Esercizi Capitolo 9 - Grafi

Alberto Montresor

19 Agosto, 2014

Alcuni degli esercizi che seguono sono associati alle rispettive soluzioni. Se il vostro lettore PDF lo consente, è possibile saltare alle rispettive soluzioni tramite collegamenti ipertestuali. Altrimenti, fate riferimento ai titoli degli esercizi. Ovviamente, si consiglia di provare a risolvere gli esercizi personalmente, prima di guardare la soluzione.
Per molti di questi esercizi l'ispirazione è stata presa dal web. In alcuni casi non è possibile risalire alla fonte originale. Gli autori originali possono richiedere la rimozione di un esercizio o l'aggiunta di una nota di riconoscimento scrivendo ad `alberto.montresor@unitn.it`.

1 Problemi

1.1 DFS iterativa (Esercizio 9.2 del libro)

Scrivere una versione iterativa della DFS basata su pila dove l'ordine di visita dei nodi e degli archi sia lo stesso di quello della versione ricorsiva descritta nel libro ed eseguirla sul grafo di Figura 1.

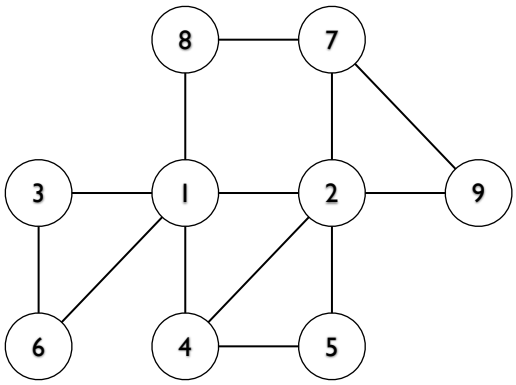

Figura 1: Un grafo non orientato

Soluzione: Sezione 2.1

1.2 Grafo bipartito (Esercizio 9.3 del libro)

Un grafo non orientato G è *bipartito* se l'insieme dei nodi può essere partizionato in due sottoinsiemi tali che nessun arco connette due nodi appartenenti alla stessa parte. G è 2-colorabile se ogni nodo può essere colorato di bianco o di rosso in modo che nodi connessi da archi siano colorati con colori distinti. Si dimostri che G è bipartito: (1) se e solo se è 2-colorabile; (2) se e solo se non contiene circuiti di lunghezza dispari.

Soluzione: Sezione 2.2

1.3 Larghezza albero radicato (Esercizio 9.5 del libro)

Si modifichi una visita BFS per calcolare la larghezza di un albero radicato.

Soluzione: Sezione 2.3

1.4 Albero libero, non orientato (Esercizio 9.6 del libro)

Si scriva una procedura per verificare se un grafo non orientato è un albero libero.

Soluzione: Sezione 2.4

1.5 Quadrato di grafo orientato (Esercizio 9.9 del libro)

Il *quadrato* di un grafo orientato $G = (V, E)$ è il grafo $G^2 = (V, E^2)$ tale che $(x, y) \in E^2$ se e solo se $\exists u : (x, u) \in E \wedge (u, y) \in E$. In altre parole, se esiste un percorso di due archi fra i nodi x e y.
Scrivere un algoritmo che, dato un grafo G rappresentato con matrice d'adiacenza, restituisce il grafo G^2.

Soluzione: Sezione 2.5

1.6 Diametro (Esercizio 9.11 del libro)

Il *diametro* di un grafo è la lunghezza del "più lungo cammino breve", ovvero la più grande lunghezza del cammino minimo (in termini di numero di archi) fra tutte le coppie di nodi. Progettare un algoritmo che misuri il diametro di un grafo e valutare la sua complessità.

Soluzione: Sezione 2.6

1.7 In-degree, out-degree

Data una rappresentazione con liste di adiacenza di un grafo orientato, quanto tempo/spazio occorre per calcolare e stampare il grado uscente di ogni vertice? Quanto tempo occorre per calcolare e stampare il grado entrante di ogni vertice?

Soluzione: Sezione 2.7

1.8 Grafo trasposto

Si consideri un grafo orientato, rappresentato o con liste di adiacenza o con matrice di adiacenza. Quanto tempo occorre per ottenere il corrispondente grafo trasposto?

Soluzione: Sezione 2.8

1.9 Pozzo universale

Data una rappresentazione con matrice di adiacenza, mostrare un algoritmo che opera in tempo $\Theta(n)$ in grado di determinare se un grafo orientato contiene un pozzo universale: ovvero un nodo con out-degree uguale a zero e in-degree uguale a $n - 1$. È possibile ottenere la stessa complessità con liste di adiacenza?

Soluzione: Sezione 2.9

1.10 Massima distanza

Scrivere un algoritmo che prenda in input un grafo orientato $G = (V, E)$ rappresentato tramite liste di adiacenza e un nodo $s \in V$, e restituisca il numero dei nodi in V raggiungibili da s che si trovano alla massima distanza da s.

Soluzione: Sezione 2.10

1.11 Spazio di memoria

Sia $G = (V, E)$ un grafo orientato con n vertici ed m archi. Si supponga che puntatori e interi richiedano 32 bit. Indicare per quali valori di n ed m la lista di adiacenza richiede meno memoria della matrice di adiacenza.

Soluzione: Sezione 2.11

1.12 Triangoli

Dato un grafo $G = (V, E)$ non orientato, chiamiamo triangolo un insieme di tre nodi distinti di G che formano una cricca, ovvero un insieme $\{a, b, c\}$ tale che gli archi $(a, b), (b, c), (a, c)$ appartengono ad E: $\{(a, b), (b, c), (a, c)\} \subseteq E$.

1. Sia G un grafo non orientato completo; qual è il numero dei suoi triangoli?
2. Scrivere un algoritmo che prenda in input un grafo non orientato G e restituisca il numero dei suoi triangoli. Calcolare la complessità dell'algoritmo. Quale rappresentazione è preferibile (matrice o liste di adiacenza?)

Soluzione: Sezione 2.12

1.13 Algoritmo di Kruskal, migliorato

Supponiamo che gli archi abbiano pesi compresi fra 1 e n (numero di nodi); come si può sfruttare questa informazione per migliorare le prestazioni dell'algoritmo di Kruskal?

Soluzione: Sezione 2.13

1.14 Cammini massimi in DAG

Dato un grafo orientato $G = (V, E)$, scrivere un algoritmo per calcolare i cammini di lunghezza massima a partire da un nodo $s \in V$.

Soluzione: Sezione 2.14

1.15 DAG: struttura

Dimostrare che in ogni grafo diretto aciclico esiste almeno un vertice che non ha archi entranti, e almeno un vertice che non ha archi uscenti.

Soluzione: Sezione 2.15

1.16 Algoritmo alternativo per MST

È possibile provare la seguente proprietà degli alberi di copertura minima: se rimuoviamo un arco, otteniamo due sottoalberi che sono alberi di copertura minima per i rispettivi sottoinsiemi di nodi.
Basandosi su questo fatto, suggeriamo il seguente algoritmo basato su divide-et-impera: dividiamo l'insieme di nodi in due parti, troviamo l'albero di copertura minima per i due sottografi, e poi li colleghiamo tramite un arco leggero.
È corretto?

Soluzione: Sezione 2.16

1.17 Cammini di peso massimo in DAG

Scrivere un algoritmo che dato un grafo orientato aciclico pesato sugli archi ed un vertice s calcola i cammini di peso massimo da s ad ogni altro vertice.

Soluzione: Sezione 2.17

1.18 BFS in matrice

Modificare l'algoritmo per la visita BFS per ricevere in input grafi rappresentati dalla matrice di adiacenza e discuterne la complessità.

Soluzione: Sezione 2.18

1.19 Quanto è grande la foresta?

Scrivere un algoritmo che, dato in ingresso un grafo non orientato $G = (V, E)$, conti il numero di componenti connesse di G che sono anche alberi. Si discuta la correttezza e si determini la complessità dell'algoritmo proposto.

Soluzione: Sezione 2.19

1.20 Distanza media

Sia $G = (V, E)$ un grafo non orientato connesso e sia s un nodo in V. Si ricorda che la distanza da s ad un nodo $v \in V$ è pari al numero minimo di archi che separa s da v. Scrivere un algoritmo che ritorni la distanza media di s da tutti gli altri nodi del grafo (quindi escluso s).

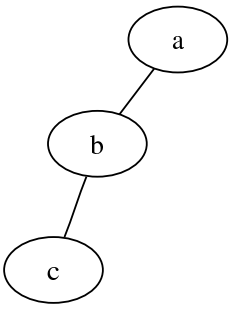

Figura 2: Esempio

Nell'esempio della Figura 2, la distanza media di b è 1, mentre la distanza media di a e c è 1.5.

Soluzione: Sezione 2.20

1.21 Componenti fortemente connesse

Sia dato un grafo orientato G costituito da 8 nodi etichettati con le lettere da a a h. Gli archi sono forniti come liste di adiacenze.

$a.\mathsf{adj}() = \{b, e\}$, $b.\mathsf{adj}() = \{c, f\}$, $c.\mathsf{adj}() = \{f, d, h, g\}$, $d.\mathsf{adj}() = \{h\}$, $e.\mathsf{adj}() = \{f, a\}$, $f.\mathsf{adj}() = \{b\}$, $g.\mathsf{adj}() = \{h\}$, $h.\mathsf{adj}() = \{d\}$

Applicare l'algoritmo per individuare le componenti fortemente connesse del grafo G, partendo dal nodo b. In caso di "ripartenza" (ovvero, se la visita non riesce a raggiungere tutti i nodi), continuate a partire dal nodo a. Disegnare il grafo e indicare chiaramente (i) i tempi di fine visita di ogni nodo (relativi alla prima visita in profondità) e (ii) le componenti fortemente connesse individuate.
Nota bene: l'ordine in cui i nodi adiacenti ad un nodo vengono visitati è esattamente quello in cui compaiono sopra; ad esempio, per quanto riguarda il nodo b verranno visitati prima il nodo adiacente c e poi il nodo adiacente f.

Soluzione: Sezione 2.21

1.22 Facebook

Avete a disposizione il grafo delle amicizie di Facebook, realizzato tramite liste di incidenza. Volete calcolare la persona che ha il maggior numero di "amici e amici di amici", ovvero nodi a distanza 1 o 2 da esso. Scrivere un algoritmo che ritorna questo valore. Calcolare la complessità.

Soluzione: Sezione 2.22

1.23 Cicli verdi

Sia $G = (V, E)$ un grafo orientato e si assuma che i nodi in G siano colorati mediante il colore rosso oppure verde. Si consideri il problema di determinare se G possiede un ciclo semplice contenente solo nodi verdi.

- Si proponga un algoritmo efficiente per il problema proposto;
- Si valuti la correttezza e la complessità della procedura delineata.

Soluzione: Sezione 2.23

1.24 Sorgenti e pozzi

Preso in input un grafo orientato $G = (V, E)$ rappresentato mediante liste di adiacenza, implementare una funzione che stampa tutti i vertici "sorgente" e tutti i vertici "pozzo" di G. Un vertice v è una sorgente se non archi entranti; è un pozzo se non ha archi uscenti. Discutere la complessità dell'algoritmo risultante.

Soluzione: Sezione 2.24

1.25 Diametro di alberi

In un grafo non orientato G, il diametro è la massima distanza fra due vertici in G, intesa come il più lungo cammino minimo fra tutte le coppie di nodi (misurata in numero di archi).
Come sappiamo, gli alberi binari radicati sono particolari tipi di grafi; quindi la definizione di diametro si applica anche ad essi. Scrivere un algoritmo che ritorni il diametro di un albero radicato, sfruttando le caratteristiche speciali di un albero. Discutere la complessità dell'algoritmo risultante.

Soluzione: Sezione 2.25

1.26 Stessa distanza

Si consideri il seguente problema: dato un grafo orientato G (con n nodi e m archi) e due nodi s_1 e s_2 di G, trovare il numero di nodi di G raggiungibili da s_1 e s_2, e che si trovano alla stessa distanza da s_1 e da s_2. Scrivere un algoritmo che risolva il problema e discuterne la complessità.
Ricordiamo che in un grafo orientato G, dati due nodi s e v, si dice che v è raggiungibile da s se esiste un cammino da s a v e, in questo caso, la distanza di v da s è la lunghezza del più breve cammino da s a v (misurato in numero di archi).

Soluzione: Sezione 2.26

2 Soluzioni

2.1 DFS iterativa (Esercizio 9.2 del libro)

La procedure si ottiene semplicemente sostituendo una pila alla coda utilizzata nella visita BFS.

```
dfs(GRAPH G, NODE r)
    STACK S = Stack( )
    S.push(r)
    boolean[ ] visitato = new boolean[1 ... G.n] = {false}        % Initalized to false
    visitato[r] = true
    while not S.isEmpty() do
        NODE u = S.pop()
        if not visitato[u] then
            { esamina il nodo u }
            visitato[u] = true
            foreach v ∈ G.adj(u) do
                { esamina l'arco (u, v) }
                S.push(v)
```

2.2 Grafo bipartito (Esercizio 9.3 del libro)

Date le tre affermazioni (i) G è bipartito; (ii) G è 2-colorabile e (iii) G non contiene cicli di lunghezza dispari, dimostriamo (i) $\Rightarrow$ (ii), (ii) $\Rightarrow$ (iii), (iii) $\Rightarrow$ (i). Questo dimostra che (i) $\Leftrightarrow$ (ii) e (i) $\Leftrightarrow$ (iii).

1. Se G è bipartito, è 2-colorabile. Semplicemente, diamo colore 1 a tutti i nodi in una partizione, diamo colore 2 a tutti i nodi nell'altra. Non essendoci archi fra i nodi di una partizione, la colorazione è valida.

2. Se G è 2-colorabile, non contiene cicli di lunghezza dispari. Supponiamo per assurdo che esista un ciclo $(v_1, v_2), (v_2, v_3) \ldots, (v_{k-1}, v_k), (v_k, v_1)$, con k dispari. Se il nodo v_1 ha colore 1, il nodo v_2 deve avere colore 2; il nodo v_3 deve avere colore 1, e così via fino al nodo v_k, che deve avere colore 1. Poichè v_1 è successore di v_k, v_1 deve avere colore 2, assurdo.

3. Se non esistono cicli di lunghezza dispari, il grafo è bipartito. Dimostriamo questa affermazione costruttivamente. Si prenda un nodo x lo si assegna alla partizione S_1. Si prendono poi tutti i nodi adiacenti a nodi in S_1 e li si assegna alla partizione S_2. Si prendono tutti i nodi adiacenti a nodi in S_2 e li si assegna alla partizione S_1. Questo processo termina quando tutti i nodi appartengono ad una o all'altra partizione. Un nodo può essere assegnato più di una volta se e solo se fa parte di un ciclo. Ma affinché venga assegnato a due colori diversi, deve far parte di un ciclo di lunghezza dispari, e questo non è possibile.

2.3 Larghezza albero radicato (Esercizio 9.5 del libro)

Assumendo di valutare il grafo a partire da una radice r, si mantiene un vettore *count*, dove *count*$[d]$ memorizza il numero di nodi che hanno distanza d da r. Si cerca poi il massimo all'interno di questo vettore.

```
int graphWidth(GRAPH G, NODE r)
    int max = 0
    QUEUE S = Queue( )
    S.enqueue(r)
    int[ ] dist = new int[1 ... G.n] = {−1}          % Initalized to −1
    int[ ] count = new int[1 ... G.n] = {0}          % Initalized to 0
    dist[r] = 0
    while not S.isEmpty() do
        NODE u = S.dequeue()
        foreach v ∈ G.adj(u) do
            if dist[v] < 0 then
                dist[v] = dist[u] + 1
                count[dist[v]] = count[dist[v]] + 1
                S.enqueue(v)
    return max(count, n)
```

2.4 Albero libero, non orientato (Esercizio 9.6 del libro)

La procedura hasCycle() seguente ritorna vero se il grafo ha un ciclo; se il grafo non orientato è connesso ed non ha cicli, è un albero libero.

```
boolean hasCycleRec(GRAPH G, NODE u, NODE p, boolean[ ] visitato)
    visitato[u] = true
    foreach v ∈ G.adj(u) − {p} do
        if visitato[v] then
            return true
        else if hasCycleRec(G, v, u, visitato) then
            return true
    return false
```

```
boolean hasCycle(GRAPH G)
    boolean[ ] visitato = new boolean[G.size()]
    foreach u ∈ G.V() do
        visitato[u] = false
    foreach u ∈ G.V() do
        if not visitato[u] then
            if hasCyclerec(G, u, nil, visitato) then
                return true
    return false
```

2.5 Quadrato di grafo orientato (Esercizio 9.9 del libro)

Vediamo la versione basata su matrice di adiacenza. Il costo computazionale dell'algoritmo è $O(n^3)$.

```
int[][] square(int[][] A, int n)
    int[][] A^2 = new int[1...n][1...n]
    for i = 1 to n do
        for j = 1 to n do
            int k = 1
            found = false
            while k ≤ n and not found do
                found = A[i][k] = 1 ∧ A[k][j] = 1 ∧ i ≠ k ∧ j ≠ k
                k = k + 1
            if found then
                A^2[i][j] = 1
            else
                A^2[i][j] = 0
    return A^2
```

2.6 Diametro (Esercizio 9.11 del libro)

Si effettuano n visite in ampiezza, una a partire da ogni nodo, e si memorizza il massimo valore di distanza trovato in una visita.

```
int diameter(GRAPH G)
    int max = 0
    foreach r ∈ G.V() do
        QUEUE S = Queue( )
        S.enqueue(r)
        int[] dist = new int[1...G.n] = {−1}          % Initalized to −1
        dist[r] = 0
        while not S.isEmpty() do
            NODE u = S.dequeue()
            foreach v ∈ G.adj(u) do
                if dist[v] < 0 then
                    dist[v] = dist[u] + 1
                    if dist[v] > max then
                        max = dist[v]
                    S.enqueue(v)
    return max
```

2.7 In-degree, out-degree

Per il grado uscente: con un singolo contatore, è possibile contare la lunghezza della lista di adiacenza e poi stamparla. Per il grado entrante: è necessario un vettore c di contatori, uno per vertice. Quando un arco incidente in un vertice i viene trovato, il corrispondente contatore $c[i]$ viene incrementato. In entrambi i casi, il tempo è $\Theta(n + m)$.

2.8 Grafo trasposto

Nel caso delle liste di adiacenza, la soluzione è simile al problema precedente. Bisognare scorrere tutti gli archi e inserirli in modo invertito nel nuovo grafo. Costo: $\Theta(n+m)$ Nel caso di matrice d'incidenza, il costo è ovviamente $\Theta(n^2)$.

2.9 Pozzo universale

L'idea è evitare controlli non necessari. Dati due vertici i, j, se $A[i,j] = 1$, allora il vertice i non è un pozzo universale (c'è un arco uscente da i); se $A[i,j] = 0$, allora j non è un pozzo universale (manca arco entrante in j). Si noti inoltre che può esistere un solo pozzo universale.
Partiamo dalla prima riga: $i = 1$.

1. Cerchiamo il minore indice j tale $j > i$ e $A[i,j] = 1$.
2. Se tale vertice non esiste, i non ha archi uscenti ed è l'unico candidato per essere un pozzo universale
3. Se invece tale j esiste, ne possiamo dedurre che tutti i vertici h tali che $1 \leq h < j$ non possono essere pozzi universali, perché manca un arco da i. Quindi ci spostiamo nella riga $i = j$, e torniamo al passo 1.

Si noti che un possibile candidato viene trovato sempre; al limite è dato dall'ultima riga. A quel punto, si verifica che sia effettivamente un pozzo universale, guardando la riga i (deve contenere tutti 0) e la colonna i (deve contenere tutti 1, tranne $A[i,i]$). Il costo dell'algoritmo è pari a $\Theta(n)$.

```
boolean universalSink(int[][] A, int n)
    int i = 1
    int candidate = −1
    while i < n ∧ candidate < 0 do
        j = i + 1
        while j ≤ n and A[i][j] == 0 do
            j = j + 1
        if j > n then
            candidate = i
        else
            i = j
    int rowtot = 0
    int coltot = 0
    for j = 1 to n do
        rowtot = rowtot + A[candidate][j]
        coltot = coltot + A[j][candidate]
    return rowtot = 0 ∧ coltot = n − 1
```

2.10 Massima distanza

È sufficiente eseguire un algoritmo di visita in ampiezza del grafo, mantenendo un contatore associato alla

coda, che memorizza il numero di elementi alla massima distanza trovata fino ad un certo punto.

```
maxdist(GRAPH G, NODE s)
boolean[] dist = new boolean[1...G.n] = {-1}        % Initalized to -1, meaning not visited
dist[s] = 0
QUEUE Q = Queue()
Q.enqueue(s)
current = 0
count = 0
while not Q.isEmpty() do
    u = Q.dequeue()
    foreach v ∈ G.adj(u) do
        if dist[v] < 0 then
            dist[v] = dist[u] + 1
            if dist[u] > current then
                current = dist[u]
                count = 0
            count = count + 1
return count
```

2.11 Spazio di memoria

Per rappresentare la matrice, abbiamo bisogno di n^2 bit; per rappresentare le liste di adiacenza, abbiamo bisogno di n puntatori di inizio lista (32 bit) più m strutture dati con $32 + 32$ bit (32 per il puntatore al next della lista e 32 per il puntatore al nodo della lista).
La matrice di adiacenza è meno efficiente delle liste quando $n^2 > 64m + 32n$, ovvero quando

$$m < \frac{n^2 - 32n}{64}$$

Per fare un esempio, con 1000 nodi, $m \leq 15125$; in altre parole, 15.125 archi per nodo.

2.12 Triangoli

Il numero di triangoli in un grafo completo equivale al numero di combinazioni semplici di 3 elementi scelti in un insieme di n oggetti; quindi, se $n < 3$ tale numero è 0, altrimenti è pari a

$$\binom{n}{3} = \frac{n(n-1)(n-2)}{6}$$

Nella rappresentazione con matrice, l'algoritmo richiede semplicemente di provare tutte le possibile combinazioni di tre elementi distinti ($O(n^3)$ e vedere se esistono gli archi corrispondenti.
Nella rappresentazione con liste di adiacenza, dobbiamo guardare per ogni nodo e per ogni vicino se esiste un nodo in comune nella lista dei nodi.

2.13 Algoritmo di Kruskal, migliorato

Basta utilizzare PigeonHole sort, che ci permette di ordinare gli archi in tempo $O(n)$. In questo modo, il costo dell'algoritmo non è più $O(n+m \log n+m) = O(n \log m)$, ma diventa $O(n+m+m) = O(n+m)$,

2.14 Cammini massimi in DAG

Basta cercare i cammini minimi da s ad ogni altro vertice rispetto ai pesi $w'(u,v) = -w(u,v)$. Si noti che l'assenza di cicli garantisce l'assenza di cicli negativi; in caso di presenza di cicli in generale, il problema diventa esponenziale.

2.15 DAG: struttura

Supponiamo per assurdo che esista un grafo aciclico in cui tutti i vertici hanno almeno un arco entrante. Partendo da un vertice qualunque, potremmo quindi continuare a percorrere archi all'indietro quante volte vogliamo. Osserviamo che, poiché il grafo è aciclico, il vertice raggiunto ad ogni passo deve essere necessariamente diverso da quelli visitati precedentemente. Ne consegue che, dopo aver visitato tutti i vertici, dovremmo necessariamente incontrare un vertice già visitato, il che contraddirrebbe l'ipotesi che il grafo sia aciclico.
Lo stesso ragionamento può essere utilizzato per dimostrare che esiste un nodo senza archi uscenti.

2.16 Algoritmo alternativo per MST

No. Si consideri il seguento grafo:

```
A,B,1
B,C,10
C,D,1
D,A,10
```

e la suddivisione $\{A, D\}$, $\{B, C\}$. Questo avviene perché la proprietà dimostrata all'inizio contiene un'implicazione in senso opposto a quello che serve per dimostrare la correttezza dell'algoritmo.

2.17 Cammini di peso massimo in DAG

Basta cercare i cammini minimi da s ad ogni altro vertice rispetto ai pesi $w'(u,v) = -w(u,v)$. Si noti che l'assenza di cicli garantisce l'assenza di cicli negativi; in caso di presenza di cicli in generale, il problema richiede algoritmi esponenziali.

2.18 BFS in matrice

Nell'algoritmo bfs(), è sufficiente sostituire lo statement:

foreach $v \in G.\mathsf{adj}(u)$ **do**

con lo statement seguente:

for $v = 1$ **to** $G.\mathsf{V}()$ **do**
  new $M[u, v]$

Il costo passa da $O(m + n)$ a $O(n^2)$.

2.19 Quanto è grande la foresta?

Si utilizza lo schema di visita basato su discovery/finish time visto nel libro per visitare l'intero grafo e verificare se ogni componente connessa è un albero oppure no. Si inizializza i discovery/finish time di tutti i nodi a 0, in modo da distinguere nodi già visitati da quelli da visitare. Tutte le volte che viene identificato un nuovo nodo di partenza, si esegue ciclico e si incrementa *count* se questa chiamata ritorna **false**. La variabile *count* viene restituita al chiamante.

Il costo di una tale procedura è $O(m+n)$ (il semplice costo della visita).

```
int treeCount(GRAPH G)
    int count = 0
    int time = 0
    int[] dt = new int[1...G.n] = {0}        % Discovery time, initialized to 0
    int[] ft = new int[1...G.n] = {0}        % Finish time, initialized to 0
    foreach u ∈ G.V() do
        if dt[u] == 0 then
            if not hasCycle(G, u, dt, du, time) then
                count = count + 1
    return count
```

```
boolean hasCycle(GRAPH G, NODE u, int[] dt, int[] ft, &time)
    time = time + 1
    dt[u] = time
    foreach v ∈ G.adj(u) do
        if dt[v] == 0 then
            if hasCycle(G, v) then
                return true
        else if dt[u] > dt[v] and ft[v] == 0 then
            return true
    time = time + 1
    ft[u] = time
    return false;
```

2.20 Distanza media

Si effettua una semplice visita BFS di costo $O(m+n)$ e si calcola la distanza di ogni nodo dal nodo s. Tutte le volte che viene calcolata una nuova distanza, viene aggiunta alla variabile tot che poi viene divisa per $n-1$, il numero di nodi del grafo meno il nodo s. Si noti che se il grafo non fosse connesso, avremmo anche

dovuto contare il numero di nodi raggiungibili da s, per poi dividere la distanza totale per questo numero.

```
int averageDistance(GRAPH G, NODE r)
    QUEUE S = Queue()
    S.enqueue(r)
    int[] dist = new int[1...G.n]
    int tot = 0
    foreach u ∈ G.V() − {r} do
        dist[u] = −1
    dist[r] = 0
    while not S.isEmpty() do
        NODE u = S.dequeue()
        foreach v ∈ G.adj(u) do
            if dist[v] < 0 then
                dist[v] = dist[u] + 1
                tot = tot + dist[v]
                S.enqueue(v)
    return tot/(G.n − 1)
```

2.21 Componenti fortemente connesse

Il grafo di input con i valori di inizio e fine visita per la visita che parte da b.

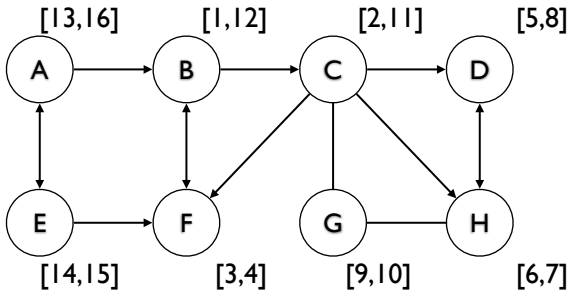

Figura 3: Grafo con tempi di discovery e finish per la visita che parte dal nodo b

Componenti in ordine di individuazione, per la visita che parte dal nodo b. Il primo nodo dell'insieme è la radice dell'albero depth-first.
$\{a, e\}$
$\{b, f, c\}$
$\{g\}$
$\{d, h\}$

2.22 Facebook

In questo esercizio non era spiegato chiaramente come calcolare il numero di amici e amici di amici. Erano possibili tre soluzioni diverse:

1. tre cicli annidati su tutti i nodi; questo approccio, oltre a non tener conto dei duplicati, è inefficiente –$O(n^3)$

2. utilizzando "programmazione dinamica", è possibile salvare in ogni nodo il numero di amici e ottengo lo stesso risultato di cui sopra, ma in $O(n + m)$ (che è al limite $O(n^2)$).

3. volendo evitare i duplicati, possiamo fare n visite BFS, in cui però limitiamo a due il numero massimo di passi; l'algoritmo resta comunque $O(n^3)$, ma abbiamo evitato i duplicati.

Mostriamo qui la terza soluzione.

```
int facebook3(GRAPH G)
    int max = 0
    foreach v ∈ G.V() do
        int t = count(GRAPH G, NODE v)
        if t > max then
            max = t
    return max
```

```
int count(GRAPH G, NODE r)
    boolean[] visited = new boolean[1...G.n] = {false}        % Initalized to not visited
     visited[r] = true
    int f = 0
    foreach u ∈ G.adj(r) do
        visited[u] = true
        f = f + 1
        foreach v ∈ G.adj(u) do
            if not visited[v] then
                visited[v] = true
                f = f + 1
    return f
```

2.23 Cicli verdi

È sufficiente modificare l'algoritmo ciclico() visto nel libro, che effettua una visita in profondità, partendo solo e unicamente dai nodi verdi non già visitati, e non visitando mai nodi rossi - il che corrisponde ad eliminare i nodi rossi dal grafo, ed ottenere potenzialmente un grafo non connesso. Il costo totale è $O(m + n)$.
Nel codice seguente, si assume che i colori siano memorizzati nel vettore $color$ passato in input.

```
boolean hasGreenCycle(GRAPH G, int[]color, NODE u)
    time = time + 1;  dt[u] = time
    foreach v ∈ G.adj(u) do
        if color[v] == GREEN then
            if dt[v] == 0 then
                if hasGreenCycle(G, v) then
                    return true
            else if dt[u] > dt[v] and ft[v] == 0 then
                return true
    time = time + 1;  ft[u] = time
    return false;
```

2.24 Sorgenti e pozzi

L'idea consiste nel visitare il grafo e marcare tutti i nodi che non hanno archi entranti. Dopo di che, per ogni nodo u si decide se è una sorgente (non ha archi entranti) o un pozzo (non ha archi uscenti, ovvero $G.\mathsf{adj}(u) = \emptyset$). Il costo dell'algoritmo è ovviamente $O(m + n)$.

```
printSinkSources(GRAPH G)
    boolean[] in = new boolean[1...n]
    foreach u ∈ G.V() do
        in[u] = false
    foreach u ∈ G.V() do
        foreach v ∈ G.adj(u) do
            in[v] = true
    print "Sorgenti:"
    foreach u ∈ G.V() do
        if not in[u] then
            print u
    print "Pozzi:"
    foreach u ∈ G.V() do
        if G.adj(u) == ∅ then
            print u
```

2.25 Diametro di alberi

Il *diametro* di un albero consiste nel più lungo cammino esistente fra due nodi foglia. (N.b. non necessariamente esso deve passare dal nodo radice, si veda la Figura 4).

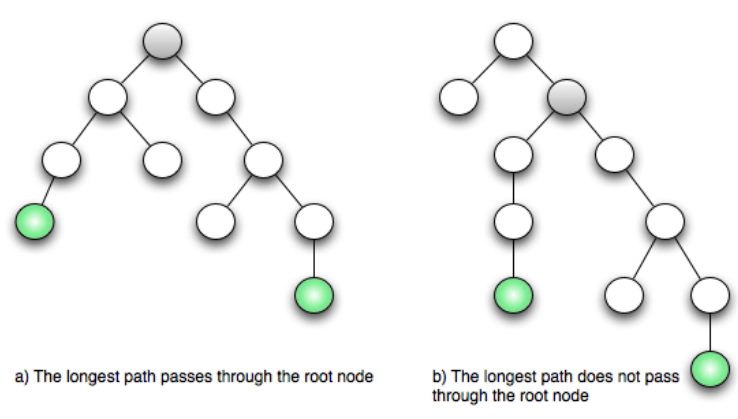

Figura 4: Diametro di un albero

Quindi, calcoliamo il diametro tramite una funzione ricorsiva `diameter(T)` che calcoli e valuti come risultato, la più grande fra le seguenti quantità:

- Il diametro del sotto-albero sinistro di T;
- il diametro del sotto-albero destro di T;
- il percorso più lungo fra due foglie che passa per la radice di T (altezze dei sotto-alberi sinistro e destro, più due archi che connetto la radice a questi sottoalberi).

La funzione ricorsiva diameter() ritorna una coppia di interi, rispettivamente l'altezza dell'albero e il diametro dell'albero passato in input. Per semplificare la gestione, la funzione ritorna un'altezza di -1 nel caso l'albero di input sia vuoto. La complessità corrisponde a quella di una visita di un albero, che quindi è $O(n)$, dove n è il numero dei nodi.

```
⟨int, int⟩ diameter(TREE t)
    if t == nil then
        return ⟨−1, 0⟩
    ⟨alt_sx, dia_sx⟩ = diameter(t.left)
    ⟨alt_dx, dia_dx⟩ = diameter(t.right)
    return ⟨max(alt_sx, alt_dx) + 1, max(dia_sx, dia_dx, (alt_sx + 1) + (alt_dx + 1))⟩
```

2.26 Stessa distanza

L'algoritmo esegue una esplorazione in ampiezza del grafo diretto G a partire dal nodo s_1 calcolando così le distanze di tutti i nodi da s_1, e poi a partire da s_2 calcolando così le distanze di tutti i nodi da s_2. Conta poi il numero di nodi che hanno la stessa distanza. Il costo di due visite è banalmente $O(m + n)$.

```
int sameDistance(GRAPH G, NODE s_1, NODE s_2)
    QUEUE S = Queue( )                          % Queue containing nodes to be visited
    S.enqueue(s_1)
    int[ ] dist_1 = new int[1 ... G.n] = −1     % Initialized to −1, meaning not visited
    dist_1[r] = 0
    while not S.isEmpty() do
        NODE u = S.dequeue()
        foreach v ∈ G.adj(u) do
            if dist_1[v] < 0 then
                dist_1[v] = dist_1[u] + 1
                S.enqueue(v)
    QUEUE S = Queue( )                          % Queue containing nodes to be visited
    S.enqueue(s_1)
    int[ ] dist_2 = new int[1 ... G.n] = −1     % Initialized to −1, meaning not visited
    dist_2[r] = 0
    while not S.isEmpty() do
        NODE u = S.dequeue()
        foreach v ∈ G.adj(u) do
            if dist_2[v] < 0 then
                dist_2[v] = dist_2[u] + 1
                S.enqueue(v)
    int counter = 0                             % Counter of same-distance nodes
    foreach u ∈ G.V() do
        if dist_1[s_1] == dist_2[s_2] then
            counter = counter + 1
    return counter
```

Una soluzione più semplice e compatta è la seguente, che sfrutta la procedura per il calcolo delle distanze

BFS vista nel libro.

```
int sameDistance(GRAPH G, NODE s1, NODE s2)
    int dist1 = new int[1...G.n]          % Distances from s1
    int dist2 = new int[1...G.n]          % Distances from s2
    distance(G, s1, dist1)
    distance(G, s2, dist2)
    int counter = 0
    for i = 1 to nG. do
        if dist1[s1] == dist2[s2] then
            counter = counter + 1
    return counter
```

3 Problemi aperti

3.1 Da multigrafo a grafo

Data una rappresentazione con liste di adiacenza di un multigrafo $G = (V, E)$ (un grafo in cui ci possono essere più archi per la stessa coppia di nodi), scrivere un algoritmo con tempo $O(V + E)$ per calcolare la rappresentazione con liste di adiacenza del grafo non orientato $G' = (V, E')$, dove E' è formato dagli archi di E con tutti gli archi multipli fra due vertici distinti sostituiti da un singolo arco e con tutti i cappi rimossi. Per la soluzione, è possibile utilizzare una struttura di appoggio.

3.2 Lemma - Visite

Provare il seguente lemma: In un grafo, indicando con $\delta(s, v)$ il minimo numero di archi in un cammino da s a v si ha che per ogni $(u, v) \in E : \delta(s, v) \leq \delta(s, u) + 1$.

3.3 BFS in matrice

Qual è il tempo di esecuzione della procedura BFS se il suo grafo di input è rappresentato da una matrice delle adiacenze e l'algoritmo viene modificato per gestire questa forma di input?

- Scrivere l'algoritmo modificato.
- Calcolare il tempo di esecuzione dell'algoritmo giustificando la risposta

3.4 Numero di nodi raggiugibili

Descrivere un algoritmo che prenda in input un grafo non orientato $G = (V, E)$ rappresentato mediante liste di adiacenza e dia in output il numero di nodi raggiungibili da ogni nodo v nel grafo.

3.5 Componenti connesse, conteggio

Scrivere un algoritmo che conti le componenti connesse di un grafo $G = (V, E)$ usando le operazioni merge-find.

3.6 Doppia raggiungibilità

Scrivere un algoritmo che, dati in ingresso un grafo $G = (V, E)$ non orientato e due vertici $u, v \in V$, determini l'insieme dei vertici raggiungibili sia da u che da v. Fare poi l'analisi del tempo di calcolo dell'algoritmo proposto.

3.7 Sottografo a distanza k

Scrivere un algoritmo che, dato in ingresso un grafo $G = (V, E)$ non orientato, e specifi- cato un vertice $v \in V$ e un intero $k > 0$, stampi i vertici che hanno distanza da v minore o uguale a k. Si indichi, giustificandolo, il tempo di esecuzione dell'algoritmo.

3.8 Dimensione della massima componente connessa

Fornire un algoritmo per determinare la dimensione di una massima componente connessa in un grafo non orientato. Fare l'analisi del tempo di calcolo dell'algoritmo proposto.

3.9 Albero di copertura unitario

Scrivere un algoritmo che, dato in ingresso un grafo $G = (V, E)$ non orientato, completo (in cui, cioè, per ogni coppia $u, v \in V$ di vertici, esiste l'arco $(u, v) \in E$), pesato (esiste una funzione $w : E \leftarrow \mathbf{R}^+$ che ad ogni arco $(u, v) \in E$ associa un peso reale positivo $w(u, v)$), con $|V| = n$ vertici, determini se esiste un albero di copertura del grafo costituito solamente da archi di peso unitario. Fornire il peso dell'albero.

3.10 Max-peso

Sia $G = (V, E, w)$ un grafo orientato pesato e sia $s \in V$ un nodo di G tale che ogni altro nodo è raggiungibile da s. Dato un cammino $p = (v_0, v_1, \ldots, v_k)$ definiamo il max-peso $mw(p)$ di p come il massimo tra i pesi degli archi che compongono p.

- Si descriva un algoritmo per deteminare l'albero dei cammini aventi max- peso minimo radicato in s. Si dimostri la correttezza e si determini la complessità dell'algoritmo proposto.
- Nel caso in cui G sia aciclico si descriva un algoritmo per determinare l'albero dei cammini aventi max-peso massimo radicato in s. Si dimostri la correttezza e si determini la complessità dell'algoritmo proposto.