

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 $\text{mindist}(\text{GRAPH } G, \text{SET } V_1, \text{SET } V_2)$ 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 $O(m + n)$.

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).

Grafi – Connetti il grafo

- Progettare un algoritmo efficiente che dato un grafo non orientato, **restituisca il numero minimo di archi** da aggiungere per renderlo connesso.
- Progettare un algoritmo efficiente che dato un grafo non orientato, **aggiunga** il numero minimo di archi necessari a renderlo connesso.

Maggioranza

Scrivere una funzione

```
boolean hasMajority(int[] A, int n)
```

che prenda in input un vettore ordinato *A* contenente *n* interi e restituisca **true** se *A* contiene un valore di maggioranza, ovvero un valore che compare più di $n/2$ volte; restituisca **false** altrimenti. Soluzioni di costo computazionale lineare non verranno prese in considerazione.

Discutere correttezza e complessità dell'algoritmo proposto.

Spoiler alert!

Distanza fra partizioni

mindist(GRAPH G , SET V_1 , SET V_2)

QUEUE $Q = \text{Queue}()$

int[] $dist = \text{new int}[0 \dots G.n - 1]$

foreach $u \in G.V()$ **do**

if $V_1.\text{contains}(u)$ **then**

$Q.\text{enqueue}(u)$

$dist[u] = 0$

if $V_2.\text{contains}(u)$ **then**

return 0

else

$dist[u] = \infty$

Distanza fra partizioni

```
while not  $Q$ .isEmpty() do
  NODE  $u = Q$ .dequeue()
  foreach  $v \in G.\text{adj}(u)$  do
    if  $\text{dist}[v] == \infty$  then
       $\text{dist}[v] = \text{dist}[u] + 1$ 
      if  $V_2.\text{contains}(v)$  then
        return  $\text{dist}[v]$ 
       $Q.\text{enqueue}(v)$ 
return  $+\infty$ 
```

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  $1 \leq nr \leq n$  and  $1 \leq nc \leq n$  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( $\langle nr, nc \rangle$ )
```

Grafi – Connetti il grafo - 1

```
int numberToConnect(GRAPH  $G$ )

---

int[]  $id$  = new int[1... $G.n$ ]  
foreach  $u \in G.V()$  do  $id[u] = 0$   
  
int  $counter = 0$   
foreach  $u \in G.V()$  do  
    if  $id[u] == 0$  then  
         $counter = counter + 1$   
         $ccdfs(G, counter, u, id)$   
  
return  $counter - 1$   
  
 $ccdfs(\text{GRAPH } G, \text{int } counter, \text{NODE } u, \text{int}[] \text{ } id)$   
     $id[u] = counter$   
    foreach  $v \in G.adj(u)$  do  
        if  $id[v] == 0$  then  
             $ccdfs(G, counter, v, id)$ 
```

Grafi – Connetti il grafo - 2

numberToConnect(GRAPH G)

int[] id = **new int**[1... $G.n$]

int[] *representative* = **new int**[1... $G.n$]

foreach $u \in G.V()$ **do** $id[u] = 0$

int $counter = 0$

foreach $u \in G.V()$ **do**

if $id[u] == 0$ **then**

$counter = counter + 1$

representative[$counter$] = u

 ccdfs($G, counter, u, id$)

for $i = 2$ **to** $counter$ **do**

$G.insertEdge(representative[i - 1], representative[i])$

ccdfs(GRAPH G , **int** $counter$, NODE u , **int**[] id)

$id[u] = counter$

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

if $id[v] == 0$ **then**

 ccdfs($G, counter, v, id$)

Maggioranza (03/07/20)

Se esiste un valore di maggioranza, questo deve essere il valore contenuto nella posizione mediana $\lfloor n/2 \rfloor$.

- Se n è dispari, questo è l'elemento centrale. Qualunque maggioranza deve includere questo elemento e altri $(n - 1)/2$ elementi; infatti, a sinistra e destra di tale elemento ci stanno esattamente $(n - 1)/2$ elementi.
- Se n è pari, sia l'elemento in $n/2 - 1$ che l'elemento in $n/2$ devono appartenere alla maggioranza, per lo stesso ragionamento di cui sopra.

Dato il valore candidato $\lfloor n/2 \rfloor$, si effettua ricerca dicotomica modificata che restituisca l'indice più alto in cui il candidato si presenta.

Conoscendo l'indice *last* più alto in cui il candidato si presenta, è necessario vedere se l'elemento in posizione $A[\text{last} - \lfloor n/2 \rfloor]$ è uguale ad esso, nel qual caso esiste una maggioranza di elementi uguali, essendo il vettore ordinato.

```
boolean hasMajority(int[] A, int n)
```

```
int candidate = A[ $\lfloor n/2 \rfloor$ ]
```

```
int last = searchLast(A, 0, n - 1, candidate)
```

```
return A[last] == A[last -  $\lfloor n/2 \rfloor$ ]
```

```
int searchLast(int[] A, int start, int end, int v)
```

```
if start == end then
```

```
    | return start
```

```
else
```

```
    | int mid =  $\lfloor (start + end)/2 \rfloor$ 
```

```
    | if v < A[mid] then
```

```
        | return searchLast(A, start, mid - 1, v)
```

```
    | else
```

```
        | return searchLast(A, Mid, end, v)
```
