Idea di base

Cominciamo con il suddividere i vertici in tre sottoinsiemi:

- **bianco**: vertici non ancora scoperti(visitati).
- **grigio**: vertice scoperto i cui adiacenti non sono ancora stati tutti scoperti.
- **nero**: vertice scoperti i cui adiacenti sono stati tutti scoperti.

```
INIZIALIZZA(G)
  for ∀u ∈ V do
      u.color ← bianco
VISITA(G, s)
  s.color ← grigio
  while ∃ nodo grigio do
      u ← nodo grigio
      if ∃v bianco ∈ adj[u] then
          v.color ← grigio
      else
          u.color ← black
```

Versione astratta dell'algoritmo

Proprietà ed invarianti

Il colore di un nodo può solo passare da **bianco** → **grigio** → **nero**
Inoltre valgono anche le seguenti invarianti:

1. Se $(u, v) \in E$ e u è nero, allora v è grigio o nero
2. Tutti i vertici grigi o neri sono raggiungibili da s
3. Qualunque cammino da s ad un nodo bianco deve contenere almeno un vertice grigio
    - Se **s** è ancora grigio, questo è banalmente vero
    - Se **s** è nero, se non ci fosse nessun vertice grigio, allora ci sarebbe un nodo bianco adiacente che è impossibile per l'invariante 1

Sottografo dei predecessori

Per ogni vertice che viene scoperto, ci conviene ricordare quale vertice grigio ci ha permesso di scoprirlo.
Dunque dobbiamo ricordare anche l'arco che ci ha portato alla scoperta del nodo.

Per fare ciò associamo ad ogni vertice(nodo) un **attributo** che memorizza il nodo dal quale era stato scoperto:

```
INIZIALIZZA(G)
  for ∀u ∈ V do
      u.color ← bianco
      u.π ← nil
```

Aggiungiamo l'attributo π ad ogni attributo

```
VISITA(G, s)
  s.color ← grigio
  while ∃ nodo grigio do
      u ← nodo grigio
      if ∃v bianco ∈ adj[u] then
          v.color ← grigio
          v.π ← u
      else
          u.color ← black
```

Definiamo dunque
$G_\pi = (V_\pi, E_\pi)$ con
$V_\pi = \{v \in V : v.\pi \neq nil\} \cup \{s\}$
$E_\pi = \{(v.\pi, v) \in E : v \in V.\pi - \{s\}\}$

Al termine di $Visita(G, s)$, V_π è l'insieme di tutti i vertici neri(ovvero tutti i vertici raggiungibili da s)

Il sottografo dei predecessori ottenuto è un **albero**, detto **albero di scoperta**

Una volta costruito l'**albero di scoperta**, possiamo stampare il percorso da un nodo iniziale s ad un nodo u nel seguente modo.

```
PRINT-PATH(G, s, u)
  if u = s then
      stampa u
  else
      if u.π = nil then
          stampa "non esiste cammino da s ad u"
      else
          PRINT-PATH(G, s, u.π)
          stampa u
```

Algoritmo BFS

Dobbiamo prima definire una struttura dati D d'appoggio per gestire l'insieme di vertici grigi.
Servono inoltre le seguenti operazioni:

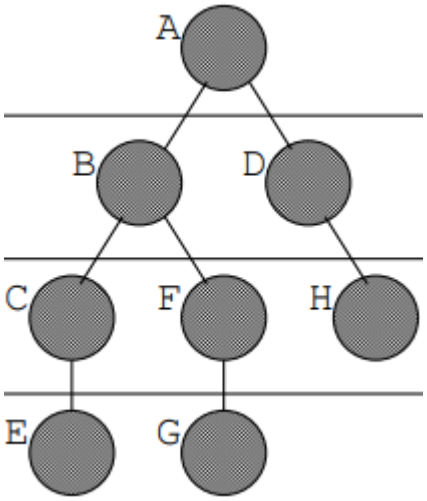
- **Make-empty**: crea una struttura nuova vuota
- **First(D)**: restituisce il primo elemento di **D** senza modificare la lista
- **Add(D,x)**: aggiunge l'elemento x alla struttura **D**
  - Se **D** è uno stack, allora x viene aggiunto come primo elemento
  - Se **D** è una queue(pila), allora x viene aggiunto come elemento
- **Remove-first(D)**: toglie da **D** il primo elemento
- **Not-empty(D)**: restituisce **true** se **D** non è vuota, false altrimenti

Inoltre supponiamo di usare le **liste di adiacenza** per rappresentare il grafo.
Se supponiamo che le operazioni di ADD e $REMOVE$ siano costanti come costo, allora la **complessità della visita** diventa $O(|V| + |E|)$

```
VISITA(G, s)
  D ← MAKE-EMPTY
  s.color ← grigio
  ADD(D, s)
  while NON-EMPTY(D) do
      u ← FIRST(D)
      for ∀v : v è bianco ed è ∈ adj[u] do
          v.color ← grigio
          v.π ← u
          ADD(D, v)
      u.color ← black
      REMOVE-FIRST(D)
```

Se la struttura d'appoggia D è una coda, allora diventa una visita in ampiezza (**BFS**) dal momento che vengono esplorati prima tutti i vertici adiacenti.

Dunque, durante la costruzione dell'albero corrispondente, prima di passare al livello successivo, viene completato tutto il livello corrente.



Per calcolare il livello, basta associare ad ogni nodo l'attributo d.

```
VISITA(G, s)
  D ← MAKE-EMPTY
  s.color ← grigio
  s.d ← 0
  ADD(D, s)
  while NON-EMPTY(D) do
      u ← FIRST(D)
      for ∀v : v è bianco ed è ∈ adj[u] do
          v.color ← grigio
          v.π ← u
          v.d ← u.d + 1
          ADD(D, v)
      u.color ← black
      REMOVE-FIRST(D)
```

L'attributo d viene inizializzato a ∞.
In questo modo viene calcolato il livello.

Al termina della visita **BFS**

$$\forall v \in V, v.d = \delta(s, v)$$

con $\delta(s, v)$ indica la distanza di v dal sorgente s della visita

Per ogni vertice v raggiungibile da s, il cammino da s a v è un cammino minimo nell'albero **BFS**

Algoritmo DFS

Se come struttura d'appoggio D usiamo invece una pila(**stack**), l'algoritmo diventa un **DFS**.
Anche la complessità rimane uguale.

Intervalli di attivazione

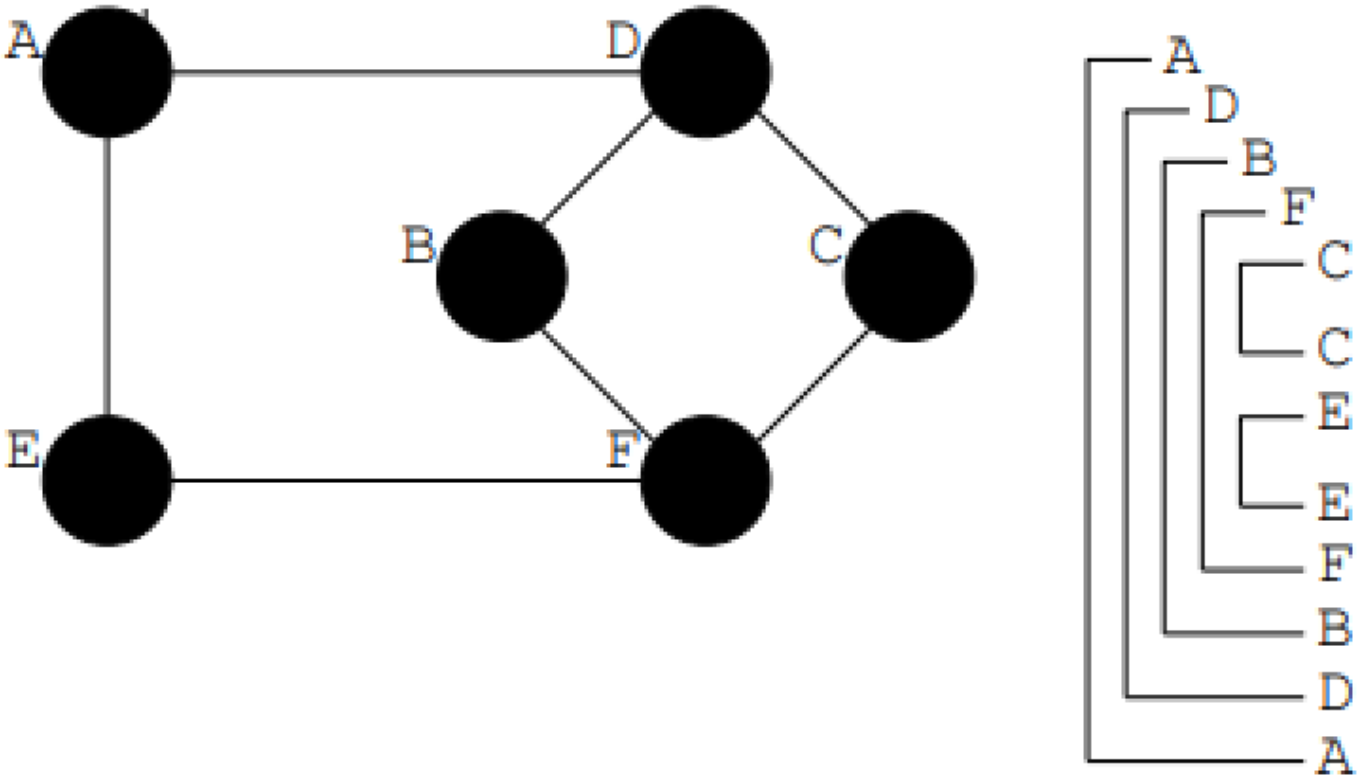
Possiamo aggiungere, ad ogni nodo, informazioni in più riguardante:

- **inizio visita** ovvero quando il nodo viene visitato per la prima volta

- **fine visita** ovvero quando finisco di visitare tutti i suoi figli per poi visitarlo per l'ultima volta.

Questi **intervalli di attivazione** possono essere disgiunti o contenuti in altri intervalli.

Esempio

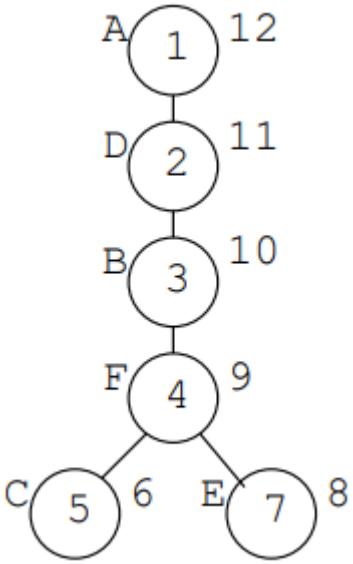


A destra vediamo la struttura d'attivazione di una visita **DFS** applicata al grafo a sinistra.

Possiamo notare che:

- **A** viene visitato all'inizio e alla fine della visita **DFS**
- L'ordine in cui vengono visitati i vertici è il seguente:
  - Visito **A**, visito poi il vertice adiacente **D**.
  - Visito **B**, vertice adiacente a **D**.
  - Visito **F**, vertice adiacente a **B**.
  - Visito **C**, vertice adiacente a **F**.

    - Ho già visitato tutti i vicini di **C**,quindi lo rivisito di nuovo, ritorno a **F** e visito il suo adiacente **E**.
    - Visito e rivisito **E**.
    - A cascata rivisito tutti gli altri vertici, finendo con il nodo **A**.
- L'albero risultante dalla visita **DFS** è il seguente:



    - **A** ha intervallo d'attivazione $(1, 12)$
    - **D** ha intervallo d'attivazione $(2, 11)$ e così via..

Teorema legato al DFS

Iniziamo classificando gli archi durante una visita **DFS**:

- **arco dell'albero:** arco inserito nella foresta **DFS**
- **arco all'indietro:** arco che collega un nodo ad un suo antenato
- **arco in avanti:** arco che collega un nodo ad un suo discendente
- **arco di attraversamento:** arco che collega due vertici non in relazione *antenato-discendente* tra di loro

Per classificare un generico arco (u, v), bisogna vedere il colore di v nella lista di adiacenti di u:

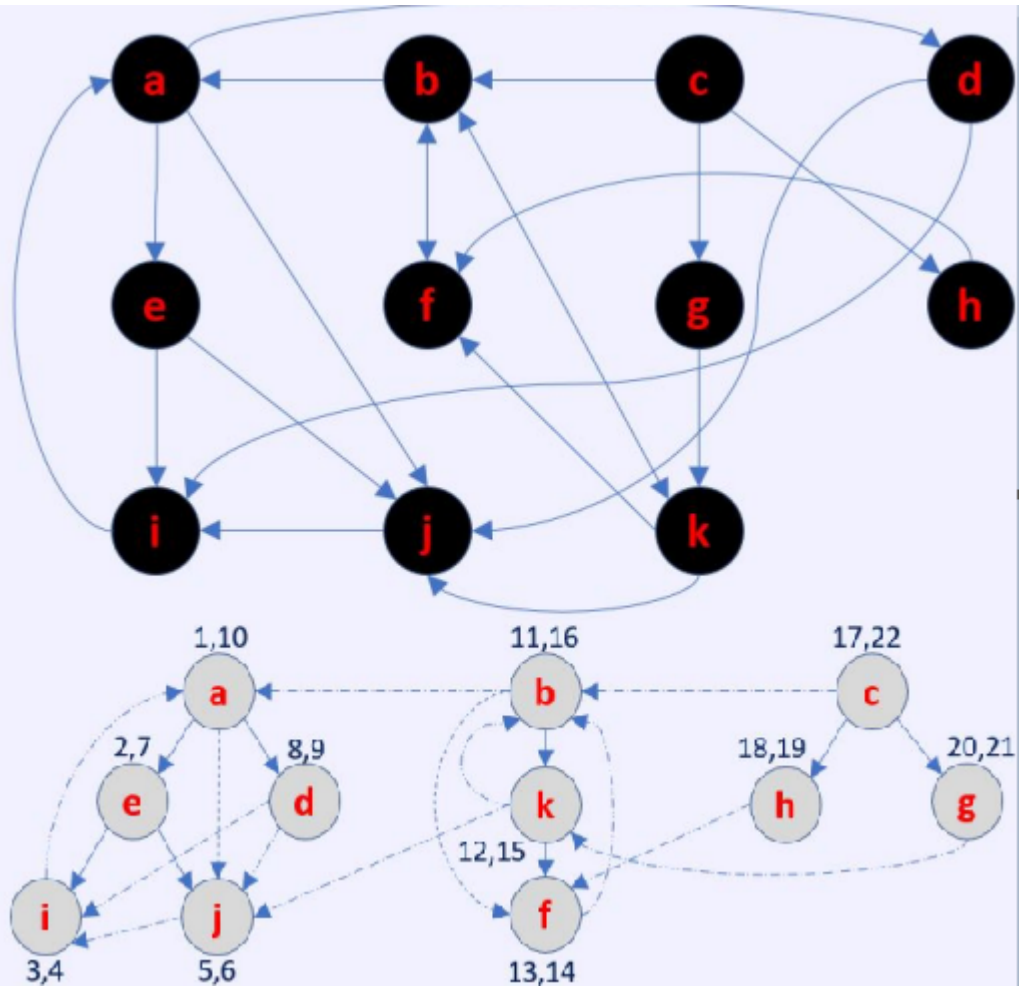
- **bianco:** (u, v) è un arco di foresta
- **grigio:** u è un discendente di v, dunque l'arco (u, v) è all'indietro.
- **nero:** ho finito di visitare v dunque:

- l'arco (u, v) è in avanti se v è un discendente di u
- altrimenti l'arco (u, v) è di attraversamento.

Teorema: In ogni visita **DFS** di un grafo non orientato, ogni arco è un *arco dell'albero* o un *arco all'indietro*

Teorema: un grafo(orientato o non) è **aciclico** se e solo se una visita **DFS** non produce archi all'indietro.

Esempio



La visita del grafo in alto produce la foresta in basso.
Procedimento:

- Iniziamo visitando **A**, dunque lo coloriamo di grigio.
- Procediamo a visitare **E** e quindi lo coloriamo di grigio.
- Visitiamo e coloriamo di grigio **I**.

- **I** è adiacente ad **A**
- Analizziamo dunque l'arco (I, A)
- **I** è grigio, e **A** pure, dunque l'arco (I, A) è all'indietro

Esiste un arco all'indietro, dunque questo grafo è **CICLICO**.
Ciò era intuibile anche guardando il disegno dal momento che esiste un ciclo dato dal cammino A, E, I, A