

# Grafi

## Palestra di algoritmi

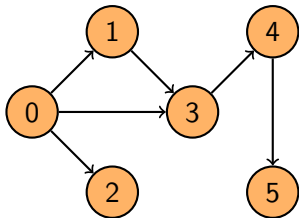
2020-2021

# Cos'è un grafo

## Definizione

Un *grafo*  $G$  è una coppia  $(V, E)$  dove:

- $V$  è un insieme di *nodi*
- $E$  è un insieme di *archi* che collegano coppie di nodi.



## Nota

I nodi possono rappresentare 'qualsiasi cosa'. Per semplicità (anche nell'implementazione) li identifichiamo con dei numeri.

# Cosa rappresentano

## Perchè si studiano

Moltissimi problemi possono essere visti come problemi su grafi. Anche se i problemi hanno forma astratta, le loro applicazioni si trovano poi negli ambiti più disparati.

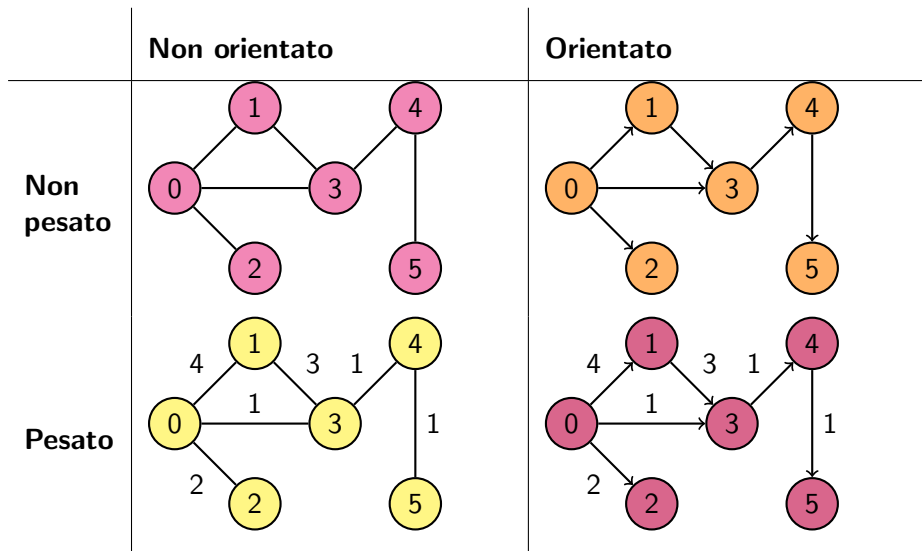
## Esempi

Tantissimi utilizzi 'pratici':

- **Mappe:** luoghi collegati da strade
- **Social networks:** utenti collegati da follow/amicizie
- **Grafi delle dipendenze:** un'attività deve essere svolta prima di un'altra

In generale, qualsiasi modello dove sono presenti *entità* collegate da qualche tipo di *relazione*. Trovano applicazioni in informatica, intelligenza artificiale, fisica, biologia, linguistica, ...

# Tipi di grafo

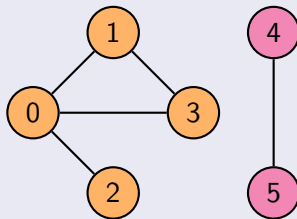


## Grafi (non) orientati

Un grafo non orientato è un caso particolare di grafo orientato: un arco bidirezionale può essere visto come una coppia di archi monodirezionali. Tuttavia, in alcuni problemi sono più facili da trattare.

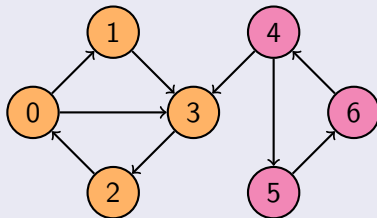
## Grafi non connessi e componenti connesse

Un grafo può non essere connesso. In un *grafo non orientato*, un sottografo in cui ogni nodo è raggiungibile da ogni altro è detto *componente connessa*.



## Componenti fortemente connesse

In un *grafo orientato*, un sottografo in cui ogni nodo è raggiungibile da ogni altro è detto *componente fortemente connessa*.



## Problemi in grafi non pesati

- Ricerca del cammino più breve (misurato in numero di archi)
- Componenti (fortemente) connesse
- Ordinamento topologico
- ...

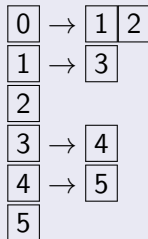
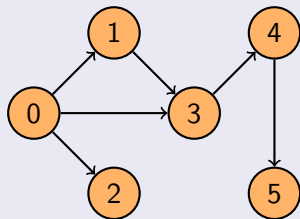
## Problemi in grafi pesati

- Cammini di peso minimo
- ...

# Memorizzare un grafo in C++

## Liste di adiacenza

Nella maggior parte dei casi il modo più 'comodo' di memorizzare un grafo è tramite liste di adiacenza: per ogni nodo del grafo memorizziamo la lista di nodi adiacenti.



```
vector<vector<int>> g(6);  
g[1].push_back(2);  
g[1].push_back(3);
```

```
g[2].push_back(4);  
g[4].push_back(5);  
g[5].push_back(6);
```



# Memorizzare un grafo in C++

```
vector<vector<int>> g(6);  
// stampo i vicini del nodo 0  
for (int v : g[0]) cout << v << endl;
```

## Grafi non orientati

Per i grafi orientati devo aggiungere gli archi in entrambe le direzioni.

## Grafi pesati

Nell'arco devo memorizzare anche il peso:

```
struct Edge{int v, weight};  
vector<vector<Edge>> g;  
g[x].push_back({v, w}); // aggiungo gli archi...  
for (Edge &e : g[x]){ // visito i vicini del nodo x  
    cout << "Edge from " << x << " to "  
        << e.v << ", weight: " << e.weight;  
}
```

## Tipi di visita

La maggior parte degli algoritmi sui grafi richiedono di effettuare una o più visite del grafo.

Esistono due 'strategie' principali per visitare i grafi:

**DFS (Depth-First-Search)** : visita del grafo 'in profondità'

**BFS (Breadth-First-Search)** : visita del grafo 'in ampiezza'

## Idea

Visita ricorsiva: per ogni nodo adiacente, si visita ricorsivamente tale nodo, visitando ricorsivamente i suoi nodi adiacenti, etc. [Video](#)

## Pseudocodice

**DFS**(**Graph**  $g$ , **int**  $node$ )

- 1 Segno  $node$  come visitato;
- 2 Visito il nodo  $node$
- 3 **For** nodo  $v$  adiacente a  $node$ :
- 4     Visito l'arco  $(node, v)$
- 5     **If**  $v$  non è ancora stato visitato:
- 6         **DFS**( $g, v$ )

## Idea

Visitare i nodi a distanze crescenti dalla sorgente. Visitare i nodi a distanza  $k$  prima di visitare i nodi a distanza  $k + 1$ . [Video](#)

## Pseudocodice

**BFS**(**Graph**  $g$ , **int**  $node$ )

- ① Queue  $q$ , inserisco  $node$  in  $q$  e segno che  $node$  è visitato
- ② **While** la coda  $q$  non è vuota:
  - ③  $u$  = estraggo l'elemento in testa a  $q$ , visito il nodo  $u$
  - ④ **For** nodo  $v$  adiacente a  $node$ :
    - ⑤ Visito l'arco  $(u, v)$
    - ⑥ **If**  $v$  non è ancora stato visitato:
      - ⑦ Segno che  $v$  è visitato
      - ⑧  $q.push(v)$

## Idea

Visitare i nodi a distanze crescenti dalla sorgente. Visitare i nodi a distanza  $k$  prima di visitare i nodi a distanza  $k + 1$ . [Video](#)

## Cammini minimi

L'ordine in cui visitiamo i nodi dipende dal nodo da cui partiamo. Se partiamo dal nodo  $v$ , raggiungiamo ogni altro nodo con *distanza minima* (misurata in numero di archi attraversati). Per questo motivo, la visita BFS ci permette di trovare la distanza minima da un nodo verso ogni altro nodo del grafo!

## Video

DFS e BFS sono fondamentali per tutti gli algoritmi su grafi. Questi video permettono di visualizzarle su grafi più grandi.

- [DFS](#)
- [BFS](#)

## Esercizi

Vediamo due esercizi insieme:

- [Ponti](#)
- [Spesa](#)