Grafi – Tutte le strade portano a Roma

Un vertice v in un grafo orientato G si dice di tipo “Roma” se ogni altro vertice w in G può raggiungere v con un cammino orientato che parte da w e arriva a v.

1. Descrivere un algoritmo che dati un grafo G e un vertice v, determina se v è un vertice di tipo “Roma” in G.
2. Descrivere un algoritmo che, dato un grafo G, determina se G contiene un vertice di tipo “Roma”.

In entrambi i casi è possibile trovare un algoritmo con complessità $O(m + n)$, ma anche altre complessità verranno considerate.

Grafi – Pozzo universale

- Un pozzo universale è un nodo con out-degree uguale a zero e in-degree uguale a $n - 1$.
- Dato un grafo orientato G rappresentato tramite matrice di adiacenza, scrivere un algoritmo che opera in tempo $\Theta(n)$ in grado di determinare se G contiene un pozzo universale.
- È possibile ottenere la stessa complessità con liste di adiacenza?

Griglia quadrata

Si consideri un griglia quadrata $n \times n$ celle.

- Ogni cella è colorata con un colore in $\{1, 2, 3\}$
- Per semplicità, supponete che nella griglia sia presente almeno una cella di colore 1 e almeno una cella di colore 3.
- Supponete di partire da una cella di colore 1
- Ad ogni passo potete muovervi di una cella in alto, in basso, a destra o a sinistra
- L'obiettivo è raggiungere una cella con colore 3

Scrivere un algoritmo che prende in input una griglia rappresentata da una matrice di interi e restituisca il numero minimo di passi *necessari* per raggiungere una qualunque cella di colore 3 a partire da una qualunque cella di colore 1.
Discutere correttezza e complessità dell'algoritmo proposto.

Griglia quadrata

Ad esempio, si consideri la matrice seguente:

1	2	2	3
2	1	2	3
2	2	2	**3**
3	2	**1**	**2**

La risposta da dare è 2, perchè non esistono celle 1 e 3 adiacenti ma esistono percorsi formati da due passi (come quello evidenziato in grassetto, che però non è l'unico).

Good vs bad guys

Fra ogni coppia di wrestler professionisti può esserci una rivalità oppure no. Per ragioni di marketing, è una buona idea dividere i wrestler professionisti in due gruppi, "buoni" e "cattivi", e farli combattere fra di loro.

Supponete di avere in input un insieme di rivalità, rappresentate come un vettore di coppie (x, y), dove x e y sono identificatori di wrestler compresi fra 1 ed n.

Scrivere un algoritmo che restituisca **true** se è possibile suddividere i wrestler professionisti in due sottoinsiemi non vuoti ("buoni" e "cattivi"), non necessariamente della stessa dimensione, in modo tale che non ci siano rivalità all'intero dei due gruppi; **false** altrimenti.

Spoiler alert!

Tutte le strade portano a Roma – 2012/05/03

Operando sul grafo trasposto – un nodo è Roma se da esso è possibile raggiungere tutti i nodi.

```
isRoma(GRAPH G^T, NODE v)
boolean[] id = new int[1 ... G^T.n]
foreach u ∈ G^T.V() do
    id[u] = 0
ccdfs(G^T, 1, v, id)
foreach u ∈ G^T.V() do
    if id[u] == 0 then
        return false
return true
```

Tutte le strade portano a Roma – 2012/05/03

E' possibile ripetere Roma a partire da tutti i nodi, con un costo pari a $O(n(m+n)) = O(mn)$. Altrimenti, si consideri un ordinamento topologico del grafo trasposto: se il primo non è di tipo Roma, allora nessuno lo è; se è di tipo Roma, allora potrebbero essercene altri ma basta il primo. Chiamiamo quindi isRoma() a partire da esso.

```
Roma(GRAPH G^T)
STACK S = topsort(G^T)
NODE v = S.pop()
return isRoma(G^T, v)
```

Il costo è $O(m+n)$.

Pozzo universale

```
universalSink(int[][] A)
i = 1
candidate = false
while i < n ∧ candidate == false do
    j = i + 1
    while j ≤ n ∧ A[i, j] == 0 do
        j = j + 1
    if j > n then
        candidate = true
    else
        i = j
rowtot = ∑_{j∈{1...n}−{i}} A[i, j]
coltot = ∑_{j∈{1...n}−{i}} A[j, i]
return rowtot = 0 ∧ coltot = n − 1
```

Griglia quadrata

```
int grid(int[][] M, int n)
int[] dr = [−1, 0, +1, 0]                              % Mosse possibili sulle righe
int[] dc = [0, −1, 0, +1]                              % Mosse possibili sulle colonne
int[] distance = new int[1 . . . n][1 . . . n]
QUEUE Q = Queue()
for r = 1 to n do
    for c = 1 to n do
        distance[r][c] = iif(M[r][c] == 1, 0, −1)
        if M[r][c] == 1 then
            Q.enqueue(⟨r, c⟩)
[...]
```

Griglia quadrata

```
int grid(int[][] M, int n)
[...]
while not Q.isEmpty() do
    int, int r, c = Q.dequeue()        % Riga, colonna della cella visitata correntemente
    for i = 1 to 4 do
        nr = r + dr[i]                 % Nuova riga
        nc = c + dc[i]                 % Nuova colonna
        if 1 ≤ nr ≤ n and 1 ≤ nc ≤ n and distance[nr][nc] < 0 then
            distance[nr][nc] = distance[r][c] + 1
            if M[nr][nc] == 3 then
                return distance[nr][nc]
            else
                Q.enqueue(⟨nr, nc⟩)
```

Good vs bad guys

Il problema proposto è quello della bi-colorazione di un grafo, che è possibile se e solo se il grafo è bipartito. La bi-colorazione può essere ottenuta facilmente tramite una visita DFS:

boolean good-bad-guys(Graph G)

int[] C = **new int**[$1 \ldots G.n$]
for $u = 1$ **to** n **do**
  $C[u] = -1$

foreach $u \in G.V()$ **do**
  if $C[u] < 0$ **then**
    if not dfsVisit($G, u, 0, C$) **then**
      return false

return true

Good vs bad guys

```
boolean dfsVisit(GRAPH G, int u, int c, int[] C)
C[u] = c
foreach v ∈ G.adj(u) do
    if C[v] < 0 then
        if not dfsVisit(G, v, 1 − c, C) then
            return false
    else if C[v] == c then
        return false
return true
```