Progettazione di Algoritmi

Alessio Bandiera

Informatica, La Sapienza

Indice

Capitolo 1

Grafi

1.1 Grafi

1.1.1 Definizioni

Definizione 1.1.1.1 (Grafo)**.** Un grafo è una struttura matematica descritta da vertici, collegati da archi. Un grafo viene descritto formalmente come $G = (V, E)$, dove i $v \in V$ sono i *vertici* del grafo, mentre gli $e \in E$ sono gli *archi* (dall'inglese *edges*). In particolare, $V(G)$ è l'insieme dei vertici di G, comunemente indicato con n, mentre $E(G)$ è l'insieme degli archi di G, comunemente indicato con m. Presi due vertici $v_1, v_2 \in V(G)$, allora $(v_1, v_2) \in E(G)$ è l'arco che li collega.

Osservazione 1.1.1.1. $E(G) \subseteq V^2$.

Definizione 1.1.1.2 (Vertici adiacenti)**.** $v_1, v_2 \in V(G)$ sono detti *adiacenti* se $(v_1, v_2) \in E(G)$; in tal caso, si usa la notazione $v_1 \sim v_2$.

Definizione 1.1.1.3 (Grafo indiretto)**.** Un grafo è detto *indiretto* se gli archi non hanno direzione, o equivalentemente

$$\forall v_1, v_2 \in V(G) \quad (v_1, v_2) = (v_2, v_1) \in E(G)$$

Esempio 1.1.1.1 (Grafo indiretto)**.** Ad esempio, si consideri questo grafo indiretto:

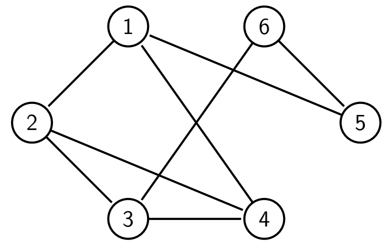

Figura 1.1: Un grafo indiretto.

in esso, si hanno

$$V(G) = \{1, 2, 3, 4, 5, 6\}$$

$$E(G) = \{(1, 2), (1, 4), (1, 5)(2, 3), (2, 4), (3, 4), (3, 6), (5, 6)\}$$

Definizione 1.1.1.4 (Grafo diretto)**.** Un grafo è detto *diretto* se gli archi hanno direzione, o equivalentemente

$$\forall v_1, v_2 \in V(G) \quad (v_1, v_2) \neq (v_2, v_1) \in E(G)$$

Esempio 1.1.1.2 (Grafo diretto)**.** Ad esempio, si consideri questo grafo diretto:

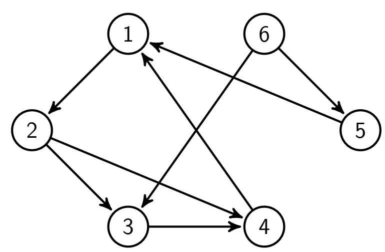

Figura 1.2: Un grafo diretto.

in esso, si hanno

$$V(G) = \{1, 2, 3, 4, 5, 6\}$$

$$E(G) = \{(1, 2), (2, 3), (2, 4), (3, 4), (4, 1), (5, 1), (6, 3), (6, 5)\}$$

Definizione 1.1.1.5 (Grado)**.** Il *grado* di un vertice $v \in V(G)$ è il numero di archi incidenti su v, indicato con $\deg(v)$.

Teorema 1.1.1.1 (Somma dei gradi)**.** *Dato un grafo G, la somma dei gradi dei vertici è pari a $2|E(G)|$.*

Dimostrazione. Sia G un grafo. Allora, ogni arco $e \in E(G)$ collega due vertici; allora necessariamente $\sum_{v \in V(G)} \deg(v) = 2|E(G)|$. □

Definizione 1.1.1.6 (Cappio)**.** Un arco con estremi coincidenti è detto *cappio.*

Esempio 1.1.1.3 (Grafo con cappio)**.** Un esempio di grafo con cappio è il seguente:

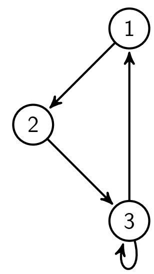

Figura 1.3: Un grafo diretto con cappio in 3.

Definizione 1.1.1.7 (Grafo semplice)**.** Un grafo è detto *semplice* se non contiene cappi, né lati multipli, ovvero più archi per due vertici.

1.1.2 Visite

Definizione 1.1.2.1 (Passeggiata)**.** Una *passeggiata* è una sequenza di vertici ed archi, della forma $v_0, e_1, v_1, e_2, \ldots, e_{n-1}, v_{n-1}, e_n, v_n$, dove $e_i = (v_{i-1}, v_i)$. È la visita di un grafo più generale, ed è possibile ripercorrere ogni arco ed ogni vertice.

Osservazione 1.1.2.1. La lunghezza massima di una passeggiata su un grafo è infinita.

Definizione 1.1.2.2 (Passeggiata chiusa)**.** Una passeggiata si dice *chiusa* se è della forma $v_0, e_1, v_1, e_2, \ldots, e_{n-1}, v_{n-1}, e_n, v_0$, dunque il primo e l'ultimo vertice coincidono.

Definizione 1.1.2.3 (Traccia)**.** Una *traccia* è una passeggiata aperta, in cui non è possibile ripercorrere gli archi, ma è possibile ripercorrere i vertici.

Esempio 1.1.2.1 (Traccia di un grafo)**.** Ad esempio, si consideri questo grafo indiretto:

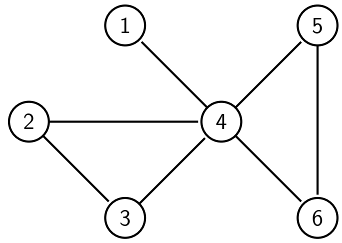

Figura 1.4: Un grafo indiretto.

in esso, si ha la traccia

$$\{5, (5, 4), 4, (4, 3), 3, (3, 2), 2, (2, 4), 4, (4, 6), 6\}$$

Definizione 1.1.2.4 (Circuito)**.** Un *circuito* è una traccia chiusa.

Definizione 1.1.2.5 (Cammino)**.** Un *cammino* è una traccia aperta, in cui non è possibile ripercurrere i vertici.

Osservazione 1.1.2.2. In una passeggiata in cui non si ripercorrono i vertici, non è possibile ripercorrere gli archi

Definizione 1.1.2.6 (Ciclo)**.** Un *ciclo* è un cammino chiuso.

Esempio 1.1.2.2 (Cicli di un grafo)**.** Ad esempio, si consideri questo grafo indiretto:

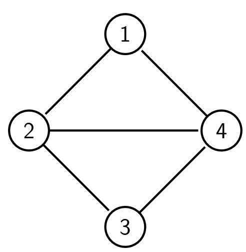

Figura 1.5: Un grafo indiretto.

in esso, si hanno tre cicli:

$$\{2, (2, 4), 4, (4, 3), 3, (3, 2), 2\}$$

$$\{2, (2, 4), 4, (4, 1), 1, (1, 2), 2\}$$
$$\{1, (1, 2), 2, (2, 3), 3, (3, 4), 4, (4, 1), 1\}$$

Definizione 1.1.2.7 (Grafo connesso)**.** Un grafo è detto *connesso* se per ogni $v_1, v_2 \in V(G)$ esiste una passeggiata che li collega.

Esempio 1.1.2.3 (Grafo non connesso)**.** Ad esempio, si consideri questo grafo:

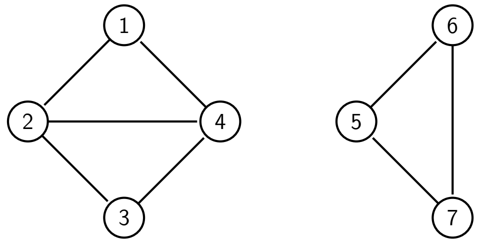

Figura 1.6: Un grafo non connesso.

Poiché non esiste una passeggiata che possa collegare 4 e 5, il grafo non è connesso.

Definizione 1.1.2.8 (Grafo fortemente connesso)**.** Un grafo diretto è detto *fortemente connesso* se per ogni $v_1, v_2 \in V(G)$ esistono due cammini diretti, che li collegano in entrambe i versi.

Esempio 1.1.2.4 (Grafo fortemente connesso)**.** Un esempio di grafo fortemente connesso è il seguente:

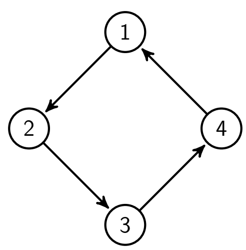

Figura 1.7: Un grafo fortemente connesso.

Definizione 1.1.2.9 (Passeggiata euleriana)**.** Una passeggiata si dice *euleriana* se attraversa ogni arco del grafo, senza ripercorrerne nessuno.

Osservazione 1.1.2.3. Una passeggiata euleriana è una traccia passante per ogni arco del grafo.

Esempio 1.1.2.5 (Passeggiata euleriana)**.** Ad esempio, si consideri il seguente grafo indiretto:

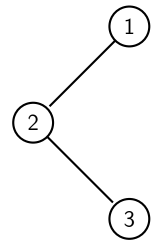

Figura 1.8: Un grafo indiretto.

in esso, l'unica passeggiata euleriana è

$$\{1, (1, 2), 2, (2, 3), 3\}$$

Teorema 1.1.2.1. *Dato un grafo G, esiste un circuito euleriano su G se e solo se G è connesso, e per ogni v, $\deg(v)$ è pari.*

Dimostrazione. Prima implicazione. Sia G un grafo avente un circuito euleriano; per assurdo, sia $v \in V(G) \mid \deg(v)$ non sia pari. Allora, percorrendo G secondo il circuito euleriano, giungendo a v non si potrebbe più lasciare tale vertice senza riattraversare uno degli archi gia visitati. Inoltre, se G non fosse connesso, il circuito non potrebbe essere euleriano poiché non potrebbe attraversare tutti gli archi di G. *Seconda implicazione.* TODO □

Definizione 1.1.2.10 (Passeggiata hamiltoniana)**.** Una passeggiata si dice *hamiltoniana* se TODO

1.2 Rappresentazione

1.2.1 Matrici di adiacenza

Definizione 1.2.1.1 (Matrice di adiacenza)**.** Sia $G = (V, E)$ un grafo; allora, è possibile rappresentare G attraverso una matrice $M_G \in \mathrm{Mat}_{n\times n}(\{0, 1\})$, dove

$$\forall m_{i,j} \in M_G \quad m_{i,j} = \begin{cases} 1 & i \sim j \\ 0 & i \nsim j \end{cases}$$

Osservazione 1.2.1.1 (Spazio di una matrice)**.** Lo spazio utilizzato da una matrice di adiacenza è pari a $O(n^2)$, poiché è necessario rappresentare l'adiacenza di ogni vertice con ogni altro.

Osservazione 1.2.1.2 (Aggiornamento di una matrice)**.** Per ogni grafo G, si ha che M_G è simmetrica; di conseguenza, il costo per aggiornare una matrice di adiacenza è $2O(1) = O(2) = O(1)$, poiché per $v_i, v_j \in V(G)$ non coincidenti, sarà necessario aggiornare $M_G[i, j]$ e $M_G[j, i]$.

Osservazione 1.2.1.3 (Controllo di adiacenza)**.** Per controllare che $v_i, v_j \in V(G)$ non coincidenti siano adiacenti, sarà sufficiente controllare $M_G[i, j] = M_G[j, i]$, e dunque il costo di un controllo è $O(1)$.

1.2.2 Liste di adiacenza

Definizione 1.2.2.1 (Liste di adiacenza)**.** Sia $G = (V, E)$ un grafo; allora, è possibile rappresentare G attraverso liste di adiacenza, salvando dunque una lista per ogni vertice, contenente i vertici ad esso adiacenti; in simboli

$$\forall v \in V(G) \quad v : [\hat{v} \in V(G) - \{v\} \mid \hat{v} \sim v]$$

Osservazione 1.2.2.1 (Spazio delle liste)**.** Dato un certo $v \in V(G)$, la lista di adiacenza corrispondente ha lunghezzadeg(v); allora, il numero di elementi nelle liste di adiacenza, per il Teorema 1.1.1.1, è pari a $O\left(\sum_{v \in V(G)} \deg(v)\right) = O\left(2|E(G)|\right) = O(2m) = O(m)$. Si noti inoltre che, per un grafo con pochi archi, nonostante si abbiano le liste poco riempite, è comunque necessario salvare i puntatori a tali liste, e dunque è necessario introdurre un $O(n)$ nel costo totale dello spazio, ottenendo allora $O(n) + O(m) = O(n + m)$.

Osservazione 1.2.2.2 (Controllo di adiacenza)**.** Nel caso peggiore, il grafo rappresentato da liste di adiacenza sarà composto da una sola lista per un certo $v \in V(G)$, contenente ogni altro vertice del grafo $\hat{v} \in V(G) - \{v\}$, e la lunghezza della lista di adiacenza di v sarà $n - 1$. Di conseguenza, il costo per controllare se due vertici sono adiacenti, utilizzando tale rappresentazione, è $O(n)$.

Esempio 1.2.2.1 (Rappresentazione di un grafo)**.** Ad esempio, si consideri il seguente grafo G:

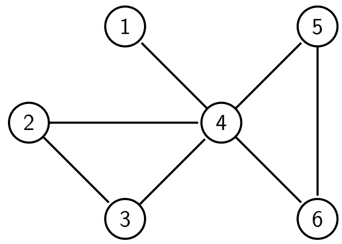

Figura 1.9: Un grafo indiretto.

allora, la sua corrispondente matrice di adiacenza è

$$M_G = \begin{pmatrix} 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 \end{pmatrix}$$

mentre le corrispondenti liste di adiacenza sono

$$\begin{cases} 1 : [4] \\ 2 : [4, 3] \\ 3 : [2, 4] \\ 4 : [1, 3, 5, 6] \\ 5 : [4, 6] \\ 6 : [4, 5] \end{cases}$$

1.3 Algoritmi

1.3.1 Trovare un ciclo

Algoritmo 1.3.1.1: Dato un grafo indiretto G, con ogni vertice avente grado almeno pari a 2, l'algoritmo restiuisce un ciclo di G.

Input : G grafo indiretto, tale che $\forall v \in V(G) \quad \deg(v) \geq 2$
Output: Un ciclo di G

1 **Function** `findCycle`(G)
2     $v \in V(G)$ `// un vertice qualsiasi di` G
3   `visited` $:= [v]$ `// conterrà i vertici visitati`
4     $v' \in V(G) : v \sim v'$
5   **while** $v' \notin$ `visited` **do**
6   `visited.add`(v')
7         $v' := v'' \in V(G) : \begin{cases} v' \sim v'' \\ v'' \neq \texttt{visited}[\texttt{visited.length} - 2] \end{cases}$
8   **end**
9   **return** `visited`[`visited.indexOf`(v')`:visited.length`]
10 **end**

Dimostrazione. L'algoritmo inizia scegliendo un qualsiasi vertice di G, denotato alla riga 2 con v; successivamente, alla riga 3 viene inizializzato un array `visited` che contterà tutti i vertici visitati attraverso l'algoritmo; inotre, alla riga 4 viene scelto un altro vertice v', che sia adiacente al v di partenza.

All'interno del ciclo `while`, alla riga 6 l'algoritmo salva v' all'interno dell'array di vertici visitati, mentre alla riga 7 viene rimpiazzato v', scegliendo un nuovo vertice, adiacente a v', che sia diverso dal penultimo vertice inserito all'interno di `visited`. Il motivo per cui quest'ultimo controllo è necessario, è che il penultimo vertice inserito sarà il vertice dal quale v' proveniva, di conseguenza si rischierebbe di ripercorrere uno stesso vertice più di una volta, e dunque non si formerebbe un ciclo. Si noti che è necessaria l'ipotesi per cui G abbia ogni vertice di grado almeno pari a 2, altrimenti non sarebbe possibile trovare un vertice differente dal penultimo di `visited`. Il ciclo termina nel momento in cui viene scelto un v' già presente all'interno di `visited`, in quanto, poiché non è possibile ripercorrere i propri passi, l'unica possibilità in cui si è giunti ad un vertice già visitato è se si è concluso un ciclo.

L'algoritmo termina restituendo uno slice dell'array, partendo dal primo indice di v' disponibile (si noti che alla fine dell'algoritmo anche l'ultimo elemento di `visited` sarà v'), fino alla fine. $\square$

Osservazione 1.3.1.1. Si noti che `visited` contiene esclusivamente i nodi visitati, dunque restituire interamente l'array potrebbe non fornire un ciclo, come nel seguente grafo

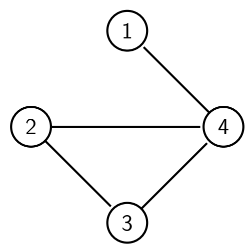

Figura 1.10: Un grafo contenente un ciclo.

in cui, ad esempio partendo da $v = 4$, l'unico ciclo è

$$\{4, (4, 3), 3, (3, 2), 2, (2, 4), 4\}$$

nonostante al termine dell'algoritmo si avrebbe `visited` $= [1, 4, 3, 2, 4]$, che non costituisce un ciclo.

Osservazione 1.3.1.2 (Costo dell'algoritmo)**.** Il costo di questo algoritmo dipende dalla struttura dati utilizzata per rappresentare il grafo in input: nel caso in cui G è rappresentato attraverso una matrice di adiacenza, il costo del ciclo `while` è pari a $O(n)$, poiché la riga 7 richiede di trovare un $v'' \in V(G) : v' \sim v''$, il ché potrebbe portare a dover scorrere tutta la riga/colonna di v'', dunque nel caso peggiore $O(n)$; differentemente, rappresentando G attraverso liste di adiacenza, basta scegliere il primo vertice contenuto nella lista di v'', e se questo dovesse coincidere con il penultimo elemento di `visited`, sarà sufficiente scegliere il secondo elemento della lista (sicuramente presente per come G è scelto in ipotesi), dunque si ha $O(2) = O(1)$.

Infine, si noti che il ciclo `while` ha costo $O(n)$, poiché nel caso peggiore si ha un ciclo che percorre tutto il grafo.

Allora, tramite matrice l'algoritmo ha costo $O(n) \cdot O(n) = O(n^2)$, mentre tramite liste si ha $O(1) \cdot O(n) = O(n)$.

1.3.2 DFS (Depth-First Search)

Algoritmo 1.3.2.1: Prima versione del DFS; dato un grafo indiretto G, e un vertice v di esso, l'algoritmo restituisce tutti i vertici raggiungibili attraverso un cammino, partendo da v.

Input : G grafo indiretto
Output: v un vertice di G

```
Function DFS₁(G, v)
    visited := [0] * n                          // array di n zeri
    visited[v] = 1
    Stack S = [v]
    while !S.isEmpty() do
        v_top = S.top()
        if ∃z ∈ V(G) : { z ~ v_top ; visited[z] = 0 } then
            S.push(z)
            visited[z] = 1
        else
            S.pop()
        end
    end
    return visited
end
```

Dimostrazione. TODO □

Osservazione 1.3.2.1 (Costo dell'algoritmo)**.** TODO

Algoritmo 1.3.2.2: Seconda versione del DFS; dato un grafo indiretto G, e un vertice v di esso, l'algoritmo restituisce tutti i vertici raggiungibili attraverso un cammino, partendo da v.

Input : G grafo indiretto
Output: v un vertice di G

```
Function DFS₂(G, v)
    TODO
end
```