GRAFI: VISITA GENERICA

[Deme, seconda edizione] cap. 12

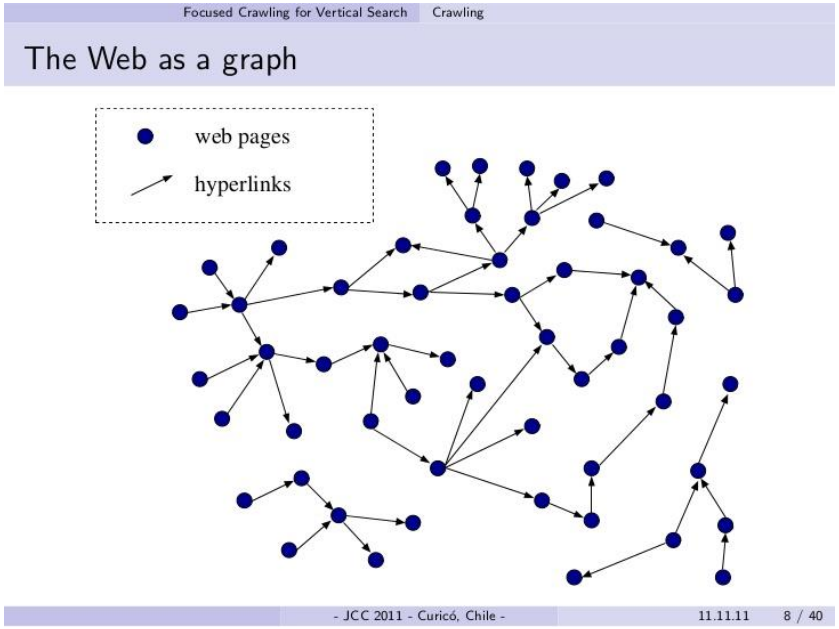
Sezione 12.3 fino alla sez. 12.3.1 esclusa

Scopo e tipi di visita

Una **visita** (o attraversamento) di un grafo G permette di **esaminare i nodi e gli archi di G in modo sistematico.**

È un problema di base in molte applicazioni.

ESEMPIO: Web Crawling. Un motore di ricerca (ad es. Google) deve indicizzare tutte le pagine web esistenti. Abbiamo già visto che il web può essere modellato con un grafo.



Scopo del motore di ricerca è di partire da una pagina web e «navigare» tutti i suoi hyperlink per indicizzare tutte le pagine da essa raggiungibili.
Non vogliamo però indicizzare la stessa pagina più volte (l'algoritmo non terminerebbe mai!).

Tipi di visita

Esistono vari tipi di visite con diverse proprietà. In particolare, due di esse assumono una «rilevanza» importante

- visita in **ampiezza** (**BFS=breadth first search**) e
- visita in **profondità** (**DFS=depth first search**)

Qui vediamo una versione astratta, istanziabile, denominata **visita generica**. Poi la «istanzieremo» sulle due politiche di vista.

Visita generica - inizializzazione

La nostra prima preoccupazione è di **non tornare «indietro»** (cioè visitare un nodo più volte).

IDEA: Dividiamo l'insieme dei vertici in tre insiemi colorati in modo diverso

- **Bianco:** Vertici non ancora scoperti (e quindi non visitati). Il libro li chiama **inesplorati**.
- **Grigio:** Vertici scoperti (e visitati) i cui adiacenti non sono ancora stati tutti scoperti, quindi vertici utili per continuare la visita. Il libro li chiama **aperti**.
- **Nero:** Vertici scoperti (e visitati) non più utili per continuare a scoprire altri vertici (i loro adiacenti sono stati tutti già scoperti). Il libro li chiama **chiusi**.

Algoritmo generico - inizializzazione

Dati n nodi, usiamo un vettore **color** di colori (in un'implementazione «reale» vanno bene degli interi) di grandezza n (n = numero dei vertici).

color[u] rappresenta lo **stato** del nodo u in una particolare visita.

Inizializziamo color[u] a white per ogni u (all'inizio della visita, tutti i nodi sono inesplorati).

INIZIALIZZA (G)

    color <- un vettore di lunghezza n

    for ogni u ∈ V **do**

        color [u] <- white

Visita generica - algoritmo

Partiamo da un nodo s (**sorgente**).

VISITA (G, s)

```
    INIZIALIZZA (G)
    color [s] <- gray
    {visita s}
    while ci sono vertici grigi do begin
        u <- scegli un vertice grigio
        if esiste v bianco adiacente ad u then
            color [v] <- gray
            {visita v}
        else color [u] <- black
    end
```

Partiamo da un vertice s, lo marchiamo come «scoperto» e lo visitiamo

Per ogni vertice grigio u incontrato, aggiungiamo i vertici v ad esso adiacenti (bianchi) a quelli scoperti e li visitiamo

Se u non ha vertici bianchi adiacenti, lo marchiamo come chiuso

Proprietà (cambiamento di colore nella visita).
Il colore di un vertice può solo passare da "white" a "gray" a "black".

Visita generica - invarianti

Partiamo da un nodo s.

VISITA (G, s)

INIZIALIZZA (G)

color [s] <- gray

{visita s}

while ci sono vertici grigi **do begin**

u <- scegli un vertice grigio

if esiste v bianco adiacente ad u **then**

color [v] <- gray

{visita v}

else color [u] <- black

end

INV1: Se $(u, v) \in E$ e u è nero, allora v è grigio o nero.

Dimostrazione: u è nero se e solo se non ha vertici adiacenti bianchi

INV2: Tutti i vertici grigi o neri sono raggiungibili da s.

Dimostrazione per induzione: caso base: s è raggiungibile da se stesso. Passo: se u è raggiungibile da s, lo è anche v tramite il cammino s, … u, v

Visita generica - invarianti

Partiamo da un nodo s.

VISITA (G, s)

    INIZIALIZZA (G)
    color [s] <- gray
    {visita s}
    while ci sono vertici grigi **do begin**
        u <- scegli un vertice grigio
        if esiste v bianco adiacente ad u **then**
            color [v] <- gray
            {visita v}
        else color [u] <- black
    end

INV3: Qualunque cammino da s a un vertice bianco deve contenere almeno un vertice grigio.

Dimostrazione.

Se s è grigio: vero.

Se s è nero: se non ci fosse nessun vertice grigio tra s ed il vertice bianco ci sarebbe un vertice bianco adiacente ad uno nero, che è impossibile per l'invariante INV1.

Teorema del colore dei vertici al termine della visita

Al termine dell'algoritmo di visita,

v è nero $\Leftarrow\Rightarrow$ v è raggiungibile da s.

Dimostrazione.

($\Rightarrow$) Per **INV2** (Tutti i vertici grigi o neri sono raggiungibili da s) all'uscita del while **tutti i vertici neri sono raggiungibili da s**.

($\Leftarrow$) Da **INV3** (Qualunque cammino da s a un vertice bianco deve contenere almeno un vertice grigio) si ricava che tra s e v (compresi) esiste almeno un vertice grigio, oppure v non è bianco.
Da quanto sopra, e dalla condizione di uscita del ciclo (non ci sono vertici grigi) si ricava che v non è bianco e non può essere grigio.
Quindi all'uscita del ciclo, **tutti i vertici v raggiungibili da s sono neri**.

Predecessori

L'algoritmo può essere modificato in modo da ricordare, per ogni vertice che viene scoperto, quale vertice grigio (**predecessore**) ha permesso di scoprirlo, ossia ricordare l'arco percorso.

Ad ogni vertice u si associa un attributo **π[u]** che rappresenta il vertice che ha permesso di scoprirlo.

L'algoritmo di inizializzazione deve essere esteso per inizializzare π:

INIZIALIZZA (G)

…

for ogni vertice u ∈ V[G] **do**

color [u] <- white

π[u] <- NULL

Visita generica con predecessori

VISITA (G, s)

color [s] <- gray

{visita s}

while ci sono vertici grigi **do begin**

u <- scegli un vertice grigio

if esiste v bianco adiacente ad u **then**

color [v] <- gray

π[v] <- u

{visita v}

else color [u] <- black

end

Ogni volta che un nodo passa da bianco a grigio (quindi nel momento in cui lo scopro) assegno a π[u] il vertice v da cui ho raggiunto u

Proprietà (predecessori dei vertici neri al termine della visita).
Al termine dell'esecuzione di VISITA (G,s)
tutti e soli i vertici neri diversi da s hanno
un predecessore diverso da **NULL**.

Sottografo/Albero dei predecessori

Dati

$$V_\pi = \{v \in V : \pi[v] \neq \text{NULL}\} \cup \{s\} \text{ e}$$
$$E_\pi = \{(\pi[v], v) \in E : v \in V_\pi - \{s\}\}$$

Il grafo $G_\pi = \{V_\pi, E_\pi\}$ è detto **sottografo dei predecessori**.

Si dimostra che **il sottografo dei predecessori è un albero** di radice s (è connesso e $|E_\pi| = |V_\pi| - 1$), e per ogni vertice u, G_π contiene il cammino «attraversato» dalla visita per raggiungere u partendo da s.

DOMANDA: il cammino all'interno del grafo, è il cammino **minimo** tra s e u?

Stampa del cammino

Algoritmo per stampare il cammino da s a u, dato π.

```
Print-Path (G, s, u)
        if u = s then stampa u
        else if π[u] = NULL
                then stampa "non esiste cammino"
        else Print-Path (G, s, π[u])
        stampa u
```

L'algoritmo funziona «all'indietro»: se u è diverso da s ma è comunque raggiungibile da s, stampo prima i suoi predecessori (chiamata ricorsiva «all'indietro») e poi stampo u

E se il grafo non è connesso?

***VISITA_TUTTI_I_VERTICI* (G)**

INIZIALIZZA (G)

for ogni u ∈ V **do**

if color [u] = white

then *VISITA* (G, u)

Al termine di una visita, controllo se ci sono vertici bianchi. Se sì applico la visita ad essi. Il risultato è una **foresta**.

DOMANDA: in questo modo visito alcuni vertici più volte?

Come gestisco i vertici grigi?

Abbiamo tralasciato un aspetto importante. Come gestisco l'insieme di vertici grigi?

Utilizziamo una struttura dati ordinata D (nel libro, D è chiama **frangia**). Su D possiamo fare queste operazioni:

Create() : restituisce una nuova D vuota

Add(D,x) : aggiunge un elemento x a D

First(D) : restituisce il primo elemento di D

RemoveFirst(D) : elimina il primo elemento di D

NotEmpty(D) : true se D contiene almeno un elemento, false altrimenti

Che struttura è D? è una **coda** se Add(D,x) aggiunge l'elemento in coda a D, è uno **stack** se Add(D,x) aggiunge l'elemento in testa a D

Visita con gestione di D

```
VISITA (G, s)
        D <- Create()
        color [s] <- gray
        {visita s}
        Add(D,s)
        while ci sono vertici grigi NotEmpty(D) do begin
                u <- scegli un vertice grigio First(D)
                if esiste v bianco adiacente ad u then
                        color [v] <- gray
                        π[v] <- u
                        {visita v}
                        Add(D,v)
                else
                        color [u] <- black
                        RemoveFirst(D)
        end
```

Complessità dell'algoritmo di visita generica

```
VISITA (G, s)
        D <- Create()
        color [s] <- gray
        {visita s}
        Add(D,s)
        while NotEmpty(D) do begin
                u <- First(D)
                if esiste v bianco adiacente ad u then
                        color [v] <- gray
                        π[v] <- u
                        {visita v}
                        Add(D,v)
                else
                        color [u] <- black
                        RemoveFirst(D)
        end
```

Bisogna, ad ogni ciclo (il numero di cicli è O(n)), controllare che esista un v bianco adiacente a u. Questo costo **(adj) dipende dalla rappresentazione**.

Ogni vertice viene inserito in D (al massimo) una volta (white->gray), e viene rimosso da D una volta (gray->black). Quindi **O(n)** operazioni totali su D

Costo visita = $O(n + adj)$

Costo della scansione dei vertici adiacenti (adj)

Come abbiamo detto, capire se esiste un vertice v bianco adiacente al vertice u (grigio) scelto dipende dalla rappresentazione usata. L'operazione va fatta O(n) volte, e per il costo della visita totale dobbiamo sommare questo costo alle **O(n)** operazioni fatte su D.

LISTA DI ARCHI:

Ogni volta (O(n) volte), bisogna scandire l'intera lista (costo O(m)).

Costo totale visita: O(n) + O(n*m) = O(mn)

MATRICI DI ADIACENZA

Ogni volta, bisogna scandire l'intera riga della matrice (costo O(n)).

Costo totale visita: O(n) + O(n*n) = $O(n^2)$.

LISTE DI ADIACENZA:

Possiamo ottimizzare la ricerca. Non è necessario scandirle ogni volta tutte, ma basta ricordarsi a che punto si è arrivati su ogni lista. Quindi in totale si ha un costo di O(m) per la ricerca (non moltiplicato per le O(n) volte).

Costo totale visita: O(n) + O(m) = O(n + m).

La rappresentazione con liste di adiacenza è la più efficiente per l'algoritmo di visita.

NB: è possibile ottimizzare anche le matrici di adiacenza.

Cosa devo aver capito fino ad ora

- Cos'è un algoritmo di visita
- Com'è fatto l'algoritmo di visita generica
- Teorema del colore dei vertici alla fine della visita
- Cos'è il sottografo dei predecessori e come si costruisce
- Come uso il sottografo dei predecessori
- Come visitare un grafo connesso
- Come gestire i nodi grigi nella visita generica
- Complessità della visita generica (in generale e dipendente dalla rappresentazione)

…se non ho capito qualcosa

- Alzo la mano e chiedo
- Ripasso sul libro
- Chiedo aiuto sul forum
- Chiedo o mando una mail al docente