

# Progettazione di algoritmi

Leonardo Ganzaroli

## Indice

<b>Introduzione</b>	<b>1</b>
<b>1 Teoria dei grafi</b>	<b>3</b>
1.1 Visite . . . . .	5
1.2 Studio dei grafi . . . . .	6
1.2.1 Classificazione archi . . . . .	6
1.2.2 Ordinamento . . . . .	7
1.2.3 Componenti . . . . .	8
<b>2 Algoritmi Greedy</b>	<b>9</b>
2.1 Grafi pesati . . . . .	10
2.1.1 MST . . . . .	11
<b>3 Programmazione dinamica</b>	<b>12</b>

## Introduzione

Questi appunti sono derivanti principalmente dalle slide del corso di *Progettazione di algoritmi* che ho seguito durante la laurea Triennale di informatica all'università "La Sapienza".

**N.B.** Questo corso è il naturale proseguimento di *Introduzione agli algoritmi*, quindi molte cose saranno date per scontate.

# 1 Teoria dei grafi

**N.B** Se non meglio specificato  $|V(G)| = n, |E(G)| = m$ .

**Definizione** Un cappio è un arco di un vertice in se stesso.

**Definizione** Un multigrafo è un grafo che ammette archi ripetuti e cappi.

**Definizione** 2 vertici  $x, y$  collegati da un arco sono detti adiacenti, l'arco è detto incidente in  $x, y$ .

**Definizione** Un grafo è detto diretto se i suoi archi hanno un orientamento.

**Definizione** Il grado di un vertice ( $deg()$ ) è pari al numero di archi incidenti in esso, se il grafo è diretto si distinguono grado entrante e uscente.

Per rappresentare un grafo ci sono 2 modi:

## 1. Matrice di adiacenza

$M \in Matr_{n \times n}(\{0, 1\})$  t.c.:

$$m_{i,j} = \begin{cases} 1 & \text{se } (v_i, v_j) \in E(G) \\ 0 & \text{altrimenti} \end{cases}$$

## 2. Liste di adiacenza

$\forall x \in V(G)$ :

$$L_x = [v \in V(G) \mid (x, v), (v, x) \in E(G)]$$

Se il grafo è diretto ce ne sono 2 per vertice:  $L_x^{in}, L_x^{out}$ .

( $x$ vertice)	Matrice	Liste
<b>Spaziale</b>	$O(n^2)$	$O(n + m)$
<b>Verificare esistenza arco</b>	$O(1)$	$O(deg(x))$
<b>Trovare adiacenti</b>	$O(n)$	$O(deg(x))$
<b>Aggiungere/Rimuovere arco</b>	$O(1)$	$O(deg(x))$

Tabella 1: Costi

**Definizione** Una traccia è una passeggiata senza archi ripetuti.

**Definizione** Una cammino è una traccia senza vertici ripetuti.

**Definizione** Una passeggiata è detta chiusa se il primo vertice è anche l'ultimo, altrimenti si dice aperta.

**Definizione** Dati  $x, y$  vertici.  $y$  è visitabile da  $x$  ( $x \rightarrow y$ ) se esiste una passeggiata da  $x$  a  $y$ .

**Definizione** Una passeggiata è detta Euleriana se contiene tutti gli archi ed ogni arco è presente una sola volta.

**Teorema 1 (Eulero)**

$$\exists \text{ passeggiata Euleriana chiusa} \iff \begin{cases} \forall v_1, v_2 \in V(G) \quad \exists v_1 \rightarrow v_2 \\ \forall v \in V(G) \quad \exists k \in \mathbf{Z} \mid \deg(v) = 2k \end{cases}$$

**Definizione** Un grafo è fortemente connesso se  $\forall x, y \in V(G) \quad \exists 2 \text{ cammini} \mid x \rightarrow y \wedge y \rightarrow x$ .

**Definizione** Un'arborescenza è un albero diretto.

I grafi possono essere usati in molti ambiti, un esempio di applicazione è il seguente:

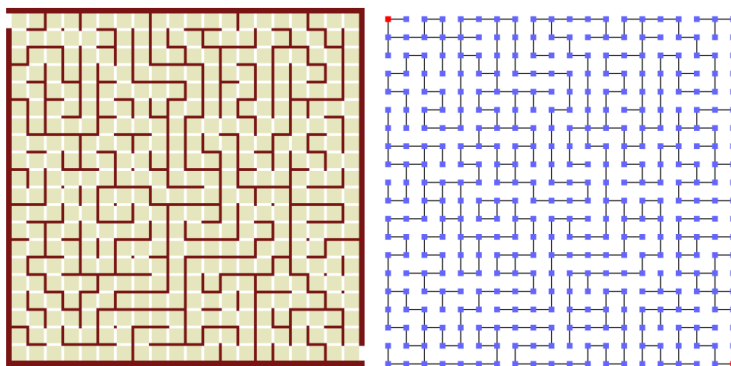


Figura 1: Labirinto  $\rightarrow$  Grafo

In questo modo basta trovare un cammino tra l'entrata e l'uscita per attraversarlo.

## 1.1 Visite

---

**Algorithm 1** DFS (Ricorsiva)

---

G grafo, x nodo iniziale

Vis = {x}  
DFS(G,x,Vis)  
Return Vis

```
function DFS(G,x,Vis)
  for  $y \in x.out$  do
    if  $y \notin Vis$  then
      Vis.add(y)
      DFS(G,y,Vis)
    end if
  end for
end function
```

---

Il costo è  $O(n + m)$ , effettua una visita in profondità (simile a quella di un albero).

---

**Algorithm 2** BFS

---

G grafo, x nodo iniziale

Vis = {x}  
Q = coda vuota  
Q.enqueue(x)

```
while  $Q \neq \emptyset$  do
  y = Q.dequeue()
  for  $z \in y.out$  do
    if  $z \notin Vis$  then
      Q.enqueue(z)
    end if
  end for
end while
```

Return Vis

---

Il costo è  $O(n + m)$ , effettua una visita in ampiezza. Inserendo un opportuno array si possono trovare anche le distanze dei nodi da quello iniziale (numero di archi).

## 1.2 Studio dei grafi

### 1.2.1 Classificazione archi

**Definizione** Dato un contatore  $C$  incrementato ad ogni vertice visitato. Se si esegue una DFS si definisce  $\forall x \in V(G)$ :

- **Tempo di visita di  $x$**  ( $t(x)$ )  
Valore di  $C$  quando  $x$  è aggiunto allo stack.
- **Tempo di chiusura di  $x$**  ( $T(x)$ )  
Valore di  $C$  quando  $x$  è rimosso dallo stack.
- **Intervallo di visita di  $x$**

$$Int(x) = [t(x), T(x)]$$

**Definizione** Data  $A$  arborescenza generata da una DFS su  $G$  grafo. Si possono usare gli intervalli per catalogare gli archi  $(u, v) \in E(G) - E(A)$ :

- **All'indietro**

$$Int(u) \subseteq Int(v)$$

- **In avanti**

$$Int(u) \supseteq Int(v)$$

- **Di attraversamento**

$$Int(u) \cap Int(v) = \emptyset$$

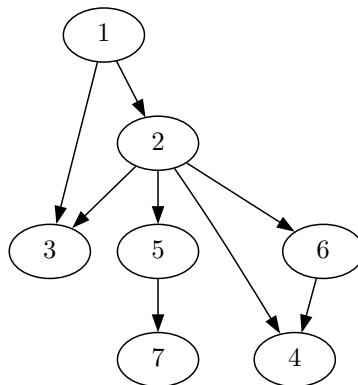
### Teorema 2 (Cicli)

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

**Definizione** Un ponte è un arco che non appartiene a nessun ciclo.

### 1.2.2 Ordinamento

**Definizione** Un ordinamento topologico è un ordinamento dei vertici tale che ogni vertice viene prima dei vertici raggiungibili da esso.



2 possibili ordinamenti di questo grafo sono:

1. 1,2,3,6,4,5,7
2. 1,2,6,4,3,5,7

**Teorema 3 (Cicli)**

$$\exists \text{ ordinamento topologico} \iff \nexists \text{ ciclo}$$

---

**Algorithm 3** Trova ordinamento DAG

---

G grafo

L = lista vuota

```
while  $V(G) \neq \emptyset$  do  
   $v = v \in V(G) \mid \text{deg}_{out}(v) = 0$   
  L.insert_head(v)  
  G.remove(v)  
end while
```

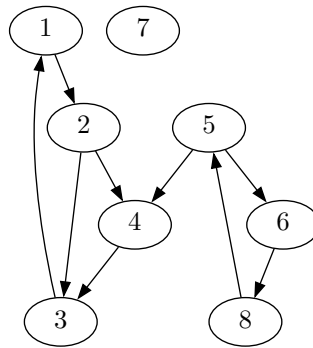
Return L

---

Se  $G$  è rappresentato con le liste ha costo  $O(n(n+m))$ .

### 1.2.3 Componenti

**Definizione** Dato  $G$  grafo. Un componente di  $G$  è un suo sottografo fortemente connesso e massimale.



I componenti di questo grafo sono:

- $H_1 = \{1, 2, 3, 4\}$
- $H_5 = \{5, 6, 8\}$
- $\{7\}$

**Definizione** La contrazione di un componente in un vertice ( $G/V(H)$ ) è l'operazione con cui:

- Si rimuovono dal grafo i vertici del componente
- Si inserisce un vertice apposito
- Si sistemano gli archi

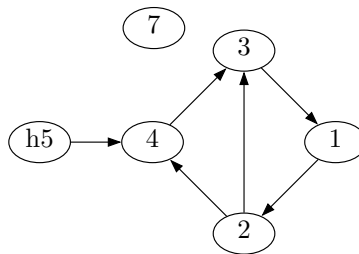


Figura 2:  $G/V(H_5)$



## 2 Algoritmi Greedy

**Definizione** Un algoritmo è Greedy se cerca una soluzione ammissibile da un punto di vista globale attraverso la scelta della soluzione più conveniente ad ogni passo locale.

Per questo tipo di algoritmi è sempre necessario dimostrare la correttezza, per farlo si deve:

1. Dimostrare che l'output abbia le caratteristiche previste
2. Dimostrare (con induzione) che ogni istanza di output sia nella soluzione ottimale
3. Dimostrare che l'output finale sia la soluzione ottimale

---

Si supponga di dover effettuare un viaggio dalla località A alla località B con un'auto che ha un'autonomia di  $k$  chilometri e con serbatoio vuoto all'inizio. Lungo la strada ci sono  $n + 1$  distributori di benzina ciascuno distante dal precedente meno di  $k$  chilometri. Sia  $d_i$  la distanza che separa il distributore  $i$  dal distributore  $i + 1$ , dove il distributore 1 è in A e il distributore  $n + 1$  è in B.

Descrivere un algoritmo greedy che preso in input la lista delle distanze  $d_1, \dots, d_n$  dei distributori, seleziona un numero minimo di distributori in cui far rifornimento durante il viaggio.

---

### Algorithm 4

---

d array distanze, k autonomia

```
R = array di 0
aut = 0
for i=1 to n do
  if aut < d[i] then
    aut = k
    R[i] = 1
  end if
  aut -= d[i]
end for
Return R
```

---

- Grazie all'IF ogni soluzione prodotta è ammissibile
- Induzione

Base:  $i = 0$ , niente da provare

Sia  $R_i$  l'array  $R$  dopo l' $i$ -esima iterazione,  $R^*$  la soluzione ottima.

Per ipotesi fino ad  $i$  coincidono, analizzo i casi non coincidenti in  $i + 1$ :

1.  $R_{i+1}[i+1] = 1 \wedge R^*[i+1] = 0$

Non può essere  $i+1 = n+1$  perché ci sono massimo  $n$  iterazioni e  $R$  non avrebbe 1 in quella posizione.

Se  $i+1 < n+1$  il fatto che venga scelto il distributore  $i+1$  implica che l'autonomia non basti per arrivare a  $i+2$ , la soluzione sarebbe quindi sbagliata.

2.  $R_{i+1}[i+1] = 0 \wedge R^*[i+1] = 1$

Non può essere  $i+1 = n+1$  altrimenti la soluzione avrebbe un rifornimento di troppo.

Se  $i+1 < n+1$  il fatto che non venga scelto implica che c'è abbastanza autonomia per arrivare a  $i+2$ , quindi la soluzione avrebbe un rifornimento di troppo.

## 2.1 Grafi pesati

**Definizione** Un grafo pesato è un grafo in cui ogni arco ha associato un valore reale.

**Definizione** Il peso di un cammino  $(p())$  è la somma dei pesi degli archi del cammino.

**Definizione** La distanza pesata tra 2 vertici  $x, y$  ( $dist(x, y)$ ) è il cammino tra i 2 con peso minimo.

### Algorithm 5 Dijkstra (Grafo non diretto)

```

dist = array di  $+\infty$ 
Padri = array di -1
dist[u] = 0                                ▷ u nodo iniziale
Padri[u] = u
H = min-Heap                               ▷ inizializzato con tutti i nodi, priorità=dist
while  $H \neq \emptyset$  do
    v = H.get_min()                         ▷ Rimuovi il nodo minore
    for  $w \in v.out$  do
        if  $dist[w] > dist[v] + p[v, w]$  then
             $dist[w] = dist[v] + p(v, w)$ 
            Padri[w] = v
            Aggiorna H
        end if
    end for
end while
Return dist, Padri

```

Il costo è  $O((n + m) \log n)$ .

### 2.1.1 MST

**Definizione** Un sottografo di un grafo connesso e non diretto è un albero di copertura se è aciclico e contiene tutti i nodi del grafo.

**Definizione** L'albero di copertura minima di un grafo (MST) è il suo albero di copertura la cui somma degli archi è la minima.

---

**Algorithm 6** Kruskal

---

```
Sol =  $\emptyset$ 
Sort( $E(G)$ )
for  $e \in E(G)$  do
    if Trova_ciclo( $Sol \cup e$ )= $\emptyset$  then
        Sol.add( $e$ )
    end if
end for
Return Sol
```

---

---

**Algorithm 7** Prim

---

```
 $v = v \in V(G)$ 
Sol =  $\emptyset$ 
R = { $v$ }
while  $R \neq V(G)$  do
     $(x, y) = \min[w(a, b)]$  con  $a \in R \wedge b \in V(G) - R$ 
    Sol.add( $(x, y)$ )
    R.add( $y$ )
end while
Return Sol
```

---

Entrambi ritornano un MST del grafo, il costo è  $O(mn)$  per entrambi.

### 3 Programmazione dinamica

**Definizione** La programmazione dinamica è basata sulla risoluzione di un problema partendo dalle soluzioni dello stesso problema ma di dimensione inferiore.

**Definizione** La memoization consiste nel salvare in memoria i valori dati da una funzione per poterli usare successivamente senza ricalcolarli, si può usare solo se la funzione non ha effetti collaterali e dà sempre lo stesso output con un certo input.

Per fare ciò si usa una matrice.

---

Esempi:

- **Knapsack Problem**

Dati degli oggetti  $x_1, \dots, x_n$  ognuno con un suo peso  $w_i$  ed un suo valore  $v_i$  trovare un sottoinsieme che massimizzi il valore totale tenendo il peso sotto la soglia  $W$ .

Si definisce la tabella  $T$  di dimensione  $n + 1 \times W + 1$  tale che:

$T[k, x] = (\text{max valore trasportabile con uno zaino di capacità } x \leq W \text{ con i primi } k \text{ oggetti})$

Nello specifico:

$$T[k, x] = \begin{cases} 0 & \text{se } x = 0 \vee k = 0 \\ T[k - 1, x] & \text{se } w_k > x \\ \max(T[k - 1, x], T[k - 1, x - w_k] + v_k) & \text{se } w_k \leq x \end{cases}$$

Per trovare la soluzione basta partire dalla cella in basso a destra e risalire una riga alla volta, nel caso  $T[k, x] > T[k - 1, x]$  si aggiunge l'elemento corrente alla soluzione e ci si sposta verso sinistra di  $w_k$  colonne.

- **Cammino peso max**

Dato un DAG pesato e due vertici  $x, y$  trovare il cammino con peso massimo tra i 2.

Essendo un DAG con  $n$  vertici ci sono massimo  $n - 1$  archi, si definisce quindi la tabella  $n \times n$  tale che:

$$T[k, z] = (\text{peso max del cammino } x \rightarrow z \text{ passante per max } k \text{ archi})$$

Nello specifico:

- $T[0, x] = 0$
- $\forall z \neq x \quad T[0, z] = -\infty$
- $\forall k \in [0, n - 1] \quad T[k, z] = -\infty$  se  $\nexists$  cammino  $x \rightarrow z$  lungo  $k$
- $T[1, z] = w(x, z)$  se  $\exists (x, z) \in E(G)$ ,  $T[0, z]$  altrimenti

Quindi:

$$T[k, z] = \max(T[k - 1, z], T[k - 1, v_1] + w(v_1, z), \dots, T[k - 1, v_h] + w(v_h, z))$$

Il cammino si trova con procedimento simile al precedente.

- **CPM**

Un progetto si può dividere in attività  $1, 2, \dots, n$ , ogni attività ha un suo tempo di svolgimento e possono esistere dipendenze tra 2 attività. Si vogliono sapere i tempi di inizio delle attività ed il tempo totale necessario.

Si segue il procedimento:

1. Costruire un DAG i cui nodi sono le attività e gli archi le dipendenze, il peso di quest'ultime è il tempo di esecuzione del nodo di partenza
2. Aggiungere un nodo *Start* con archi uscenti ad ogni altro nodo con costo 0
3. Il cammino con peso maggiore da *Start* ad un certo nodo fornisce il tempo di inizio per lo stesso
4. Il tempo totale è dato dal costo del cammino massimo all'ultimo nodo + il suo tempo di completamento

- **Bellman-Ford**

Trovare il cammino minimo tra 2 nodi in presenza di costi negativi (no cicli negativi).

Come visto in precedenza:

$$T[k, z] = (\text{peso min cammino } x \rightarrow z \text{ passante per max } k \text{ archi})$$

Nello specifico:

- $T[0, x] = 0$
- $\forall z \neq x \quad T[0, z] = +\infty$
- $\forall k \in [0, n-1] \quad T[k, z] = +\infty$  se  $\nexists$  cammino  $x \rightarrow z$  lungo  $k$
- $T[1, z] = w(x, z)$  se  $\exists (x, z) \in E(G)$ ,  $T[0, z]$  altrimenti

Quindi:

$$T[k, z] = \min(T[k-1, z], T[k-1, v_1] + w(v_1, z), \dots, T[k-1, v_h] + w(v_h, z))$$

---