1 Grafi come strutture dati

1.1 Introduzione e terminologia

Un grafo è una coppia di elementi (insiemi) **G=(V,E)** e consiste in:

- un insieme V di **vertici** (o **nodi**)
- un insieme E (E sottoinsieme del prodotto cartesiano $V \times V$) di coppie di vertici, detti **archi** o **spigoli**; ogni arco connette due vertici

I grafi possono essere:

- **orientati**: relazioni asimmetriche, insieme di coppie ordinate
- **non orientati**: relazioni simmetriche, insieme di coppie non ordinate

Un arco è **incidente** per i nodi che si toccano.
Il **grado** di un vertice è dato dal numero di archi incidenti.
Un vertice B è **adiacente** ad A se da B si può percorrere un solo arco e giungere ad A.
Un **sottografo** è una porzione di grafo (notazione $H \subseteq G$): i vertici di H sono sottoinsieme dei vertici di G e gli archi di H sono sottoinsieme degli archi di G.
Un **cammino** è una sequenza ordinata di archi che collegano due nodi. I cammini devono rispettare l'orientamento degli archi. La **lunghezza** è il numero di archi di cui è composto un cammino.
Un cammino si dice **semplice** se non passa due volte per lo stesso vertice. Se esiste almeno un cammino p tra i vertici v e w, si dice che w è **raggiungibile** da v. Inoltre v è un **antenato** di w e w è un **discendente** di v.
Un cammino tra due nodi v e w si dice **minimo** se tra v e w non esiste nessun altro cammino di lunghezza minore. La lunghezza del cammino minimo è detta **distanza** ($\delta(v, w)$).
Un grafo può essere **pesato**. La funzione peso è definita come $W : E \to \mathbb{R}$; per ogni arco $(v, w) \in E, W(v, w)$ definisce il **peso** di (v, w). In un grafo pesato, la lunghezza/peso di un cammino si calcola sommando i pesi degli archi che contiene.
I grafi non orientati possono essere:

- **connessi**: esiste un cammino da ogni vertice verso ogni altro vertice
- **non connessi**

I grafi orientati possono essere:

- **fortemente connessi**: esiste un cammino da ogni vertice verso ogni altro vertice
- **debolmente connessi**: ignorando il verso degli archi

Un cammino $\langle w_1, w_2, ..., w_n\rangle$ si dice **chiuso** se $w_1 = w_n$. Un cammino chiuso, semplice, di lunghezza almeno 1 si dice **ciclo**. Se un grafo non contiene cicli, si dice **aciclico**.
Un **grafo completo** è un grafo con un arco per ogni coppia di vertici. Un grafo completo ha numero di archi E pari a $|E| = \frac{|V|(|V|-1)}{2}$.
Un grafo non orientato, connesso e aciclico è definito **albero libero**. Se un vertice è designato ad essere radice, si definisce **albero radicato**. Un grafo non orientato, aciclico ma non connesso è definito **foresta**.

1.2 Rappresentazione

Per valutare un approccio di rapppresentazione, bisogna considerare lo **spazio** occupato dalla struttura dati e il **costo computazionale** delle operazioni da effettuare su di essa.

1.2.1 Lista di archi

Dati n (numero di vertici) e m (numero di archi), lo spazio occupato è $\mathcal{O}(n+m)$: è una rappresentazione inefficiente, in quanto bisogna percorrere tutto il grafo per scandire la lista di archi. Introdurre un vertice o arco ha costo $\mathcal{O}(1)$, ma la rimozione ha costo $\mathcal{O}(m)$.

1.2.2 Liste di adiacenza

Ogni vertice v ha una lista contenente i vertici ad esso adiacenti. Calcolare il grado di un vertice è un'operazione semplice, in quanto basta scorrere la lista di adiacenza. Occupa spazio $\mathcal{O}(n+m)$, ed è adatta per grafi **sparsi** (il numero di archi è molto minore del numero di vertici).

1.2.3 Liste di incidenza

Ogni vertice v ha una lista contenente un riferimento agli archi ad esso incidenti. Occupa spazio $\mathcal{O}(n+m)$.

1.2.4 Matrici di adiacenza

Il grafo è rappresentato tramite una matrice di interi di grandezza $n \times n$ (spazio occupato $\mathcal{O}(n^2)$); è adatta per grafi **densi**. Calcolare il grado e archi incidenti ha costo $\mathcal{O}(n)$ (basta scorrere la matrice). La modifica dei vertici ha costo $\mathcal{O}(n^2)$ in quanto bisogna ricostruire completamente la matrice. Una matrice di adiacenza rappresenta anche la presenza di un cammino di lunghezza 1 tra ogni coppia di vertici v e w. In particolare, $v \rightarrow_1 w$ se e solo se $M[v, w] \neq 0$: moltiplicando la matrice per sè stessa, il risultato è diverso da 0 solo se esiste un cammino di lunghezza 2 (e via dicendo).

1.2.5 Matrici di incidenza

Il grafo è rappresentato tramite una matrice di interi di grandezza $n \times m$ (spazio occupato $\mathcal{O}(n \times m)$), in cui le righe indicizzano i vertici e le colonne indicizzano gli archi.

2 Visite

2.1 Visita generica

Una **visita** di un grafo G permette di esaminare i nodi e gli archi in maniera sistematica, senza passare due volte per lo stesso nodo.

2.1.1 Inizializzazione

Una tattica per evitare di visitare un nodo più volte è quella di mappare lo stato della visita ad un colore:

- **bianco** (o **nodi inesplorati**): vertice non ancora esplorato
- **grigio** (o **nodi aperti**): vertice visitato, ma con nodi adiacenti ancora inesplorati
- **nero** (o **nodi chiusi**): vertice visitato, con adiacenti esplorati

Dati n nodi, si utilizza un vettore *color* di colori, di grandezza n: all'inizio della visita, tutte le celle del vettore *color* sono impostate a *white*.

Algoritmo 1 INIZIALIZZA(G)

color $\leftarrow$ vettore di lunghezza n
for ogni $u \in V$ **do**

color[u] $\leftarrow$ white
end for

La visita parte da un nodo s, detto **nodo sorgente**.

Algoritmo 2 VISITA(G,s)

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

Il cambiamento di colore è **monotono** (bianco $\rightarrow$ grigio $\rightarrow$ nero).

2.1.2 Invarianti

Un'**invariante** è una condizione che è verificabile come vera sia all'inizio sia alla fine di un ciclo:

- Invariante 1: se esiste un arco $(u, v) \in E$ ed u è nero, allora v è grigio o nero
- Invariante 2: tutti i vertici grigi o neri sono raggiungibili dalla sorgente
- Invariante 3: qualunque cammino dalla sorgente ad un vertice bianco deve contenere almeno un vertice grigio

Teorema. *Al termine dell'algoritmo di visita, v è nero se e solo se v è raggiungibile dalla sorgente.*

Dimostrazione. Per l'invariante 2, all'uscita dal ciclo tutti i vertici neri sono raggiungibili da s. Dall'invariante 3 si ricava che tra s e v esiste almeno un vertice grigio, oppure v non è bianco. Dato che la condizione di uscita dal ciclo è quella che non esistano più vertici grigi, si ricava che v non è bianco (cambiamento monotono) e non può essere grigio. Quindi, all'uscita dal ciclo, tutti i vertici raggiungibili dalla sorgente sono neri. $\square$

2.1.3 Predecessori

L'algoritmo può essere modificato in modo da ricordare, per ogni vertice che viene scoperto, quale vertice grigio ha permesso di scoprirlo, ossia ricordare l'arco percorso. Ad ogni vertice u si associa un attributo $\pi[u]$ che rappresenta il vertice che ha permesso di scoprirlo.

Algoritmo 3 VISITA(G,s)

INIZIALIZZA(G)
color $\leftarrow$ gray
$\{$visita $s\}$
while ci sono vertici grigi **do**

```
    u ← scegli un vertice grigio
    if esiste v bianco adiacente ad u then
        color[v] ← gray
        π[v] ← u
        {visita v}
    else color[v] ← black
    end if
end while
```

Proprietà. Al termine dell'esecuzione di VISITA(G,s), tutti e soli i vertici neri diversi da s hanno predecessore diverso da NULL.

Il sottografo dei predecessori è un albero (**albero dei predecessori**) di radice s.
Se il grafo non è connesso:

Algoritmo 4 VISITA TUTTI I VERTICI(G)

```
INIZIALIZZA(G)
for ogni u ∈ V do
    if color[u] = white then
        VISITA(G,u)
    end if
end for
```

2.1.4 Gestione dei vertici grigi

Per gestire i nodi grigi si usa una struttura dati ordinata D (**frangia**). Sulla frangia è possibile eseguire le seguenti operazioni:

- **Create()**: restituisce una 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)**: restituisce vero se D contiene almeno un elemento, falso altrimenti

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

Algoritmo 5 VISITA(G,s)

```
INIZIALIZZA(G)
Create()
color[s] ← gray
{visita s}
Add(D,s)
while NotEmpty(D) do
    u ← First(D)
    if esiste v bianco adiacente ad u then
        color[v] ← gray
```

```
        π[v] ← u
        {visita v}
        Add(D,v)
    else
        color[v] ← black
        RemoveFirst(D)
    end if
end while
```

2.1.5 Complessità

Il costo di visita è $\mathcal{O}(n+adj)$: adj è il tempo impiegato a controllare se esiste un nodo v bianco adiacente ad u, e dipende dalla rappresentazione; n è il numero di vertici, che vengono inseriti e rimossi da D.
Il costo di adj è:

- con lista di archi: bisogna scandire l'intera lista ($\mathcal{O}(m)$) per n volte ($\mathcal{O}(n)$), quindi $\mathcal{O}(n)+\mathcal{O}(n*m) = \mathcal{O}(mn)$
- con matrice di adiacenza: bisogna scandire l'intera riga della matrice ($\mathcal{O}(n)$), quindi $\mathcal{O}(n) + \mathcal{O}(n * n) = \mathcal{O}(n^2)$
- con liste di adiacenza: si possono ottimizzare le prestazioni utilizzando dei puntatori che puntano all'inizio delle liste di adiacenza. Se l'elemento è grigio, il puntatore è spostato all'elemento successivo; quando il puntatore giunge alla fine della lista, il primo elemento è colorato di nero. Ogni lista è percorsa una volta sola, in tutte le iterazioni del ciclo. Complessità: $\mathcal{O}(n + m)$.

2.2 Visita in ampiezza

2.2.1 Inizializzazione

La **visita in ampiezza** (**BFS**, Breadth First Search), esamina i vertici del grafo in un ordine ben preciso, costruendo un albero di visita chiamato **albero BFS**. Nell'albero BFS, ogni vertice si trova il più vicino possibile alla radice. La visita è realizzata usando la frangia come coda: quando un nodo grigio ha tutti gli adiacenti grigi, esso è rimosso dalla cosa (il vertice in testa rimane nella coda finchè non diventa nero).

Algoritmo 6 VISITA BFS(G,s)

```
INIZIALIZZA(G)
queue()
color[s] ← gray
{visita s}
enqueue(D,s)
while NotEmpty(D) do
    u ← head(D)
    if esiste v bianco adiacente ad u then
        color[v] ← gray
        π[v] ← u
        {visita v}
        enqueue(D,v)
```

```
    else
        color[v] ← black
        dequeue(D)
    end if
end while
```

2.2.2 Albero di visita

L'albero BFS viene costruito a livelli; l'albero rappresenta i cammini minimi. Anche se le liste di adiacenza vengono invertite, i nodi per livello non cambiano. Si può inizializzare un **vettore di distanze** (stimate) d, inizializzato ad infinito: se un determinato vertice non è stato trovato (distanza ∞ a fine BFS), allora non è raggiungibile da s.

2.2.3 Proprietà

Proprietà (1)**.** *In D ci sono tutti e soli i vertici grigi.*

Proprietà (2)**.** *Se $\langle v_1, v_2, \ldots, v_n \rangle$ è il contenuto di D, allora:*

i $d[v_i] \leq d[v_{i+1}]$*: i vertici sono ordinati per livelli nella coda*

ii $d[v_n] \leq d[v_1] + 1$*: la coda contiene al massimo due livelli*

Dimostrazione. Nel caso base, in D è presente solo la sorgente. La proprietà 2 è vera.
Il passo ha due casi:

- dequeue(D): o D rimane vuota (banalmente vera), o rimangono $\langle v_2, \ldots, v_n \rangle$, e
  i. le disuguaglianze sono ancora vere e quindi
  ii. anche $d[v_n] \leq d[v_1] + 1 \leq d[v_2] + 1$
- enqueue(D,v): v è reso figlio di v_1 e accodato, quindi $d[v] = d[v_1] + 1$ e
  i. $d[v_n] \leq d[v_1] + 1 = d[v]$
  ii. $d[v] = d[v_1] + 1 \leq d[v_1] + 1$

□

2.2.4 Dimostrazione $d[v] = \delta(s, v)$

Lemma (Invariante 4)**.** $d[v] = \delta(s, v)$ *per tutti i vertici grigi o neri.*

Dimostrazione $d[v] \geq \delta(s, v)$**.** Dato che l'albero dei predecessori π contiene solo archi appartenenti a G, il cammino da s a v è un cammino che appartiene anche a G, quindi la lunghezza del cammino da s a v nell'albero è maggiore o uguale alla distanza tra s e v.

Dimostrazione $d[v] \leq \delta(s, v)$**.** Definiamo l'insieme dei vertici a distanza k dalla sorgente nel grafo come $V_k = v \in V | \delta(s, v) = k$ (v_0 contiene solo la sorgente).
Nel caso base, $d[v_0] \leq \delta(s, v)$ (distanza di s da sè stesso: $0 \leq 0$). Sia $v \in V_k$: allora $\delta(s, v) = k$ (per definizione).
Con $k > 0$ (passo), esisterà almeno un vertice w tale che $\delta(s, w) = k - 1$ e $(w, v) \in E$, ovvero un arco che va da v a w. Definiamo l'insieme dei vertici appartenenti a V_{k-1} con arco entrante in v come $U_{k-1} = w \in V_{k-1} | (w, v) \in E$. Tra questi, sia u il primo vertice di U_{k-1} ad essere scoperto ed inserito nella coda: per politica FIFO, u sarà anche il primo ad essere estratto dalla coda. Quando guarderò i vertici adiacenti a u, v sarà ancora bianco (perchè più lontano), e v verrà inserito nell'albero come figlio di u,

con $d[v] = d[u] + 1$. Inoltre, per ipotesi induttiva, $d[u] \leq k - 1$.
Quindi, quando inseriremo v nell'albero:

- $d[v] = d[u] + 1$
- ma $d[u] \leq k - 1$, quindi $d[v] \leq (k - 1) + 1$
- $d[v] \leq k$