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.

Minecraft

In Minecraft, una carta geografica è rappresentata da una matrice $n \times n$ di celle, ognuna delle quali rappresenta un'altitudine intera. Un valore ≤ 0 corrisponde ad un mare, mentre un valore > 0 corrisponde ad un'isola.

Scrivere un algoritmo che prenda in input una matrice A di interi e la sua dimensione positiva n, e restituisca l'altezza media dell'isola più elevata.

Discutere informalmente la correttezza della soluzione proposta e calcolare la complessità computazionale.

Minecraft

Ad esempio, nella matrice seguente ci sono tre isole, una 2×3 in alto a sinistra con altezza media 1.5, una 4×1 a destra con altezza media 1, una 1×2 in basso a sinistra con altezza media 2. L'isola con altezza media più alta è quella in basso a sinistra, e quindi l'algoritmo deve restituire 2.

1	1	1	0	1
2	2	2	-1	1
0	-1	0	-2	1
0	0	-1	-2	1
2	2	-1	0	0

Per fare un albero (binario di ricerca) ci vuole...

Dato un vettore V contenente n interi ordinati e distinti, scrivere un algoritmo che restituisca un albero binario di ricerca di altezza minima.

Discutere correttezza e complessità dell'algoritmo proposto.

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?

Famiglia

Trovare i limiti superiori e inferiori più stretti possibili per la seguente famiglia di equazioni di ricorrenza, per valori di a interi positivi.

$$T(n) = \begin{cases} aT(\lfloor n/2 \rfloor) + n^{a-1} & n \geq 2 \\ 1 & n < 2 \end{cases}$$

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.

---

boolean isRoma(Graph G, Node v)

```
Graph G^T = transpose(G)
boolean[] id = new int[1...G.n] = {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

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^T, v)

Il costo è $O(m+n)$.

Minecraft (21/08/18)

Utilizziamo la matrice M di input come vettore *visited*; in altre parole, una cella è visitabile se il suo valore è maggiore di zero. Una volta scoperta, il valore viene posto a zero. Se la matrice deve essere conservata, sarà necessario farne una copia prima (con costo $O(n^2)$).

L'algoritmo scritto qui simula una ricerca delle componenti connesse nel grafo. Si parte da ogni cella non visitata, e si lancia una visita DFS. Per ogni nodo visitato, si aggiunge l'altezza ad un accumulatore di altezze *height*, si aggiunge 1 ad un contatore *count* e si marca il nodo come visitato. Si continua quindi la visita affrontando le quattro caselle vicine. Al termine della visita ricorsiva, si calcola l'altezza media e la si confronta con il massimo.

Visto che ogni cella può essere visita al più una volta, la complessità è $O(n^2)$.

Minecraft (21/08/18)

```
int topIsland(int[][] M, int n)
float max = 0
for r = 1 to n do
    for c = 1 to n do
        if M[r][c] > 0 then
            int count = 0
            int height = 0
            dfsRec(M, n, r, c, &count, &height)
            max = max(max, height/count)
return max
```

Minecraft (21/08/18)

---

dfsRec(**int**[][] M, **int** n, **int** r, **int** c, **int** $count$, **int** $height$)

---

if $1 \leq r < n$ **and** $1 \leq c < n$ **and** $M[r][c] > 0$ **then**

- $height = height + M[r][c]$
- $count = count + 1$
- $M[r][c] = 0$ % Visited
- dfsRec$(M, n, r-1, c, \&count, \&height)$
- dfsRec$(M, n, r+1, c, \&count, \&height)$
- dfsRec$(M, n, r, c-1, \&count, \&height)$
- dfsRec$(M, n, r, c+1, \&count, \&height)$

Per fare un albero (binario di ricerca) ci vuole...

```
TREE build-tree-rec(int[] A, int i, int j)
if i ≤ j then
    int m = ⌊(i + j)/2⌋
    TREE T = new TREE
    T.left = build-tree-rec(A, i, m − 1)
    T.right = build-tree-rec(A, m + 1, j)
    T.key = V[m]
    return T
else
    return nil
```

L'equazione di ricorrenza è pari a: $T(n) = 2T(n/2) + 1$, che dà origine a $T(n) = \Theta(n)$.

Per fare un albero (binario di ricerca) ci vuole...

Correttezza: per induzione sulla dimensione dell'albero e per il fatto che qualunque altra divisione dei vali fra sottoalbero sinistro e sottoalbero destro dà origine a sottoalberi di altezza maggiore o uguale.

Pozzo universale – Esercizio 1.9 degli esercizi su grafi

Dati due vertici i, j:

- se $A[i][j] = 1$, allora 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, 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.
Il costo dell'algoritmo è pari a $\Theta(n)$.

Pozzo universale – Esercizio 1.9 degli esercizi su grafi

boolean universalSink(**int**[][] A, **int** n)

int $i = 1$
int $candidate = -1$
while $i < n$ **and** $candidate < 0$ **do**
    $j = i + 1$
    while $j \leq n$ **and** $A[i][j] == 0$ **do**
        $j = j + 1$
    if $j > n$ **then**
        $candidate = i$
    else
        $i = j$
$rowtot = \sum_{j \in \{1 \ldots n\}} A[candidate][j]$
$coltot = \sum_{j \in \{1 \ldots n\}} A[j][candidate]$
return $rowtot == 0$ **and** $coltot == n - 1$

Famiglia (24/07/20)

Data la forma della ricorrenza, è possibile utilizzare il Master Theorem, versione base.

- Per $a = 1$, abbiamo $\alpha = \log_2 1 = 0$, $\beta = 0$; siamo nel secondo caso, $T(n) = \Theta(\log n)$.
- Per $a = 2$, abbiamo $\alpha = \log_2 2 = 1$, $\beta = 1$; siamo nel secondo caso, $T(n) = \Theta(n \log n)$.
- Per $a = 3$, abbiamo $\alpha = \log_2 3 < 2$, $\beta = 2$; siamo nel terzo caso, $T(n) = \Theta(n^2)$.
- In generale, è possibile dimostrare che $\log_2 a < a - 1$ per tutti i valori interi $a \geq 3$; siamo nel terzo caso e si ottiene $T(n) = \Theta(n^{a-1})$.