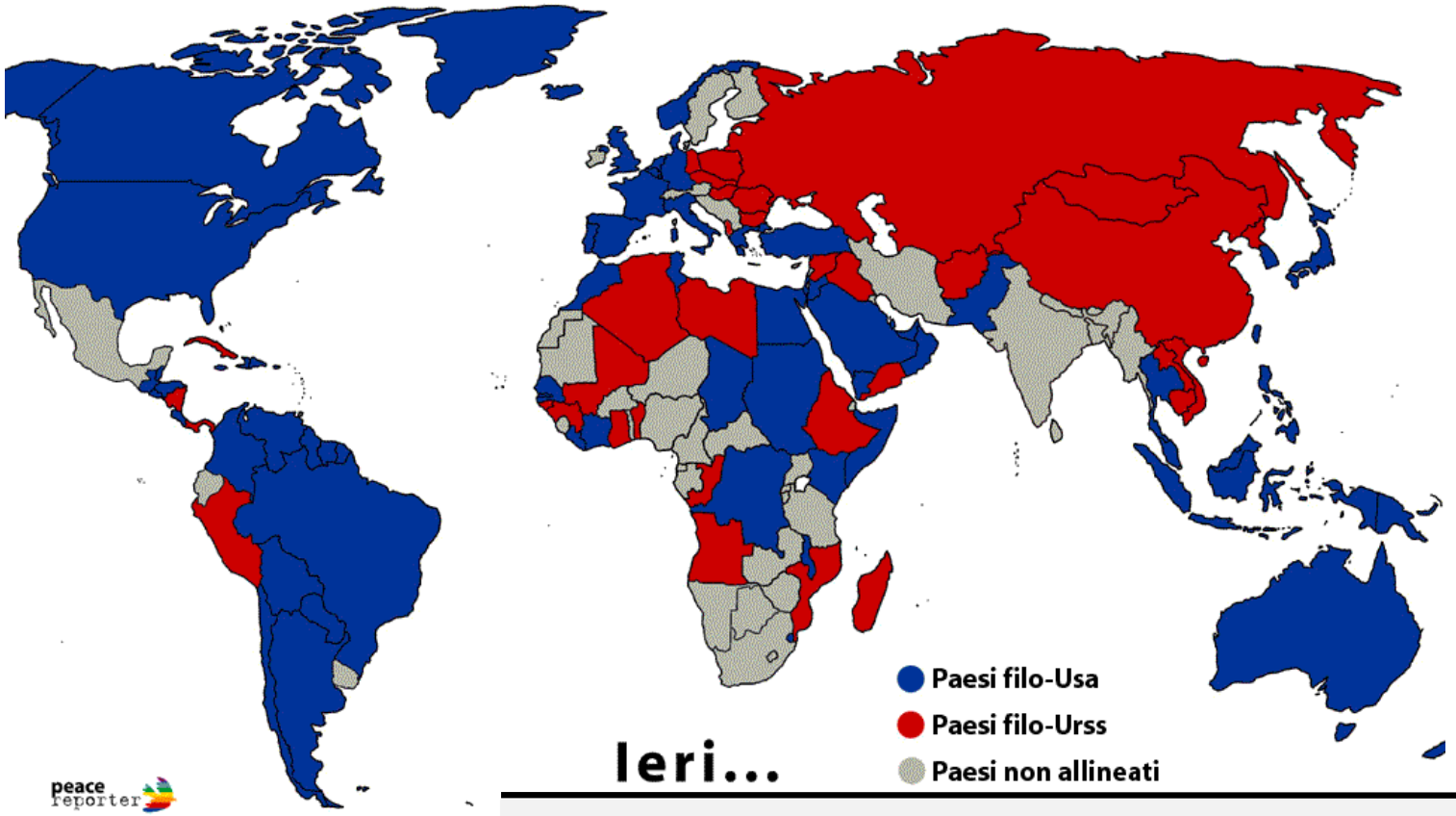
Soluzioni quarto esonero

corso di laurea in **Matematica**

Informatica Generale, **Ivano Salvo**

Lezione **27(a)** **[12/1/24]**

SAPIENZA
UNIVERSITÀ DI ROMA



sfere d'influenza

Sfere di influenza

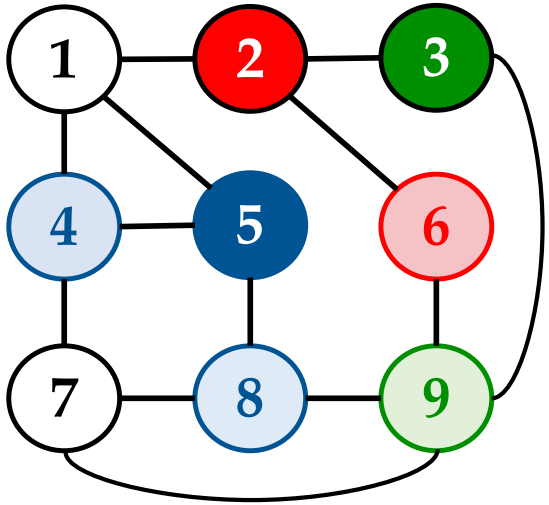
Problema 1: Dato un grafo non orientato $G = (V, E)$ e un insieme $C \subseteq V$ di nodi detti *centri*, diciamo che $v \in V$ appartiene alla sfera di influenza $\mathbf{S}(c)$ di $c \in C$ se e solo se $d(c, v) < d(c', v)$ per ogni $c' \in C$ ($c' \neq c$).
Viceversa, un nodo v è detto di frontiera se esistono almeno due nodi $c, c' \in C$ tali che $d(c, v) = d(c', v)$.

1. Dato il grafo in Fig. 1, determinare le sfere d'influenza dei centri (nodi grigi) e i nodi di frontiera;

Ecco le **sfere d'influenza**: i centri sono i nodi con numero bianco su sfondo colorato.

Le loro sfere di influenza sono i nodi con lo stesso colore.

I nodi di frontiera sono i nodi bianchi: il **nodo 1** dista 1 da 2 e 5, il **nodo 7** dista 2 da 5 e 3.

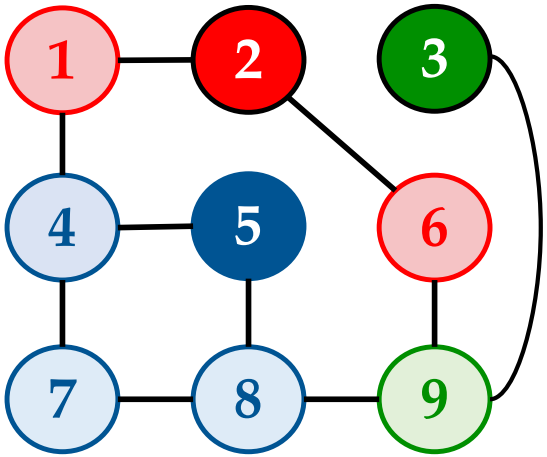


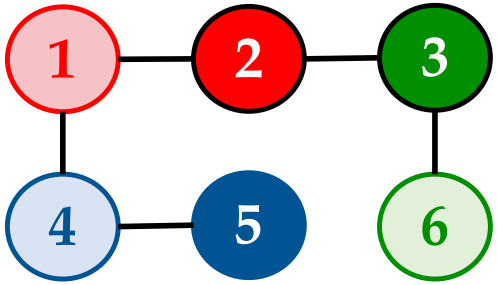
Grafi senza nodi di frontiera

2. disegnare un grafo connesso con un insieme $C \subseteq V$ di centri tali che: - $|C| = 3$; - per ogni $c \in C$, $|\mathbf{S}(c)| > 1$; - e non ci sia nessun nodo di frontiera;

Ovviamente ci sono tantissime soluzioni. Forse la più facile è modificare opportunamente il grafo dell'esempio, rimuovendo/aggiungendo archi in modo da evitare nodi equidistanti da due nodi. Basta rimuovere 2 archi: (7, 9) e (5, 1).

Ovviamente, la soluzione minimale ha almeno 6 nodi [destra] e 5 archi.





Idee per codifica e algoritmo

3. proporre una codifica per classificare i nodi in sfere d'influenza e in nodi di frontiera. Dare un algoritmo che dato un grafo $G = (V, E)$ e un insieme di centri $C \subseteq V$, calcola la classificazione dei nodi di V nella codifica scelta. Dare la complessità in in funzione di $m = |E|$, $n = |V|$ e $k = |C|$.

Un modo facile di codificare la classificazione dei nodi è usare il solito **vettore indicizzato sui nodi**. Chiamiamo il vettore sf e avremo che $sf[u] = v$, se $u \in \mathbf{S}[v]$ e $sf[u] = -1$ (o qualsiasi altro valore impossibile) se u è un nodo di frontiera.

L'algoritmo può seguire due strategie:

1. **per ogni centro $c \in C$**, si fa una **BFS radicata in c** e si calcola il vettore delle distanze dc da c. Si mantiene un vettore di distanze minime dai centri finora analizzati. Per ogni nodo i, se $dc[i] < d[i]$, allora $sf[i]$ diventa c e aggiorno $d[i]$ con $dc[i]$, mentre se $dc[i] = d[i]$ allora i è (per ora) un nodo di frontiera e $sf[i]$ diventa -1.

2. **per ogni nodo $v \in V$** si fa una **BFS radicata in v** che si ferma non appena si esplora **tutto il livello in cui si trova il primo centro**: se c'è un unico centro c nell'ultimo livello, $sf[u]$ diventa c (per sempre), altrimenti v è un nodo di frontiera e $sf[v]$ diventa -1.

Algoritmo 1

La strategia **1** ha il vantaggio di **poter riutilizzare una normale BFS** (completa del grafo). La complessità è $\Theta(k(n+m))$, in quanto il riaggiornamento dei vettori d e sf si fanno in tempo $\Theta(n)$.

Anche la complessità in spazio è $\Theta(n)$ (i vettori dc e sf .

```
def sfereDInfluenza(G=(V,E), C):
    sf = init(|G.V|, -1)
    dc = init(|G.V|, +∞)
    forall c ∈ C:
        d = bfs(G, c)
        aggiornaSfere(d, dc, sf, c)
    return sf
```

```
def aggiornaSfere(d, dc, sf, c):
    for i = 1 to len(d):
        if d[i] < dc[i]:
            dc[i] = d[i]
            sf[i] = c
        if d[i] == dc[i]:
            sf[i] = -1
```

Algoritmo 2

La strategia **2** ha il vantaggio di essere **più efficiente** se i **centri sono molti** e la **distanza di ogni nodo da un centro** è molto **piccola**.

Occorre tuttavia (al fine di renderla efficiente) mettere **un po' mano alla BFS**.

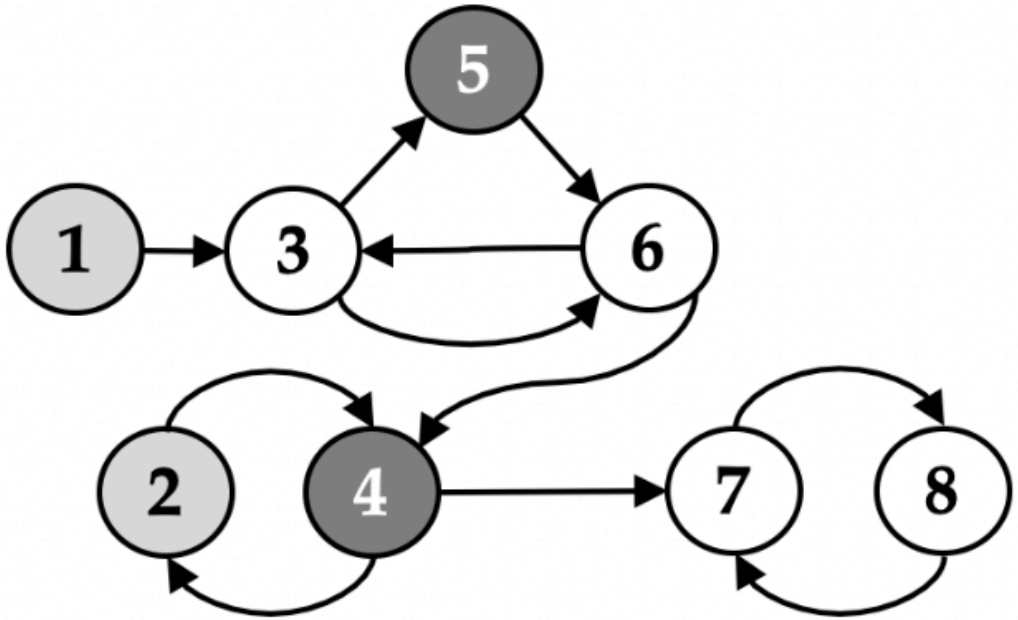
```
def dist(G, s, t):
    Q , d = newQueue(), init(|G.V|, -1)
    d[s], dist, dc = 0, 0, +∞
    enqueue(Q, s)
    while not isEmpty(Q) and dist < dc:
        u = dequeue(Q)
        forall v ∈ G.adj(u):
            if d[v] < 0:
                enqueue(Q, v)
                d[v], dist = d[u] + 1, d[u] + 1
                if v ∈ C and d[v] < dc:
                    dc, sf[v] = d[v], c
                if v ∈ C and d[v] == dc:
                    sf[v] = -1
    return sf
```

Complessità e confronti

La complessità del primo algoritmo è sempre $\Theta(k(n+m))$ mentre il secondo è $\mathcal{O}(n(n+m))$ in quanto fa n BFS, ma molte di queste potrebbero fermarsi prima.

Quindi: sicuramente $k \leq n$, e se k è molto minore di n, probabilmente il primo algoritmo è da preferirsi.

Il secondo algoritmo è da preferirsi se i centri sono molti ed è noto che tutti i nodi sono "vicini" ad almeno un centro.



Passeggiate Infinite

Passeggiate infinite sui DAG…

Problema 2: In un grafo orientato finito c'è un insieme *I* di nodi iniziali (grigio chiari in Fig. 2) e un insieme *F* di nodi finali (grigio scuri in figura). Consideriamo le proprietà: - (**E**) esiste in *G* una passeggiata infinita che parte da un nodo iniziale e passa infinite volte su un nodo finale; - (**A**) ogni passeggiata infinita in *G* che parte da un nodo iniziale passa infinite volte su un nodo finale.

1. In un DAG può valere (**E**)? Ed (**A**)?

In un **grafo finito**, possiamo avere una **passeggiata infinita** solo **percorrendo un ciclo**, o più in generale **passeggiando all'interno di una componente fortemente connessa *non banale***, cioè una componente fortemente connessa che non sia un singolo nodo (senza neanche un cappio).

Di conseguenza, in un **DAG** non esiste mai nessuna passeggiata infinita e quindi la proprietà (**E**) **non vale mai**.

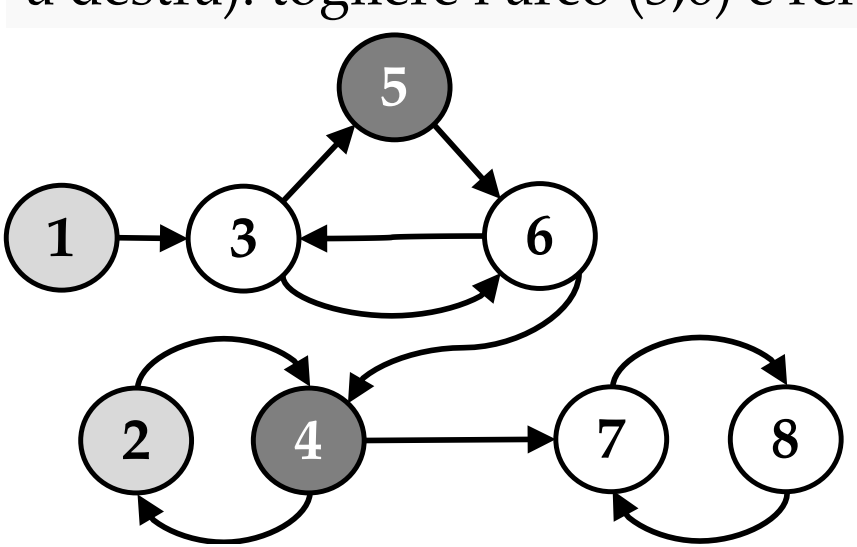
Tuttavia **vale sempre** (**A**) perché è una **proprietà universale sull'insieme vuoto** di passeggiate infinite.

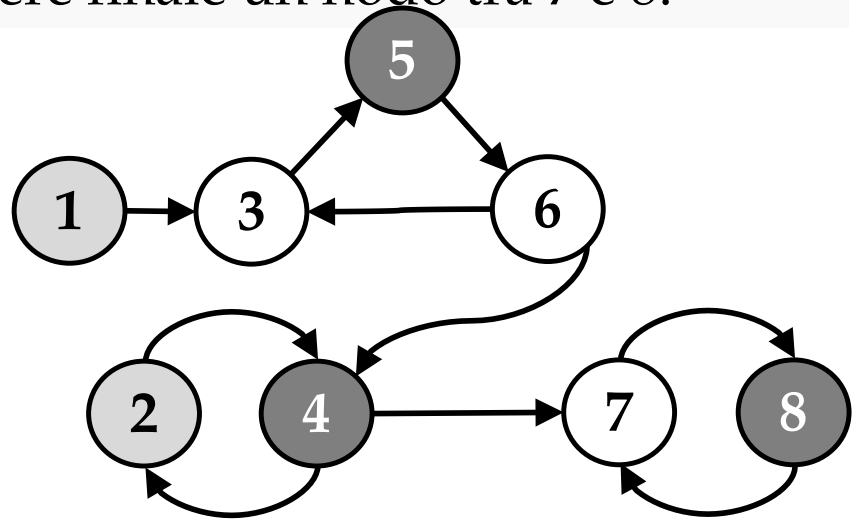
Esempi di proprietà (E) & (A)

2. Dire perché nel grafo in Fig. in basso a sinistra vale (**E**) ma non vale (**A**). Rendere finale un nodo e togliere un arco affinché valga (**A**).

Nel grafo in figura ci sono diverse passeggiate infinite che passano infinite volte per un nodo finale: ad esempio semplicemente la sequenza 2, 4 ripetuta all'infinito (notazione: **(2,4)$^{\omega}$**). Ma anche la sequenza **1, (3, 5, 6)$^{\omega}$**. Quindi (**E**) vale.

D'altro canto, ci sono passeggiate infinite che **entrano nella componente fortemente connessa {7, 8}** che incontrano un numero finito di nodi finali, ma anche la passeggiata **1, (3,6)$^{\omega}$**. Detto questo, dovrebbe essere chiaro cosa fare per fare in modo che (**A**) valga (fig. a destra): togliere l'arco (3,6) e rendere finale un nodo tra 7 e 8.





Condizioni che assicurano (E) & (A)

3. Determinare delle condizioni affinché valga (**E**) e delle condizioni per (**A**)

Affinché valga (**E**) è necessario che ci sia **almeno una componente fortemente connessa** *C non banale* che **contenga un nodo finale**. E che *C* sia **raggiungibile da almeno un nodo iniziale**.

(**A**) è un po' più complicata: in prima battuta si potrebbe pensare che sia sufficiente che tutte le componenti fortemente connesse non banali raggiungibili da un nodo iniziale contengano un nodo finale: ma questo non basta, come visto nell'esempio. È necessario che **tutti i cicli** contengano un **nodo finale**.

Quest'ultima proprietà è decisamente poco maneggiabile: per avere algoritmi efficiente per verificare (**A**) è meglio verificare **la sua negazione** e cercare quindi se esiste una passeggiata infinita che **non ripete infinite volte un nodo finale**, riconducendosi a verificare (**E**) su un grafo opportunamente modificato.

Algoritmo per verificare (E)

4. Descrivere un algoritmo che dati un grafo G, e $I, F \subseteq V$, determina se è soddisfatta la proprietà (**E**). Si può usare lo stesso algoritmo per verificare (**A**)?

Visto quanto sopra, per verificare (**E**) si procede come segue:

1. si decompone G nelle sue **componenti fortemente connesse**;

2. nel grafo condensato G^{SCC}, si marcano le componenti connesse $I' \subseteq V^{SCC}$ che contengono stati iniziali e quelle *non banali* $F' \subseteq V^{SCC}$ che contengono stati finali.

3. si verifica (con una qualsiasi visita) se ***F'* è raggiungibile da *I'***.

Tutte le operazioni sono dominate dal calcolo delle componenti fortemente connesse, ed è quindi lineare $\theta(n+m)$.

Per verificare che una componente fortemente connessa sia **non banale** è sufficiente verificare che abbia **almeno 2 nodi** oppure se ha un **unico nodo *u***, è necessario verificare che **esista il cappio** $u \rightarrow u$.

Algoritmo per verificare (A)

L'idea è **verificare la negazione**, cercando una **passeggiata infinita** che attraversa solo un **numero finito di nodi finali**.

A tale scopo, preso si può procedere come segue:

1. Si costruisce ***G'*** in cui si **rimuovono tutti i nodi finali**. G' è il sotto-grafo indotto dall'insieme di nodi $V \setminus F$.
2. Si calcolano le **componenti fortemente connesse** non banali in ***G'***.
3. Se ci sono, si vede se sono raggiungibili dagli stati iniziali **in *G*** (**non in *G'***). Infatti, qualche **nodo finale** potrebbe essere **essenziale per la raggiungibilità**!

Osservate che i passi **2-3** sono (circa) la verifica se vale (**E**), prendendo come F' tutti i nodi **non finali**, cioè $F' = V \setminus F$.

Osservate che i due algoritmi forniscono facilmente esempi (per (**E**)) e controesempi (per (**A**)).

That's all Folks!

corso di laurea in **Matematica**

Informatica Generale, **Ivano Salvo**

Lezione **27(a)** **[12/1/24]**

SAPIENZA
UNIVERSITÀ DI ROMA