Progettazione di Algoritmi

Simone Lidonnici

10 luglio 2024

Indice

1 Teoria dei grafi **2**

1.1 Tipi di grafi . . . 2
1.1.1 Grafi diretti e non diretti . . . 2
1.1.2 Passeggiate e cammini . . . 2
1.1.3 Grafi connessi e fortemente connessi . . . 3
1.1.4 Grafi ciclici . . . 3
1.2 Rappresentare un grafo . . . 4
1.2.1 Matrici di adiacenza . . . 4
1.2.2 Liste di adiacenza . . . 4
1.3 Trovare il ciclo in un grafo . . . 5
1.4 DFS (Ricerca in profondità) . . . 6
1.4.1 DFS ottimizzata . . . 7
1.4.2 DFS ricorsiva . . . 8
1.4.3 DFS in grafi diretti . . . 8
1.4.4 Componenti e DFS con componenti . . . 9
1.5 Ordinare un grafo . . . 9
1.5.1 Trovare l'ordine topologico in grafi diretti . . . 10
1.5.2 Trovare l'ordine topologico in grafi non diretti . . . 11
1.6 Intervalli di visita e tipi di archi . . . 11
1.6.1 Tipi di archi . . . 12
1.6.2 Algoritmo per controllare i tipi di archi . . . 14
1.7 Alberi di visita e cicli . . . 15
1.7.1 Grafi non diretti . . . 15
1.7.2 Grafi diretti . . . 15
1.7.3 Vettore dei padri . . . 16
1.8 Ponti . . . 17
1.8.1 Algoritmo per trovare i ponti . . . 18
1.9 Componenti fortemente connessi . . . 18

1

Teoria dei grafi

Definizione di Grafo

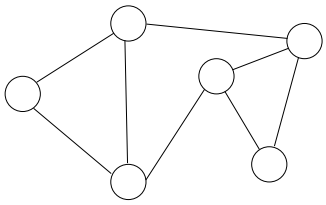
Un **grafo** G è una coppia (V, E) in cui V è un insieme di nodi e E un insieme di archi che collegano due nodi. Un grafo si dice **semplice** se:

- Non ha cappi, cioè nessun nodo è collegato con se stesso
- Ogni coppia di nodi è collegata da massimo un arco

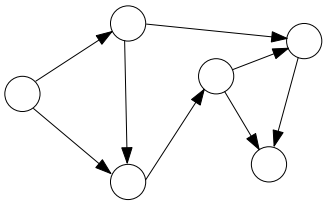
1.1 Tipi di grafi

1.1.1 Grafi diretti e non diretti

I grafi possono essere di due tipologie in base a se gli archi sono **orientati**, cioè partono da un nodo e arrivano ad un altro senza essere percorribili al contrario. Se il grafo ha archi orientati si dice **diretto**.



Grafo non diretto



Grafo diretto

1.1.2 Passeggiate e cammini

Nodi adiacenti

Due nodi collegati da un arco si dicono **adiacenti** (o vicini) e l'arco che li collega viene detto incidente. Per indicare che due nodi sono adiacenti scriviamo $x \backsim y$.
Si definisce il grado di un nodo $\deg(x)$ come il nomero dei suoi nodi adiacenti, uguale al numero di archi incidenti.

Definizione di passeggiata

Una **passeggiata** su un grafo è una sequenza di archi e nodi:

$$v_0 e_1 v_1 e_2 \dots e_n v_n$$

In cui ogni arco e_i collega il nodo v_{i-1} al nodo v_i.
Un **cammino** è una passeggiata in cui non si ripetono i nodi.

1.1.3 Grafi connessi e fortemente connessi

Definizione di grafo connesso

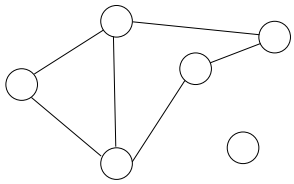
Un grafo G si dice **connesso** se per qualsiasi coppia di nodi esiste un cammino che li collega:

$$\forall v_i, v_j \in V(G) \exists \text{cammino} | v_1 \to v_j \vee v_j \to v_i$$

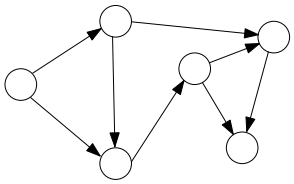
Un grafo G si dice **fortemente connesso** se per qualsiasi coppia di nodi esiste un cammino che li collega partendo da entrambi i nodi:

$$\forall v_i, v_j \in V(G) \exists \text{cammino} | v_1 \to v_j \wedge v_j \to v_i$$

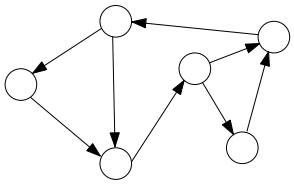
Nel caso di grafi non diretti ogni grafo connesso è anche fortemente connesso.



Grafo non connesso



Grafo connesso



Grafo fortemente connesso

Esiste un tipo specifico di passeggiata detta **passeggiata Euleriana** in cui si attraversano tutti i nodi una sola volta. Può esistere una passeggiata Euleriana in un grafo solo se il grafo è connesso e ci sono al massimo 2 nodi con grado dispari, che saranno inizio e fine.

1.1.4 Grafi ciclici

Definizione di grafo ciclico

Un grafo G è ciclico se esiste un sottogrppo connesso in cui ogni vertice ha grado ≥ 2.
Se nel grafo tutti i vertici hanno grado ≥ 2 allora il grafo è sicuramente ciclico.

$$\forall v \in V(G) \deg(v) \geq 2 \implies G \text{ ciclico}$$

In un grafo diretto se ogni nodo ha almeno un arco uscente allora il grafo è ciclico.

1.2 Rappresentare un grafo

1.2.1 Matrici di adiacenza

I grafi possono essere rappresentati con delle matrici di adiacenza in cui se v_i è adiacente a v_j la matrice conterrà 1 nella posizione (i, j):

	v_1	...	v_j	...	v_n
v_1	0				
...					
v_i			1		
...					
v_n					0

Costo per controllare se x è vicino di y: $O(1)$
Spazio necessario per l'archiviazione: $O(n^2)$

1.2.2 Liste di adiacenza

Per rappresentare i grafi si può anche usare una lista di adiacenza in cui ogni nodo ha una lista contenente tutti i suoi vicini:

$$v_1.\text{neighbors} = [\dots]$$
$$\dots$$
$$v_n.\text{neighbors} = [\dots]$$

Nel caso di un grafo diretto, ogni nodo avrà due liste:

- v_i.neighbors_out che contiene i nodi collegati da archi uscenti da v_i
- v_i.neighbors_in che contiene i nodi collegati da archi entranti in v_i

Costo per controllare se x è vicino di y: $O(n)$
Spazio necessario per l'archiviazione: $O(n^2)$
Lunghezza della lista di vicini di un determinato nodo v_i: $\deg(v_i)$
Grandezza totale delle liste: $O(n) + O(\sum_{i=1}^{n} \deg(v_i)) = O(n + m)$

1.3 Trovare il ciclo in un grafo

Dato un grafo G in cui ogni vertice ha grado ≥ 2, l'algoritmo per trovare il ciclo:

Algoritmo: Ricerca di un ciclo in un grafo G

Input:

- G: grafo

Output:

- C: nodi che formano il ciclo

```
def FindCiclo(G):
    x=V[0]
    C=[x]
    current=x
    next=x.neighbors[0]
    while next not in C : // finchè non trovo un nodo già visitato
        C.append(next)
        current=next
        if current.neighbors[0]!=C[-2] :
            next=current.neighbors[0]
        else
            next=current.neighbors[1]
    while C[0]!=next :
        C.pop(0)
    return C
```

1.4 DFS (Ricerca in profondità)

La **DFS** (Depth first search) è un modo per visitare un grafo che consiste nel partire da un nodo e spostarsi in un vicino casuale non ancora visitato e nel caso tutti i vicini di un nodo siano già stati visitati ritornare al nodo precedente. Per implementare questo roll-back si utilizza uno Stack. L'algoritmo ritorna tutti i nodi visitabili dal nodo di partenza, quindi nel caso di grafo non connesso, ritornerà solo i vertici nel sottografo contenente il nodo di partenza.

Algoritmo: DFS

Input:

- G: grafo
- x: nodo di partenza

```
def DFS(G, x):
    Vis=set()
    Stack S=[x]
    while len(S)!=0 :
        y=S.top()
        if ∃ z in y.neighbors | z ∉ Vis :  // O(deg(y) · n)
            Vis.add(z)
            S.push(z)
        else
            S.pop()
    return Vis
```

Dimostrazione per assurdo:
Supponiamo esista $y | \exists$cammino $x \to y$ ma $y \notin$ Vis e sia i un indice per cui $v_i \in \text{Vis} \wedge v_{i+1} \notin \text{Vis}$.
$v_i \in \text{Vis} \implies \begin{cases} v_i \text{ è stato inserito in } S \\ v_i \text{ è stato tolto da } S \end{cases} \implies$ ogni vicino di v_i è stato inserito in Vis $\implies v_{i+1}$ è stato inserito in Vis

1.4.1 DFS ottimizzata

L'algoritmo di base della DFS è poco ottimizzato per via del costo dell'if che richiede $O(\deg(y) \cdot n)$, per ottimizzarlo si cambia la struttura di Vis rendendolo un array lungo n in cui:

$$Vis[v] = \begin{cases} 0 & v \text{ non è stato visitato} \\ 1 & v \text{ è stato visitato} \end{cases}$$

Con questo cambiamento l'algoritmo diventa:

Algoritmo: DFS ottimizzata

Input:

- G: grafo
- x: nodo di partenza

```
def DFS_ott(G, x):
    Vis[x]=1
    Stack S=[x]
    while len(S)!=0 :
        y=S.top()
        if Vis[y.neighbors[0]]==1 :  // O(deg(y) · n)
            z=y.neighbors[0]
            Vis[z]=1
            S.push(z)
        y.neighbors.remove(0)
        if len(y.neighbors==0) :
            S.pop()
    return Vis
```

Avendo tutto costo $O(1)$ tranne il ciclo while con costo $O(n+m)$, l'algoritmo ha costo complessivo $O(n+m)$.

1.4.2 DFS ricorsiva

Della DFS si può fare anche una versione ricorsiva:

Algoritmo: DFS ricorsiva

Input:

- G: grafo
- x: nodo di partenza
- Vis: array nodi visitati

```
def DFS_ric(G, x):
    Vis[x]=1
    for y in x.neighbors :
        if Vis[y]==0 :
            DFS_ric(G, y, Vis)
    return Vis
```

Il costo di questo algoritmo è $O(n + m)$.

1.4.3 DFS in grafi diretti

Nel caso di grafi diretti bisogna cambiare l'algoritmo per controllare solo gli archi uscenti e non quelli entranti quando si cambia nodo:

Algoritmo: DFS

Input:

- G: grafo
- x: nodo di partenza

```
def DFS_dir(G, x):
    Vis[x]=1
    Stack S=[x]
    while len(S)!=0 :
        y=S.top()
        if ∃ z in y.neighbors_out | Vis[z]==0 :
            Vis[z]=1
            S.push(z)
        else
            S.pop()
    return Vis
```

1.4.4 Componenti e DFS con componenti

Definizione di componente

Un **componente** è l'insieme di nodi di un sottografo connesso, però non connesso al resto del grafo.

$$Comp[x] = \text{nodi nello stesso componente che contiene } x$$
$$Comp[x] = Comp[y] \iff x, y \text{ appartengono allo stesso sottografo}$$

L'algoritmo che visita tutti i componenti è una modifica della DFS ricorsiva in cui:

$$Comp[v] = \begin{cases} 0 & v \text{ non è ancora stato visitato} \\ i & v\text{è nel componente } i \end{cases}$$

Si aggiunge inoltre una funzione per cambiare componente in cui si trova il nodo corrente:

Algoritmo: DFS per trovare componenti

Input:
- G: grafo

```
def CComp(G):
    comp_count=0
    for x in V :
        if Comp[x]==0 :
            comp_count+=1
            DFS_ric_comp(G, x, Comp, comp_count)
    return Comp


def DFS_ric_comp(G, x, Comp, comp_count):
    Comp[x]=comp_count
    for y in x.neighbors :
        if Comp[y]==0 :
            DFS_ric_comp(G, y, Comp, comp_count)
    return Comp
```

1.5 Ordinare un grafo

Un grafo diretto G ha un **ordine topologico** se esiste un ordine per cui ogni nodo ha archi uscenti che vanno solo verso nodi successivi nell'ordine e archi entranti solo da nodi precedenti nell'ordine. Inoltre:

$$G \text{ ciclico} \iff \nexists \text{ ordine topologico}$$

Corollario:

$$G \text{ non ciclico} \implies \exists v \in V | v \text{ non ha archi uscenti}$$

1.5.1 Trovare l'ordine topologico in grafi diretti

Per trovare l'ordine topologico in grafi diretti si usa un'algoritmo:

Algoritmo: DFS per trovare l'ordine topologico in grafi diretti

Input:
- G: grafo

```
def DFS_ord(G):
    l=[]
    while len(G)!=0 :  // O(n)
        x=no_archi(G)
        l.insert(x,0)
        elimina(x)
    return l


def no_archi(G): // O(n)
    for v in V :
        if len(v.neighbors_out)==0 :
            return v


def elimina(x):  // O(m)
    for e in E :
        if x in e :
            E.remove(e)
```

Il ciclo while esegue n volte le funzioni no_archi e elimina, quindi il costo dell'algoritmo sarà: $O(n(n+m))$

1.5.2 Trovare l'ordine topologico in grafi non diretti

Per trovare l'ordine topologico in grafi non diretti si usa un'algoritmo:

Algoritmo: DFS per trovare l'ordine topologico in grafi non diretti

Input:
- G: grafo

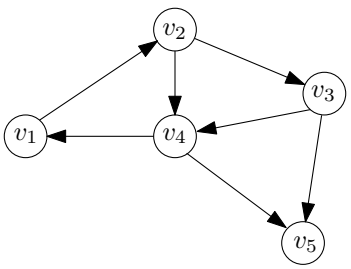
```
def ord_top(G):
    L=[]
    for v in V :
        if Vis[v]==0 :
            DFS_ord(G, v, Vis, L)
    return L


def DFS_ord(G, v, Vis, L):
    Vis[v]=1
    for w in v.neighbors :
        if Vis[w]==0 :
            DFS_ord(G, w, Vis, L)
    L.insert(v,0)
```

1.6 Intervalli di visita e tipi di archi

Dato un grafo G aggiungiamo un contatore C alla DFS, che parte da 1 e viene aumentato di uno ogni volta che si visita un nodo nuovo.
Ad ogni nodo $v \in V$ associamo:

- $t(v)$: valore di C quando v viene visitato per la prima volta
- $T(v)$: valore di C quando v viene rimosso dallo Stack
- $\text{Int}(v) = [t(v), T(v)]$

Esempio:



Una possibile tabella contenente gli intervalli usando una DFS partendo da v_1 è:

v	$t(v)$	$T(v)$
v_1	1	5
v_2	2	5
v_3	3	5
v_4	5	5
v_5	4	4

Dalla tabella e dal grafico possiamo osservare che:

- $t(v_i) \neq t(v_j)\ \forall i, j$
- $t(v_i) \leq T(v_i)$
- $t(v_i) = T(v_i) \iff v_i$ non ha archi uscenti e non è radice
- v_i radice $\iff \text{Int}(v_i) = [1, n]$ con G che ha n nodi

Inoltre confrontando gli intervalli tra due nodi v_1 e v_2 ci sono 3 possibilità:

- $\text{Int}(v_1) \subset \text{Int}(v_2)$
- $\text{Int}(v_1) \supset \text{Int}(v_2)$
- $\text{Int}(v_1) \cap \text{Int}(v_2) = \emptyset$

1.6.1 Tipi di archi

Albero di visita

Un **albero di visita** è un sottografo connesso e aciclico composto solo dagli archi che sono stati usati per raggiungere i vertici visitati. Nel caso di grafi diretti viene detto **arborescenza** ed è un'albero con tutti gli archi orientati dalla radice verso le foglie.

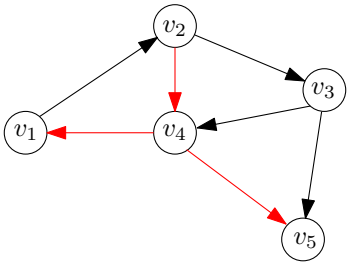
Preso un'arborescenza A creata tramite una DFS su un grafo G, ogni arco $(v_i, v_j) \in E$ non in A può essere classificato in 3 categporie:

1. **Arco all'indietro**: se va da un discendente ad un antenato, cioè $\text{Int}(v_i) \subset \text{Int}(v_j)$
2. **Arco in avanti**: se va da un antenato a un discendente, cioè $\text{Int}(v_i) \supset \text{Int}(v_j)$
3. **Arco di attraversamento**: se i due nodi non hanno correlazioni, cioè $\text{Int}(v_i) \cap \text{Int}(v_j) = \emptyset$

Nei grafi non diretti non essendoci differenza tra gli archi (v_i, v_j) e (v_j, v_i), l'unico caso possibile è che sia un arco all'indietro perché:

$$t(v_i) < t(v_j) \implies \text{Int}(v_i) \subset \text{Int}(v_j)$$

Esempio:



Gli archi non presenti nell'arborescenza A sono (v_2, v_4), (v_4, v_1) e (v_4, v_5).
Questi archi sono classificati:

- (v_2, v_4) è in avanti perchè $[2,5] \supset [4,5]$
- (v_4, v_1) è indietro perchè $[5,5] \supset [1,5]$
- (v_4, v_5) è di attraversamento perchè $[5,5] \cap [4,4] = \emptyset$

1.6.2 Algoritmo per controllare i tipi di archi

Per controllare i tipi di archi usiamo un'algoritmo modificato della DFS che da in output 3 insiemi *Back*, *Forward* e *Cross* che contengono rispettivamente gli archi appartenenti alle tre categorie. Aggiungo un contatore C e anche due array t e T in cui segno gli intervalli dei vari nodi.

Algoritmo: DFS per classificare gli archi

Input:

- G: grafo
- x: nodo di partenza

```
def DFS_archi(G, x):
    C=0
    Vis[x]=1
    t[x]=1
    Stack S=[x]
    while len(S)!=0 :
        y=S.top()
        while len(y.neighbors_out)!=0 :
            z=y.neighbors_out[0]
            y.neighbors_out.remove(0)
            if Vis[z]==0 :
                C+=1
                t[z]=C
                Vis[z]=1
                S.push(z)
                break
            if t[z]<t[y] and T[z]==0 :
                Back.add((y,z))
            elif t[z]<t[y] and T[z]!=0 :
                Cross.add((y,z))
            else
                Forward.add((y,z))
        if y==S.top() :
            S.pop()
            T[y]=C
    return Back, Cross, Forward
```

1.7 Alberi di visita e cicli

1.7.1 Grafi non diretti

Dato un grafo non diretto G connesso con un albero di visita T generato da una DFS, allora:

$$\exists \text{ arco all'indietro} \iff G \text{ ciclico}$$

1.7.2 Grafi diretti

Dato un grafo diretto G con un'arborescenza T generata da una DFS, definiamo che:

- un nodo u è discendente di un altro nodo v se esiste un cammino $v \to u$, cioè $\text{Int}(u) \subseteq \text{Int}(v)$
- un nodo v è antenato di un altro nodo u se un arco (u, v) è un arco all'indietro. Gli antenati di u sono tutti i nodi nel cammino radice $\to u$

Anche in questo caso:

$$\exists \text{ arco all'indietro} \iff G \text{ ciclico}$$

Esempio:
Un pozzo universale è un nodo x per cui:

- $\nexists (x, y) \in E \; \forall y \in V(G)$
- $\exists (y, x) \in E \; \forall y \in V(G)$

Scrivere un algoritmo con costo $O(n)$ per stabilire se esiste un pozzo universale avendo in input il grafo come matrice di adiacenza. La matrice se ci fosse un pozzo x sarebbe:

	v_1	...	x	...	v_n
v_1	0		1		
...		0	1		
x	0	0	0	0	0
...			1	0	
v_n			1		0

Il codice dell'algoritmo:

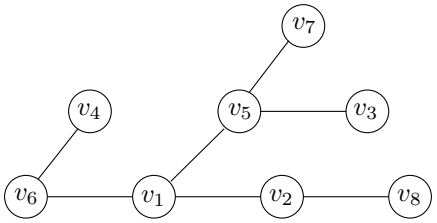
Algoritmo: Ricerca di un pozzo

Input:
- M: matrice di adiacenza del grafo

```
def SearchPozzo(M):
    pozzo=1
    for i in range(2,n) :
        if M[pozzo][i]==1 :
            pozzo=i
    for i in range(n) :
        if M[pozzo][i]==1 :
            return False
        if M[i][pozzo]==0 and i!=pozzo :
            return False
    return pozzo
```

1.7.3 Vettore dei padri

Un modo di salvare un albero di visita è il vettore dei padri, cioè un vettore P in cui $P[v] =$ nodo tramite cui si è arrivati a v. Per la radice $P[v] = v$.
Esempio:



n questo caso partendo da v_6 il vettore dei padri sarebbe:

$$P = [6, 1, 5, 6, 1, 6, 5, 2]$$

Per trovare gli antenati di un nodo v si può usare un algoritmo con costo $O(n)$:

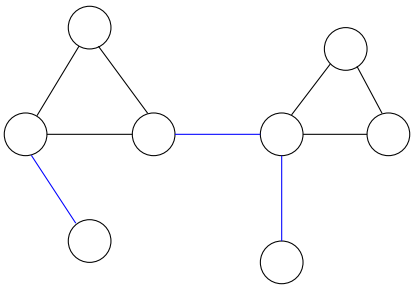
Algoritmo: Trovare gli antenati di un nodo v

```
def Ant(P, v):
    A.add(v)
    while P[v]!=v :
        A.add(P[v])
        v=P[v]
    return A
```

1.8 Ponti

Definizione di ponte

Dato un grafo non diretto G, si dice **ponte** un arco che se tolto fa diventare il grafo non connesso:



Per controllare se un determinato arco (u, v) è un ponte lo elimino e controllo se esiste un altro cammino $u \to v$:

- esiste $\implies$ (u, v) non ponte
- non esiste $\implies$ (u, v) ponte

Se volessimo trovare tutti i ponti in un grafo controllando ogni arco il costo computazionale sarebbe $O(m(n + m))$.

Dato T l'albero di visita di una DFS su un grafo G:

$$(u, v) \text{ ponte} \iff \nexists \text{ arco all'indietro da } T_v \text{ a fuori } T_v$$

Dove T_v è l'insieme dei discendenti di v.

1.8.1 Algoritmo per trovare i ponti

Dato un grafo G per trovare tutti i ponti si usa un'algoritmo che tiene segnato con $Back[v]$ il punto più indietro che si può raggiungere da un determinato nodo v:

Algoritmo: DFS per trovare i ponti

```
def Ponti(G):
    C=0
    v=V[0]
    DFS_ponte(G, v, v, t, C, P, Ponti)
    return Ponti


def DFS_ponte(G, u, v, t, C, P, Ponti):
    C+=1
    t[v]=C
    Back[v]=t[v]
    for u in v.neighbors_out :
        if t[u]==0 :
            P[u]=v
            DFS_ponte(G, v, u, , C, P, Ponti)
            if Back[u]<Back[v] :
                Back[v]=Back[u]
        elif u!=P[v] and t[u]<Back[v] :
            Back[v]=t[u]
    if Back[v]==t[v] :
        P.add((u,v))
```

1.9 Componenti fortemente connessi