Progettazioni di Algoritmi
Esercitazioni

Ivano Salvo

Corso di Laurea in Informatica

SAPIENZA
UNIVERSITÀ DI ROMA

Esercitazione **1D**, 17 aprile 2020

Esercizi 1D(a)

BFS

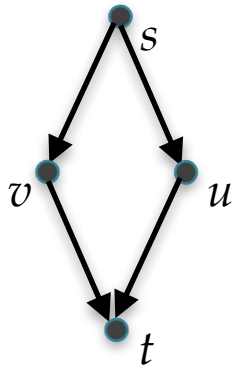
Capire la BFS [1]

Esercizio 22.2-5 [Cormen]

Dimostrare che in una BFS ***1****) il valore u.d è indipendente dall'ordine delle liste di adiacenza.* ***2****) Viceversa l'albero di visita può dipendere dall'ordine in cui appaiono i nodi nelle liste di adiacenza.*

Soluzione: **1**) Ricordiamo che l'attributo $u.d$ rappresenta la distanza del nodo u dalla radice s della BFS. O se preferite il livello del nodo nell'albero di visita BFS (la radice s a livello 0). Questa proprietà dipende dalla correttezza della BFS, ma si può provare direttamente, ad esempio per induzione su d, provando appunto che ogni nodo sta al livello uguale alla sua distanza dalla radice.

2)



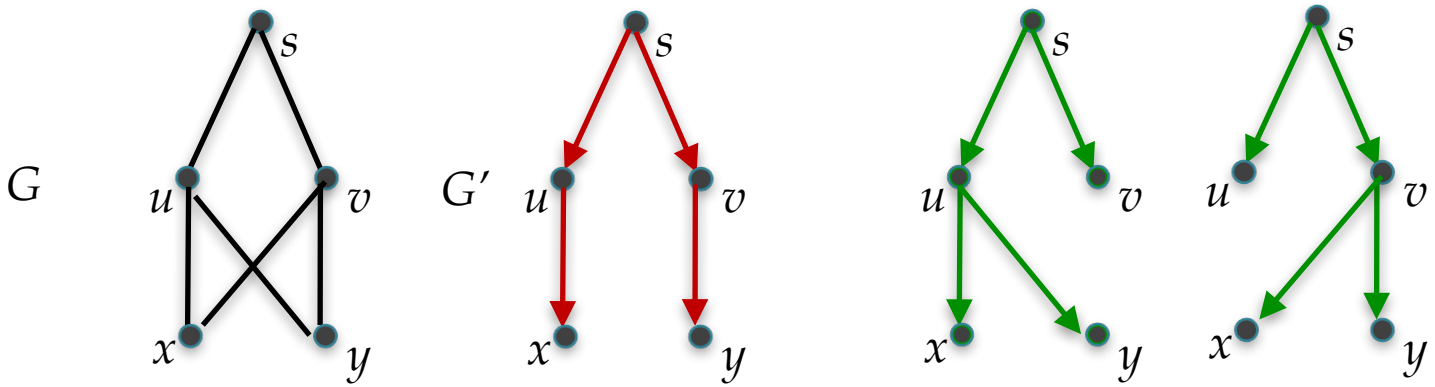
Ovviamente se v viene prima di u nella lista di adiacenza di s, l'albero di visita conterrà l'arco (v, t) altrimenti l'arco (u, t).

Capire la BFS [2]

Esercizio 22.2-6 [Cormen]

Fare un esempio di un grafo G=(V,E), una sorgente s ∈ V e un insieme di archi E′⊆E tali che per ogni vertice in v ∈ V l'unico cammino semplice da s a v sia minimo in G′=(V, E′), ma G′ non sia il risultato di nessuna BFS (indipendentemente dall'ordinamento delle liste di adiacenza).

Soluzione:



possibili BFS radicate in s

BFS per calcolare distanze

Esercizio

Descrivere un algoritmo che dato un grafo G non diretto e connesso e due nodi u e v trova, in tempo $\mathcal{O}(n+m)$ tutti i nodi che hanno la stessa distanza da u e v.

Soluzione: Sappiamo che in una BFS possiamo etichettare ciascun nodo con la sua distanza dalla radice della BFS.

In tempo $\mathcal{O}(n+m)$ possiamo fare una BFS radicata in u e decorare ogni nodo x con l'attributo $x.d_u$ che rappresenta la sua distanza da u.

Allo stesso modo, con una BFS radicata in v decoriamo ogni nodo x con l'attributo $x.d_v$ che rappresenta la sua distanza da v sempre con tempo di esecuzione $\mathcal{O}(n+m)$.

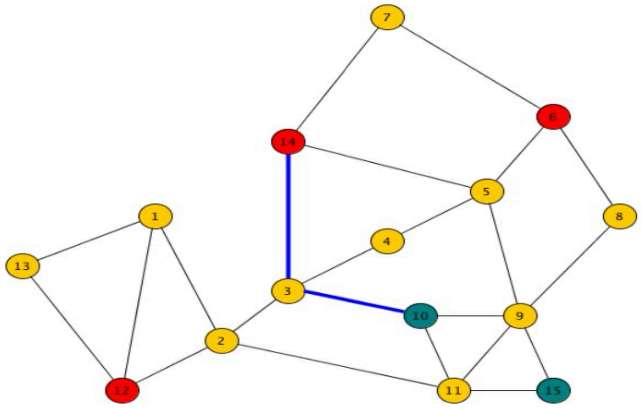
Infine, in tempo $\mathcal{O}(n)$ possiamo scorrere la lista dei nodi e selezionare tutti i nodi x per cui $x.d_u = x.d_v$.

Distanze tra insiemi di nodi

Esercizio 9

Dato un grafo G e due sottoinsiemi V_1 e V_2 dei suoi vertici si definisce distanza tra V_1 e V_2 la distanza minima per andare da un nodo in V_1 ad un nodo in V_2. Nel caso V_1 e V_2 non sono disgiunti allora il valore 0.
Descrivere un algoritmo che, dato un grafo G e i due sottoinsiemi dei vertici V_1 e V_2 calcola la loro distanza. L'algoritmo deve avere complessità $O(n+m)$.

- Ad esempio per il grafo G in figura, dove i nodi dell'insieme A sono in verde mentre i nodi dell'insieme B sono in rosso, la distanza tra i due insiemi è 2 come evidenziato dal cammino in blu.



Soluzione: modificare la BFS

Una soluzione naïf consiste nel fare **una BFS in ogni nodo di V_1 e fermarsi non appena si trova un nodo di V_2** e fare il minimo tra tutte le distanze così trovate. Il tempo di esecuzione in questo caso sarebbe nell'ordine di $\mathcal{O}(n(n+m))$, quindi **quadratico**, perché devo fare (alla peggio) $\mathcal{O}(n)$ BFS. (La cardinalità di V_1 e V_2 può essere nell'ordine della cardinalità di V).

Se avete studiato alcuni algoritmi su grafi avrete scoperto che spesso però si riescono a fare molti conti in **una sola visita** del grafo.

In questo caso, possiamo modificare la BFS come segue: mettiamo nella coda **tutti i nodi di V_1** con attibuto d=0. E poi procediamo come in una normale BFS. Dopo che abbiamo estratto e processato tutti i nodi di V_1 avremmo trovato tutti i nodi di G che distano 0 da V_1 (quelli originarimente in V_1) e tutti quelli che distano 1 da V_1. E così via finchè non incontriamo il primo nodo di V_2: la sua distanza sarà la distanza di V_2 da V_1. Il tutto con una sola BFS!

Calcolare distanze: vettore dei padri

Esercizio

Dare lo pseudocodice di un algoritmo che preso in input un grafo diretto connesso G, un suo nodo s, un vettore dei padri P relativo a una BFS da s in G e un arco $u \rightarrow v$ di G e determina se la rimozione dell'arco $u \rightarrow v$ non cambia le distanze da s. L'algoritmo deve avere complessità $\mathcal{O}(n)$.

Soluzione: La rimozione di un arco $u \rightarrow v$ **non** modifica mai le distanze se tale arco non è nell'albero di visita. Se invece lo è, occorre verificare se esiste un altro nodo t tale che $d(s, t)=d(s, u)$ tale che l'arco $t \rightarrow v$ sia in G.

Per fare questo controllo è necessario tuttavia calcolarsi le distanze $d(s, x)$ partendo dal vettore dei padri per ogni nodo x. Ovviamente la versione naïf di ricalcolare ogni volta le distanze da ogni x non funziona perché alla peggio è $\mathcal{O}(n^2)$. Tuttavia, ancora una volta occorre chiedersi se non sia possibile fare tutto in un'unica passata, eventualmente approfittando del fatto di aver già calcolato le distanze di altri nodi.

È una sorta di **programmazione dinamica** (altro argomento del corso).

Calcolare distanze: vettore dei padri

Idea: usiamo un vettore d indicizzato sui nodi per calcolare le distanze da s. All'inizio mettiamo un valore abbastanza alto in $d[i]$ (scrivo ∞). Per calcolare un certo nodo u, controllo il padre $p[u]$: se $d[p[u]]$ è disponibile, pongo $d[u]= d[p[u]] +1$. Altrimenti continuo ricorsivamente fino alla peggio alla radice. Ogni distanza viene calcolata una sola volta in $\mathcal{O}(n)$.

```
def computeDistance(p, d, i):
    if (d[i]==∞)
            if (p[i]=i) d[i]=0;
               else d[i]=1+computeDistance(p, d, p[i])
    return d[i]

def modifyDistance(G, p, n, u, v):
    if p[v]≠u return FALSE
    forall i ∈ [1,n] do d[i]=∞
    computeDistance(p, d, i)
    forall i ∈ [1,n] do
        if (d[i]=d[u] && i→v ∈ E) return FALSE
    return TRUE
```

Dimostrare che questa funzione è $\mathcal{O}(n)$! Osservate che ogni $d[i]$ viene assegnato una volta sola e ogni elemento di $p[i]$ percorso al più 2 volte.

Esercizi 1D(b)

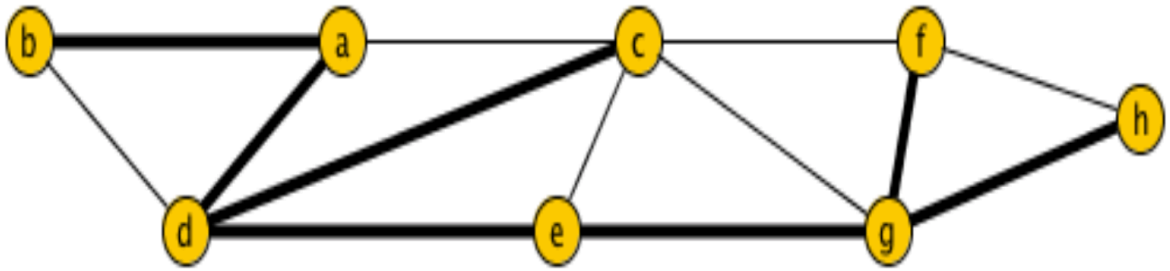
DFS

1: DFS su grafo NON orientato

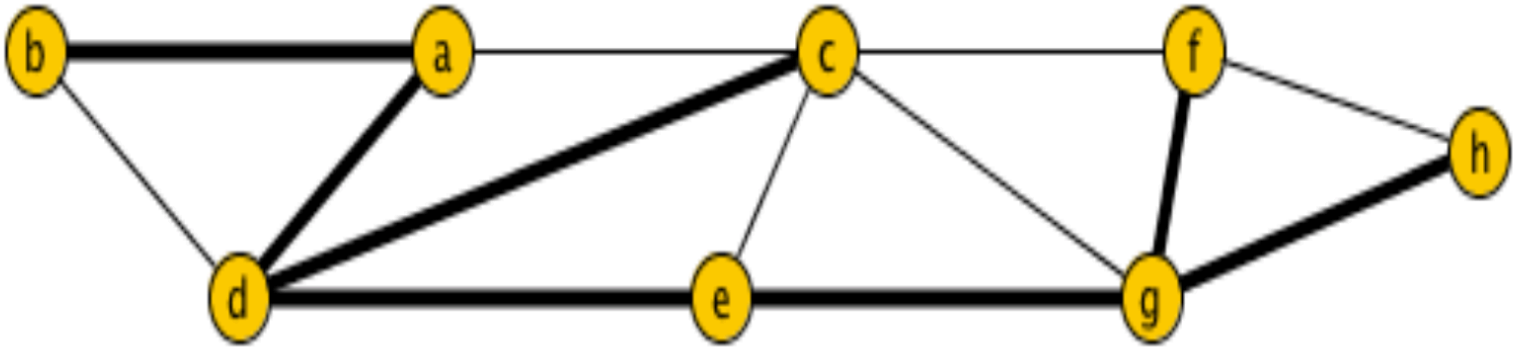
Esercizio 1:

Si consideri il grafo G nella figura qui sotto e l'albero T formato dagli archi marcati.

- L'albero T può essere stato prodotto da una DFS? In caso affermativo, esibire una rappresentazione di G tramite liste di adiacenza in grado di produrre T e specificare il nodo da cui parte la visita e il numero di archi all'indietro che si ottengono a seguito della visita.



Soluzione: occorre esaminare radici



Ad esempio: **b** può essere la radice di una DFS? **NO**.

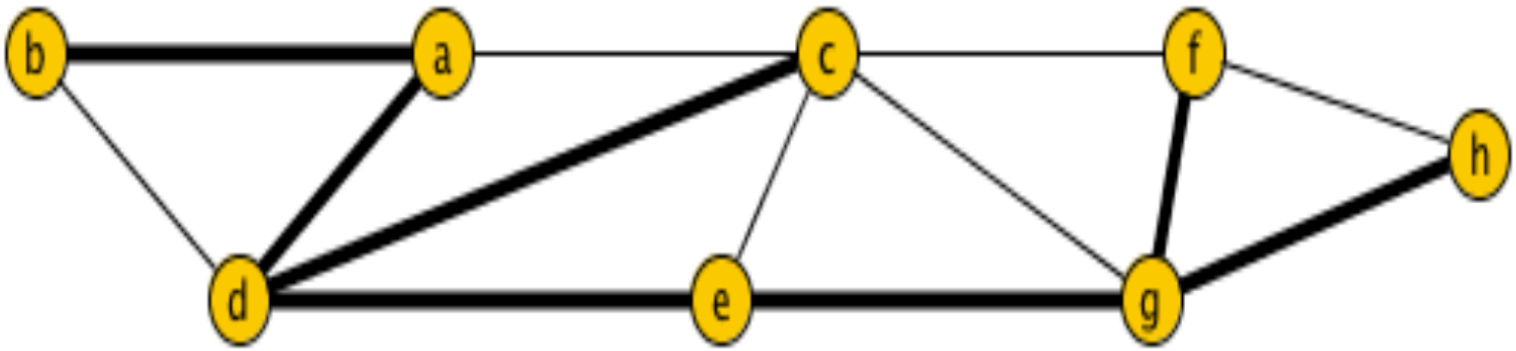
Infatti, la sequenza **b→a→d** è legittima, ma partendo da **d** i nodi **c**, **e** non sarebbero visitati entrambi da **d**, ma almeno uno da un suo "successore" (nell'albero di visita DFS).

Questo esclude anche **a** e **d** come possibili radici (per lo stesso motivo).

c sembra un buon candidato: **c→d→a→b** è decisamente una possibile sequenza di una DFS, e poi ritornando a **d** si può proseguire con **d→e→g→f** tuttavia a questo punto, **h** dovrebbe essere visitato da **f** e non da **g**.

Anche **f** è un buon candidato, ma ci si arena su **c** (che dovrebbe essere raggiunto da **a** e non da **d** verificare)

Esercizi supplettivi



Esercizio 1: Modificare l'albero di visita (togliendo un arco dall'arco dall'albero sopra e aggiungendone un altro) dimodoché possa effettivamente essere un albero di visita DFS.

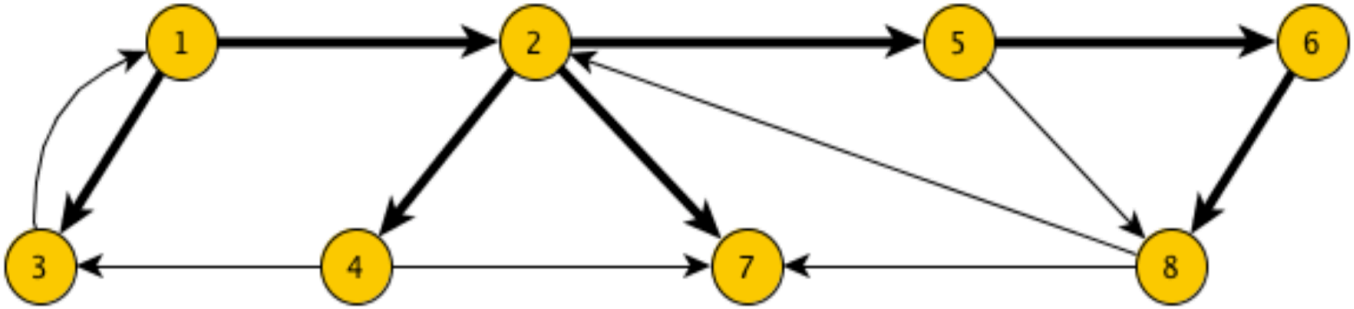
Esercizio 2: In tal caso, scrivere la rappresentazione del grafo in liste di adiacenza che ha prodotto tale albero di visita.

2. DFS su grafo diretto

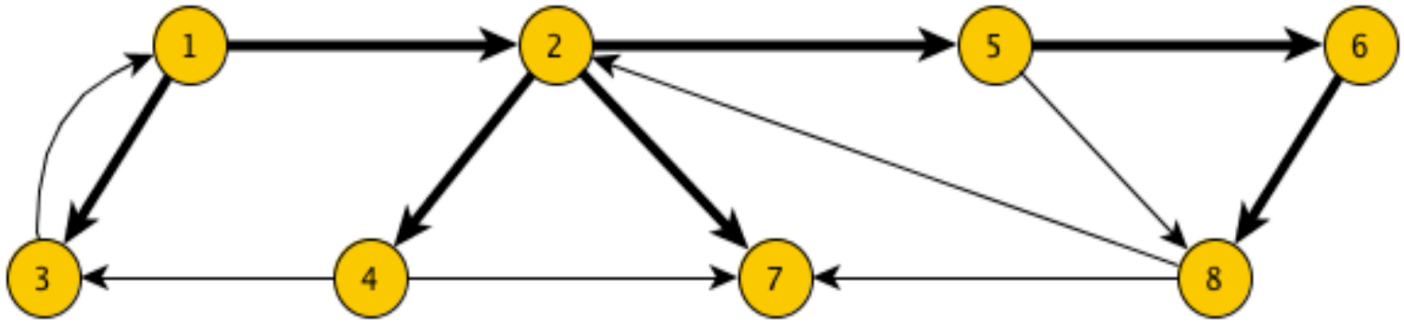
Esercizio 2:

Si consideri il grafo diretto G nella figura qui sotto e l'albero (arborescenza) T formato dagli archi marcati.

- L'albero T può essere stato prodotto da una DFS? In caso affermativo, esibire una rappresentazione di G tramite liste di adiacenza in grado di produrre T e specificare il numero di archi all'indietro, in avanti e di attraversamento che si ottengono a seguito della visita.



Soluzione: occorre esaminare radici



Cominciamo dal nodo **1**: effettivamente si può visitare subito **3** (da dove non si può andare da nessun'altra parte), poi **2→5→6→8**, e qui ci si blocca perché da **8** si va a **7**.

Osservando che da **7** non si può andare da nessuna parte, **7** deve essere visitato prima di **8** e di **4**. Quindi ecco un possibile ordine:

1→3, poi ricominciando da **1**, **1→2→7** e infine (indifferentemente in quale ordine) **2→4** e **2→5→6→8.**

Capire in profondità la DFS [1]

Esercizio 22.3-8 [Cormen]

Trovate un controesempio alla congettura che, se esiste un cammino da u a v in un grafo orientato G e se u.d < v.d in una DFS di G, allora v è un discendente di u nella foresta di visita risultante.

Soluzione: Ricordiamo innanzitutto che nel Cormen viene usato l'attributo *d* come tempo di "entrata" in un nodo e con l'attributo *f* il tempo di "uscita" dal nodo.

Ovviamente, ogni eventuale cammino da *u* a *v* deve essere "tagliato" da qualche nodo già visitato. Quindi, un controesempio minimale è il seguente, in cui la visita comincia dal nodo *s*.

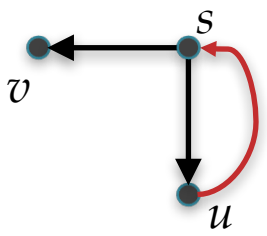
Capire in profondità la DFS [2]

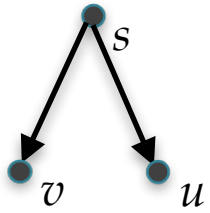
Esercizio 22.3-9 [Cormen]

Trovate un controesempio alla congettura che, se esiste un cammino da u a v in un grafo orientato G, allora in qualsiasi DFS deve essere v.d ≤ u.f.

Soluzione: Ovviamente posso avere $v.d \leq u.f$ per due ragioni: u viene visitato come "discendente" di v (in questo caso ho $u.f < v.f$), oppure semplicemente v viene visitato in un sottoalbero (di visita) che viene visitato prima di un sottoalbero contenente u (in questo caso ho $v.f < u.d$).

Il controesempio precedente funziona lo stesso. Osservate che tutte le visite radicate in s non avranno mai v discendente di u nell'albero di visita. Tuttavia al fine del controesempio, è necessario visitare prima v.



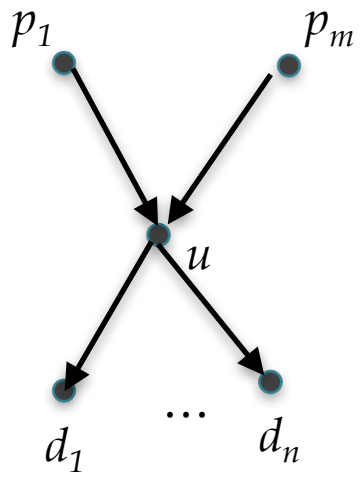


Capire in profondità la DFS [3]

Esercizio 22.3-11 [Cormen]

Spiegate come un vertice u di un grafo orientato G possa finire in un albero DFS contenente solo u, allora in qualsiasi DFS deve essere v.d ≤ u.f.

Soluzione: Ovviamente, affinché u sia solo nell'albero di visita, devo visitare u **dopo** aver già visitato i suoi discendenti, ma **prima** dei suoi predecessori. Quindi, in figura, l'ordine di visita deve essere d_1, …, d_m, u, p_1, …, p_n e non ci devono essere cicli contenenti u.



Capire la DFS e BFS [1]

Esercizio

*Come deve essere fatto un grafo **non** orientato **connesso** G affinché la BFS e la DFS (radicate nello stesso nodo) producano lo stesso albero di visita*

Soluzione:
In una visita DFS ho **solo** archi dell'albero e archi all'indietro.
In una visita BFS ho **solo** archi dell'albero e archi trasversali.

Quindi DFS(G)=BFS(G) solo se contengono solo archi dell'albero di visita e quindi G è esso stesso un albero.

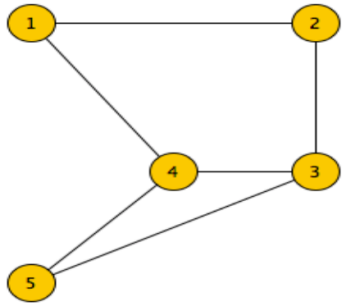
D'altro canto è ovvio che se G fosse un albero, allora DFS(G)=BFS(G) =G. Possiamo quindi dire che **DFS(G)=BFS(G) se e solo se G è un albero**.

Capire la DFS e BFS [2]

[Anche **Esercizio 22.2-9 [Cormen]**]

Esercizio 3

Descrivere un algoritmo che, dato un grafo connesso G, trova un cammino in G che attraversa tutti gli archi una e una sola volta in ognuna delle due direzioni. L'algoritmo deve avere complessità $O(m)$.



Ad esempio per il grafo in figura che ha 7 archi una possibile soluzione è il seguente cammino di lunghezza 14:

1 – 4 – 5 – 4 – 1 – 5 – 1 – 2 – 3 – 4 – 3 – 5 – 3 – 2 – 1

Soluzione

L'esempio può trarre un po' in inganno, perché l'ordine di percorrenza è abbastanza arbitrario (non segue né una BFS, né una DFS).

Una visita qualsiasi (DFS o BFS) visita **tutti** i nodi. Tuttavia, nelle liste di adiacenza incontra tutti gli archi, evitando di ripercorrere nodi già marcati visitati. Gli archi in più (all'indietro nella DFS e trasversali nella BFS) si considerano, ma non si percorrono.

Per ottenere una soluzione al problema basterà fare una visita (DFS o BFS). Gli archi in più si percorrono immediatamente avanti/indietro, mentre gli alberi della visita si percorrono all'indietro **dopo che un nodo è stato completamente esplorato**.

Per tornare al nodo padre è sufficiente aggiungere un attributo ai nodi (che corrisponde al vettore dei padri). Nella **DFS ricorsiva** ciò non è neanche necessario, è sufficiente percorrere l'arco **indietro al ritorno dalle chiamate ricorsive**.

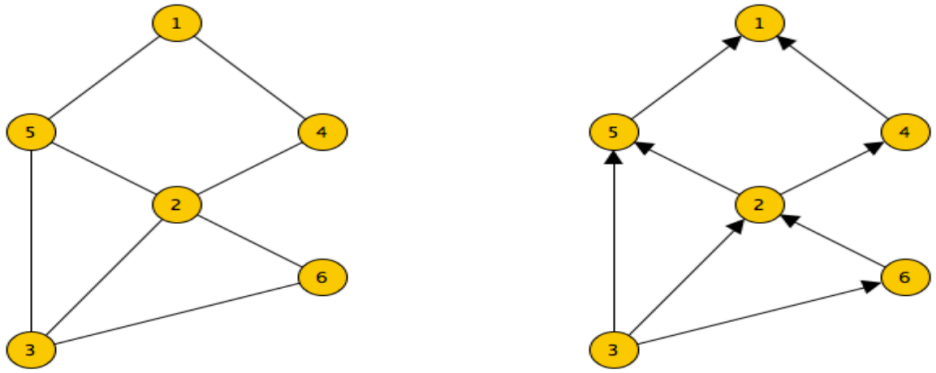
Esercizi 1D(c)

Topological Sort

Capire il Topological Sort [1]

Esercizio 4

Dato un grafo G, descrivere un algoritmo che ne orienta gli archi in modo da creare un grafo G' diretto e aciclico. L'algoritmo deve avere complessità $O(n+m)$.



Ad esempio per il grafo sopra a sinistra un orientamento degli archi lecito è quello riportato sopra a destra

Soluzione

Questo problema è più semplice di quanto non sembri.

Sappiamo che G è un DAG se e solo se è possibile ordinare i nodi in modo che $u \leq v$ implica che esiste un cammino da u a v in G. Per assicurare questa proprietà, è sufficiente ordinare in modo arbitrario i nodi e poi per ogni arco (u, v) nel grafo non orientato si mette un arco $u \rightarrow v$ se $u \leq v$ oppure $v \rightarrow v$ se $v \leq u$.

Più concretamente, avendo la **matrice di adiacenza** di G, e prendendo semplicemente l'ordine derivante dal posizionamento della matrice, è sufficiente ricopiare la matrice e cancellare tutti gli 1 nella parte inferiore della matrice (in cui $i>j$) [$\mathcal{O}(n^2)$].

Avendo le liste di adiacenza (e immaginando ancora di rappresentare i nodi con numeri interi) è sufficiente cancellare nella lista di adiacenza di u tutti i nodi v tali che $v \leq u$ [$\mathcal{O}(m+n)$].

Capire il Topological Sort [3]

Esercizio 6

Un vertice v in un grafo diretto G, si dice *principale* se ogni altro vertice in G può essere raggiunto con un cammino diretto che parte da v.

a) Descrivere un algoritmo che dati un grafo G e un vertice v, determina se v è un vertice principale in G. L'algoritmo deve avere complessità $O(n+m)$.

b) Descrivere un algoritmo che, dato un grafo G, determina se G contiene un vertice principale. L'algoritmo deve avere complessità $O(n+m)$.

Soluzione

Per quanto riguarda il punto **a**) è sufficiente eseguire una qualsiasi visita (BFS o DFS) dal nodo v, marcare i nodi visitati (ad esempio con un vettore di 0 e 1) e poi determinare se tutti i nodi del grafo sono visitati [costo $\mathcal{O}(n+m)$]..

Per quanto riguarda **b**) limitiamoci **inizialmente al caso in cui G sia un DAG**. È sicuramente preferibile avere **un nodo candidato** ad essere il nodo principale. La cosa migliore sembra essere quello di ordinare topologicamente i nodi [costo $\mathcal{O}(n+m)$].

Chiaramente l'unico nodo che può essere principale è il nodo minore nell'ordine topologico dei nodi del grafo. A quel punto, si esegue una qualsiasi visita da quel nodo e si procede come nel punto **a**). Anche questo passo ha costo $\mathcal{O}(n+m)$.

E se G non è un DAG? È possibile semplicemente calcolare **il grafo condensato delle componenti fortemente connesse** G^{SCC} e verificare se in quel grafo c'è un nodo principale. Ovviamente tutti i nodi che appartengono a una eventuale SCC principale sono essi stessi principali.

Capire il Topological Sort [3]

Esercizio 7

Descrivere un algoritmo che dato un grafo diretto G trova il minimo numero di vertici da cui è possibile raggiungere tutti gli altri vertici del grafo. L'algoritmo deve avere complessità $O(n+m)$.

Ancora una volta, cominciamo **assumendo G un DAG**. Osserviamo subito che tutti i nodi senza archi entranti **devono** stare necessarieamente nella soluzione. D'altra parte tutti gli altri nodi **non possono** stare nella soluzione. Quindi è sufficiente calcolare il grado entrante di ogni nodo (vedi sotto in $\mathcal{O}(n+m)$) e poi selezionare (o contare) i nodi con grado entrante 0 (costo $\mathcal{O}(n+m)$).

Se G non è un DAG calcolare **il grafo condensato delle componenti fortemente connesse** G^{SCC}, procediamo come prima e poi è sufficiente prendere **un nodo qualsiasi per ogni componente connessa senza archi entranti**.

forall $u \in V$ **do** d[u]=0
forall $u \in V$ **do**
    forall $v \in G.adj(u)$ d[v]=d[v]+1

Soluzione

Per quanto riguarda il punto **a**) è sufficiente eseguire una qualsiasi visita (BFS o DFS) dal nodo v, marcare i nodi visitati (ad esempio con un vettore di 0 e 1) e poi determinare se tutti i nodi del grafo sono visitati [costo $\mathcal{O}(n+m)$]..

Per quanto riguarda **b**) è sicuramente preferibile avere **un nodo candidato** ad essere il nodo principale. La cosa migliore sembra essere quello di ordinare topologicamente i nodi [costo $\mathcal{O}(n+m)$].

Chiaramente l'unico nodo che può essere principale è il nodo minore nell'ordine topologico dei nodi del grafo. A quel punto, si esegue una qualsiasi visita da quel nodo e si procede come nel punto **a**). Anche questo passo ha costo $\mathcal{O}(n+m)$.

Lezione 1D

That's all Folks…

…Domande?