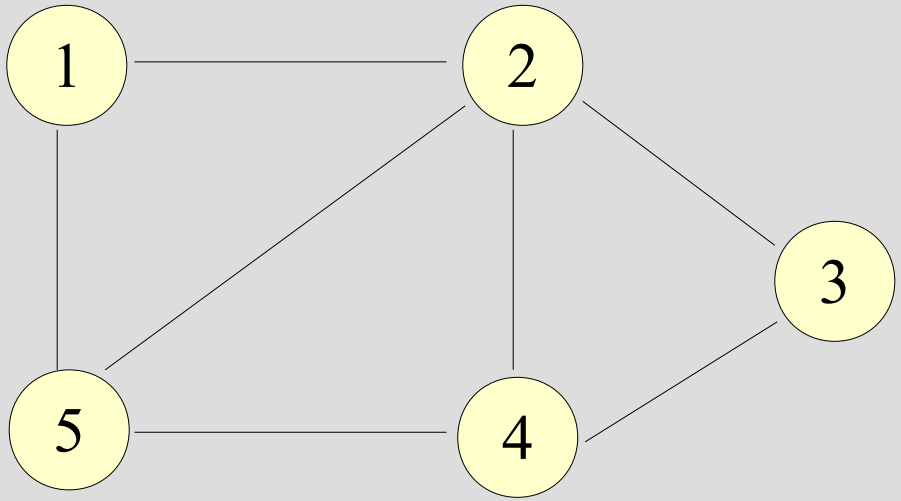
Esercitazione 7

Grafi

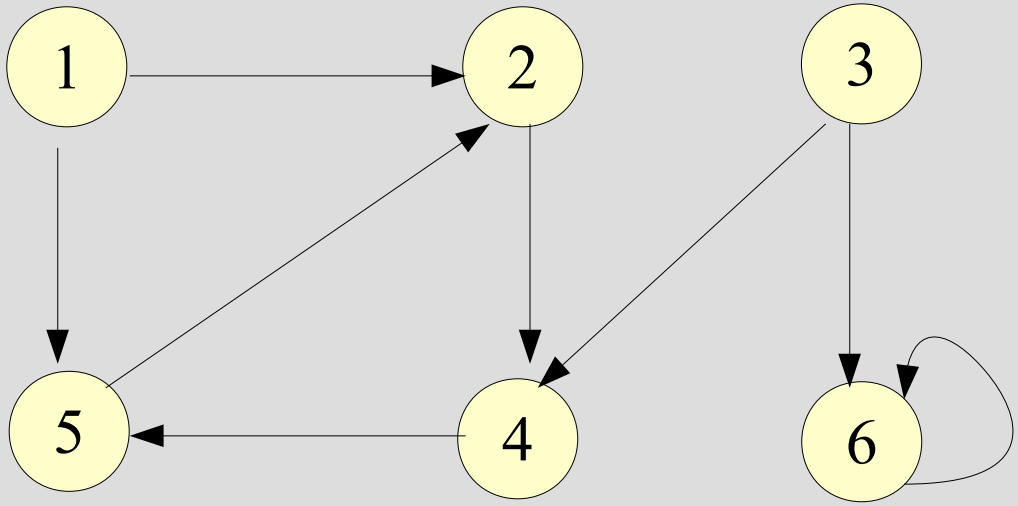
Rappresentazione e algoritmi di visita

Grafo

G = (V,E)
non orientato



G = (V,E)
orientato



Rappresentazione

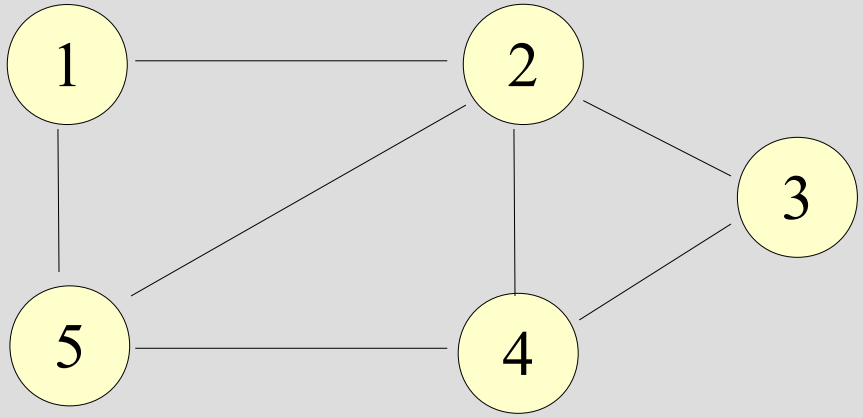
- **Grafo G = (V,E)**
- 2 metodi standard per la rappresentazione
  - ***Liste di adiacenza***
  - ***Matrici di adiacenza***
- Entrambi validi sia per **grafi orientati** che **non orientati**

Liste di adiacenza

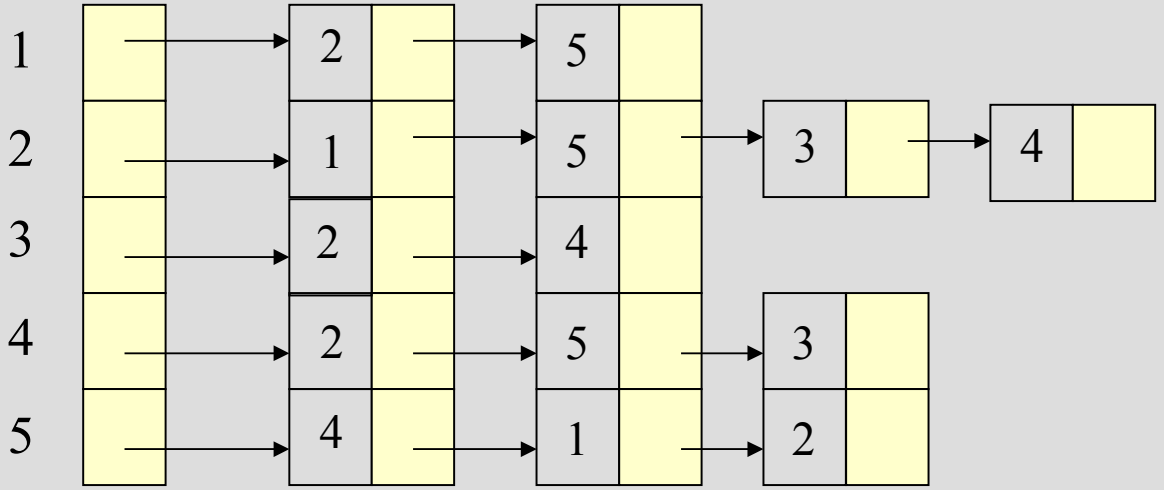
- **Grafo G = (V,E)**
- Array **Adj** di **|V|** liste, una per ogni vertice in **V**
- Per ogni vertice **u** in **V**, **Adj[u]** contiene tutti i vertici **v** in **V** tali che esista un arco **(u,v)** in **E** (tutti i ***vertici adiacenti*** a **u** in **G**, memorizzati in ***ordine arbitrario***)
- A livello implementativo, una soluzione è che **Adj[u]** contenga un ***puntatore*** a tali vertici

Grafo non orientato

G = (V,E)
non orientato

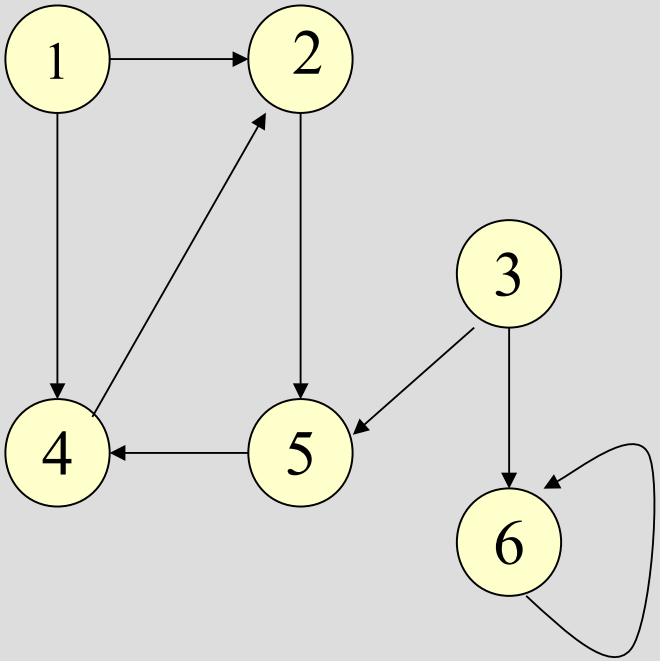


Lista di adiacenza

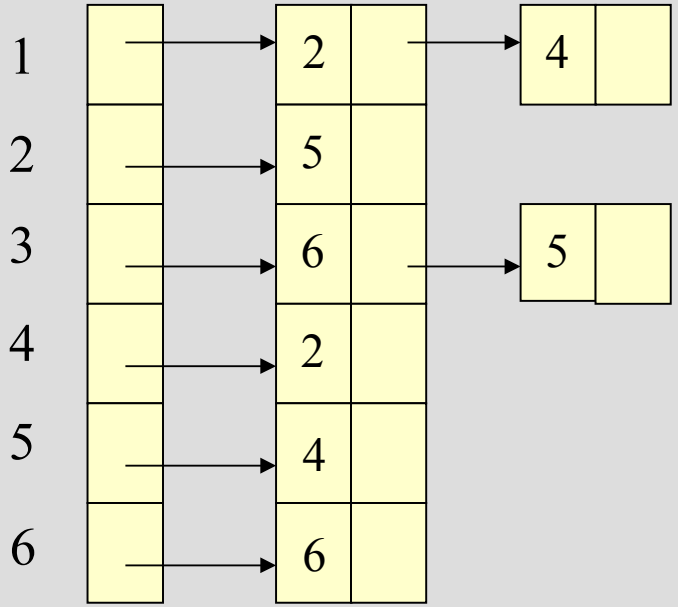


Grafo orientato

G = (V,E)
orientato

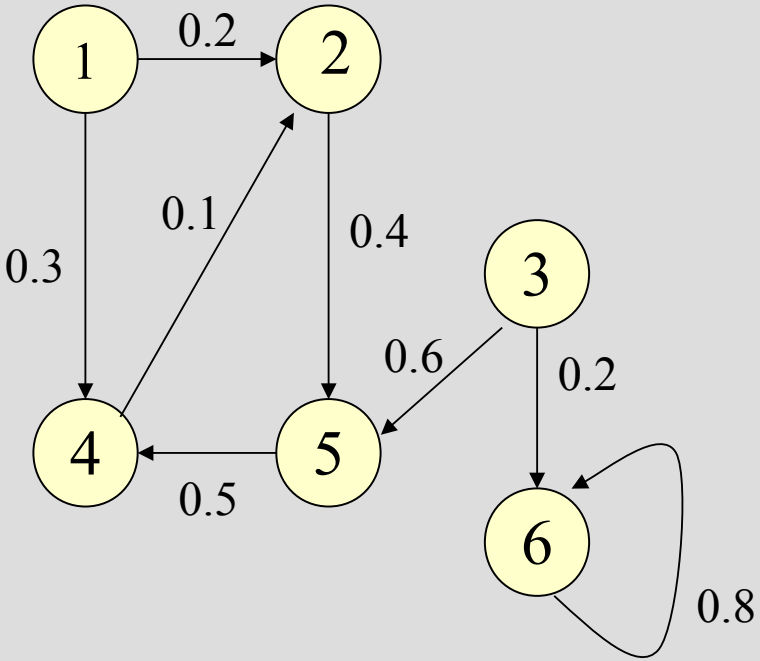


Lista di adiacenza:
La somma delle lunghezze di tutte le liste di adiacenza è pari ad **|E|**

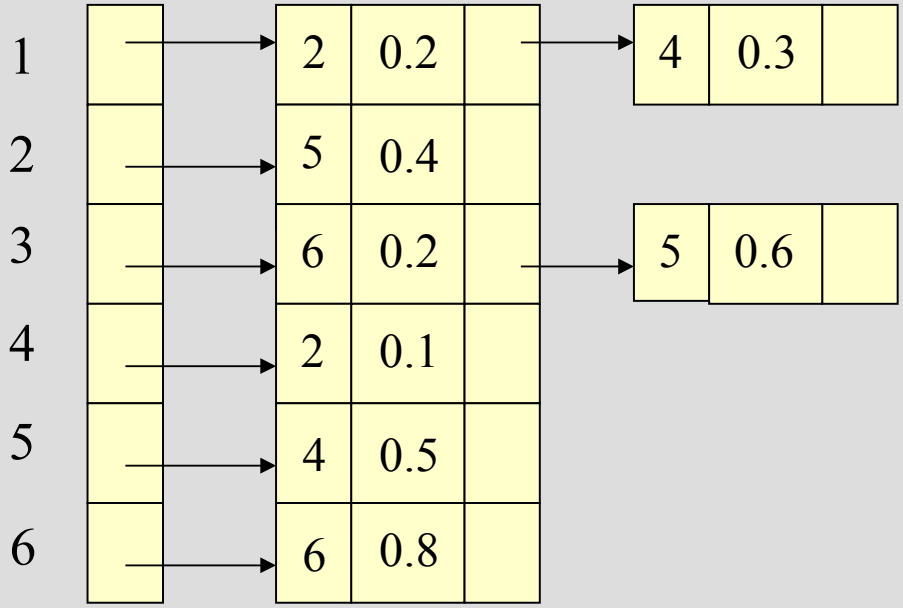


Grafo orientato e pesato

G = (V,E)
orientato e pesato



Lista di adiacenza:
il peso dell'arco **(u,v)** è memorizzato col vertice **v** nella lista di **u**



Matrici di adiacenza

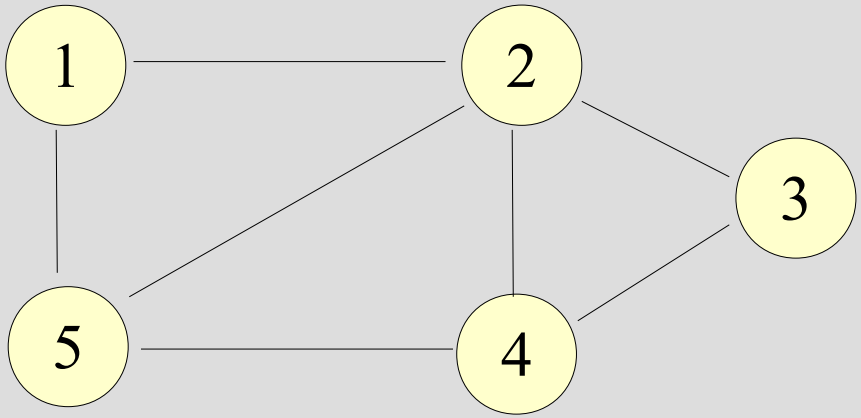
- **Grafo G = (V,E)**
- Matrice **A = ($a_{i\,j}$)** di dimensione **|V|x|V|**
- $a_{i\,j}$ =

$$\begin{cases} \mathbf{1} & \text{se } \mathbf{(i,j)} \text{ appartiene a } \mathbf{E} \\ \mathbf{0} & \text{altrimenti} \end{cases}$$

- Per ***archi pesati memorizzo il peso*** anziché il valore **1**

Grafo non orientato

G = (V,E)
non orientato

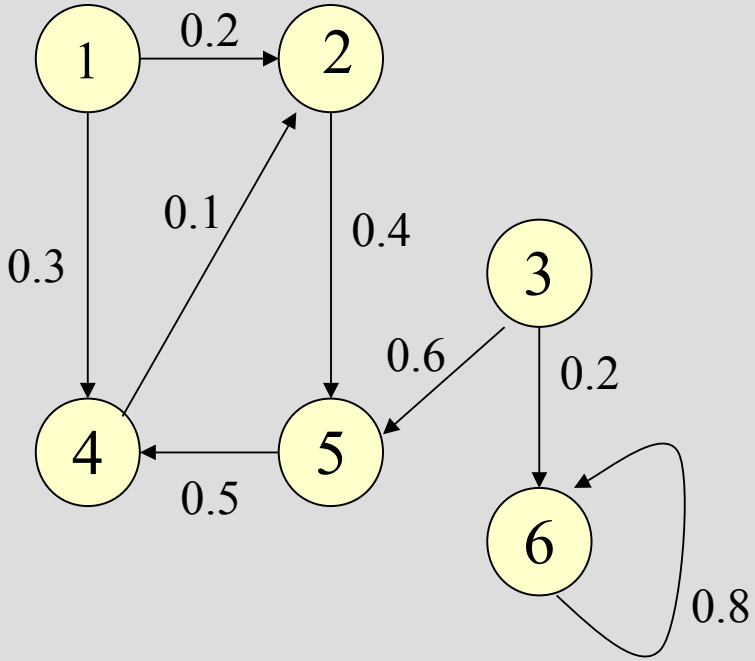


Matrice di adiacenza (simmetrica)

	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

Grafo orientato (e pesato)

G = (V,E)
orientato e pesato



Matrice di adiacenza (asimmetrica)

	1	2	3	4	5	6
1	0	.2	0	.3	0	0
2	0	0	0	0	.4	0
3	0	0	0	0	.6	.2
4	0	.1	0	0	0	0
5	0	0	0	.5	0	0
6	0	0	0	0	0	.8

Implementazione con liste

Struttura nodo

```
struct adj_node {
  int node;
  float weight;
  struct adj_node* next;
};
```

Puntatore al prossimo elemento della lista

Grafo G

```
struct adj_node **Adj;
```

Puntatore a puntatore a struttura nodo

Per n nodi

```
Adj = new adj_node*[n+1];
```

Array di puntatori a struttura nodo

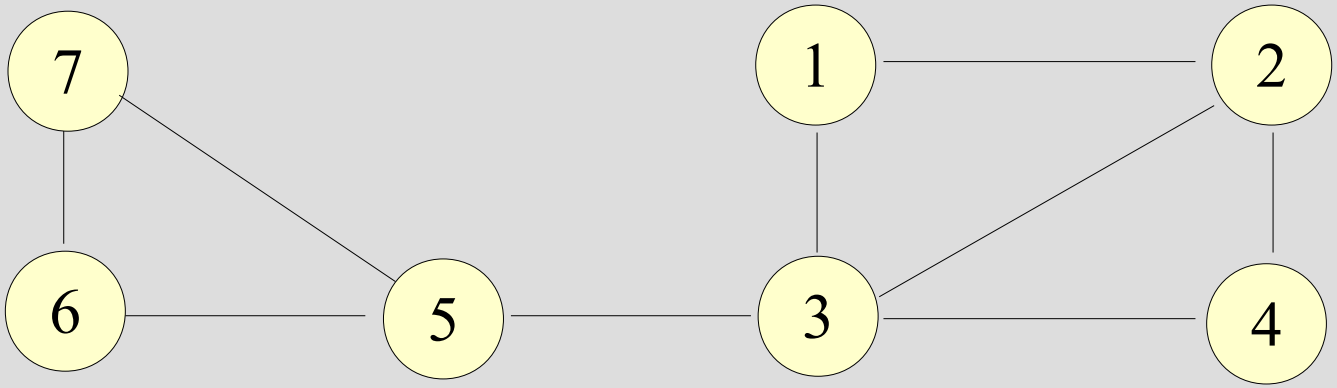
Lettura grafo da file

graph1 e *graph2*: **file di input** che contengono un **elenco di archi**

Es.

```
7
1 2
1 3
2 3
2 4
4 3
3 5
5 6
5 7
7 6
```

Numero di nodi componenti il grafo



Nota: archi interpretabili come orientati o non orientati

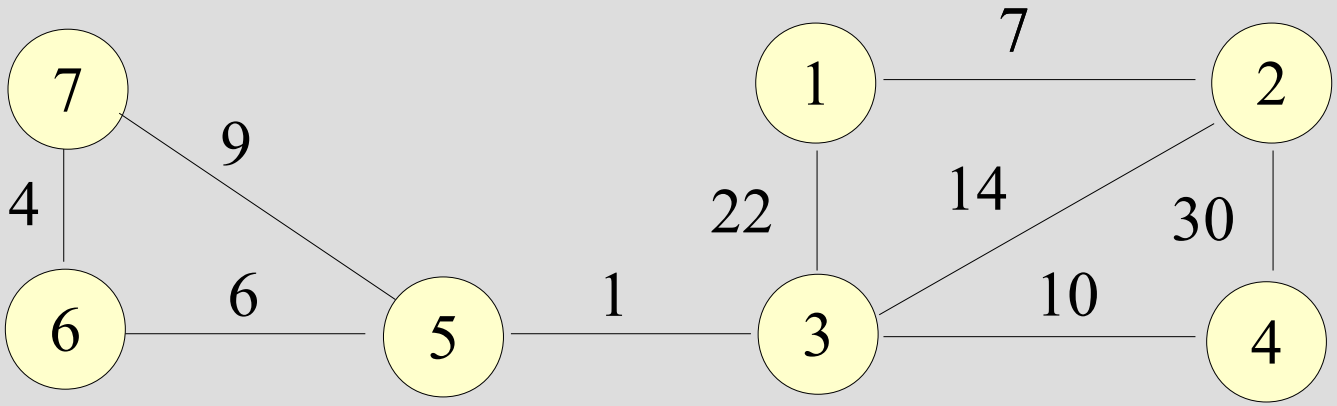
Lettura grafo da file

graph1w e *graph2w*: **file di input** che contengono un **elenco di archi pesati**

Es.

```
7
1 2 7
1 3 22
2 3 14
2 4 30
4 3 10
3 5 1
5 6 6
5 7 9
7 6 4
```

Numero di nodi componenti il grafo



Nota: archi interpretabili come orientati o non orientati

Inserimento di un arco

- Inserire un arco **(u,v)** nelle liste di adiacenza che rappresentano il grafo implica diverse operazioni a seconda del tipo di grafo considerato
- ***Grafi orientati***: inserisco **v** in **Adj[u]**
- ***Grafi non orientati***: inserisco **v** in **Adj[u]** e **u** in **Adj[v]**

Programma

- *graph.cc*
- Programma che implementa la **rappresentazione di un grafo** mediante **liste di adiacenza**
- La struttura del grafo viene presa da un **file di input** che contiene l'elenco degli archi (potenzialmente orientati)
  - Programma predisposto per archi pesati
- **Suggerimento per l'implementazione:** iniziare col considerare un **grafo non orientato e non pesato**

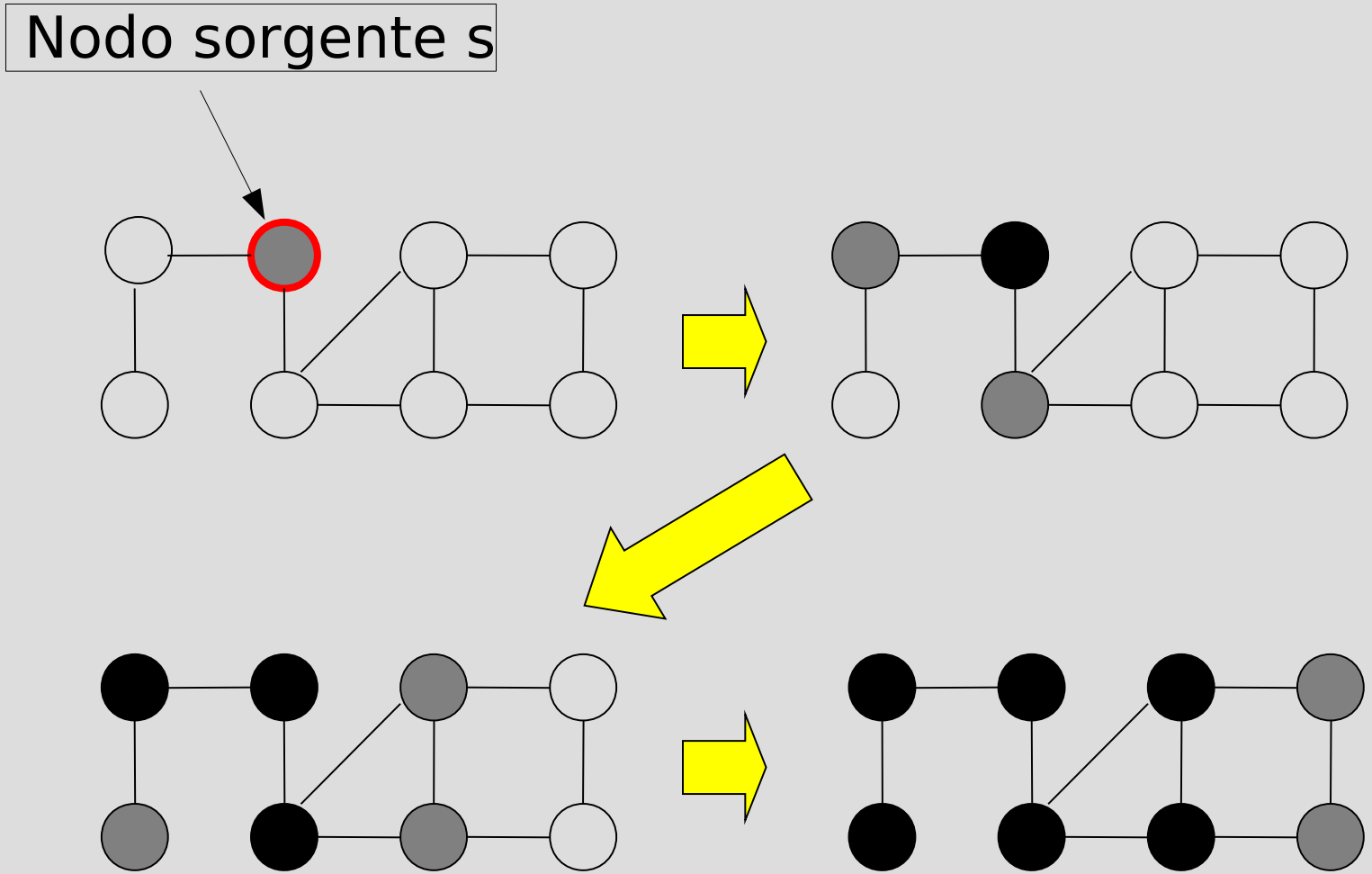
Visita in ampiezza

- Algoritmo di visita del grafo
- ***Breadth-First Search (BFS)***
- ***Grafo*** **G=(V,E)** e ***vertice sorgente*** **s**
- Ispezione sistematica di tutti i vertici raggiungibili da **s**
- L'algoritmo scopre tutti i vertici che hanno distanza **k** da **s** prima di scoprire quelli a distanza **k+1**

Idea intuitiva

- **Tenere traccia dello stato di ogni vertice** (già scoperto, appena scoperto, ancora da scoprire), “colorandolo” di un colore diverso
  - ***bianco:*** ancora non scoperto
  - ***grigio:*** appena scoperto ed appartenente alla **frontiera**
  - ***nero:*** terminata la visita (tutti i vertici adiacenti sono già stati scoperti --> non può avere nodi adiacenti bianchi)

Visualizzazione



Strutture ausiliarie

- Per ogni vertice **u** possono essere mantenuti altri attributi (non tutti necessari):
  - ***colore***: `color[u]`
  - ***padre*** (o ***predecessore***): `parent[u]`
  - la ***distanza*** dalla sorgente **s**: `d[u]`
- L'algoritmo fa uso di una ***coda Q*** con schema **FIFO** (**First In First Out**) per gestire l'insieme dei vertici grigi (frontiera)
  - Inizialmente nella coda ci sarà soltanto il nodo sorgente **s**

Coda Q

- Coda **Q** implementata come una **lista**
- ***Schema di gestione FIFO***
- Operazioni previste sulla coda **Q**: ***enqueue*** e ***dequeue***
  - ***enqueue:*** inserisce un elemento in fondo alla lista (inserisce in coda)
  - ***dequeue:*** toglie il primo elemento dalla lista (estrae dalla testa)

Albero BFT

- ***La visita in ampiezza costruisce un albero BFT (Breadth-First Tree)***
- L'**albero BFT** mantiene l'informazione su chi è il **padre** (o **predecessore)** di ogni nodo **i** nella sequenza della visita in ampiezza
  - Il nodo da cui si è arrivati a scoprire **i**
- Non pensiamo in realtà un albero nel senso di struttura dati (es. albero binario)
  - E' sufficiente utilizzare il vettore `parent[u]`

Costruzione albero BFT

- L'albero inizialmente contiene solo il ***nodo sorgente*** **s**, che ne è la ***radice***
- Quando un vertice bianco **v** viene scoperto durante la scansione della lista di adiacenza di un vertice **u** già scoperto (grigio), si aggiunge all'albero il vertice **v** e l'arco **(u,v)**: **u** è ***padre*** di **v**
  - Ogni vertice viene scoperto al massimo una volta -> ha al massimo un padre
  - Aggiungere all'albero significa aggiornare il relativo elemento nel vettore: **parent[v]=u**

Programma (1)

bfs_visit.cc

- Programma che implementa la **rappresentazione di un grafo** mediante **liste di adiacenza**
- La struttura del grafo viene presa da un **file di input** che contiene l'elenco degli archi (potenzialmente orientati e pesati)
- (Fin qui già fatto...)

Programma (2)

- Il programma effettua poi la **visita in ampiezza di un grafo non orientato,** costruendo l'**albero di visita**
- **Suggerimento per l'implementazione:** utilizzare le seguenti strutture ausilarie:
  - **color[u]**: mantiene il colore di ogni nodo durante la visita in ampiezza
  - **parent[u]**: mantiene il predecessore di **u** nella visita del grafo (il nodo da cui si è arrivati a visitare **u**)