Grafi: visite, ordinamento topologico, componenti fortemente connessi

May 7, 2020

Obiettivi: visitare grafi in modo sistematico raccogliendo informazioni utili durante la visita. Argomenti: visita generale, visita in ampiezza, visita in profondità, test di aciclicità, ordinamento topologico, componenti fortemente connessi

1 Visita generica

Si vuole visitare tutti i nodi di un grafo $G = (V, E)$. La difficoltà rispetto a visitare alberi è dovuta agli eventuali cicli presenti nel grafo: possono esserci più cammini verso lo stesso nodo e quindi bisogna tener traccia dei nodi già visitati. Utilizziamo tre colori:

- **bianco**: nodi non ancora scoperti (non visitati),
- **grigio**: vertici scoperti di cui adiacenti non sono ancora tutti scoperti (nodi da cui bisogna andare ancora avanti; la frangia),
- **nero**: nodi scoperti di cui adiacenti sono già stati scoperti (nodi da cui non bisogna andare avanti più).

L'algoritmo che inizializza la visita su G:

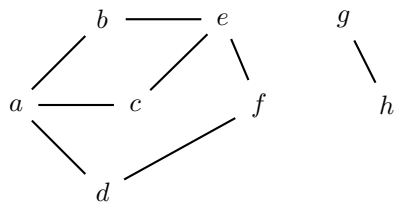
```
INIZIALIZZA(G)
for ∀v ∈ V do
    v.color ← bianco
end for
```

L'algoritmo che visita il grafo a partire dal nodo s:

```
VISITA(G, s)
s.color ← grigio
while ∃u ∈ V tale che u.color = grigio do
    u ← un nodo grigio
    if ∃v ∈ V tale che v.color = bianco ∧ (u, v) ∈ E then
        v ← un nodo tale che v.color = bianco ∧ (u, v) ∈ E
        v.color ← grigio
    else
        u.color ← nero
    end if
end while
```

L'algoritmo precedente è astratto nel senso che non definisce in che modo si sceglie un nodo fra quelli grigi e in che modo si rappresenta il grafo.

Consideriamo il grafo seguente:



Simuliamo una possibile esecuzione di VISITA(G, a) dopo aver inizializzato il grafo con INIZIALIZZA(G). La tabella indica l'ordine dei cambi di colore dei nodi e per ogni cambio *bianco* $\rightarrow$ *grigio* (tranne per il nodo a) indica l'arco utilizzato (u, v) dove u è il nodo grigio e v è il nodo bianco:

$$
\begin{aligned}
a &: bianco \xrightarrow{1} grigio \xrightarrow{8} nero \\
b &: bianco \xrightarrow{2,(a,b)} grigio \xrightarrow{4} nero \\
c &: bianco \xrightarrow{5,(e,c)} grigio \xrightarrow{7} nero \\
d &: bianco \xrightarrow{6,(a,d)} grigio \xrightarrow{12} nero \\
e &: bianco \xrightarrow{3,(b,e)} grigio \xrightarrow{10} nero \\
f &: bianco \xrightarrow{9,(e,f)} grigio \xrightarrow{11} nero \\
g &: bianco \\
h &: bianco
\end{aligned}
$$

L'insieme dei nodi grigi cambia come segue:

$$\{\} \xrightarrow{1} \{a\} \xrightarrow{2} \{a,b\} \xrightarrow{3} \{a,b,e\} \xrightarrow{4} \{a,e\} \xrightarrow{5} \{a,c,e\} \xrightarrow{6} \{a,c,d,e\} \xrightarrow{7} \{a,d,e\} \xrightarrow{8} \{d,e\} \xrightarrow{9} \{d,e,f\} \xrightarrow{10} \{d,f\} \xrightarrow{11} \{d\} \xrightarrow{12} \{\}$$

Un nodo diventa nero se viene scelto fra quelli grigi e non ha più adiacenti bianchi. I nodi g e h rimangono bianchi perché non sono raggiungibili dal nodo a.

Invarianti del ciclo **while** di VISITA(G, S):

- I_1: gli adiacenti dei nodi neri sono grigi o neri
- I_2: tutti i vertici grigi o neri sono raggiungibili da s
- I_3: qualunque cammino da s ad un nodo bianco deve contenere almeno un vertice grigio

Le invarianti I_1 e I_2 sono valide inizialmente ed è facile vedere che vengono mantenute dal ciclo. Per dimostrare l'invariante I_3 consideriamo due casi:

- se s è ancora grigio allora è banalmente vero,
- se s è nero: assumiamo per assurdo che non c'è nessun vertice grigio sul cammino verso un nodo bianco; allora ci sarebbe un nodo bianco adiacente ad uno nero; ma questo contraddice all'invariante I_1.

Il seguente teorema afferma la correttezza di VISITA(G, s).
Teorema: Al termine di VISITA(G, s):

$$v.color = nero \iff v \text{ è raggiungibile da } s$$

Dimostrazione:
$\Rightarrow$: segue dall'invariante I_2.
$\Leftarrow$: l'invariante I_3 e la condizione di uscita del ciclo implicano che non ci può essere nessun vertice bianco raggiungibile da s. $\square$

Visita dell'intero grafo: se rimangono nodi bianchi allora si può ripartire da uno di essi per visitare l'intero grafo:

```
VISITA-TUTTI-VERTICI(G)
INIZIALIZZA(G)
for ∀v ∈ V do
    if v.color = bianco then
        VISITA(G, v)
    end if
end for
```

La **complessità della visita completa**:

- Il costo associato con la gestione dei nodi dipende da come viene gestito l'insieme dei nodi grigi. Utilizziamo una struttura dati per rappresentare l'insieme dei nodi grigi e assumiamo che aggiungere e togliere un nodo ha costo costante ($O(1)$). Ogni nodo viene inserito e tolto dall'insieme dei grigi una volta a quindi il costo totale associato con i nodi è $O(|V|)$.
- Il costo associato con la gestione degli archi dipende da come viene effettuato il controllo "**If** $\exists v \in V$ tale che $v.color = bianco \wedge (u, v) \in E$ **then**". Dato che un nodo non può ridiventare bianco, non è necessario percorrere la lista di adiacenti di u sempre dall'inizio ma si può continuare dalla posizione dove si è trovato un nodo bianco l'ultima volta. Questo implica che ogni lista viene percorsa una volta e il costo totale associato con gli archi è $O(|E|)$.

Segue che il costo totale della visita completa è $O(|V| + |E|)$.

2 Visita in ampiezza

La visita in ampiezza utilizza una coda per gestire l'insieme dei nodi grigi. Durante la visita per ogni nodo segniamo il predecessore e il numero di passi dal nodo dove la visita ha inizio. Inizializzazione:

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

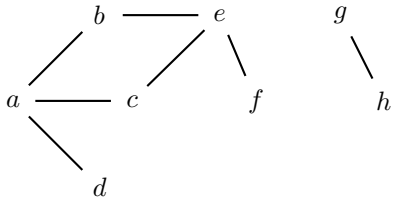
La visita in ampiezza a partire dal nodo s:

```
VISITA(G, s)
Q ← EMPTY-QUEUE
s.color ← grigio
s.π ← nil
s.d ← 0
ENQUEUE(Q, s)
while NON-EMPTY(Q) do
    u ← FRONT(Q)
    if ∃v ∈ V tale che v.color = bianco ∧ (u, v) ∈ E then
        v ← un nodo tale che v.color = bianco ∧ (u, v) ∈ E
        v.color ← grigio
        ENQUEUE(Q, v)
        v.π ← u
        v.d ← v.u + 1
    else
        u.color ← nero
        DEQUEUE(Q)
    end if
end while
```

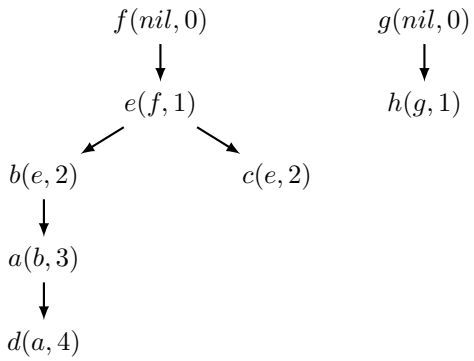
Il primo elemento della coda rimane lo stesso nodo finché ha adiacenti bianchi. Per questo motivo si può inserire tutti gli adiacenti bianchi di u con un ciclo e poi togliere u dalla coda:

```
VISITA(G, s)
Q ← EMPTY-QUEUE
s.color ← grigio
s.π ← nil
s.d ← 0
ENQUEUE(Q, s)
while NON-EMPTY(Q) do
    u ← FRONT(Q)
    while ∃v ∈ V tale che v.color = bianco ∧ (u, v) ∈ E do
        v ← un nodo tale che v.color = bianco ∧ (u, v) ∈ E
        v.color ← grigio
        ENQUEUE(Q, v)
        v.π ← u
        v.d ← v.u + 1
    end while
    u.color ← nero
    DEQUEUE(Q)
end while
```

Consideriamo il grafo:



Simuliamo l'algoritmo a partire prima dal nodo f e poi dal nodo g per visitare tutto il grafo. Gli attributi π definiscono una foresta dove i nodi con $\pi = nil$ sono le radici e dove d indica il livello del nodo (fra parentesi gli attributi π e d):



Proprietà della visita in profondità:

- al termine della visita in ampiezza a partire del nodo s:

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

  dove $\delta(s, v)$ indica la distanza di v dal dal nodo s, cioè la lunghezza del cammino minimo da s a v in termini di numero di archi;
- per ogni vertice v raggiungibile da s, il cammino da s a v nella foresta generata dalla visita in ampiezza è un cammino minimo.

3 Visita in profondità

La visita in profondità utilizza una pila per gestire l'insieme dei nodi grigi. Durante la visita per ogni nodo segniamo il predecessore e due attributi che, utilizzando un unico contatore chiamato *time*, segnano l'inizio visita (quando il nodo diventa grigio) e fine visita (quando il nodo diventa grigio) del nodo. Inizializzazione:

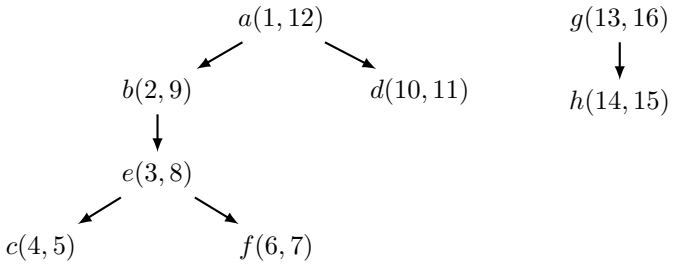
INIZIALIZZA(G)
for $\forall u \in V$ **do**
    $u.color \leftarrow bianco$
    $u.\pi \leftarrow nil$
    $u.i \leftarrow \infty$
    $u.f \leftarrow \infty$
end for
$time \leftarrow 1$

La visita in profondità a partire dal nodo s:

VISITA(G, s)
$S \leftarrow$ EMPTY-STACK
$s.color \leftarrow grigio$
$s.\pi \leftarrow nil$
$s.i \leftarrow time$
$time \leftarrow time + 1$
PUSH(S, s)
while NON-EMPTY(S) **do**
    $u \leftarrow$ TOP(S)
    if $\exists v \in V$ tale che $v.color = bianco \wedge (u, v) \in E$ **then**
        $v \leftarrow$ un nodo tale che $v.color = bianco \wedge (u, v) \in E$
        $v.color \leftarrow grigio$
        PUSH(S, v)
        $v.\pi \leftarrow u$
        $v.i \leftarrow time$
        $time \leftarrow time + 1$
    else
        $u.color \leftarrow nero$
        $u.f \leftarrow time$
        $time \leftarrow time + 1$
        POP(S)
    end if
end while

(In questo caso non si può inserire tutti gli adiacenti bianchi di u nella pila con un ciclo perché tale modo di procedere non sarebbe più una visita in profondità.)

Anche in questo caso la visita, grazie all'attributo π, genera una foresta. La visita completa del grafo precedente (a partire del nodo a a poi ripartendo dal nodo g) costruisce la seguente foresta dove per ogni nodo vengono indicati i e f:



La visita in profondità si può effettuare anche con un procedimento ricorsivo:

VISITA(G, s)
$s.color \leftarrow grigio$
$s.i \leftarrow time$
$time \leftarrow time + 1$
while $\exists v \in V$ tale che $v.color = bianco \wedge (s, v) \in E$ **do**
  $v \leftarrow$ un nodo tale che $v.color = bianco \wedge (s, v) \in E$
  $v.\pi \leftarrow s$
  VISITA(G, v)
end while
$s.f \leftarrow time$
$time \leftarrow time + 1$
$s.color \leftarrow nero$

Durante una visita in profondità è possibili classificare gli archi del grafo come segue:

- arco dell'albero: arco inserito nella foresta generata dalla visita,
- arco all'indietro: arco che collega un nodo ad un suo antenato in un albero della foresta,
- arco in avanti: arco che collega un nodo ad un suo discendente in un albero della foresta,
- arco di attraversamento: arco che collega due vertici che non sono in relazione antenato-discendente (fanno parte di due alberi distinti oppure di due rami distinti dello stesso albero).

Se si disegnano i figli da sinistra a destra e gli alberi da sinistra a destra allora gli archi di attraversamento vanno da destra a sinistra.

Un arco (u, v) viene classificato quando si esamina v nella lista di adiacenti di u. In quel momento $v.color$ può essere:

- *bianco*: allora (u, v) è un arco della foresta generata della visita
- *grigio*: allora u è un discendente di v in un albero della foresta, e quindi (u, v) è un arco all'indietro
- *nero*: allora la visita di v è già terminata (e quindi $v.f < u.f$), e quindi (u, v) è un arco
  - in avanti se v è un discendente di u e quindi se $u.i<v.i$
  - di attraversamento altrimenti e quindi se $v.i<u.i$

I precedenti casi forniscono un criterio per la classificazione.

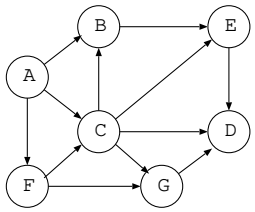
Proprietà della visita in profondità di tutti i nodi:

- teorema: in una visita in profondità di un grafo non orientato, ogni arco è un arco dell'albero o un arco all'indietro;
- teorema: un grafo, orientato o non orientato, è aciclico se e solo se una visita in profondità (qualunque) non produce archi all'indietro.

4 Ordinamento topologico

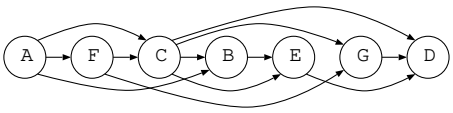
Definizione dell'ordinamento (o ordine) topologico: una funzione $\sigma : V \to \{1, ..., |V|\}$ tale che $\sigma(u) < \sigma(v)$ se esiste un cammino da u a v in G. Definizione equivalente: una funzione $\sigma : V \to \{1, ..., |V|\}$ tale che $\sigma(u) < \sigma(v)$ se esiste un arco da (u, v) in G.

Esempio:



$\sigma(A) = 1$, $\sigma(F) = 2$,
$\sigma(C) = 3$, $\sigma(B) = 4$,
$\sigma(E) = 5$, $\sigma(G) = 6$,
$\sigma(D) = 7$

Ridisegnando lo stesso grafo secondo l'ordine σ tutti gli archi vanno da sinistra a destra:



Il seguente algoritmo determina un ordine topologico sulla base di una visita in profondità.

TOPOLOGICAL-SORT(G)
$L \leftarrow$ lista vuota di vertici
INIZIALIZZA(G)
for $\forall u \in V$ **do**
  if $u.color = bianco$ **then**
    DFS-TOPOLOGICAL(G, u, L)
  end if
end for
restituisci L

DFS-TOPOLOGICAL(G, s, L)
$s.color \leftarrow grigio$
$s.d \leftarrow time$
$time \leftarrow time + 1$
while $\exists v \in V$ tale che $v.color = bianco \wedge (s, v) \in E$ **do**
  $v \leftarrow$ un nodo tale che $v.color = bianco \wedge (s, v) \in E$
  $v.\pi = s$
  DFS-TOPOLOGICAL(G, v, L)
end while
$s.color \leftarrow nero$
$s.f \leftarrow time$
$time \leftarrow time + 1$
in testa di L inserisci s

In pratica l'algoritmo crea una lista L che contiene i nodi in ordine decrescente di tempi di fine visita (l'attributo f). La correttezza dell'algoritmo si basa sul seguente ragionamento. In un grafo orientato e aciclico ogni arco (u, v) ricade in una delle tre seguenti categorie:

- arco della foresta generata dalla visita,
- arco in avanti,
- arco di attraversamento.

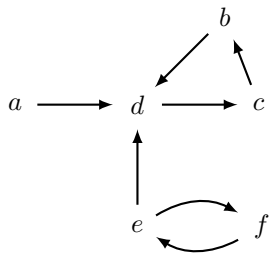
In tutti i tre casi, il tempo di fine visita di u è maggiore di quello di v ($u.f > v.f$). Quindi per ogni arco (u, v) in L il nodo u precede il nodo v. Quindi in L si ha un ordine topologico.

5 Componenti fortemente connessi

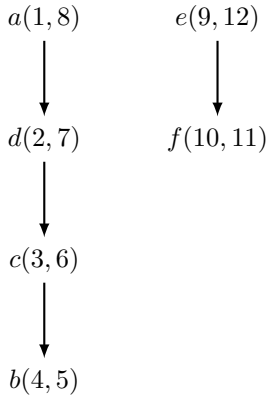
Due nodi, u e v, sono detti *mutuamente raggiungibili* (denotato con $u \leftrightarrow v$) se nel grafo esistono un cammino dal nodo u al nodo v e un cammino dal nodo v al nodo u. La relazione *mutuamente raggiungibile* è riflessiva ($u \leftrightarrow u$), simmetrica ($u \leftrightarrow v \implies v \leftrightarrow u$), e transitiva ($u \leftrightarrow v \wedge v \leftrightarrow z \implies u \leftrightarrow z$). Quindi la relazione *mutuamente raggiungibile* è una relazione di equivalenza e può essere utilizzata per partizionare i nodi del grafo. Le partizoini sono chiamate *componenti fortemente connesse* (cfc). Data una cfc, per qualunque coppia di nodi u e v della cfc abbiamo che u e v sono mutuamente raggiungibili e nessun nodo della cfc è mutuamente raggiungibile con un nodo che non fa parte della cfc.

Dato un grafo G, le sue cfc si possono determinare con due visite in profondità. Durante la prima visita si registrano i tempi di fine visita (l'attributo f). La seconda visita si fa sul trasposto del grafo (la direzione di ogni arco è invertita) e i nodi si considerano in ordine decrescente di tempi di fine visita (della prima visita) quando bisogna decidere da quale nodo far (ri)partire la visita in profondità. Nella foresta generata dalla seconda visita ogni albero corrisponde ad una cfc.

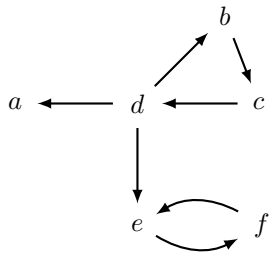
Applichiamo l'algoritmo al grafo seguente.



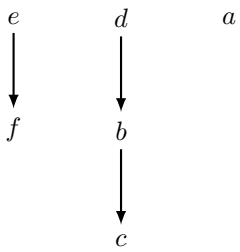
Risultato della prima visita con tempi di inizio e fine visita:



La seconda visita si fa sul trasposto riportato di seguito.



La seconda visita considera i nodi nell'ordine e, f, a, d, c, b per determinare da dove partire. Il risultato della seconda visita è la foresta seguente.



Nella foresta precedente ogni albero corrisponde ad una cfc. Quindi le cfc del grafo sono $\{e, f\}, \{d, b, c\}, \{a\}$.