Problemi su Grafi

corso di laurea in **Matematica**

Informatica Generale, Lezione **25(b)**

Ivano Salvo

SAPIENZA
UNIVERSITÀ DI ROMA

Capire le SCC [1]

Esercizio 22.5-1 [Cormen]

Come cambia il numero delle componenti fortemente connesse di un grafo orientato G se viene aggiunto un nuovo arco $u \rightarrow v$?

Soluzione: Distinguiamo due casi.

1) u e v **appartengono** alla stessa componente connessa: in questo caso esistono già due cammini, $u \rightsquigarrow v$ da u a v e $v \rightsquigarrow u$ da v a u. L'aggiunta del nuovo arco non modifica quindi le componenti connesse di G.

2) u e v **non appartengono** alla stessa componente connessa: qui distinguiamo due sotto-casi:

2a) **esiste** un cammino da $v \rightsquigarrow u$ da v a u: questo implica che c'è un cammino da $C(v)$ a $C(u)$ nel grafo condensato G^{SCC} e quindi creiamo un ciclo che fa collassare tutte le componenti connesse nel cammino $C(v)$ a $C(u)$ in G^{SCC}.

2b) **non esiste** un cammino da $v \rightsquigarrow u$: in questo caso non si generano nuovi cicli e quindi il numero rimane uguale.

In generale $|G'^{SCC}| \leq |G^{SCC}|$.

ot

Esercizio 3

Per un grafo diretto G, diciamo che un arco da u a v è *interno* se u e v appartengono alla stessa componente fortemente connessa e altrimenti diciamo che è *esterno*. In relazione ad una qualsiasi DFS, rispondere alle seguenti domande:

1) un arco in avanti può essere esterno?

2) un arco di attraversamento può essere interno?

3) un arco di attraversamento può essere esterno?

4) un arco all'indietro può essere esterno?

Per ognuna, in caso affermativo esibire un esempio e altrimenti dimostrare l'impossibilità.

Soluzione

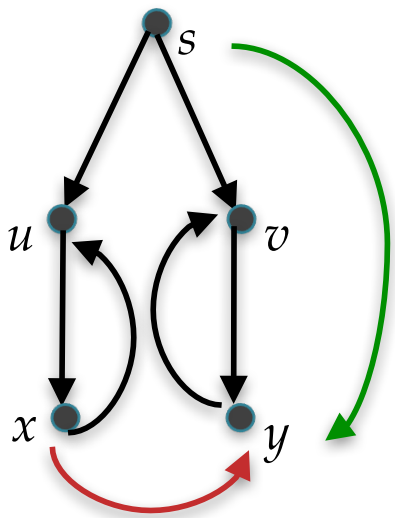
Ricordiamo che in un grafo **non orientato**, in una DFS possono esserci solo archi all'indietro e archi dell'albero di visita. Invece in un grafo **diretto** possiamo incontrare anche archi di **attraversamento** $u \rightarrow v$ (quando v viene visitato prima e u non è raggiungibile da v), oppure archi all'**avanti** $u \rightarrow v$ (quando ho più cammini diretti da u a v).

1) **SI**. Vedi **arco verde** sotto.

2) **NO**. Tutti gli archi interni a una componente fortemente connessa collegano nodi antenati uno dell'altro. Non ci possono essere per definizioni archi di attraversamento.

3) **SI**. Vedi **arco rosso** sotto.

4) **NO**. Un arco all'indietro ovviamente genera un ciclo che deve appartenere alla stessa componente fortemente connessa.



Esercizio 3 Presenza

[Compito 21/6/2022]

Esercizio 3 (10 punti) Ricordiamo che in un grafo, la distanza tra due nodi u e v è il numero minimo di archi di un cammino semplice da u a v.

Progettare un algoritmo che dati un grafo G e due nodi s e t di G costruisca un vettore $eqDist$ indicizzato sui nodi di G tale che $eqDist[u]$ vale TRUE se u è equidistante da s e t e FALSE altrimenti. Dell'algoritmo fornite:

1. la descrizione a parole;
2. lo pseudocodice;
3. l'analisi del costo computazionale, nel caso in cui G sia memorizzato con liste di adiacenza.

Soluzione naif

Supponendo di avere una funzione $dist(G, u, v)$, il problema in esame ha una soluzione banale:

```
fun eqDist(grafo G, nodi s, t):
    allocaVettoreDiBooleani(eqdist, n)
    forall u ∈ G.V do
        if dist(G, u, s) = dist(G, u, t)
            then eqDist[u] = TRUE
            else eqDist[u] = FALSE
    return eqDist
```

Cosa devo fare per trovare la distanza tra due nodi u e v?
E quanto costa?

Devo fare una **visita in ampiezza** radicata in u (o v) e fermarmi non appena trovo v, avendo avuto cura di contare i livelli visitati.

Nel caso pessimo, al solito, la visita costa $\mathcal{O}(m+n)$ e quindi il programma sopra ha costo $\mathcal{O}(nm+n^2)$

Distanza tra due nodi

Seguendo questo approccio, scriviamo la visita, osservando che possiamo fermarci non appena troviamo il secondo nodo.

```
fun dist(grafo G, nodi s, t):
    Q = emptyQueue();
    allocaVettoreDiInteri(dist, n)
    forall u ∈ G.V do dist[u] ← +∞
    dist[s] ← 0
    enqueue(Q, s)
    while notEmpty(Q) do
        u = dequeue(Q)
        forall v adiacente a u do
            if dist[v] < +∞
                then dist[v] ← dist[u] + 1
                     enqueue(Q, v)
            if v=t then return dist[v]
    return +∞
```

*osservare che l'informazione in **marked** è sussunta da **dist** e che **non serve** produrre **l'albero di visita***

È una normale visita, mi fermo non appena trovo *t*.

Distanza tra un nodo e tutti gli altri

Il programma precedente, a ben vedere, però, può restituire la distanza di tutti i nodi da s, senza avere un maggiore costo computazionale.

```
fun dist(grafo G, nodi s, t):
    Q = emptyQueue();
    allocaVettoreDiInteri(dist, n)
    forall u ∈ G.V do dist[u] ← +∞
    dist[s] ← 0
    enqueue(Q, s)
    while notEmpty(Q) do
        u = dequeue(Q)
        forall v adiacente a u do
            if dist[v] < +∞
                then dist[v] ← dist[u] + 1
                     enqueue(Q, v)
/*stavolta restituisco il vettore di distanze */
    return dist
```

Soluzione Finale

Usando la funzione *allDist*, posso calcolare tutte le distanze da s e tutte le distanze da t, in tempo $\mathcal{O}(m+n)$. Poi in tempo $\mathcal{O}(n)$ verifico quali sono i nodi che hanno queste due distanze uguali.

```
fun eqDist(grafo G, nodi s, t):
    distS = allDist(G, s)
    distT = allDist(G, t)
    forall u ∈ G.V do
        if distS[u] = distT[u]
            then eqDist[u] = TRUE
            else eqDist[u] = FALSE
    return eqDist
```

Morale: spesso nei grafi, molti problemi costano almeno una visita, ma magari una visita ci può permettere di calcolare molta più informazione utile ai fini della soluzione del problema.