Rappresentazione e utilizzo di GRAFI

Prof. Dalla Pozza

ITT Chilesotti

Grafi

Rappresentazione di un insieme di OGGETTI in RELAZIONE.

Gli oggetti sono rappresentati da NODI o VERTICI.

La relazione è rappresentata da ARCHI o FRECCE.

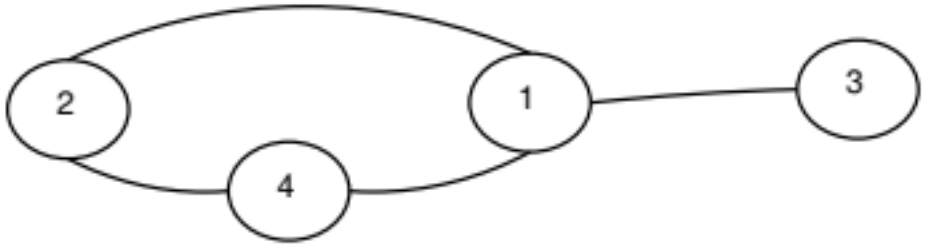


Figura 8.1: Esempio di grafo con 4 vertici e 4 lati

Grafi

Rappresentazione di un insieme di OGGETTI in RELAZIONE.

Matematicamente, è descritto dalla coppia G=(V,E)

Gli oggetti sono rappresentati da NODI o VERTICI.

Matematicamente è descritto da un insieme V che contiene (e numera) i vertici.

La relazione è rappresentata da ARCHI o FRECCE.

Matematicamente è descritto da un insieme E che contiene COPPIE DI VERTICI.

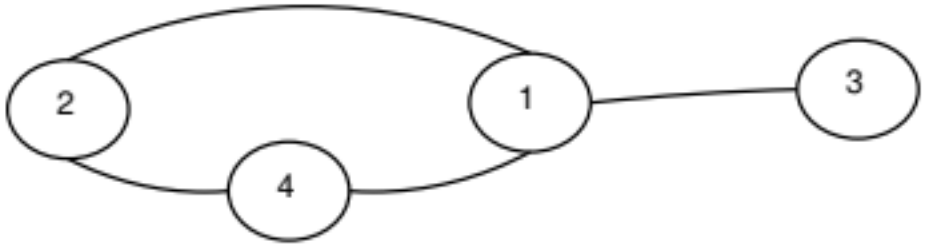


Figura 8.1: Esempio di grafo con 4 vertici e 4 lati

Esempi di applicazioni dei grafi

- Una carta stradale, dove i vertici sono le città e gli archi sono le strade che li uniscono.

- Un gruppo di persone (vertici) messi in relazione tramite da rapporti di amicizia

- Un insieme di attività da eseguire in un certo ordine per raggiungere uno scopo, dove le attività sono i vertici e le relazioni di precedenza tra le attività sono gli archi del grafo.

- La rete elettrica nazionale, dove i vertici sono le centrali elettriche e gli archi sono le linee ad alta tensione che le collegano.

Altre caratteristiche dei grafi

Grafo ORIENTATO: la relazione tra i vertici (i,j) è diversa dalla relazione tra (j,i) che potrebbe mancare

Gli archi si rappresentano con frecce.

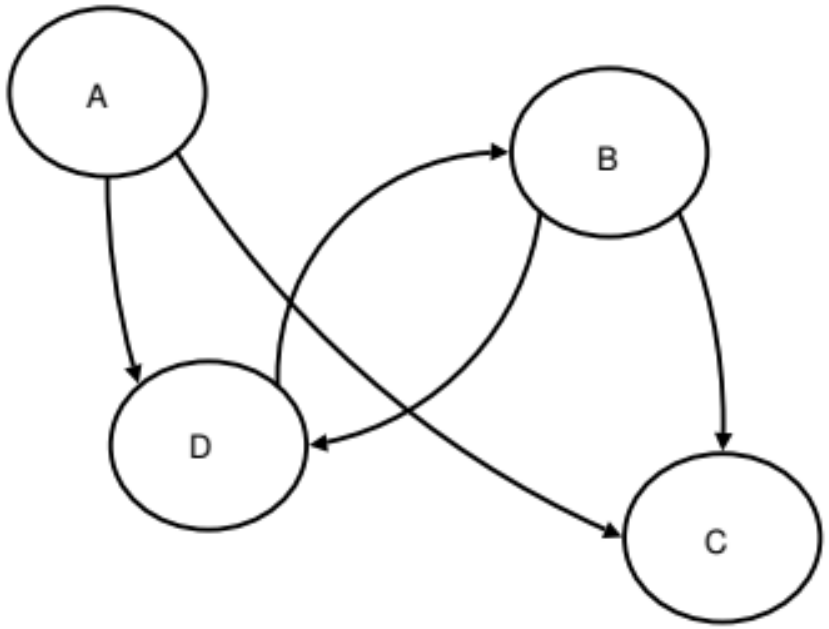


Figura 8.2: Esempio di grafo orientato

Altre caratteristiche dei grafi

Grafo CONNESSO: esiste un «percorso» che permette di collegare ciascuna coppia di nodi:

C'è una singola «componente connessa» e non più componenti «staccate»

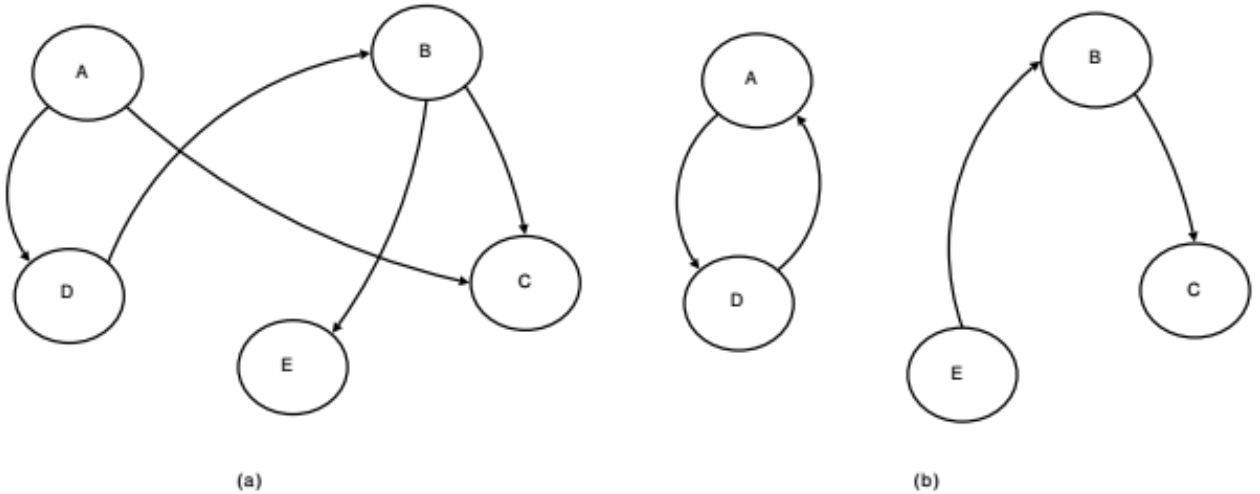


Figura 8.3: Grafo connesso (a) e grafo non connesso (b)

Altre caratteristiche dei grafi

Grafo PESATO: su ciascun arco è associato ad un valore che rappresenta un «costo»

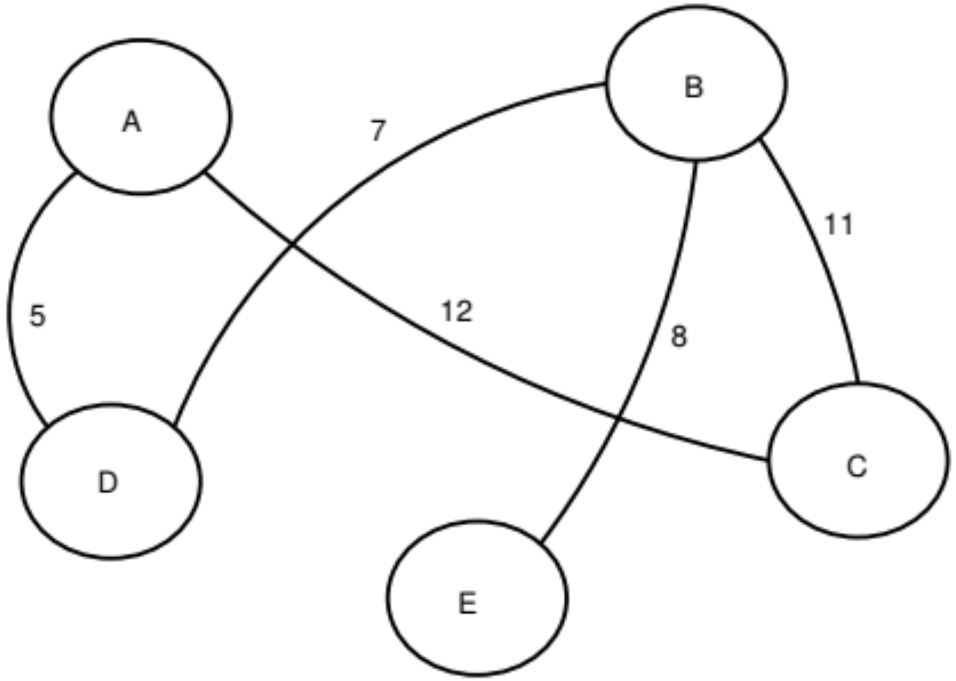


Figura 8.4: Grafo pesato

Rappresentazione informatica grafi

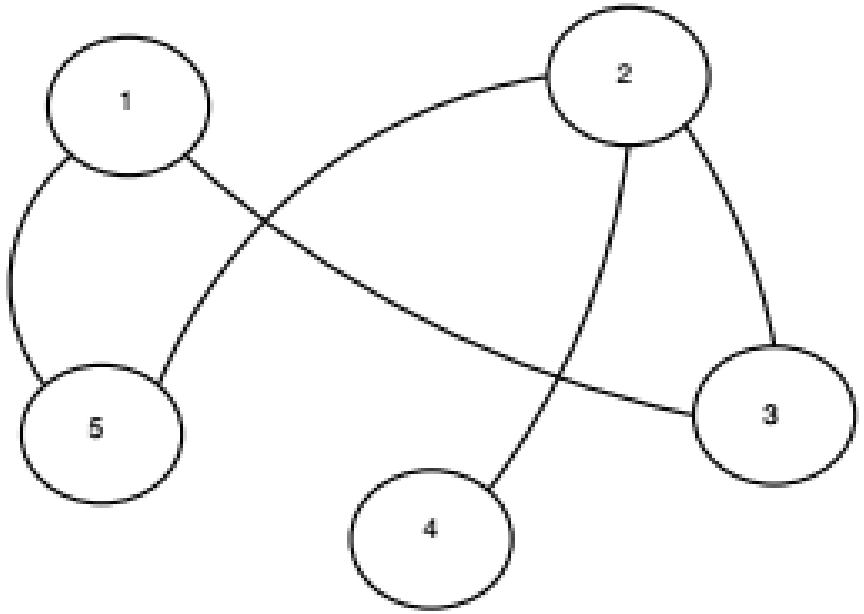


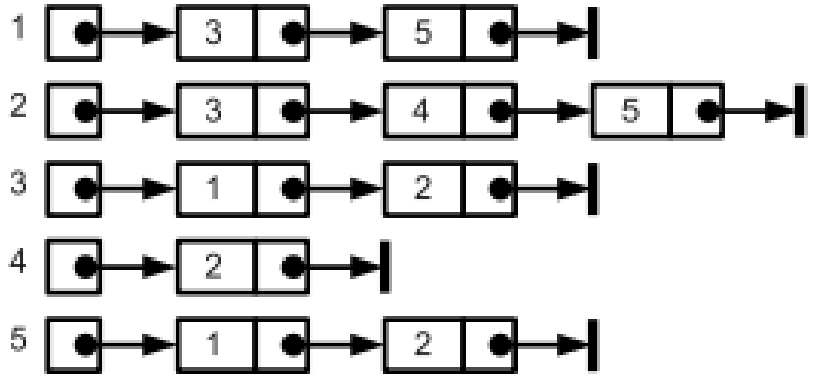
Figura 8.5: Grafo di esempio

Matrice delle adiacenze: M[i][j] indica la presenza dell'arco tra i e j

Lista delle adiacenze: per ciascun vertice si elencano i nodi connessi (più facili da realizzare in C++ con il container list)

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

(a)



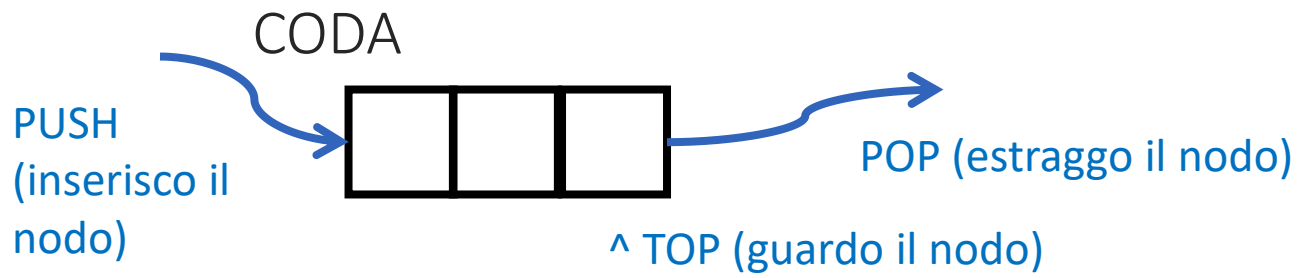
(b)

Algoritmi sui grafi: visita di un grafo

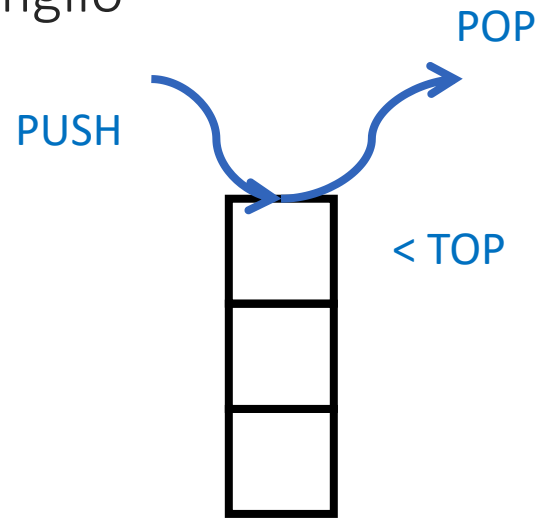
Il problema della visita consiste nel visitare tutti i vertici di un grafo. Per fare questo ci sono due modalità: in PROFONDITA' o in AMPIEZZA.

- in profondità (Depth First Search): quando si visita un nodo (non ancora visitato) , successivamente si passa ad un nodo figlio
- in ampiezza (Breadth First Search): quando si visita un nodo, successivamente si passa a visitare un nodo «fratello» prima di passare ad un nodo figlio

Per implementare gli algoritmi si usano le strutture dati

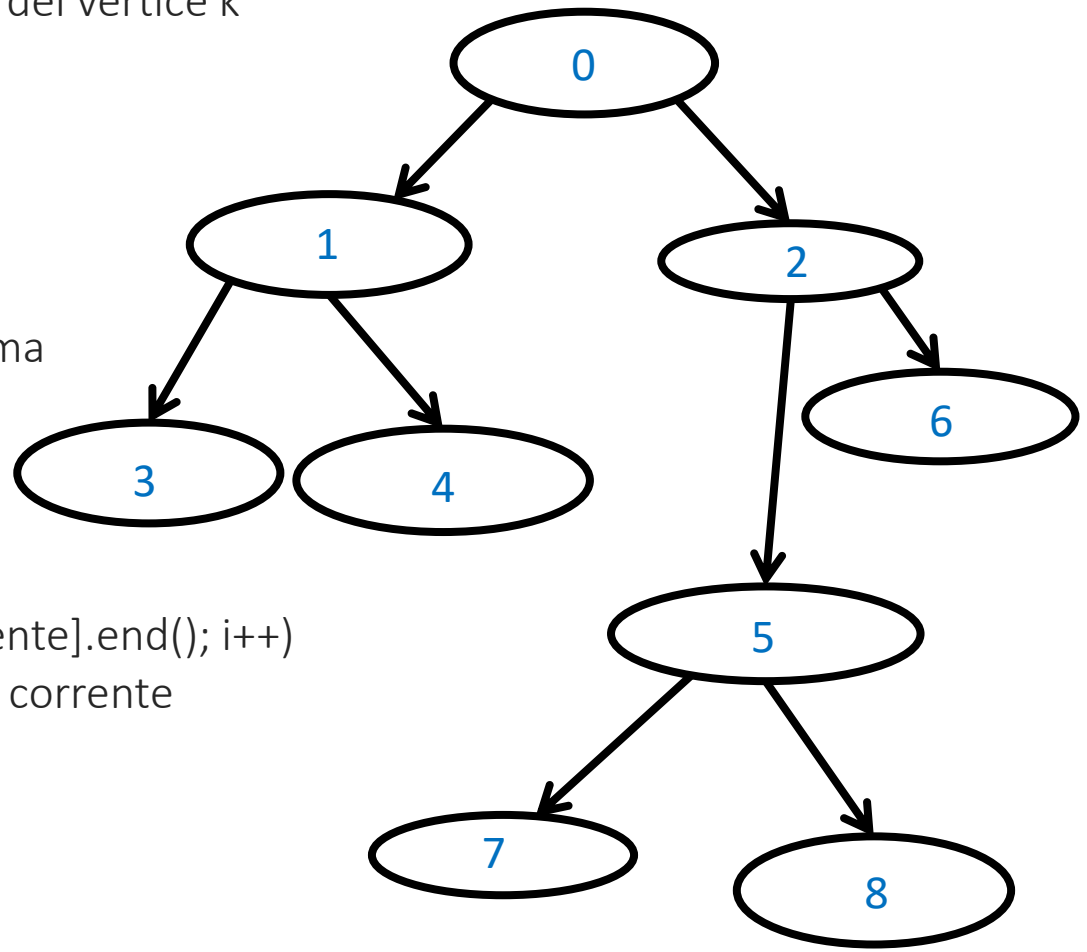


PILA



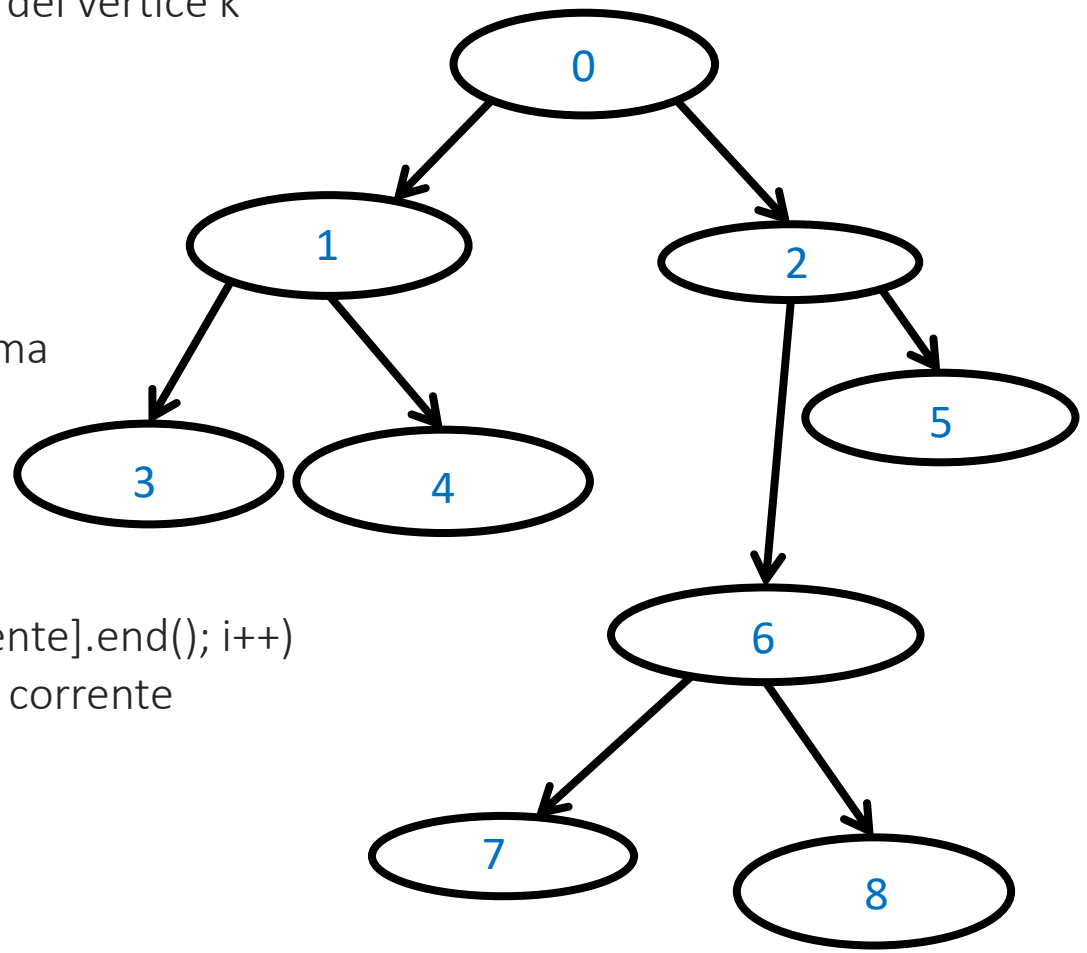
Algoritmi sui grafi: visita in profondità

```
list<int> liste[100];   // array di liste, liste[k] è la lista delle adiacenze del vertice k
int visitato[100];      // visitato[k] indica se il vertice k è stato visitato
stack <int> pila;       // struttura dati pila
void visita_profondita(int n) {  // visita il grafo partendo dal nodo n
   pila.push(n);                 // inserisce il primo vertice
   while (!pila.empty()) {       // mentre la pila non è vuota
      int corrente = pila.top(); // leggi quale vertice è presente in cima
      pila.pop();                // estrae il vertice in cima
      if (visitato[corrente] == false) { // se non è visitato
         visitato[corrente] = true;      // visita il nodo
         cout << corrente << endl;       // esegue l'elaborazione
         for (list <int>::iterator i = liste[corrente].begin(); i!=liste[corrente].end(); i++)
            pila.push(*i);               // aggiunge gli altri vertici connessi al corrente

      }
   }
}  // visita nell'ordine 0, 2, 6, 5, 8, 7, 1, 4, 3
```



Algoritmi sui grafi: visita in ampiezza

```
list<int> liste[100];   // array di liste, liste[k] è la lista delle adiacenze del vertice k
int visitato[100];      // visitato[k] indica se il vertice k è stato visitato
queue <int> coda;   // struttura dati coda
void visita_ampiezza(int n) {  // visita il grafo partendo dal nodo n
   coda.push(n);                    // inserisce il primo vertice
   while (!coda.empty()) {          // mentre la coda non è vuota
      int corrente = coda.top(); // leggi quale vertice è presente in cima
      coda.pop();                   // estrae il vertice in cima
      if (visitato[corrente] == false) { // se non è visitato
         visitato[corrente] = true;     // visita il nodo
         cout << corrente << endl;    // esegue l'elaborazione
         for (list <int>::iterator i = liste[corrente].begin(); i!=liste[corrente].end(); i++)
            coda.push(*i);          // aggiunge gli altri vertici connessi al corrente

      }
   }
}  // visita nell'ordine 0, 1, 2, 3, 4, 6, 5, 7, 8
```



Algoritmi sui grafi: cammini minimi

In un grafo un cammino è una sequenza di vertici che va da un vertice di partenza ad uno di destinazione. Si può definire sia per grafi orientati che non orientati.

Un problema noto è la ricerca del cammino minimo in un grafo, ovvero il cammino che ha la somma minore di costi partendo da un nodo e arrivando alla destinazione.

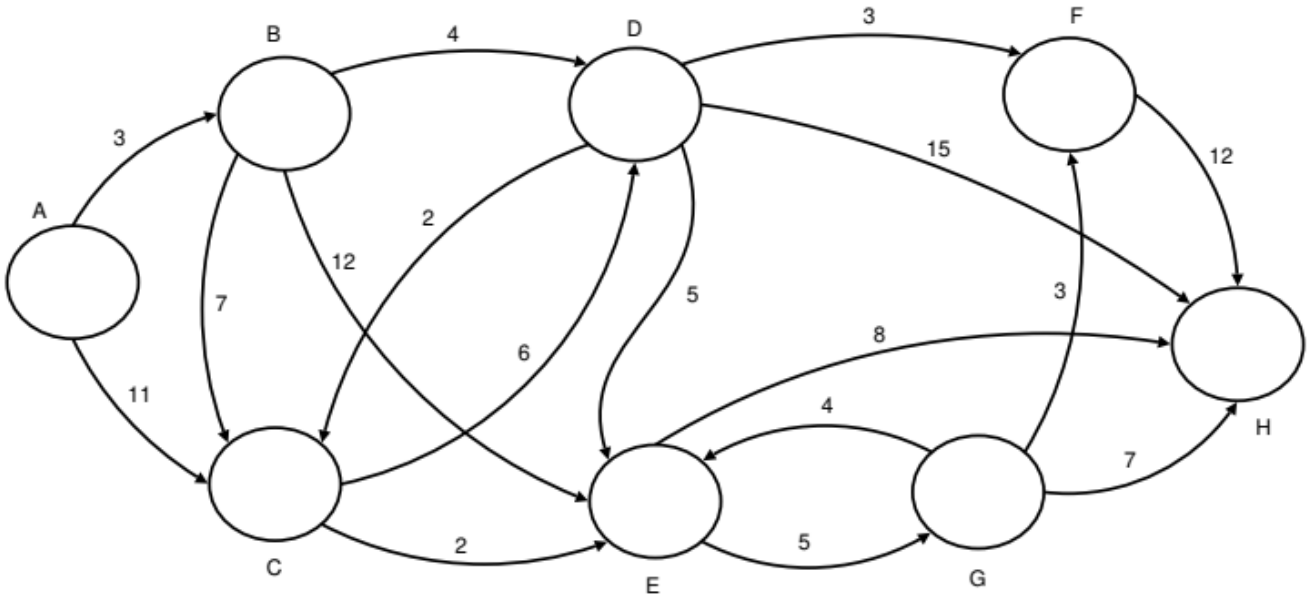


Figura 8.10: Problema del cammino minimo da A a H

Algoritmi sui grafi: Dijkstra

Intuitivamente, l'algoritmo funziona in questo modo. Ad ogni iterazione ci sono dei vertici che sono già stati visitati (inizialmente solo l'origine) e altri da visitare.

Tra tutti i vertici raggiungibili da quelli già visitati, si sceglie di visitare quel vertice che ha il valore minimo della somma per arrivare a quel vertice + costo arco.

Questo valore minimo ottenuto è il costo minimo del cammino dal vertice origine a quel nodo.

In questo modo ottengo TUTTI i cammini minimi dal vertice origine.

Una possibile implementazione è sulla guida.