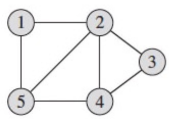
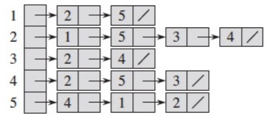
STRUTTTURE DATI
-GRAFI-

Grafi

Possono essere rappresentati sia con liste di adiacenza che con una matrice di adiacenza.
Entrambi i metodi possono essere applicati sia ai grafi orientati che non orientati

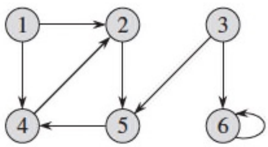


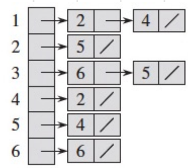


	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

Liste di adiacenza

Consiste in un array Adj di $|V|$ liste per ogni vertice in V.
Per ogni $u \in V$, la lista di adiacenza $Adj[u]$ contiene tutti i vertici v tali che esista un arco $(u,v) \in E$, oppure contiene tutti i puntatori a questi vertici

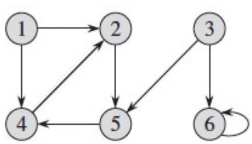




Matrice di adiacenza

La rappresentazione tramite una matrice $A=(a_{ij})$ di dimensione $|V| \times |V|$ tale che:

$$a_{ij} = \begin{cases} 1 & (i,j) \in E \\ 0 & \text{else} \end{cases}$$



	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

Tale matrice di adiacenza richiede una memoria $\Theta(V^2)$ indipendentemente dal numero di archi

Algoritmi elementari per grafi

- Breadth-First Searc (Visita in ampiezza)

Associa "colori" ai vertici

- Bianco = Vertici non ancora visitati (inizialmente tutti bianchi)
- Grigio = Vertici visitati ma non ancora esplorati (possono essere adiacenti ai vertici bianchi)
- Nero = Vertici completamente esplorati (adiacenti solo a neri e grigi)

Esplora i vertici scorrendo le liste di adiacenza di vertici grigi

```
BFS(G, s)
for ogni vertice u ∈ G.V − {s}
    u.color = WHITE
    u.d = ∞
    u.π = NIL
s.color = GRAY
s.d = 0
s.π = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
    u = DEQUEUE(Q)
    for ogni v ∈ G.Adj[u]
        if v.color == WHITE
            v.color = GRAY
            v.d = u.d + 1
            v.π = u
            ENQUEUE(Q, v)
    u.color = BLACK
```

U.Color = colore del vertice

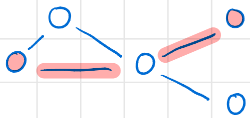
U.d = distanza dal vertice U alla sorgente s

U.π = vertice ancora da scoprire

Q = Coda con schema FiFo

Complessità in tempo: $O(V+E)$

Complessità in spazio: $O(V)$



BFS Calcola la shortest-path distance dal nodo sorgente

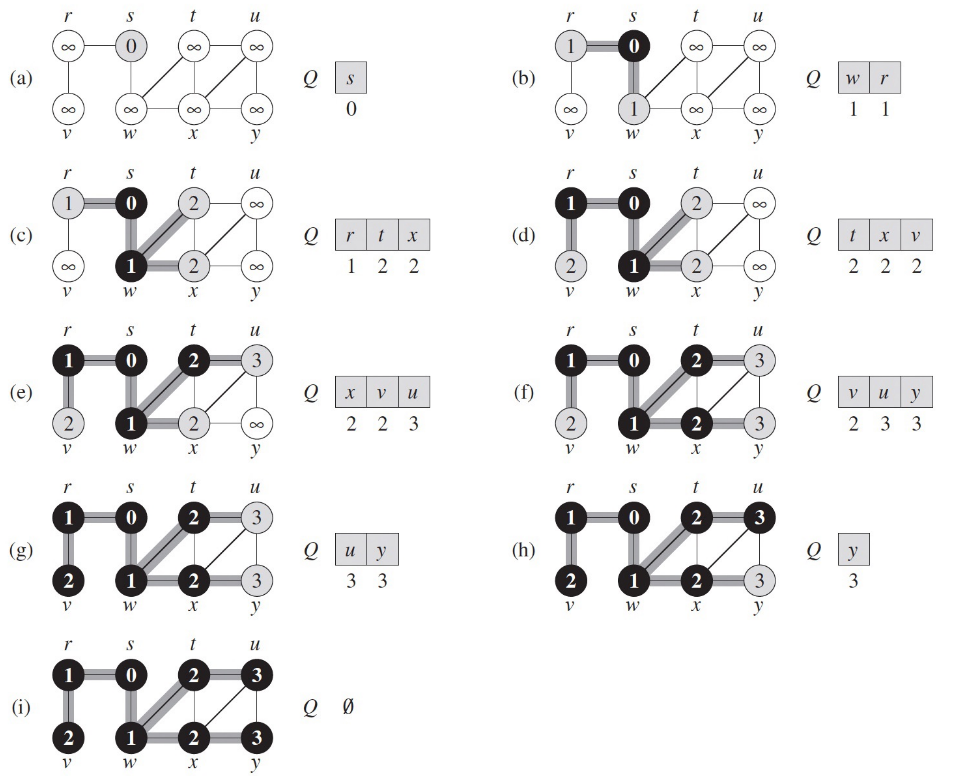
Percorso minimo è il percorso che contiene meno archi fra s e V

Distanza minima è la lunghezza del percorso minimo fra s e V

i valori delle distanze dei vertici in coda sono monotone crescenti

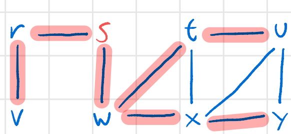
- trova sempre il cammino di lunghezza minimo
- in grafi non pesati, trova sempre il cammino di costo ottimo
- in grafi pesati non sempre trova il costo ottimo

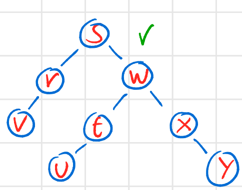
- Costruisce un albero breadth-First, dove i cammini verso la radice rappresentano i cammini minimi in G

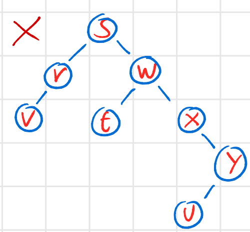


Albero breadth-First

Albero associato ad un grafo contenente tutti gli shortest-path di u messo come radice







BFS può essere usato per calcolare lo shortest-path tra nodi

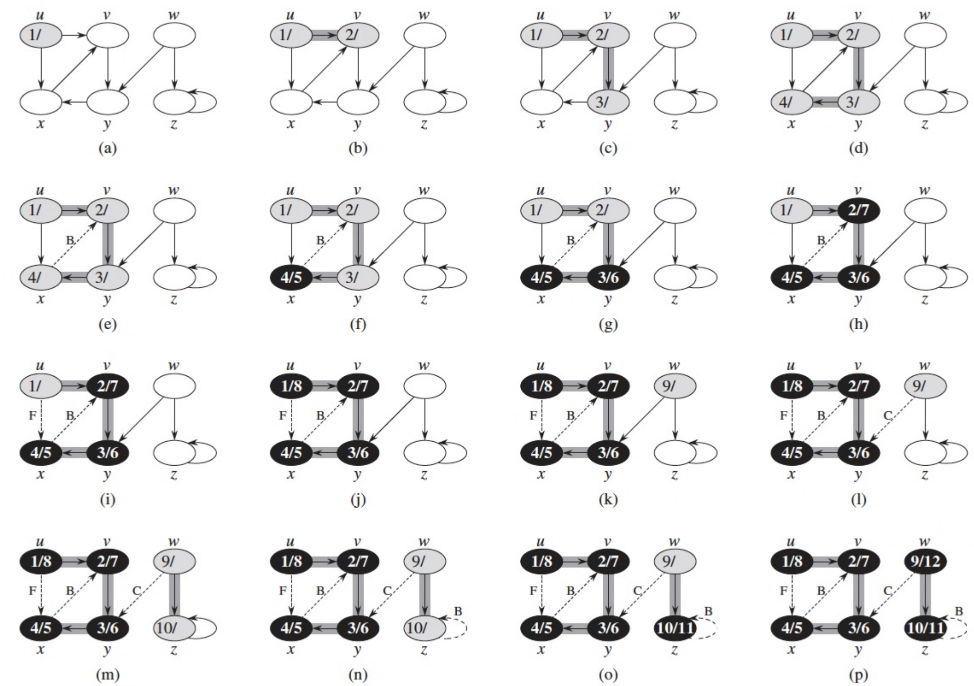
```
PRINT-PATH(G, s, v)
    if v == s
        stampa s
    elseif v.π == NIL
        stampa "non ci sono cammini da" s "a" v
    else PRINT-PATH(G, s, v.π)
        stampa v
```

Con costo temporale di $O(V+E)$

Depth-First Search

- Esplora il grafo in profondità
- Gli archi sono esplorati a partire dal nodo esplorato più di recente se ha ancora archi uscenti
- Quando ha esplorato tutti gli archi torna al vertice da cui li ha esplorati
- I nodi non devono essere raggiungibili per forza dalla sorgente

- Vertice Bianco: Inizialmente
- Vertice Grigio: Esplorato la prima volta
- Vertice Nero: Non ha più archi uscenti da visitare



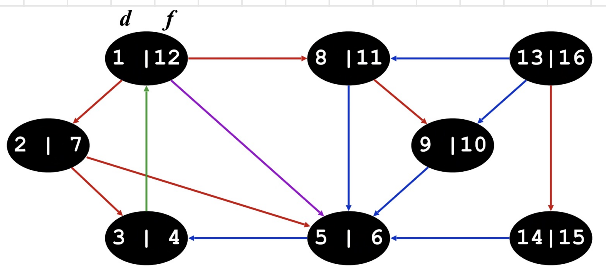

```
DFS(G)
for ogni vertice u ∈ G.V
    u.color = WHITE
    u.π = NIL
time = 0
for ogni vertice u ∈ G.V
    if u.color == WHITE
        DFS-VISIT(G, u)
```

```
DFS-VISIT(G, u)
time = time + 1              // Il vertice bianco u è stato appena scoperto.
u.d = time
u.color = GRAY
for ogni v ∈ G.Adj[u]        // Ispeziona l'arco (u, v)
    if v.color == WHITE
        v.π = u
        DFS-VISIT(G, v)
u.color = BLACK              // Colora di nero u; visita completata.
time = time + 1
u.f = time
```

Complessità temporale $= O(E+V)$

Tipologie di archi

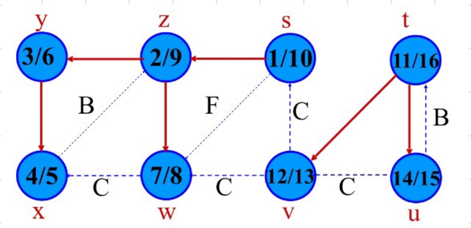
- Tree Edge: Viene incontrato un nuovo vertice
- Back Edge: Da un discendente a un antenato (chiudono un ciclo)
  - viene incontrato un vertice grigio da un nodo grigio
  - self loops
- Forward Edge: Da un antenato a un discendente
  - da un nodo grigio ad un nodo nero
- Cross Edge: gli altri



Tree edges *Back edges* *Forward edges* *Cross edges*

Strutture a parentesi

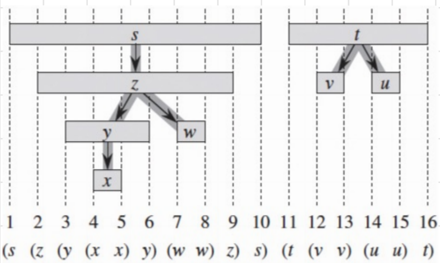
Indica i tempi di scoperta dei vertice e quindi quali vertici può raggiungere



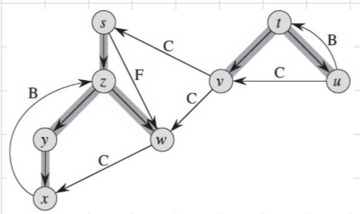
(s(z(y(x x)y)(w w)z)s)(t(v v)(u u)t)

x non ha archi uscenti

t ha 2 archi uscenti rispettivamente in v,u



Il teorema delle parentesi serve per indicare anche gli intervalli di scoperta



Topological Sort

Un ordinamento dei compiti che soddisfi i requisiti. L'obbiettivo di questo sort è trovarne uno

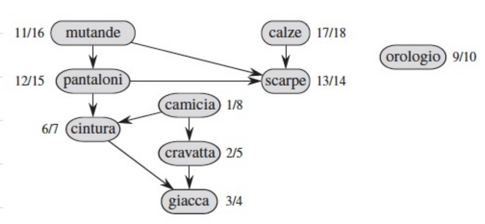
Applicazioni

- **Scheduling:** Serve per massimizzare la produttività e rispettare i vincoli di ordinamento
- **Risolvere le dipendenze:** Sequenza ammissibile con la quale un insieme di comandi può essere eseguito

Esistono più ordinamenti topologici per un singolo grafo

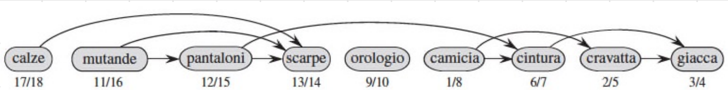
1. Esecuzione DFS per calcolare il tempo di percorrenza di ogni vertice
2. Non appena un vertice termina viene inserito nella testa della lista
3. Return lista con vertici

La lista può essere ordinata in tempo decrescente se si tiene conto del tempo di fine



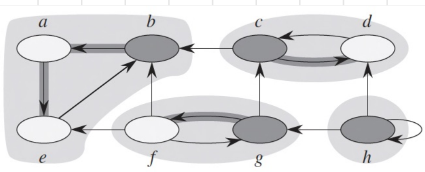
$\Theta(|E|+|V|)$ nel caso peggiore

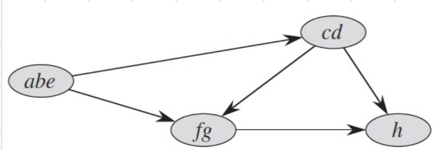
$\Theta(1)$ Inserimento nella lista



Componenti Fortemente connesse

È un insieme massimale di vertici $C \subseteq V$ tale che per ogni coppia di vertici si ha sia un arco entrante e sia un arco uscente. Il seguente algoritmo inizia effettuando questa scomposizione e finisce combinando le coppie





STRONGLY-CONNECTED-COMPONENTS(G)

1 Chiama DFS(G) per calcolare i tempi di completamento $u.f$ per ciascun vertice u

2 Calcola G^T

3 Chiama DFS(G^T), ma nel ciclo principale di DFS, considera i vertici in ordine decrescente rispetto ai tempi $u.f$ (calcolati nella riga 1)

4 Genera l'output dei vertici di ciascun albero della foresta DF che è stata prodotta nella riga 3 come una singola componente fortemente connessa

$G^T = (V, E^T)$ dove $E^T = \{(u,v) : (v,u) \in E\}$

ovvero E^T è formato dagli archi con le direzioni invertite

- il tempo richiesto per creare G^T è di $O(V+E)$

Algoritmo di Dijkstra

Algoritmo valido per trovare il percorso con costo minimo a partire da un vertice, valido per pesi positivi.

- Sceglie il vertice da cui partire
- Prende una lista Q in cui memorizzo i percorsi finali dei vertici e il peso

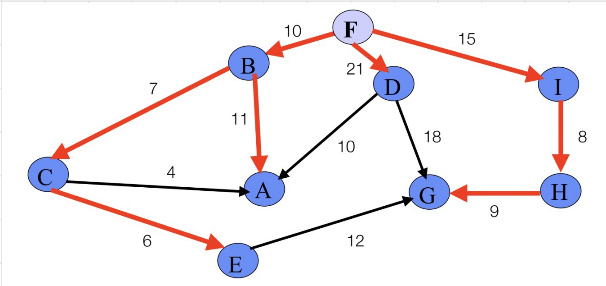
$Q = (B, nil, \infty) \xrightarrow{\text{estraggo } B} Q = (B, F, 10)$ → Costo distanza tra B e F

Costo iniziale

Predecessore inizializzato con nil

Nodo estratto

→ Nodo predecessore adiacente



Q=(F,nil,0),(B,F,10),(D,F,21)
(I,F,15),(C,B,17),(A,B,21)
(E,C,23),(H,I,23),(G,H,32)

```
function Dijkstra(G, s) {
  Q = empty vertex priority queue;
  for each v in G->V {
    if (v == s) dist[v] = 0
    else dist[v] = infinity;
    prev[v] = nil;
    add v to Q with priority dist[v]; | O(log |V|)
  }
  while (Q != empty) {
    u = vertex with min priority in Q; | O(log |V|)
    for each v in u.Adj[] {
      alt = dist[u] + weight(u,v);
      if (alt < dist[v]) {
        dist[v] = alt;
        prev[v] = u;
        decrease_priority(Q,v,alt) | O(log |V|)
      }
    }
  }
  return dist[], prev[]
}
```

O(|V|) — O(|V|log|V|)

O(|V|) — O(|E|) — O(|V|log|V| + |E|log|V|)

O((|V|+|E|)log|V|)

verificate se usate un array al posto della min heap la complessità è O(|V|^2 + |E|) = O(|V|^2)

Processo di rilassamento

Verifica se è possibile migliorare il cammino minimo e aggiornarlo

L'algoritmo per memorizzare i percorsi in una lista usa una coda di priorità con un min-heap

Coda di priorità

Struttura dati che mantiene un insieme dove ad ogni elemento è associata una chiave

Min-proprietà

- $O(\log n)$ - INSERT(S, x) = Inserisce l'elemento x nell'insieme S. $S = S \cup \{x\}$
- $\Theta(1)$ - MINIMUM(S) = Restituisce l'elemento con chiave minima in S
- $O(\log n)$ - EXTRACT-MIN(S) = Rimuove e restituisce l'elemento con chiave minima in S
- $O(\log n)$ - DECREASE-KEYS(S, x, K) = Diminuisce il valore della chiave dell'elemento x al nuovo valore K

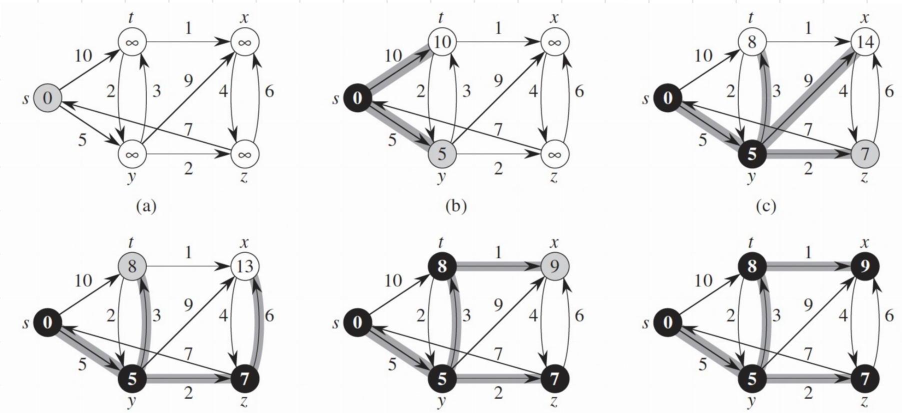
```
DIJKSTRA(G, w, s)
INITIALIZE-SINGLE-SOURCE(G, s)
S = ∅
Q = G.V
while Q ≠ ∅
    u = EXTRACT-MIN(Q)
    S = S ∪ {u}
    for ogni vertice v ∈ G.Adj[u]
        RELAX(u, v, w)
```

```
INITIALIZE-SINGLE-SOURCE(G, s)
for ogni vertice v ∈ G.V
    v.d = ∞
    v.π = NIL
s.d = 0
```

$O(V)$

```
RELAX(u, v, w)
if v.d > u.d + w(u, v)
    v.d = u.d + w(u, v)
    v.π = u
```

$O(1)$



(a) (b) (c) (d) (e) (f)

Nei passaggi c e d effettuo il processo di rilassamento cercando un percorso migliore

Algoritmo di Bellman-Ford

Risolve il problema dei cammini minimi anche con archi di peso negativo.
L'algoritmo prende in input un grafo $G=(V,E) \wedge w: E \to \mathbb{R}$ e restituisce un booleano che segnala se esiste o no un ciclo di peso negativo che è raggiungibile dalla sorgente, se esiste indica che il problema non ha soluzione altrimenti restituisce lo shortest-path tra nodo sorgente e il resto dei nodi con i rispettivi pesi

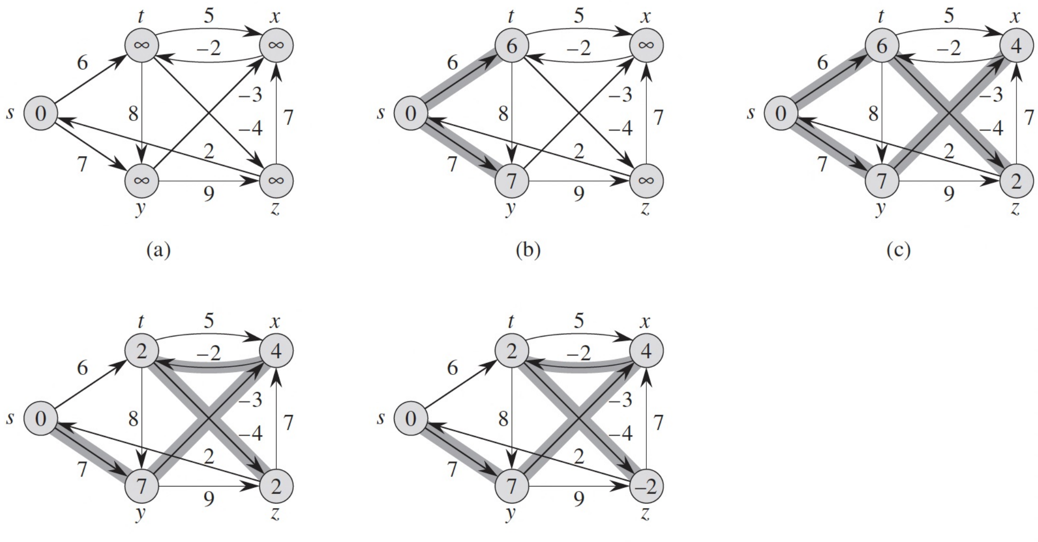
```
BELLMAN-FORD(G, w, s)
INITIALIZE-SINGLE-SOURCE(G, s)
for i = 1 to |G.V| − 1
    for ogni arco (u, v) ∈ G.E
        RELAX(u, v, w)
for ogni arco (u, v) ∈ G.E
    if v.d > u.d + w(u, v)
        return FALSE
return TRUE
```

Restituisce true $\iff$ il grafo non contiene cicli di peso negativo che partono dalla sorgente, False altrimenti

$O(V \cdot E)$

$\Theta(V^2)$

Nella prima iterazione effettua il rilassamento per ogni arco.
La seconda iterazione controlla se esiste un ciclo con peso negativo



(a) (b) (c) (d) (e)

Algoritmi vari su grafi

- Determinare se è un DAG

```
function isDAG(G) -> Bool {
  DFS(G);
  for each (u,v) in G->E {
    if (u.f <= v.f) return false
  }
  return true
}
```

O(|V|+|E|)
O(|E|)
O(|V|+|E|)

- Contare le tipologie di archi

```
function contaEdges(G) -> [Int] {
  var ris:[Int] = [0,0,0,0];
  DFS_conta(G);
  for each (u,v) in G->E {
    if ((u,v) == TREE) ris[0]++
    else if (u.f <= v.f) ris[1]++
    else if (u.d < v.d) && (v.f < u.f) ris[2]++
    else if (v.f < u.d) ris[3]++
  }
  return ris
}
```

O(|V|+|E|)
O(|E|)
O(|V|+|E|)

- Classificare le tipologie di archi

```
function classificaEdges(G) {
  DFS_conta(G);
  for each (u,v) in G->E {
    if ((u,v) != TREE) {
       if (u.f <= v.f) (u,v) = BACK
       else if (u.d < v.d) && (v.f < u.f) (u,v) = FORWARD
       else if (v.f < u.d) (u,v) = CROSS
    }
  }
  return ris
}
```

O(|V|+|E|)
O(|E|)
O(|V|+|E|)

- Verificare se è Connesso

```
function isConnessoOrientato(G) -> Bool {
  converti(G);
  return isConnesso(G)
}


function converti(G) -> G {
  for each u in G->V {
    for each v in u->adj[] {
      add u to v->adj[]
    }
  }
  return G
}
```

- verificare se è Fortemente Connesso

```
function isFortementeConnesso(G) -> Bool {
 for each v in G-> V {
  esplorati = 0;
  vertici = 0;
  for each v in G->V {
    dist[v] = infinito;
    vertici++
  }
  Q = {v};
  dist[v] = 0;
  while (Q non vuota) {
      u = removeTop(Q);
      esplorati++;
      for each v in u->adj {
        if (dist[v] == infinito) {
           dist[v] = dist[u]+1;
           enqueue(Q, v);
        }
      }
  }
  if (vertici != esplorati) return false;
 }
 return true
}
```

- Diametro di un grafo

```
function Diametro(G) -> Int {
  max = 0;
  for each v in G->V {
    for each v in G->V {
      dist[v] = -1;
    }
    Q = {v};
    dist[v] = 0;
    while (Q non vuota) {
      u = removeTop(Q);
      for each v in u->adj {
        if (dist[v] < 0) {
           dist[v] = dist[u] + 1;
           if (dist[v] > max) max = dist[v];
           enqueue(Q, v)
        }
      }
    }
  }
  return max
}
```

- Verificare se è bipartito

```
function isBipartito(G,s) -> Bool {
  for each u in G->V {
    u->d = false;
  }
  for each u in G->V {
    if (!u->d) {
       u->d = true;
       u->color = 0;
       if !DFS_Color(u) return false
    }
  }
  return true
}
```

```
DFS_Color(u) -> Bool {
    for each v ∈u->Adj[] {
        if (!v->d) {
           v->color = !u->color;
           v->d = true;
           if !DFS_Color(v) return false;
        }
        else if (v->color == u->color) return false
    }
    return true
}
```

0

- Distanza massima fra un nodo e una sorgente

```
function maxDist(G, s) -> Int {
  BFS(G, s);
  max = 0;
  for each v in G->V {
    if (dist[v] > max) {
      max = dist[v]
    }
  }
  return max
}
```

O(|V|+|E|)

O(|V|)

O(|V|+|E|)

- Determinare se un nodo y si trova lungo il percorso fra x e z

```
percorso(G, x, y, z) -> Bool {
     if (x == y && y == z) return true
     else if (x == y) return BFS_percorso(G, x, z)
     else if (y == z) return BFS_percorso(G, x, y)
     else return BFS_percorso(G, x, y) && BFS_percorso(G, y, z)
}
```

```
BFS_percorso(G, s, w) -> Bool {
    inizializza vertici;
    Q = {s};
    while (Q non vuota) {
        u = RemoveTop(Q);
        for each v ∈u->adj {
            if (v->d == infinito)
                if (v == w) return true
                v->d = u->d + 1;
                v->p = u;
                Enqueue(Q, v);
        }
    }
     return false
}
```

- Determinare il numero di vertici che si trovano a distanza massima da una sorgente

```
function maxDist(G, s) -> Int {
  BFS(G, s);                              | O(|V|+|E|)
  max = 0;
  count = 1;
  for each v in G->V {                    | O(|V|)         | O(|V|+|E|)
    if (dist[v] > max)
      max = dist[v] {
      count = 1;
    }
    else if (dist[v] == max) {
      count++;
    }
  }
  return count
}
```

- Determinare se contiene cicli - iterativamente

```
haCicli(G) > Bool {
   for each vertex u ∈G->V {
     u->color = BIANCO;
   }
   for each vertex u ∈G->V {
     if (u->color == BIANCO)
         if haCicliRec(G,u,nil) return true
   }
   return false
}
```

- Determinare se contiene cicli - ricorsivamente

```
haCicliRec(G,u,w) {
  u->colore = GRIGIO;
  for each v in u->Adj[]-w {
    if (v == GRIGIO) return true
    else if haCicliRec(G,v,u)
        return true
  }
  return false
}
```