7.4.1.6.1. Analisi

Sia $G = (V, E)$ con $|V| = n$ e $|E| = m$.
Supponiamo che G abbia k componenti connesse $G_1, G_2, \ldots, G_k$ con $k \geq 1$.
Sia m_i il numero di archi in G_i, con $lei \leq k$
$\implies \sum_{i=i}^{k} m_i = m$

La complessità della visita di tutto il grafo è ottenuta sommando i seguenti contributi:

- Primi 2 cicli forall: $\Theta(n+m)$
- Terzo ciclo forall:
  - Scansione di L_v: $\Theta(n)$
  - *Esattamente* una `BFS` per ogni G_i: $\Theta\left(\sum_{i=1}^{k} m_i\right) = \Theta(m)$ù
    $\implies$ Complessità totale: $\Theta(n+m)$ (ovvero è lineare nella taglia del grafo)

> **Osservazione**
>
> Se G non è connesso potremmo avere $n > m$, quindi è necessario tenere n e m nell'ordine di grandezza.

7.1.4.7. Applicazione della BFS

7.1.4.7.1. Connettività

Input: Grafo $G = (V, E)$.
Output: Numero delle componenti connesse di G e vertici di ciascuna componente etichettati con ID distinti.

> **Osservazione**
>
> Se il numero restituito è 1, allora G è connesso.

Idea: Visita tutto G e quando si visita la i-esima componente connessa, si impostano gli ID dei suoi vertici a i (contatore globale).

`BFS(G,s,i)`: variante di `BFS(G,s)` in cui invece di impostare gli ID dei vertici visitati a 1, si impostano a i.

```
forall v ∈ V do v.ID<-0;
forall e ∈ E do e.label<-null;
forall v ∈ V do{
    if(v.ID = 0) then {
    i<-i + 1;
    BFS(G,v,i);
    }
return i;
```

Complessità: $\Theta(n+m)$

- Stessa struttura della visita completa di G;
- La modifica alla `BFS` non ne altera la complessità

7.1.4.7.2. Spanning tree

Input: Grafo $G=(V,E)$ connesso.
Output: Spanning tree di G (rappresentato come lista di archi).

Idea: Sfruttare il fatto che i `DISCOVERY EDGE` formano uno spanning tree della componente connessa del vertice da cui si parte.

`BFS(G,s,L)` con $L=\emptyset$ lista vuota. La modifica di `BFS(G,s)` che

- Inserisce una copia di ogni `DISCOVERY EDGE` in L
- Alla fine restituisce L

```
L<-lista vuota;
s<-vertice arbitrario di V
return BFS(G,s,L)
```

Complessità: $\Theta(m)$

- G è connesso e quindi la `BFS` costa $\Theta(m)$;
- La modifica alla `BFS` non ne altera la complessità.

7.1.4.7.3. Cammini minimi

Input: Grafo $G=(V,E)$ e due vertici $s,t \in V$.
Output: Cammino di lunghezza minima da s a t in G (rappresentato come lista di archi), se esiste, `null` se non esiste alcun cammino.

Modifichiamo la `BFS` come segue:

- $\forall v \in V$ uso due campi aggiuntivi: $v.parent$, $v.edge$;

- La visita di un vertice w scoperto da v consiste nell'impostare $w.parent \leftarrow v$ e $w.edge \leftarrow (v,w)$. Se $w = s$, entrambi i campi sono `null`.

```
forall v ∈ V do{
    v.ID<-0;
    v.parent<-null;
    v.edge<-null;
}
forall e ∈ E do e.label<-null;
BFS(G,s) //con le modifiche dette sopra
if(t-ID = 0) then return null;
L<-lista vuota;
w<-t;
while(w != s) do{
    aggiungi w.edge a L;
    w<-w.parent;
}
return L;
```

Complessità: $\Theta(n+m)$

- Primi 2 cicli forall: $\Theta(n+m)$;
- `BFS`: $O(m)$;
- Ciclo while: $O(n)$.
  $\implies$ Totale: $\Theta(n+m)$

7.1.4.7.4. Ciclicità

Input: Grafo $G = (V,E)$.
Output: Un ciclo in G (rappresentato come lista di archi), se esiste o `null` se non esiste.

Idea: visitare tutto il grafo G invocando `BFS` su ciascuna componente connessa.

- Se nessun arco è stato etichettato come `CROSS EDGE`, `return null`, perché i `DISCOVERY EDGE` non formano cicli;
- Se c'è un `CROSS EDGE`, c'è sicuramente un ciclo. Sia (u,v) il `CROSS EDGE`, etichettato tale durante `BFS(G,s)`
  [disegno]
  $u \in L_i$, $v \in L_i \vee L_{i+1}$
  - Se $v \in L_{i+1}$ inserisco nel ciclo (u,v) e $(v, v.parent)$ e poi risalgo in parallelo sino a z, aggiungendo al ciclo i `DISCOVERY EDGE` incontrati.

- Se $v \in L_i$ faccio la stessa cosa omettendo l'iniziale aggiunta di $(v, v.parent)$ al ciclo.

Dettagli e analisi per esercizio.

Complessità: $\Theta(n+m)$

7.1.4.8. Proposizione

Abbiamo appena dimostrato la seguente proposizione:

Dato $G=(V,E)$, con $|V|=n$ e $|E|=m$, i seguenti problemi possono essere risolti in tempo $O(m+n)$ usando la `BFS`:

1. Testare se G è connesso;
2. Trovare le componenti connesse di G;
3. Trovare uno spanning tree di G, se G è connesso;
4. Trovare un cammino minimo tra 2 vertici s e t, se esiste;
5. Trovare un ciclo, se esiste.

7.4.2. Depth-First Search

`DFS` è un algoritmo ricorsivo che, a partire da un vertice s,

- visita tutti i vertici di C_s;
- etichetta ciascun arco di C_s come `DISCOVERY` o `BACK EDGE`
  Si usano le *stesse ipotesi implementative della* `BFS`

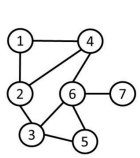
Algoritmo: `DFS(G,v)` (prima invocazione: $v=s$)
Input: grafo $G(V,E)$, $v \in V$
Output: visita ogni vertice raggiungibile da v e non ancora visitato, ed etichetta ogni arco esaminato come `DISCOVERY` o `BACK EDGE`.

> **Osservazione**
>
> Nella *prima invocazione* $(v=s)$, tutti i vertici v di C_s hanno $v-ID=0$, e tutti gli archi e di C_s hanno $e.label=\nu ll$.
> In un'invocazione generica, alcuni vertici e archi sono già stati visitati o etichettati.

```
visita il vertice v e imposta v .ID ← 1;
forall e ∈ Ga.incidentEdges(v ) do{
    if (e.label = null) then{
```

```
        w ← G.opposite(v ,e);
        if (w .ID = 0) then{
            e.label ← DISCOVERY EDGE;
            DFS(G,w);
        else e.label ← BACK EDGE;
    }
}
```

7.4.2.1. Esempio



Liste di adiacenza (ordine crescente)

1: 2 4
2: 1 3 4
3: 2 5 6
4: 1 2 6
5: 3 6
6: 3 4 5 7
7: 6

[vari passaggi sul suo pdf]