**Componente connessa**

La componente connessa di un grafo non direzionato è un sottoinsieme dei vertici del grafo tali che per ogni coppia di vertici di questo sottoinsieme esiste un percorso che li congiunge nel grafo di partenza. Quindi, per ogni coppia u,v esiste un percorso che li congiunge.

Un problema importante è trovare le componenti connesse di un grafo.

L'idea è semplice:

Se si esegue la BFS a partire da una certa sorgente andiamo a visitare tutti i vertici raggiungibili dalla sorgente. Se due vertici sono entrambi raggiungibili dalla sorgente esiste un percorso tra questi due vertici. Siano u,v due vertici raggiungibili dalla sorgente, se io vado da u ad s (o da s ad u, è uguale, sono grafi non direzionati) e poi da s a v, ho trovato un percorso da u a v. Quando faccio una BFS di fatto scopro una componente connessa, perché son tutti quanti vertici congiunti l'uno all'altro da percorsi (che ovviamente passano per s). Ce ne potrebbero essere anche altri, ma siamo sicuri che passando per s riusciamo ad arrivare da un vertice ad un altro.

Se io eseguo la BFS una sola volta a partire da una certa sorgente scoprio tutti i vertici raggiungibili da quella sorgente, e questa è una componente connessa per ciò che abbiamo appena detto. Immaginiamo che esista un percorso da u a v, e che u e v non siano presenti nel BFSTree che creo quando faccio la visita BFS a partire da s. Ovviamente questo è assurdo se si trovano nella stessa componente connessa, ma vediamolo.

Supponiamo che il vertice v non sia nella componente connessa di s e u ma che esista un percorso tra u e v. Ma se c'è un percorso tra u e v, e u è presente nella componente connessa a partire da s, vuol dire che c'è un percorso tra s ed u, ma per ipotesi c'è il percorso tra u e v, sicuramente esiste un percorso tra s e v che passa per u, quindi anche v sarà inserito nell'albero BFS quando eseguiamo la BFS a partire dalla sorgente s.

Tutti i nodi che non avranno il campo discovered a true non faranno ovviamente parte di quella componente connessa. Tra questi ne prendo uno qualsiasi ed eseguo la BFS. Trovo quindi un'altra componente connessa, e vado ancora a controllare se ci sono altri nodi con il campo discovered a falso ed eseguendo la BFS trovo altri componenti connesse, e così via.

Riassunto in un algoritmo, chiamato **AllComponents(G)**:

**AllComponents(G)**

Per ogni nodo u di G setta discovered[u]=false   
Per ogni nodo del grafo setta discovered a false.

For each node u of G   
*per ogni nodo del grafo*

If Discovered[u]= false   
*se discovered è a false*

BFS(u)   
*invoca la BFS*

Endif

Endfor

Ricordarsi che quando si richiede il codice completo non bisogna trascrivere la BFS con la re-inizializzazione dei discovered a false, è già fatta e se la si rifà si perdono le informazioni riguardo all'algoritmo esterno.

Si può usare anche la DFS, che fa esattamente la stessa cosa, esplora solo il grafo in modo diverso. Per la DFS valgono comunque tutte le regole confutate per la BFS. A differenza, La DFS non deve cambiare parte del suo codice, siccome l'inizializzazione dei discovered a false viene fatta dal programma chiamante la DFS.

**Analisi dell'algoritmo:**

La singola DFS (O BFS, tanto hanno la stessa complessità computazionale) costa tempo lineare nella somma del numero di nodi e del numero di archi. Potremmo dire che ogni componente connessa ha al più un numero di vertici pari al numero di vertici di tutto il grafo e ha al più un numero di archi pari al numero di archi di tutto il grafo, ma così non viene una buona analisi. In realtà ciascuna DFS (o ancora, BFS) richiede tempo lineare nella somma del numero di nodi e del numero di archi che stanno nella componente connessa scoperta. Io non posso stimare in ogni componente connessa quanti vertici e quanti archi ci sono, ma posso stimare in totale il numero di nodi e di archi che stanno in tutte le componenti. Qui sfruttiamo un risultato semplicissimo, dimostrato, cioè che le componenti connesse sono disgiunte e noi le scopriamo tutte. Se sono disgiunte, cioè non hanno nodi in comune, è ovvio che se sommo i nodi di tutte le componenti connesse ottengo il numero totale di nodi, stesso discorso per gli archi.

Dimostriamo che , per ogni due nodi s e t di un grafo, le loro componenti connesse o sono uguali o disgiunte. (O sono proprio la stessa cosa oppure devono essere staccate tra di loro)

Supponiamo che non sia vero, ossia che non sono né disgiunte né uguali, cioè sono diverse ma hanno almeno un vertice un comune.

La prima affermazione, ossia che non sono disgiunte, implica che c'è un vertice u che appartiene sia alla componente connessa di s sia alla componente connessa di t, ovviamente questo vertice u è diverso sia da s che da t, perché s e t sono due vertici distinti che per ipotesi stanno nella loro propria componente connessa. Questo vertice sta nell'intersezione delle due componenti connesse, e ciò significa che c'è sia un percorso che va da s ad u, sia un percorso che va da t ad u. La seconda affermazione, ossia che sono diverse, implica che c'è sicuramente un vertice che sta in una componente connessa e non nell'altra. Senza perdere di generalità supponiamo che questo nodo v sia soltanto nella componente connessa di s. Ma, dalla prima deduciamo che io posso andare da t ad u e da u ad s. Dalla seconda che io posso andare da s a v, quindi di conseguenza io posso andare da t a v passando per u: [t->u->s->v]. Questo vuol dire che t e v devono stare nella stessa componente connessa e questo contraddice la seconda affermazione, perché abbiamo detto che v è quel nodo che si trova nella componente connessa di s e non in quella di t.

Questo vuol dire che se sommo tutti i nodi delle componenti connesse ottengo sicuramente n, e la stessa cosa vale per gli archi (m).

**Torniamo ad analizzare la complessità computazionale dell'algoritmo.**

Per ogni nodo u di G setta discovered[u]=false   
*Per ogni nodo del grafo setta discovered a false, tempo lineare O(n)*

For each node u of G   
*per ogni nodo del grafo, viene fatto n volte O(n)*

If Discovered[u]= false   
*se discovered è a false, viene fanno n volte O(n)*

BFS(u)   
*invoca la BFS, n volte*

Endif

Endfor

Indipendentemente dal numero di chiamate a quella DFS, il tempo di ciascuna DFS dipende esclusivamente dal numero di archi che sono nella componente scoperta.

Supponiamo che la prima volta ci siano n1 e m1 archi, la seconda volta n\_2 e m\_2, e così via, supponiamo di invocarla k volte, quindi di avere k componenti connesse. La prima volta mi costa O(n\_1 + m\_1). La seconda volta mi costa O(n2+m2), e così via, fino ad arrivare a O(n\_k+m\_k). Ho, quindi

O(n\_1 + m\_1) + O(n\_2 + m\_2) + ... + O(n\_k + m\_k). Sotto forma di sommatoria abbiamo:

Sommatoria per i che va da 1 a k di O(n\_i + m\_i). Porto l'O-grande fuori e posso farlo perché è una somma e ottengo O(Sommatoria per i che va da 1 a k di n\_i + m\_i).

Abbiamo detto che la sommatoria degli ni è uguale ad n, la sommatoria degli mi è uguale ad m. Il tutto è quindi O(n+m).

È importante assegnare ad un nodo la componente di cui fa parte. Qual è l'idea? Numerare tutte le componenti connesse (e ci sono anche tanti altri modi).

Possiamo associare ad ogni componente un array di interi. Avremo il vertice 1, il vertice 2 etc.

Andiamo a mettere nella prima locazione dell'array la componente di cui fa parte il vertice 1, nella seconda locazione la componente di cui fa parte il vertice 2. In che punto dell'algoritmo io devo creare l'associazione tra la cella dell'array e la componente connessa? Di cosa ho bisogno? Innanzitutto ho bisogno di un contatore, inizializzato a 0, che è il contatore delle componenti connesse. Non appena scopro un vertice non scoperto (discovered a falso). Ora, dipende da dove ho fatto partire il contatore. Se l'ho fatto partire da 0 lo incremento, ponendolo ad 1, altrimenti non c'è bisogno dell'incremento e lo possiamo fare dopo.

Quindi, se lo facciamo partire da 0, dobbiamo incrementare prima di far partire la BFS/DFS, altrimenti dobbiamo incrementare dopo la BFS/DFS, sempre nell'if.

La cosa importante è che questo i va passato alla BFS/DFS. Il contatore non va per nessun motivo incrementato nella BFS/DFS perché altrimenti andiamo a dare a ciascun vertice che sta nella stessa componente connessa un intero differente. Invece lo scopo era assegnare lo stesso intero a tutti i vertici della stessa componente connessa. Quindi, passiamo i alla BFS/DFS? E questo i verrà utilizzato per creare l'associazione nodo/componente connessa. Nella DFS ricorsiva, ad esempio,se l'if(v non è marcato explored) allora ci sono due casi:

* se l'associazione tra il nodo per cui abbiamo invocato la prima volta DFS(u) non è stata creata in AllComponents, allora l'associazione dobbiamo farla non appena entriamo in DFS(u) (A[u]=i).
* se l'associazione tra il nodo per cui abbiamo invocato la prima volta DFS(u) è già stata creata nell'if di AllComponents impostando A[u]=i (dove i è il contatore), allora l'associazione dobbiamo farla nel for, ogni volta che scandiamo un nodo v adiacente ad u che non è stato visitato, poniamo A[v] = i. L'importante è che non dimentichiamo di marcare la sorgente.

Nel caso della BFS, invece, possiamo utilizzare un array.

Anch'essa deve prendere in input un indice i e:

O associamo l'indice i ad u all'interno dell'if

O possiamo farlo nella BFS.

**Pseudocodice finale con BFS**

**AllComponents(G):**

count = 0

vettore[u] = 0 per ogni nodo di G

Foreach nodo u di G

setta discovered[u] = false

EndForeach

Foreach nodo u di G

If(discovered[u] = false)

Count = count + 1

Inizializza l’albero BFSTree T

BFS(u, count, T)

Endif

EndForeach

**BFS(u, count, T):**

contatore livelli i = 0

L[i] = {u}

Vettore[u] = count

Discovered[u] = true

While(L[i] != )

Inizializza L[i+1] con una lista vuota

Foreach v L[i]

Considera ciascun arco (v,w) incidente su v

If(discovered[w] = false)

Discovered[w] = true

Aggiunge w alla lista L[i+1]

Aggiunge l’arco (v,w) all’albero T

Vettore[w] = count

Endif

EndForeach

i = i + 1

EndWhile

**Pseudocodice finale con DFS**

**AllComponents(G):**

count = 0

vettore[u] = 0 per ogni nodo di G

Foreach nodo u di G

setta explored[u] = false

EndForeach

Foreach nodo u di G

If(explored[u] = false)

Count = count + 1

Inizializza l’albero DFSTree T

DFS(u, count, T)

Endif

EndForeach

**DFS(u, count, T)**

Explored[u] = true

Vettore[u] = count

Foreach arco (u,v)

If(explored[v] = false)

Aggiungi l’arco (u,v) a T

DFS(v, count, T)

EndIf

EndForeach