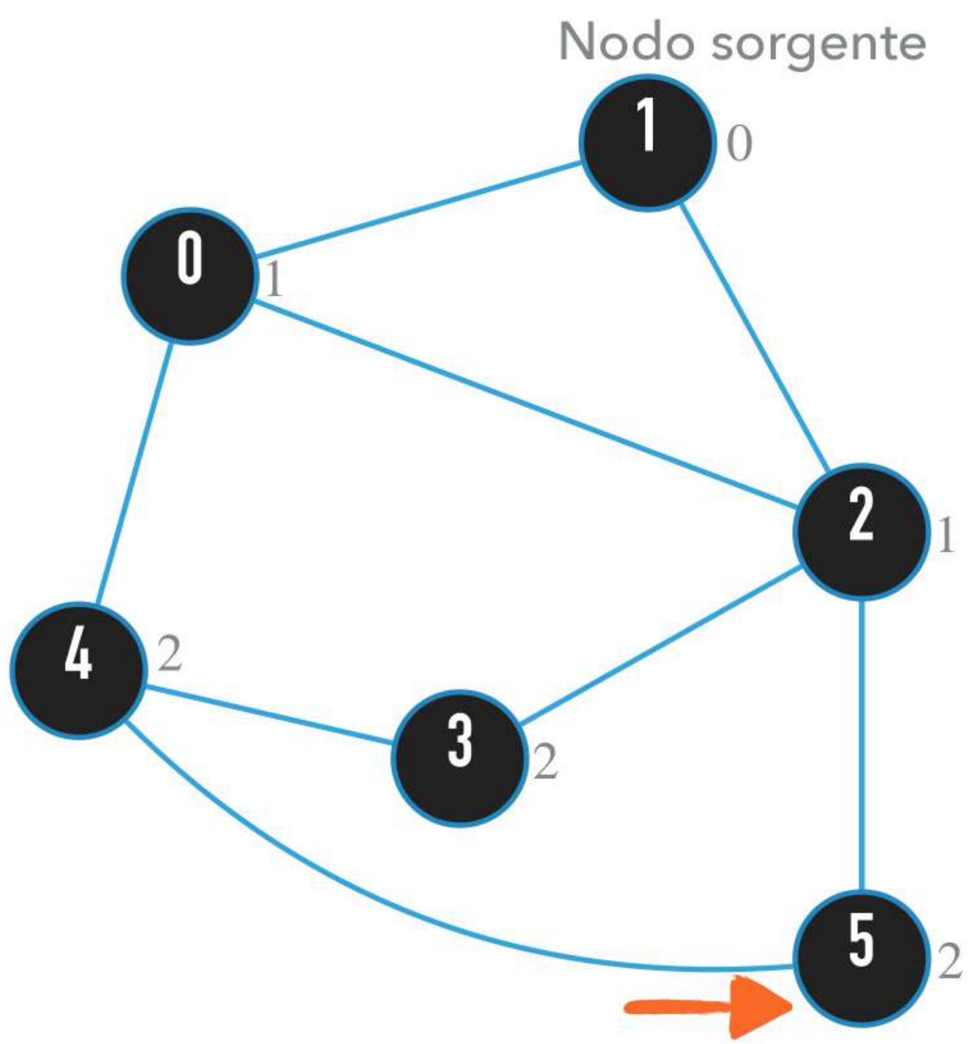
Coloriamo u di nero

	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

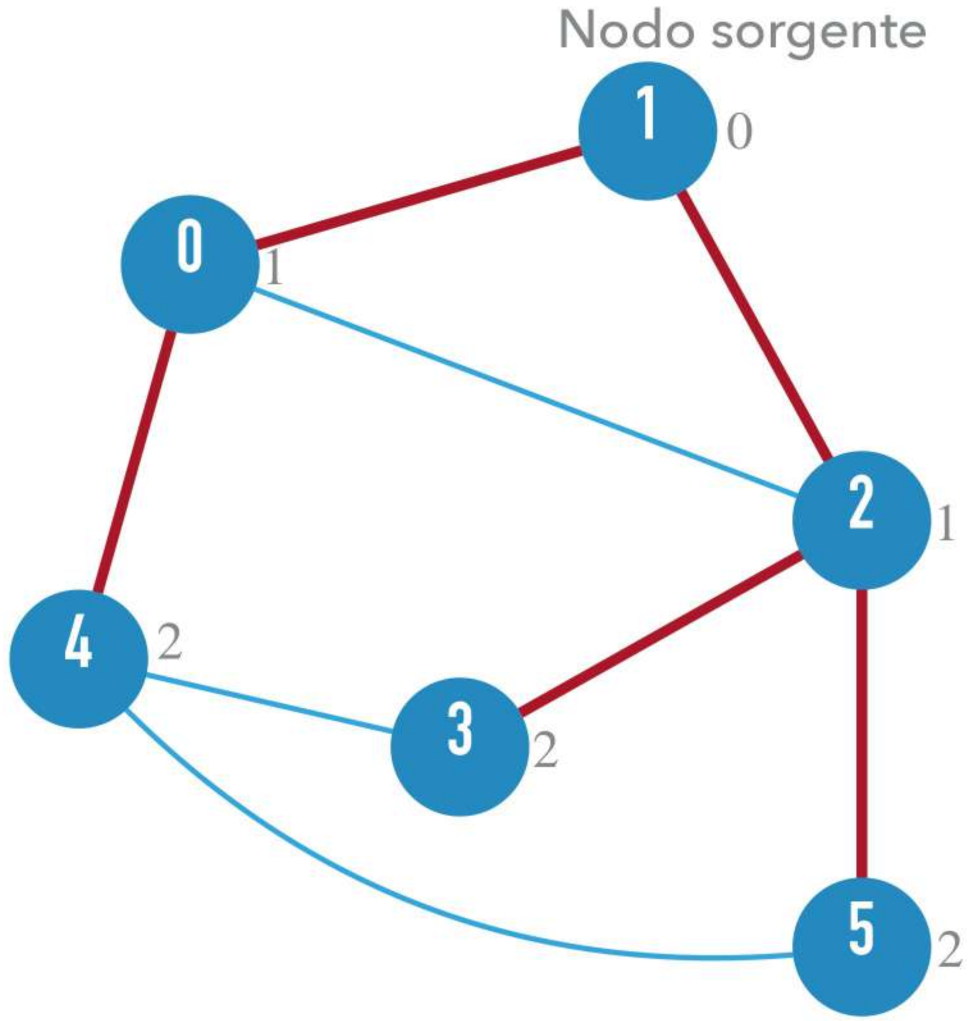
Per ogni vicino, se è bianco lo coloriamo di grigio, aggiorniamo distanza (come distanza$[u] + 1$) e precedessore (u) e lo accodiamo

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 $O(V + E)$

CORRETTEZZA di BFS

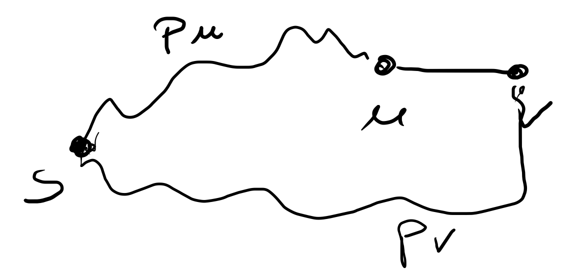
1) $d[u]$ sia la distanza di u da s $= \delta(s,u)$ ← distanza in G.

2) $\pi[u]$ è un vertice in un cammino minimo da s a u
predecessore

3) BFS scopre (colora di NERO) tutti e soli i vertici raggiungibili da s

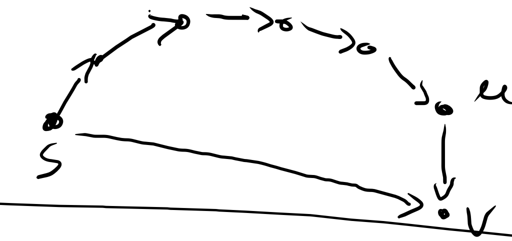
---

Lemma: $(u,v) \in E \Rightarrow \delta(s,v) \leq \delta(s,u) + 1$



Se G è non orientato

$(u,v) \in E \Rightarrow \delta(s,v) \geq \delta(s,u) - 1$



Lemma: Alla fine di BFS(G, s), vale $\forall\, u \in V$

$$d[u] \geq \delta(s,u)$$

dim: Vale per ogni passo di BFS. Per induzione sul numero di operazioni e di inserimento in coda Q

$e=1$ $\quad \forall u \in V,\ u \neq s \quad d[u] = +\infty \geq \delta(s,u)$, $d[s] = 0 = \delta(s,s)$ ✓

$\boxed{e>1}$: ip. induttiva dopo $e-1$ inserimenti vale $\forall v \in V,\ d[u] \geq \delta(s,v)$

Inseriamo il nodo w, mentre eseguiamo u. Dopo l'inserimento

$$d[w] = d[u]+1 \underset{\text{ip induttiva}}{\geq} \delta(s,u)+1 \underset{\text{lemma di prima}}{\geq} \delta(s,w)$$

Lemma: Durante BFS(G,s), sia $Q = \langle v_1, \dots, v_r \rangle$ lo stato della coda ad un qualche punto e $head(Q) = v_1$

1) $d[v_r] \leq d[v_1] + 1$

2) $\forall i \in \{1, \dots, r-1\}$, $d[v_i] \leq d[v_{i+1}]$

dim: per induzione sul numero e di operazioni sulla coda

$e = 1$: $Q = \langle s \rangle$, 1,2 sono banali

$e > 1$:
- CANCELLAZIONE $Q = \langle v_1, \dots, v_r \rangle \longrightarrow Q' = \langle v_2, \dots, v_r \rangle$

2) ✓

1) $d[v_r] \leq d[v_1] + 1 \leq d[v_2] + 1$

- INSERIMENTO $Q = \langle v_1, \dots, v_r \rangle \rightsquigarrow Q' = \langle v_1, \dots, v_r, v \rangle$

$d[v] = d[v_1] + 1 \implies 1$ ✓

$d[v_r] \leq d[v_1] + 1 = d[v] \implies 2$ ✓

TEOREMA (CORRETTEZZA DI BFS)

Sia $G=(V,E)$ un grafo, $s \in V$, e sia $v \in V$. Dopo l'esecuzione di BFS($\cancel{G}$, s)

a) $d[v] = \delta(s,v)$

b) Se $d[v] < \infty$, uno dei cammini minimi da s a v passa per $\pi[v]$

dim: $V_K = \{ v \in V \mid \delta(s,v) = K \}$

per induzione su K, dato $v \in V_K$, $\exists!$ iterazione dell'algoritmo t.c.

1) v viene colorato di GRIGIO
2) $d[v]$ passa da ∞ a K
3) $s \neq v \Rightarrow \pi[v]$ passa da NULL a $u \in V_{K-1}$
4) v viene inserito in coda

$K=0$ $\quad V_0 = \{s\}$ ✓

(Lemma 2)

$K \geq 1$: l'ipotesi vale per $u \in V_{K-1}$. Sia $v \in V_K$, viene inserito in coda dopo i nodi di V_{K-1}

Sia $u \in V_{K-1}$, u t.c. $(u,v) \in E$, il primo nodo inserito in coda

$\Rightarrow$ v è inserito in coda durante il processamento di u $\Rightarrow d[v] = d[u]+1 = K$ e $\pi[v] = u \in V_{K-1}$

SOTTOGRAFO DEI PREDECESSORI ~ ALBERO dei CAMMINI MINIMI

$G = (V, E)$, $s \in V$ source, $BFS(G, s)$

$G_\pi = (V_\pi, E_\pi)$ dove

1) $V_\pi = \{ v \in V \mid d[v] < \infty \}$

2) $E_\pi = \{ (\pi[v], v) \in E \mid v \in V_\pi \setminus \{s\} \}$

1) G_π è connesso (G non orientato) (per induzione su V_k)

2) G_π è un albero (quindi aciclico) perché $|E_\pi| = |V_\pi| - 1$

Esercizio

Sia $G=(V,E)$ un grafo non orientato e connesso

Usando BFS, stabilire se G è ciclico (se contiene almeno un ciclo)