Grafi – Distanza fra partizioni

- 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, misurata in numero di archi.
- Nel caso V_1 e V_2 non siano disgiunti, allora la distanza è 0.
- Scrivere un algoritmo mindist(Graph G, Set V_1, Set V_2) che restituisce la distanza minima fra V_1 e V_2.
- Discutere complessità e correttezza, assumendo che l'implementazione degli insiemi sia tale che il costo di verificare l'appartenenza di un elemento all'insieme abbia costo $O(1)$.
- Nota: è facile scrivere un algoritmo $O(nm)$; esistono tuttavia algoritmi di complessità $O(n^2)$ (con matrice di adiacenza) e $O(m+n)$.

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. Scrivere un algoritmo che dati un grafo G e un vertice v, determina se v è un vertice di tipo "Roma" in G.
2. Scrivere 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.

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).

Spoiler alert!

Distanza fra partizioni

```
mindist(GRAPH G, SET V1, SET V2)
QUEUE Q = Queue()
int[] dist = new int[1 ... G.n]
foreach u ∈ G.V() do
    if V1.contains(u) then
        Q.enqueue(u)
        dist[u] = 0
        if V2.contains(u) then
            return 0
    else
        dist[u] = ∞
```

Distanza fra partizioni

```
while not Q.isEmpty() do
    Node u = Q.dequeue()
    foreach v ∈ G.adj(u) do
        if dist[v] == ∞ then
            dist[v] = dist[u] + 1
            if V_2.contains(v) then
                return dist[v]
            Q.enqueue(v)
return +∞
```

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.

boolean $\mathsf{isRoma}(\textsc{Graph}\ G, \textsc{Node}\ v)$

$\textsc{Graph}\ G^T = \mathsf{transpose}(G)$
boolean$[\,]$ id = **new int**$[1 \ldots G.n] = \{0\}$
$\mathsf{ccdfs}(G^T, 1, v, id)$
foreach $u \in G^T.\mathsf{V}()$ **do**
    if $id[u] == 0$ **then**
        return false
return true

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

È 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.

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

Il costo è $O(m+n)$.

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⟩)
```