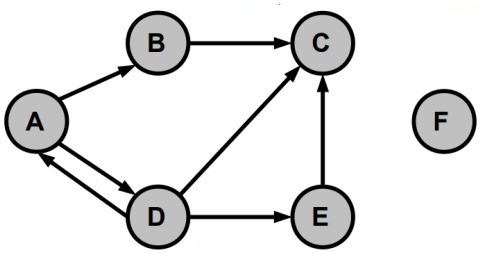
▼ **11.0 - Grafi**

▼ **11.1 - Introduzione ai grafi**

Grafi orientati e non orientati

Un **grafo orientato** è una coppia (V, E) tale che:

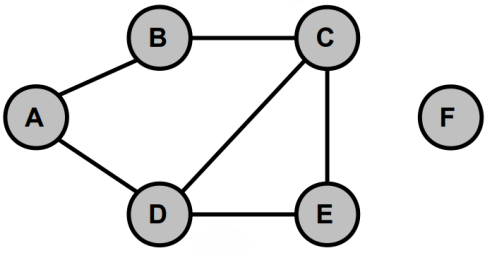
- V è un insieme di **vertici**.
- E è un insieme di **archi**, ovvero coppie non ordinate di vertici.
- Esistono archi (X, X) e vengono chiamati **cappi**.



Grafo orientato.

Un **grafo non orientato** è una coppia (V, E) tale che:

- V è un insieme di **vertici**.
- E è un insieme di **archi**, ovvero coppie ordinate di vertici.
- Non esistono archi (X, X).



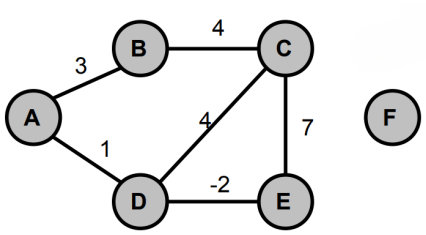
Grafo non orientato.

Grafi pesati

Un **grafo pesato** è un grafo in cui ogni arco ha un valore numerico che corrisponde al suo peso.

In alcune implementazioni, come ad esempio nella matrice di adiacenza, viene utilizzato il peso ∞ per due vertici tra i quali non esiste un arco.

Solitamente esiste una **funzione** $w : E \to \mathbb{R}$ che dato un arco restituisce il suo peso.



Grafo pesato.

Incidenza e adiacenza

In un grafo orientato l'arco (v, w) è **incidente** da v in w.

In un grafo (orientato/non orientato) w è **adiacente** a v se e solo se $(v, w) \in E$.

Notiamo che l'adiacenza è una relazione simmetrica se il grafo non è orientato, mentre se il grafo è orientato è importante l'ordine dei vertici nella coppia (v, w) (es. se $(v, w) \in E$ allora w è adiacente a v, ma non è detto che anche v sia adiacente a w).

Grado di un vertice

Grafo non orientato

In un grafo non orientato, il **grado** (δ) di un vertice corrisponde al numero di archi che partono da esso.

Grafo orientato

In un grafo orientato, il **grado entrante (uscente)** di un vertice corrisponde al numero di archi incidenti in (da) da esso.

In un grafo orientato, il **grado** (δ) di un vertice corrisponde alla somma del suo grado entrante e del suo grado uscente.

Grafi completi

Un **grafo non orientato completo** è un grafo non orientato che ha un arco tra ogni coppia di vertici.

Il numero di archi in un grafo non orientato completo con n vertici è equivalente a $\binom{n}{2} = \frac{n(n-1)}{2}$.

Cammini

Un **cammino** in un grafo è una sequenza di vertici $\langle w_0, w_1, \ldots, w_k \rangle$ tali che w_{i+1} è adiacente a w_i per $0 \leq i \leq k-1$.

La l**unghezza di un cammino** corrisponde al numero di archi attraversati, equivalente al numero di vertici meno 1.

Un cammino si dice **semplice** se tutti i suoi vertici sono distinti, ovvero compaiono una sola volta nella sequenza.

Si dice che w è **raggiungibile** da v se esiste almeno un cammino che partendo da v arriva a w.

In un grafo non orientato, si dice che tale grafo è **connesso** se esiste un cammino da ogni vertice verso ogni altro vertice.

In un grafo orientato, si dice che tale grafo è **fortemente connesso** se esiste un cammino da ogni vertice verso ogni altro vertice.

Versione non orientata

Se un grafo è orientato, il grafo ottenuto ignorando la direzione degli archi e i cappi è detto grafo non orientato sottostante oppure **versione non orientata** del grafo.

Se un grafo orientato non è fortemente connesso, ma la sua versione non orientata è connessa, allora diciamo che tale grafo è **debolmente connesso**.

Cicli

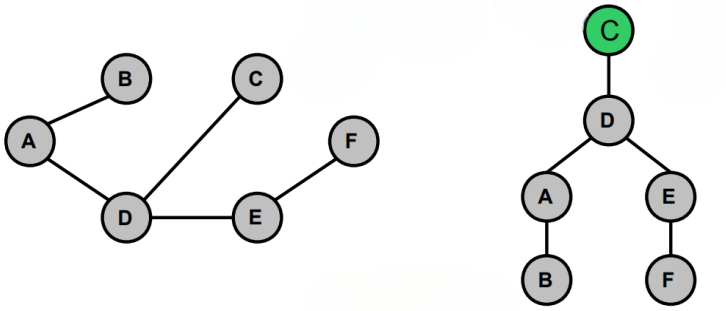
Un **ciclo** è un cammino $\langle w_0, w_1, \ldots, w_n \rangle$ tale che $w_0 = w_n$ e con una lunghezza di almeno 1 nei grafi orientati e di almeno 3 nei grafi non orientati.

Un ciclo è **semplice** se tutti i vertici che compongono il suo cammino sono distinti.

Un grafo non orientato è **aciclico** se è privo di cicli semplici.
Un grafo orientato è **aciclico** se è privo di cicli.

Alberi liberi

Un **albero libero** è un grafo non orientato connesso e aciclico. Se un vertice di tale albero è detto radice, otteniamo un **albero radicato**.



Albero libero e albero radicato.

Operazioni basilari

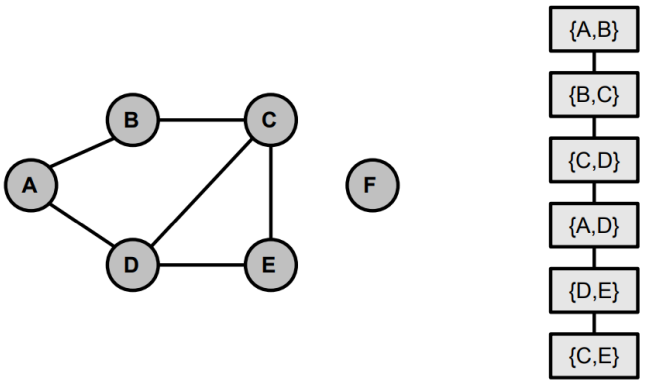
Le **operazioni basilari** che ogni grafo deve supportare sono le seguenti:

- **int numVertici()**
- **int numArchi()**
- **int grado(vertice v)**
- **(arco, arco, …, arco) archiIncidenti(vertice v)**
- **(vertice, vertice) estremi(arco e)**
- **vertice opposto(vertice x, arco e)**
- **bool sonoAdiacenti(vertice x, vertice y)**
- **void aggiungiVertice(vertice v)**
- **void aggiungiArco(vertice x, vertice y)**
- **void rimuoviVertice(vertice v)**
- **void rimuoviArco(arco e)**

Implementazioni dei grafi

Liste di archi

Un primo modo per implementare dei grafi è quello di utilizzare una semplice **lista** per elencare i suoi archi.



Lista di archi.

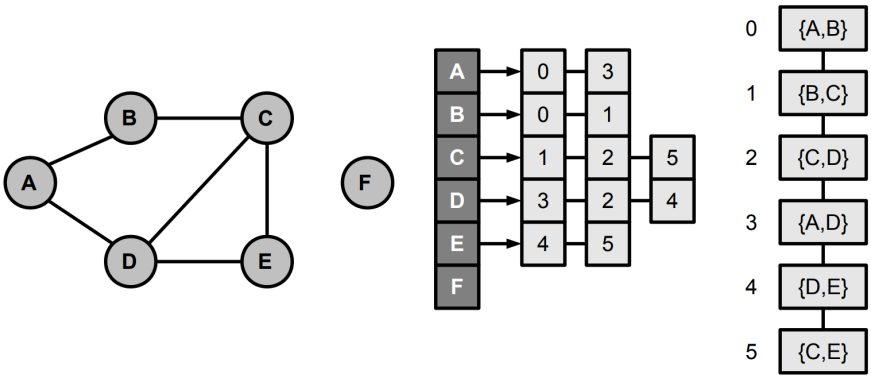
Il **costo computazionale** in termini di spazio è $\Theta(|E|)$.

I **costi computazionali** in termini di tempo delle operazioni basilari risultano però poco efficienti in quanto spesso richiedono la scansione dell'intera lista di archi:

- grado(vertice v): $O(m)$.
- archiIncidenti(vertice v): $O(m)$.
- sonoAdiacenti(vertice x, vertice y): $O(m)$.
- aggiungiVertice(vertice v): $O(1)$.
- aggiungiArco(vertice x, vertice y): $O(1)$.
- rimuoviVertice(vertice v): $O(m)$.
- rimuoviArco(arco e): $O(1)$.

Liste di incidenza

Per migliorare i costi computazionali è utile mantenere insieme alla lista di archi anche una lista detta **lista di incidenza** nella quale per ogni vertice si ha una lista degli indici dei nodi a cui esso appartiene.



Lista di incidenza.

I **costi computazionali** diventano i seguenti:

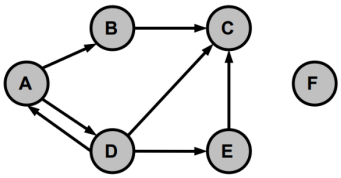
- grado(vertice v): $O(\delta(v))$.
- archiIncidenti(vertice v): $O(\delta(v))$.
- sonoAdiacenti(vertice x, vertice y): $O(min(\delta(x), \delta(y)))$.

- aggiungiVertice(vertice v): $O(1)$.
- aggiungiArco(vertice x, vertice y): $O(1)$.
- rimuoviVertice(vertice v): $O(m)$.
- rimuoviArco(arco e): $O(\delta(x) + \delta(y))$.

Matrici di adiacenza

Un'altra possibile implementazione utilizza una matrice, detta **matrice di adiacenza**, i quali valori vengono decisi in questo modo:

$$M(u,v) = \begin{cases} 1 & (u,v) \in E \\ 0 & (u,v) \notin E \end{cases}$$

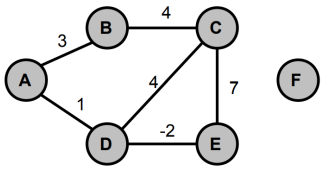


$$M = \begin{array}{c} \\ A \\ B \\ C \\ D \\ E \\ F \end{array} \begin{array}{c} \begin{array}{cccccc} A & B & C & D & E & F \end{array} \\ \begin{pmatrix} 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \end{array}$$

Matrice di adiacenza per grafo non pesato.

Oppure per un grafo pesato:

$$M(u,v) = \begin{cases} w(u,v) & (u,v) \in E \\ \infty & (u,v) \notin E \end{cases}$$

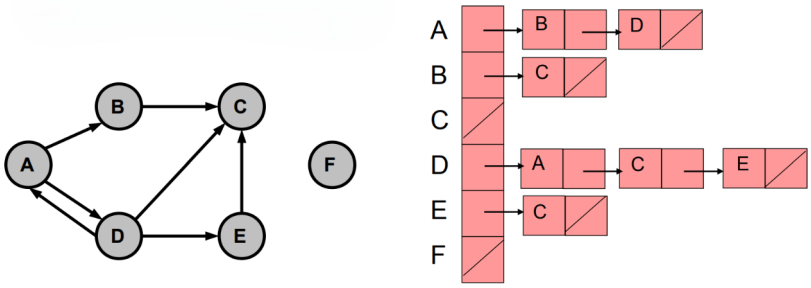


$$M = \begin{array}{c} \\ A \\ B \\ C \\ D \\ E \\ F \end{array} \begin{array}{c} \begin{array}{cccccc} A & B & C & D & E & F \end{array} \\ \begin{pmatrix} \infty & 3 & \infty & 1 & \infty & \infty \\ 3 & \infty & 4 & \infty & \infty & \infty \\ \infty & 4 & \infty & 4 & 7 & \infty \\ 1 & \infty & 4 & \infty & -2 & \infty \\ \infty & \infty & 7 & -2 & \infty & \infty \\ \infty & \infty & \infty & \infty & \infty & \infty \end{pmatrix} \end{array}$$

Matrice di adiacenza per grafo pesato.

Liste di adiacenza

Un'ulteriore implementazione comprende l'utilizzo di una lista detta **lista di incidenza** nella quale per ogni vertice si ha una lista dei nodi adiacenti ad esso.



Lista di adiacenza.

Il **costo computazionale** in termini di spazio è $\Theta(|V| + |E|)$.

I **costi computazionali** in termini di tempo sono i seguenti:

- grado(vertice v): $O(\delta(v))$.
- archiIncidenti(vertice v): $O(\delta(v))$.
- sonoAdiacenti(vertice x, vertice y): $O(min(\delta(x), \delta(y)))$.
- aggiungiVertice(vertice v): $O(1)$.
- aggiungiArco(vertice x, vertice y): $O(1)$.
- rimuoviVertice(vertice v): $O(m)$.
- rimuoviArco(arco e): $O(\delta(x) + \delta(y))$.

▼ 11.2 - Algoritmi di visita di grafi

Il problema della **visita di grafi** consiste nel, dato un vertice del grafo, visitare una sola volta tutti i vertici raggiungibili dal vertice dato.

L'idea è quella di costruire un algoritmo che, dato un vertice s, costruisce un **albero** T radicato in s che al termine della visita conterrà una sola volta tutti i vertici raggiungibili partendo da s. Durante la visita, al fine di evitare di visitare dei vertici più di una volta, ogni vertice può trovarsi in uno dei seguenti 3 stati:

- **Inesplorato**: il vertice non è ancora stato incontrato.
- **Aperto**: l'algoritmo ha incontrato il vertice la prima volta.
- **Chiuso**: il vertice è stato visitato completamente, ovvero tutti i suoi archi incidenti sono stati esplorati.

Durante l'esecuzione è utile mantenere un sottoinsieme F di T nel quale inserire tutti i nodi che stanno venendo visitati. In questo modo in base alla presenza o meno di un vertice nell'insieme F sapremo in quale dei 3 stati si trova:

- v non è in T: inesplorato.
- v è in F: aperto.
- v è in $T - F$: chiuso.

L'algoritmo generico per la visita di un grafo è il seguente:

```
albero visita(G, s) {
  for (each v in V)
    v.mark = false

  T = s
```

```
  F = {s}
  s.mark = true

  while (F != ∅) {
    u = F.extract()
    // visita il vertice u
    for (each v adiacente a u) {
      if (!v.mark) {
        v.mark = true
        T = T ∪ v
        F.insert(v)
        v.parent = u
      }
    }
  }
  return T
}
```

Il **costo computazionale** di questo algoritmo è (n: numero di vertici, m: numero di archi):

- **Liste di adiacenza**: $O(n + m)$.
- **Matrice di adiacenza**: $O(n^2)$.

Analizzeremo due tipologie di visite di un grafo, la visita in ampiezza e quella in profondità.

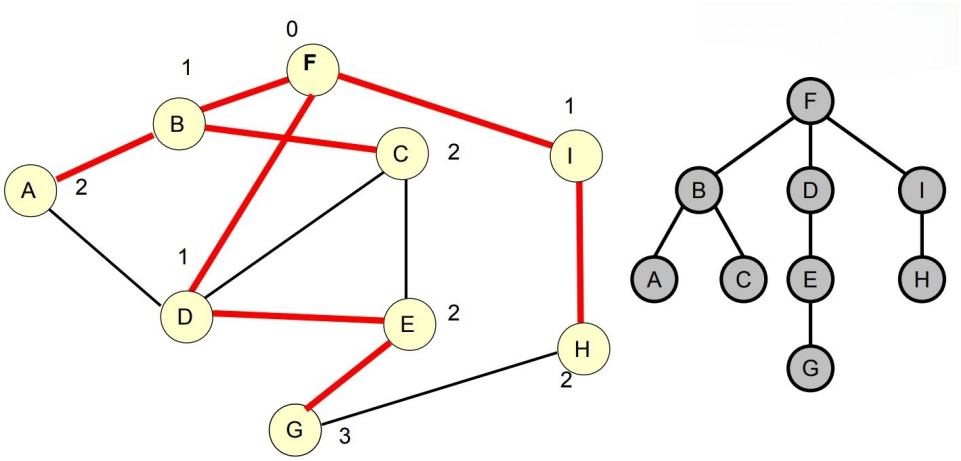
Visita in ampiezza

Tramite la **visita in ampiezza (BFS - Breadth First Search)** di un grafo vogliamo creare un albero in cui inserire tutti i vertici del grafo visitati a distanza crescente dal nodo sorgente dato in input, ovvero prima di visitare i nodi a distanza $k + 1$ dal vertice dato in input devono essere prima visitati tutti i nodi a distanza k.

È possibile utilizzare tale tipologia di visita anche per creare algoritmi che calcolino la minima distanza tra la sorgente e un altro nodo del grafo raggiungibile da essa.

Lo **pseudocodice** di un'algoritmo di visita in ampiezza è il seguente:

```
albero BFS(Grafo G, vertice s) {
  for (each v in V)
    v.mark = false

  T = s
  F = new Queue()
  F.enqueue(s)
  s.mark = true

  s.dist = 0
  while (F != ∅) {
    u = F.dequeue()
    // visita il vertice u
    for (each v adiacente a u) {
      if (!v.mark) {
        v.mark = true
        T = T ∪ v
        F.enqueue(v)
        v.parent = u
        v.dist = u.dist + 1
      }
    }
  }

  return T
}
```



Albero creato da una visita in ampiezza.

Visita in profondità

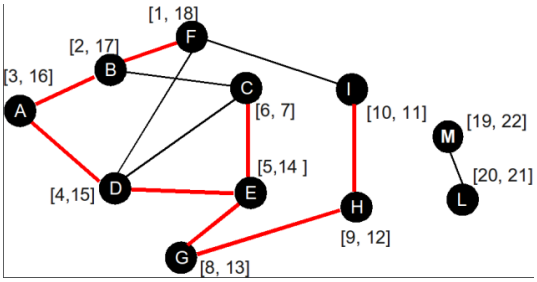
La **visita in profondità (DFS - Deapth First Search)**, a differenza di quella in ampiezza, visita tutti i nodi presenti in un grafo, dunque non solo quelli raggiungibili dal vertice dato in input. Per questo motivo la visita in profondità restituisce una foresta, ovvero un insieme di alberi nei quali vengono visitati i nodi andando il più lontano da livello attuale prima di passare al nodo sul successivo in quel livello.

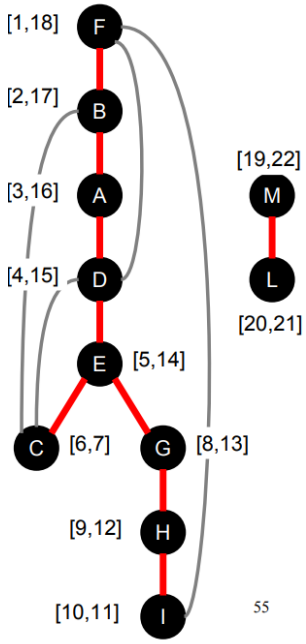
Lo **pseudocodice** di una visita in profondità in versione ricorsiva è il seguente (white: inesplorato, grey: aperto, black: chiuso, dt: discovery time, ft: finish time):

```
int time = 0

void DFS(Grafo G) {
  for (each u in V) {
    u.mark = white
    u.parent = null
  }

  for (each u in V) {
    if (u.mark == white)
      DFS-visit(u)
  }
}

void DFS-visit(vertice u) {
  u.mark = gray
  time++
  u.dt = time

  for (each v adiacente a u) {
    if (v.mark == white) {
      v.parent = u
      DFS-visit(v)
    }
  }

  time++
  u.ft = time
  u.mark = black
}
```

Poniamo ora $n.dt$ (discovery time) come l'istante in cui un vertice è stato aperto, e $n.ft$ (finish time) come l'istante in cui lo stesso vettore è stato chiuso, possiamo concludere che:

$$v \text{ è discendente di } v \text{ nella foresta} \iff u.dt < v.dt < v.ft < u.ft$$

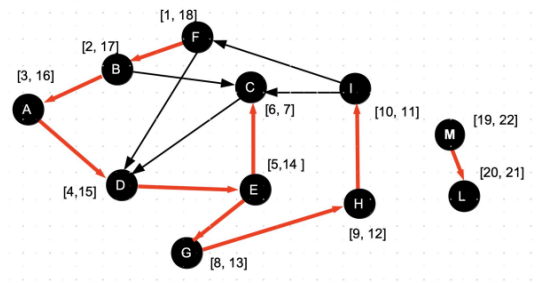


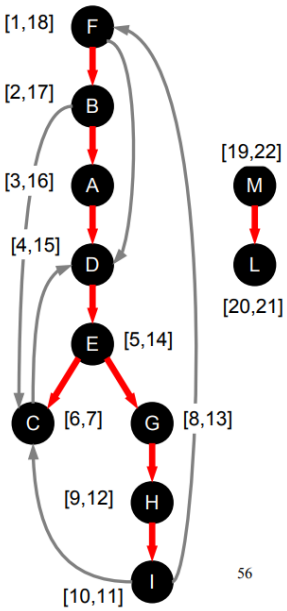


Nelle figure soprastanti $[x, y]$ corrispondono a $[n.dt, n.ft]$.

È inoltre possibile sfruttare il discovery time e il finish time per verificare la direzione di un arco (u, v) :

- Se $u.dt < v.dt$ e $v.ft < u.ft$ allora l'arco è in **avanti**, ovvero va da un nodo ascendente a un nodo suo discendente.
- Se $v.dt < u.dt$ e $u.ft < v.ft$ allora l'arco è all'**indietro**, ovvero va da un nodo discendente a un nodo suo ascendente.
- Se $v.ft < u.dt$ allora l'arco è di **attraversamento a sinistra**, ovvero i due nodi hanno un padre in comune, ma nessuno è ascendente/discendente dell'altro.





Applicazioni delle visite di grafi

Alcune **possibili applicazioni** degli algoritmi di visita di un grafo è il seguente:

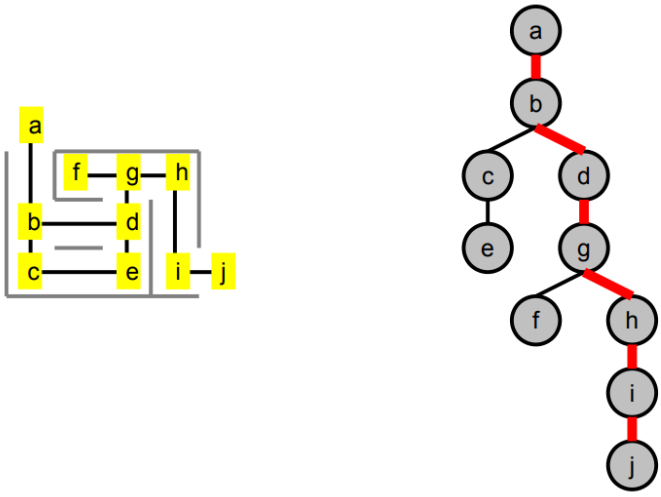
- Identificare il **cammino più breve** tra due vertici di un grafo.
- Verificare che un grafo sia **aciclico**.

  Realizzabile eseguendo una visita in profondità e verificando che non ci siano archi all'indietro.
- **Ordinamento topologico**.
- Individuare le componenti **connesse** di un grafo non orientato.
- Individuare le componenti **fortemente connesse** di un grafo orientato.

Cammino più breve

È possibile utilizzare l'albero creato dopo la visita in ampiezza di un grafo al fine di ottenere il **percorso più breve** tra due vertici:

```
void printPath(G, s, v) {
  if (v == s) print(s)
  else if (v.parent == null)
    print("no path from s to v")
  else {
    print-path(G, s, v.parent)
    print(v)
  }
}
```

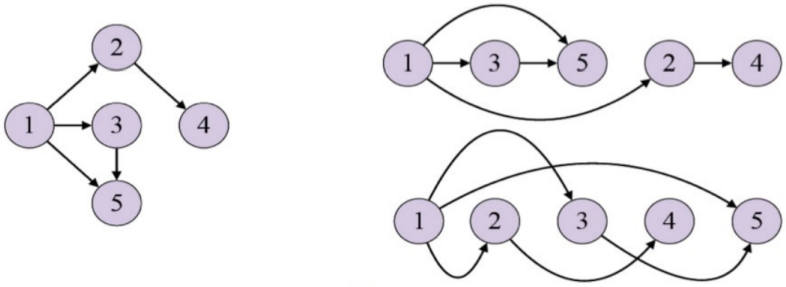


Utilizzo della visita in ampiezza per calcolare il percorso più veloce per uscire da un labirinto.

Ordinamento topologico

Dato un grafo ordinato privo di cicli, un **ordinamento topologico** consiste in un ordinamento lineare dei suoi nodi tale che, se v è raggiungibile da u, allora u compare prima di v nell'ordinamento.

Possono esistere più di un ordinamento topologico per lo stesso grafo.



È possibile realizzare ordinamenti topologici facendo uso di una visita in profondità aggiungendo ogni nodo alla testa di una lista nel momento del suo finish time. In questo modo si otterrà una lista in cui i nodi sono ordinati in base decrescente di finish time, quindi tale lista rispetta la definizione di ordinamento topologico in quanto come abbiamo visto se un nodo v è raggiungibile da un nodo u, allora il finish time di v sarà minore di u, quindi nella lista v si troverà dopo u.

Componenti connesse

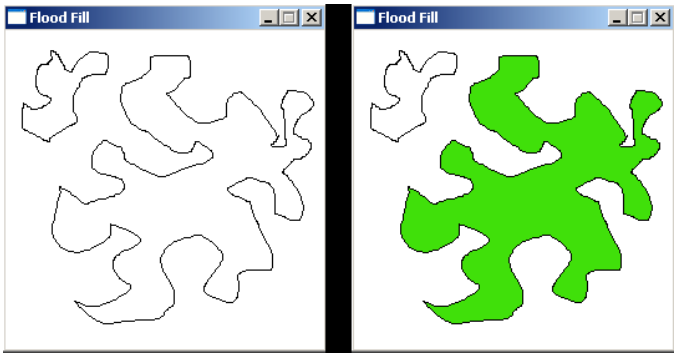
In un grafo non orientato, due vertici appartengono alla stessa **componente connessa** se da uno è possibile raggiungere l'altro. Si può utilizzare la visita in profondità per identificare tutte le componenti connesse tra loro.

Nel seguente pseudocodice etichetteremo tutti i nodi di un grafo con un intero, e al termine dell'esecuzione se due nodi sono etichettati con lo stesso intero allora sono connessi:

```
void CC(G) {
  for (each u in G) {
    u.cc = -1
    u.parent = null
  }

  k = 0
  for (each u in G) {
```

```
    if (u.cc < 0) {
      CC-visit(u, k)
      k++
    }
  }
}

void CC-visit(u, k) {
  u.cc = k
  for (each v adiacente a u) {
    if (v.cc < 0) {
      v.parent = u
      CC-visit(v, k)
    }
  }
}
```

Una possibile applicazione della ricerca delle componenti fortemente connesse è il riempimento a tinta unita di alcuni software come paint.



Floodfill, riempimento a tinta unita.

Componenti fortemente connesse

In un grafo orientato, due nodi appartengono alla stessa **componente fortemente connessi** se c'è un cammino che connette un vertice con l'altro e viceversa.

Notiamo che è possibile determinare tutti i nodi fortemente connessi a un certo vertice x calcolando prima l'insieme $D(x)$ dei nodi raggiungibili da x e l'insieme $A(x)$ dei nodi da cui si può raggiungere x per poi effettuare l'intersezione $D(x) \cap A(x)$ tra i due insiemi.

Per costruire tali insiemi è possibile, per quanto riguarda $D(x)$, richiamare una funzione di visita in ampiezza usando x come sorgente, mentre per quanto riguarda $A(x)$ occorre invertire la direzione degli archi nel grafo e richiamare ancora una funzione di visita in ampiezza usando x come sorgente.

Lo **pseudocodice** è il seguente:

```
lista SCC(Grafo G, nodo x) {
  L = lista vuota di nodi
  (1) Eseguire BFS(G, x) marcando i nodi visitati
  (2) Calcolare il grafo trasposto GT
      (invertire la direzione degli archi di G)
  (3) Eseguire BFS(GT, x), mettendo in L i nodi
      visitati che sono stati marcati durante (1)
  return L
}
```

Il **costo computazionale**, visto che tutte e 3 i passaggi hanno costo $O(n+m)$, è $O(n+m)$.

Per calcolare tutte le componenti fortemente connesse di un grafo occorre richiamare la funzione appena discussa per ogni nodo del grafo. Il costo computazionale in questo caso è $n \cdot O(n^2 + mn) = O(n^2 + mn)$. Esiste un algoritmo più sofisticato che ha costo $O(n + m)$ ma che non tratteremo.

▼ 11.3 - Minimum spanning tree

Definizione

Albero di copertura

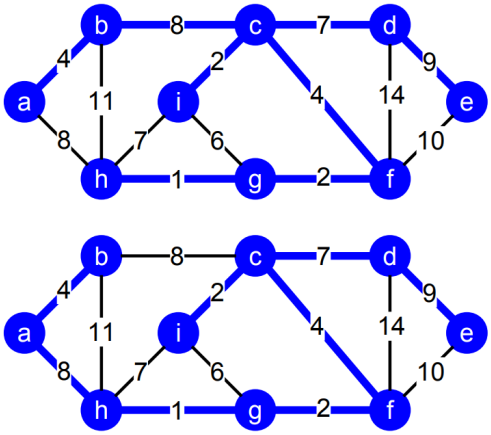
Un **albero di copertura** di un grafo $G = (V, E)$ è un sottografo $T = (V, E_T)$ tale che:

- T è un albero.
- T ha gli stessi nodi di G.
- $E_T \subseteq E$.

Minimum spanning tree

Un **minimum spanning tree (MST)**, o albero di copertura di peso minimo, è un albero di copertura il cui peso totale, ovvero la somma dei pesi dei singoli archi, sia minimo tra tutti i possibili alberi di copertura.

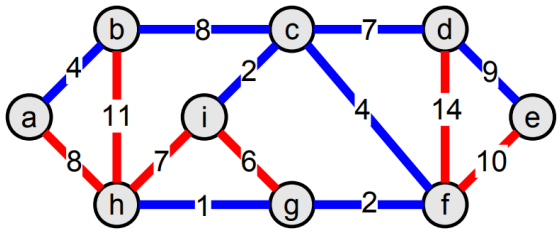
Nota: il minimum spanning tree non è necessariamente unico.



Minimum spanning tree.

Vedremo 2 algoritmi di tipo greedy che consentono di calcolare il minimum spanning tree.

Per convenzione ci riferiamo agli archi di colore **blu** per gli archi che fanno parte del MST, e agli archi di colore **rosso** per quelli che non ne fanno parte.



Minimum spanning tree.

L'idea nella costruzione di un MST tramite metodo greedy è quella di creare un albero T aggiungendo di volta in volta archi sicuri.

> Un arco (u, v) è detto **sicuro** per T se $T \cup (u, v)$ è ancora un sottoinsieme di un MST.

Lo **pseudocodice** di un tale algoritmo è il seguente:

```
Tree genericMST(Grafo G = (V, E, w)) {
  Tree T = Albero vuoto
  while (T non forma un albero di copertura) {
    trova un arco sicuro {u, v}
    T = T ∪ {u, v}
  }
  return T
}
```

Al fine di poter individuare archi sicuri occorre introdurre alcune definizioni.

> Un **taglio** $(S, V - S)$ di un grafo non orientato $G = (V, E)$ è una partizione di V in due sottoinsiemi disgiunti.

> Un arco (u, v) **attraversa il taglio** se $u \in S$ e $v \in V - S$.

> Un taglio **rispetta** un insieme di archi chiamato T se nessun arco appartenente a T attraversa il taglio.

> Un arco che attraversa il taglio è **leggero** se il suo peso è il minimo tra i pesi di tutti gli archi che attraversano il taglio.

Un metodo greedy per la costruzione di MST consiste nell'utilizzo in successione di una qualunque, purchè si possa usare, di queste due regole:

- **Regola del taglio**

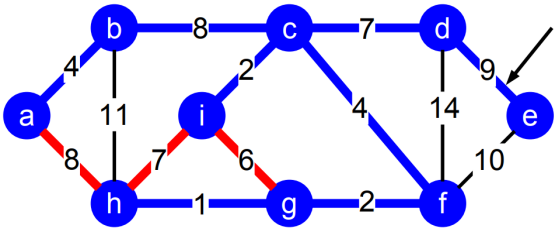
  Viene scelto un taglio che rispetta l'insieme degli archi già colorati di blu. Tra tutti gli archi che attraversano il taglio ne viene scelto uno leggero e viene colorato di blu.

- **Regola del ciclo**

Viene scelto un ciclo che non contenga archi di colore rosso e tra tutti gli archi non colorati di tale ciclo ne viene scelto uno di peso massimo e viene colorato di rosso.

Algoritmo di Kruskal

L'idea dell'**algoritmo di Kruskal** consiste nell'avere inizialmente n insiemi, uno per ogni vertice del grafo, e ognuno dei quali formato da un singolo nodo. Successivamente vengono uniti tra loro questi insiemi colorando di blu gli archi che faranno parte del MST, e di rosso quelli che non ne faranno parte. Per deciderlo analizziamo gli archi del grafo in ordine non decrescente, e se l'arco che si sta analizzando connette due insiemi distinti, allora lo si colora di blu, altrimenti di rosso.



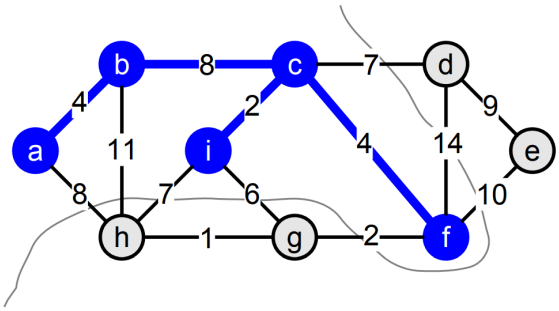
Lo **pseudocodice** di un algoritmo di Kruskal è il seguente:

```
Tree Kruskal-MST(Grafo G = (V, E, w)) {
  UnionFind UF
  Tree T = albero vuoto
  for (int i = 1; i < G.numNodi()) UF.makeSet(i)

  // ordina gli archi di E per peso w crescente
  sort(E, w)

  for (each {u, v} in E) {
    Tu = UF.find(u)
    Tv = UF.find(v)
    if (Tu != Tv) { // non fanno parte dell'insieme
      T = T ∪ {u, v} // aggiungi arco
      UF.union(Tu, Tv) // unisci componenti
    }
  }
  return T
}
```

Il **costo computazionale** nel caso in cui venga utilizzata la struttura dati QuickUnion è $O(m \log n)$.

Algoritmo di Prim

L'**algoritmo di Prim** si basa sull'idea di iniziare a costruire l'albero di copertura minimo da un certo nodo arbitrario chiamato radice, e ad ogni passo creare un taglio che divida il grafo in 2 insiemi di nodi, uno è quello dei vertici già facenti parte dell'MST e l'altro quello dei nodi non ancora raggiunti dall'MST. Tra gli archi che attraversano tale taglio ne viene scelto uno leggero e viene colorato di blu.



Lo **pseudocodice** di un algoritmo di Prim è il seguente:

```
integer[] primMST(Grafo G = (V, E, w), nodo s) {
  /* per ogni nodo viene indicato il peso
  minimo per essere collegato all'albero
  se non esiste un arco che lo collega -> infinito */
  double d[1 ... n]
  /* per ogni nodo viene indicato l'indice del padre nel MST */
  int p[1 ... n]
  /* per ogni nodo viene indicato con true se già
  presente nel MST, false altrimenti */
  boolean b[1 ... n]

  for (int v = 1; v < n; v++) {
    d[v] = ∞
    p[v] = -1
    b[v] = false
  }

  d[s] = 0
  /* coda con priorità in cui vengono inseriti
  i nodi non ancora nel MST ordinati in base
  al valore presente nell'array d */
  CodaPriorita<int, double> Q
  Q.insert(s, d[s])

  while (!Q.isEmpty()) {
    /* viene preso il nodo con il valore minimo
    nella coda e messo nel MST */
    u = Q.findMin()
    Q.deleteMin()
    b[u] = true

    /* vengono inseriti nella coda tutti i nodi
    adiacenti all'ultimo nodo aggiunto nell'albero
    con il relativo valore */
    for (each v adiacente a u && !b[v]) {
      if (d[v] == ∞) {
        Q.insert(v, w(u,v))
        d[v] = w(u,v)
        p[v] = u
      } else if (w(u,v) < d[v]) {
        Q.decreaseKey(v, d[v] - w(u, v))
        d[v] = w(u,v)
        p[v] = u
      }
    }
  }

  return p
}
```

Il **costo computazionale** nel caso in cui la coda con priorità venga implementata tramite min-heap è $O(m \log n)$.

▼ **11.4 - Cammini minimi**

Definizione

Costo di un cammino

Dato un grafo G, il **costo di un cammino** $\pi = (v_0, \dots, v_k)$ che collega il vertice v_0 con il vertice v_k è definito come:

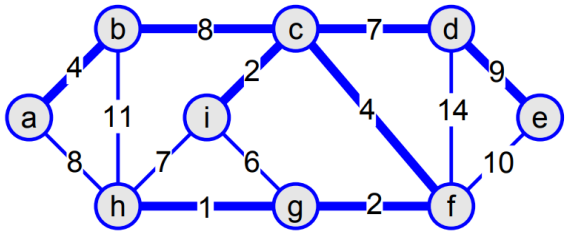
$$w(\pi) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

Cammino minimo

Dato un grafo G e due nodi v_0 e v_k il **cammino minimo** consiste nel cammino con costo minimo tra tutti i cammini che vanno da v_0 e v_k.

È importante notare che il problema della costruzione del MST e della ricerca del cammino minimo sono due problemi completamente differenti, in quanto non è detto che il cammino minimo tra due nodi si trovi all'interno dell'MST.

Ad esempio il cammino di costo minimo tra i nodi h e i nel seguente grafo non fa parte del MST.



Diverse formulazioni del problema

Esistono **diverse tipologie di problemi** che utilizzano il concetto di ricerca del cammino minimo:

1. **Cammino minimo fra una coppia di nodi**

   Determinare, se esiste, un cammino minimo tra due nodi appartenenti a un grafo.

2. **Single-source shortest path**

   Determinare i cammini di costo minimo da un nodo sorgente s fino a tutti gli altri nodi raggiungibili da s.

3. **All-pairs shortest path**

   Determinare i cammini di costo minimo tra ogni coppia di nodi in un grafo.

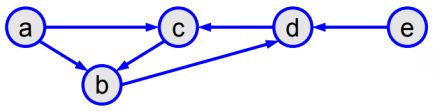
È importante tenere a mente che ancora non è stato trovato un algoritmo che consenta di risolvere il problema 1 senza risolvere anche il problema 2.

Single-source shortest path

Quando non esiste un cammino minimo?

Non esiste un cammino minimo tra due nodi se:

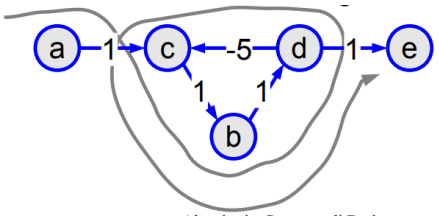
- La destinazione **non è raggiungibile** dalla sorgente.



Nodo e non raggiungibile partendo da a.

- Esiste un ciclo tra i due nodi di **costo negativo**.

  In questo caso è infatti sempre possibile trovare un cammino di costo minore facendo un ulteriore giro del ciclo.



Ciclo negativo per raggiungere e partendo da a.

Algoritmo di Bellman e Ford

L'**algoritmo di Bellman e Ford** consente di risolvere il problema single-source shortest path in un grafo.

L'idea è quella di assegnare ad ogni nodo un valore corrispondente al costo del cammino minimo trovato fino a quel momento per arrivarci, per comodità ci riferiamo a tale valore con $D[v]$, dove v è il nodo destinazione. L'algoritmo imposta quindi inizialmente il valore 0 alla sorgente e valore $+\infty$ a tutti gli altri nodi, in quanto ancora non è stato trovato un cammino minimo per nessun nodo. Ad ogni passo si controlla, per ogni arco (u, v) presente nel grafo, se il costo del cammino $D[u] + w(u, v) < D[v]$, e in quel caso $D[v]$ diventa $D[u] + w(u, v)$, in quanto è stato trovato un cammino di costo minore per arrivare a quel nodo. Dopo $n - 1$ passi, tante quante sono il numero dei possibili vertici raggiungibili da s, siamo sicuri di aver calcolato tutti i valori $D[v]$ minimi.

Lo **pseudocodice** dell'algoritmo di Bellman e Ford è il seguente:

```
double[1 ... n] bellmanFord(Grafo G = (V, E, w), int s) {
  int n = G.numNodi()
  int pred[1 ... n] // array che per ogni nodo tiene conto
  // del suo nodo precedente nel cammino minimo
  double D[1 ... n]

  for (int v = 1; v <= n; v++) {
    D[v] = +∞
    pred[v] = -1
  }

  D[s] = 0
  for (int i = 1; i <= n - 1; i++) {
    for (each (u, v) in E) {
      if (D[u] + w(u, v) < D[v]) {
        D[v] = D[u] + w(u, v)
        pred[v] = u
      }
    }
  }
```

```
  // eventuale controllo per cicli negativi
  for (each (u,v) in E) {
    if (D[u] + w(u,v) < D[v])
      error “Il grafo contiene cicli negativi”
  }

  return D
}
```

Il **costo computazionale** è $O(nm)$.

Algoritmo di Dijkstra

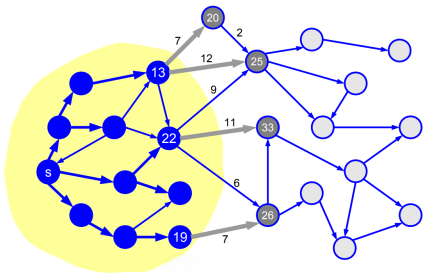
L’**algoritmo di Dijkstra** consente di risolvere il problema single-source shortest path in un grafo con archi non negativi.

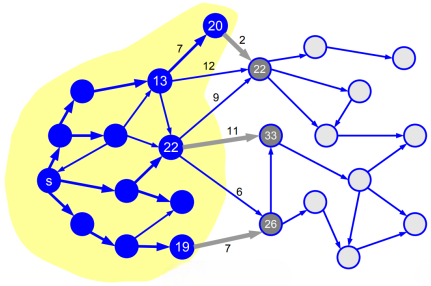
Tale algoritmo si basa sul seguente **lemma**:

> Sia G un grafo i quali archi sono ≥ 0 e sia T una parte dell’albero dei cammini di costo minimo che partono da una sorgente s, allora l’arco (u, v) che minimizza la quantità $d_{su} + w(u, v)$ appartiene al cammino minimo da s a v.

Lo **pseudocodice** dell’algoritmo è dunque il seguente:

```
double[1 ... n] dijkstra(Grafo G = (V, E, w), int s) {
  int n = G.numNodi()
  int pred[1 ... n] // array che per ogni nodo tiene conto
  // del suo nodo precedente nel cammino minimo
  double D[1 ... n]

  for (int v = 1; v <= n, v++) {
    D[v] = +∞
    pred[v] = -1
  }

  D[s] = 0
  CodaPriorita<int, double> Q
  Q.insert(s, D[s])

  while (!Q.isEmpty()) {
    u = Q.findMin()
    Q.deleteMin()

    for (each v adiacente a u) {
      if (D[v] == +∞) {
        D[v] = D[u] + w(u, v)
        Q.insert(v, D[v])
        pred[v] = u
      } else if (D[u] + w(u,v) < D[v]) {
        Q.decreaseKey(v, D[v] - D[u] - w(u, v))
        D[v] = D[u] + w(u,v)
        pred[v] = u
      }
    }
  }

  return D
}
```





Il **costo computazionale** è $O(m \log n)$

All-pairs shortest path

Algoritmo di Floyd e Warshall

L'**algoritmo di Floyd e Warshall** consente di risolvere il problema all-pairs shortest path in un grafo con archi anche negativi utilizzando la programmazione dinamica.

Tale algoritmo utilizza il concetto di D_{xy}^k, ovvero la distanza minima tra x e y, con l'ipotesi che gli eventuali nodi intermedi possano appartenere solamente all'interno di nodi $\{1, \dots, k\}$.

L'idea è quella di utilizzare un ciclo che modifichi la variabile k partendo da 1 e arrivando a n, ovvero tutti gli archi presenti nel grafo. Ad ogni passo vengono aggiornate le D_{xy}^k di ogni coppia di nodi del grafo al fine di arrivare, per ogni coppia di nodi x e y il valore di D_{xy}^n, ovvero il cammino minimo tra di essi.

All'inizio dell'algoritmo c'è un'**inizializzazione** del cammino minimo tra ogni coppia di nodi senza l'utilizzo di nodi intermedi:

$$D_{xy}^0 = \begin{cases} 0 & \text{if } x = y \\ w(x,y) & \text{if } (x,y) \in E \\ \infty & \text{if } (x,y) \notin E \end{cases}$$

Successivamente viene eseguito il ciclo di cui prima, e per calcolare il valore D_{xy}^k consideriamo i due casi seguenti:

- Il nodo k **non fa parte** del nuovo cammino minimo.

  In questo caso $D_{xy}^k = D_{xy}^{k-1}$.

- Il nodo k **fa parte** del nuovo cammino minimo.

  In questo caso $D_{xy}^k = D_{xk}^{k-1} + D_{ky}^{k-1}$.

Per scegliere se inserire o meno k nel nuovo cammino minimo occorre dunque calcolare il **minimo tra i due cammini**:

$$D_{xy}^k = min(D_{xy}^{k-1}, D_{xk}^{k-1} + D_{ky}^{k-1})$$

Lo **pseudocodice** è il seguente:

```
double[1 ... n, 1 ... n] floydWarshall(G = (V, E, w)) {
  int n = G.numNodi()
  double D[1 ... n, 1 ... n, 0 ... n] // array tridimensionale in cui
  // gli indici sono [x, y, k]

  // inizializzazione
  for (int x = 1; x <= n; x++) {
```

```
    for (int y = 1; y <= n; y++) {
      if (x == y) D[x, y, 0] = 0
      else if ((x, y) ∈ E) D[x, y, 0] = w(x, y)
      else D[x, y, 0] = +∞
  }

  // ciclo generale
  for (int k = 1; k <= n; k++) {
    for (int x = 1; x <= n; x++) {
      for (int y = 1; y <= n; y++) {
        D[x, y, k] = D[x, y, k - 1]
        if (D[x, k, k - 1] + D[k, y, k - 1] < D[x, y, k])
          D[x, y, k] = D[x, k, k - 1] + D[k, y, k - 1]
      }
    }
  }

  // eventuale controllo per cicli negativi
  for (int x = 1; x <= n; x++) {
    if (D[x, x, n] < 0)
      error “Il grafo contiene cicli negativi”
  }

  return D[1 ... n, 1 ... n, n]
}
```

Il **costo computazionale** in termini di tempo e di spazio è $O(n^3)$.

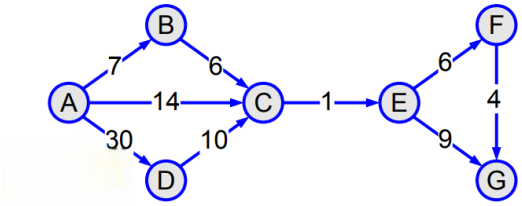
È possibile ottimizzare il costo in termini di spazio di questo algoritmo notando che una volta caricato in memoria il valore $D[x, y, k]$ per ogni coppia di nodi x e y il valore $D[x, y, k-1]$ non servirà più, dunque possiamo utilizzare una matrice bidimensionale $D[x, y]$ al posto di quella tridimensionale e calcolare $D[x, y]$ al passo k-esimo nel seguente modo:

$$D[x, y] = max(D[x, y], D[x, k] + D[k, y])$$

Lo pseudocodice è il seguente:

```
double[1 ... n, 1 ... n] floydWarshall(G = (V, E, w)) {
  int n = G.numNodi()
  double D[1 ... n, 1 ... n] // array bidimensionale in cui
  // gli indici sono [x, y]

  // inizializzazione
  for (int x = 1; x <= n; x++) {
    for (int y = 1; y <= n; y++) {
      if (x == y) D[x, y] = 0
      else if ((x, y) ∈ E) D[x, y] = w(x, y)
      else D[x, y] = +∞
  }

  // ciclo generale
  for (int k = 1; k <= n; k++) {
    for (int x = 1; x <= n; x++) {
      for (int y = 1; y <= n; y++) {
        if (D[x, k] + D[k, y] < D[x, y])
          D[x, y] = D[x, k] + D[k, y]
      }
    }
  }

  // eventuale controllo per cicli negativi
  for (int x = 1; x <= n; x++) {
    if (D[x, x] < 0)
      error “Il grafo contiene cicli negativi”
  }

  return D[1 ... n, 1 ... n, n]
}
```

Il **costo computazionale** in termini di tempo è $O(n^2)$ e in termini di spazio è $O(n^3)$.

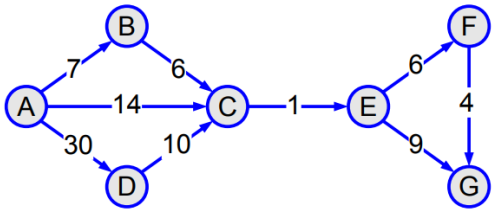


D =

	A	B	C	D	E	F	G
A	0	7	13	30	14	20	23
B	Inf	0	6	Inf	7	13	16
C	Inf	Inf	0	Inf	1	7	10
D	Inf	Inf	10	0	11	17	20
E	Inf	Inf	Inf	Inf	0	6	9
F	Inf	Inf	Inf	Inf	Inf	0	4
G	Inf	Inf	Inf	Inf	Inf	Inf	0

All-pairs shortest path tramite algoritmo di Floyd e Warshall.

Per individuare i nodi che compongono i cammini di costo minimo generati dall'algoritmo di Floyd e Warshall è possibile utilizzare una matrice $next[x, y]$ in cui, per ogni coppia di nodi x e y, viene memorizzato il nodo dopo x nel cammino minimo, in modo poi da richiamare $next[next[x, y], y]$ per trovare il nodo ancora successivo.



next =

	A	B	C	D	E	F	G
A	-1	B	B	D	B	B	B
B	-1	-1	C	-1	C	C	C
C	-1	-1	-1	-1	E	E	E
D	-1	-1	C	-1	C	C	C
E	-1	-1	-1	-1	-1	F	G
F	-1	-1	-1	-1	-1	-1	G
G	-1	-1	-1	-1	-1	-1	-1