

## 9 Grafi

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

---

Il GRAPH è una struttura dati dinamica che permette di aggiungere e rimuovere nodi ed archi.

```
Graph
SET V
int size
SET adj(NODE u)
insertNode(NODE u)
deleteNode(NODE u)
insertEdge(NODE u, NODE v)
deleteEdge(NODE u, NODE v)
```

// crea un grafo vuoto  
// restituisce l'insieme di tutti i nodi  
// restituisce il numero di nodi  
// restituisce l'insieme di NODE adiacenti a u  
// aggiunge il nodo u al grafo  
// rimuove il nodo u dal grafo  
// aggiunge l'arco (u,v) al grafo  
// rimuove l'arco (u,v) al grafo

---

```
visita(GRAPH G, NODE r)
SET S ← Set // insieme di nodi (STACK, QUEUE)
S.insert(r) // inserisco il nodo
// ho visitato il nodo
{ marca il nodo r come "scoperto" }
```

---

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

---

```

// visitare tutti i nodi a distanza k perima di visitare i nodi a distanza k+1
bfs(GRAPH G, NODE r)
    QUEUE S ← Queue
    S.enqueue(r) // inserisco la radice

    // inizializzazione
    bool[] visitato ← bool[1...G.n]
    per ciascun  $u \in G.V - \{r\}$  fai visitato[ $u$ ] ← falso

    visitato[r] ← vero // radice visitata

    // visita del grafo
    finché not S.isEmpty fai
        NODE u ← S.dequeue
        { esamina il nodo  $u$  }

        per ciascun  $u \in G.adj(u)$  fai
            { esamina l'arco  $(u, v)$  }
            se not visitato[v] allora
                visitato[v] ← vero
                S.enqueue

```

---

---

```

// il cammino più breve fra due vertici viene memorizzato tramite il vettore dei padri p
erdos(GRAPH G, NODE r, int[] erdos, NODE() parent)
    QUEUE S ← Queue
    S.enqueue(r) // inserisco la radice

    // inizializzazione
    bool[] visitato ← bool[1...G.n]
    per ciascun  $u \in G.V - \{r\}$  fai  $erdos[u] \leftarrow \infty$ 
     $erdos[r] \leftarrow \text{vero}$  // erdos ha distanza 0 da se stesso
    parent[r] ← nil

    finché not S.isEmpty fai
        NODE u ← S.dequeue

        per ciascun  $u \in G.adj(u)$  fai
            { esamina l'arco  $(u, v)$  }
            se  $erdos[v] \leftarrow \infty$  allora
                // il nodo non è stato ancora scoperto
                 $erdos[v] \leftarrow erdos[u] + 1$  // gli assegno un livello di erdos+1
                parent[v] ← u // memorizzo il figlio nel vettore dei padri
                S.enqueue(v) // è la prima volta che lo raggiungo quindi lo metto in coda

```

---

```

// stampa il cammino da r a s nell'ordine corretto
stampaCammino(GRAPH G, NODE r, NODE s, NODE[] parent)
    se  $r == s$  allora
        stampa s
    se  $parent[s] == \text{nil}$  allora
        stampa "nessun cammino da r a s"
    altrimenti
        // la chiamata ricorsiva prima della stampa per stampare in ordine
        stampaCammino(G, r, parent[s], p)
        stampa s

```

---

```

// genera un albero breadth-first
dfs(GRAPH G, NODE u, bool[] visitato)
    visitato[u] = vero
    { esamina il nodo u (caso previsita) }

    per ciascun  $u \in G.adj(u)$  fai
        { esamina l'arco  $(u, v)$  }
        se not visitato[u] allora
            // chiamata ricorsiva
            DFS(G, v, visitato)

(1) { esamina il nodo u (caso postvisita) }

(2) { esamina il nodo u (caso postvisita) }

```

---

**Analisi della complessità** Questo algoritmo ha complessità  $\mathcal{O}(n + m)$  con il grafo implementato con liste di adiacenza, e di  $\mathcal{O}(n^2)$  con matrice di adiacenza.

---

// Algoritmo iterativo, STACK esplicito, pre-order 47/101

```
dfsStack(GRAPH G, NODE r)
    STACK S ← Stack
    S.push

    // inizializzazione
    bool[ ] visitato ← bool[1...G.n]
    per ciascun  $u \in G.V - \{r\}$  fai visitato[ $u$ ] ← falso

    visitato[r] ← vero // radice visitata
    finché not S.isEmpty fai
        NODE u ← S.pop
        se not visitato[v] allora
            { esamina il nodo u (preordine) }
            ...
    
```

---

// identifica le componenti connesse di un grafo non orientato

```
int[ ] cc(GRAPH G, STACK S)
    int[ ] id = new int[1...G.n]

    per ciascun  $u \in G.V$  fai
        id[u] = 0
    int counter = 0
    per ciascun  $u \in G.V$  fai
        // per ogni nodo del grafo
        se id[u]==0 allora
            // ho trovato una nuova componente connessa
            counter++
            // effettuo una chiamata ricorsiva sul nodo trovato
            ccdfs(G, counter, u, id)

    ritorna id

// visita ricorsiva
ccdfs(GRAPH G, int counter, NODE u, int[ ] id)
    // counter: identificatore di quante cc ho trovato fin'ora
    // u: il nodo che sto visitando
    // assegno un contatore ad ogni nodo della cc
    id[v] = counter
    per ciascun  $v \in G.adj(u)$  fai
        se id[v]==0 allora
            // non è ancora stato visitato
            ccdfs(G, counter, v, id) // v: il nodo in cui vado a operare
    
```

---

## 9.1 Grafi non pesati

### 9.1.1 Applicazioni dfs: grafo *non* orientato aciclico

---

```
// ricerca di un ciclo per grafi disconnessi
bool ciclico(GRAPH G)
    // inizializzazione
    bool[] visited ← new bool[1...G.n]
    per ciascun  $u \in G.V$  fai  $visited[u] \leftarrow$  falso

    per ciascun  $u \in G.V$  fai
        se not  $visited[u]$  allora
            // non ho ancora visitato il nodo
            se ciclicoRec( $G, v, null, visited$ ) allora
                // effettuo una visita ricorsiva sul nodo vicino  $v$ 
                ritorna vero
        ritorna falso

// ricerca di un ciclo all'interno di un grafo
bool ciclicoRec(GRAPH G, NODE v, NODE p, bool[] visited)
    //  $p$ : nodo da cui provengo
     $visited[v] \leftarrow$  vero // visitato per la prima volta
    per ciascun  $v \in G.adj(u) - \{p\}$  fai
        //  $G.adj(u) - \{p\}$ : non considero  $(u, v)$  un ciclo nei grafi non orientati
        // per ogni nodo visito i suoi vicini
        se  $visited[v]$  allora
            // se il nodo è già stato visitato allora ho trovato un ciclo
            ritorna vero
        altrimenti se ciclicoRec( $G, v, u, visited$ ) allora
            // effettuo una visita ricorsiva sul nodo vicino  $v$ 
            ritorna vero
    ritorna falso
```

---

---

```

dfs-schema(GRAPH G, NODE u, &time)
    esamina il nodo u (caso previsita)
    time++
    dt[u] ← time // tempo di scoperta
    per ciascun  $u \in G.\text{adj}(u)$  fai
        esamina l'arco  $(u, v)$  di qualsiasi tipo
        se  $dt[v] == 0$  allora
            esamina l'arco  $(u, v)$  nell'albero  $T$ 
            // chiamata ricorsiva
            dfs-schema(g, v)
        altrimenti se  $dt[u] < dt[v] \text{ and } ft[v] == 0$  allora
            // se raggiungo un mio discendente e non ho ancora terminato la mia visita,
            // allora ho trovato un arco all'indietro
            esamina l'arco  $(u, v)$  all'indietro
            se  $dt[u] < dt[v] \text{ and } ft[v] \neq 0$  allora
                // se raggiungo un mio discendente e non ho ancora terminato la mia visita,
                // allora ho trovato un arco in avanti
                esamina l'arco  $(u, v)$  in avanti
            altrimenti
                esamina l'arco  $(u, v)$  di attraversamento
        esamina il nodo u (caso postvisita)
        time++
        ft[u] ← time // tempo di fine

```

---

**Definizione 9.1** (grafo diretti orientati aciclici). *Un grafo è DAG quando*

---

```
STACK topSort(GRAPH G)
    // inizializzazione
    bool[] visited ← new bool[1...G.n]
    per ciascun  $u \in G.V$  fai  $visited[u] \leftarrow$  falso

    per ciascun  $u \in G.V$  fai
        // per ogni nodo del grafo
        se not  $visited[u]$  allora
            // se non l'hai visitato
            // effettua una chiamata ricorsiva
            ts-dfs( $G, u, visited, S$ )
    ritorna STACK

    // restituisce l'ordinamento topologico dei nodi di un DAG
    // questo algoritmo funziona partendo da qualsiasi nodo
int ts-dfs(GRAPH G, NODE  $u$ , bool[]  $visited$ , NODE[] STACK  $S$ )
     $visited[u] \leftarrow$  vero // imposta il nodo come visitato
    per ciascun  $v \in G.adj(u)$  fai
        // è un grafo diretto aciclico quindi non ho bisogno di ricordare da dove sono
        // venuto
        se not  $visited[v]$  allora
            // effettua una visita in profondità
             $i \leftarrow$  ts-dfs( $G, u, visited, S$ )
     $S.push$  // aggiungi il nodo in testa alla pila
    // quando ho terminato tutte le chiamate ricorsive l'algoritmo mi restituirà l'ordine
    // topologico dei nodi del grafo dato in input, il nodo viene messo in testa in modo tale
    // che si trovi prima dei nodi che i suoi archi puntano, ossia i suoi discendenti
```

---

---

```

// applicabile solo ai DAG, in quanto non hanno archi all'indietro
bool ciclico(GRAPH G, NODE u)
    // u: il primo nodo che viene visitato
    time++
    dt[u] ← time // tempo di scoperta
    per ciascun u ∈ G.adj(u) fai
        se dt[v]==0 allora
            // non ho ancora scoperto questo nodo
            // effettuo una visita ricorsiva
            se ciclico(G,v) allora
                ritorna vero
        altrimenti se dt[u] < dt[v] and ft[v]≠0 allora
            // se raggiungo un mio discendente e non ho ancora terminato la mia visita,
            // allora ho trovato un arco all'indietro e quindi un ciclo
            ritorna vero
    time++
    ft[u] ← time // tempo di fine
    // non ho trovato un ciclo
    ritorna falso

```

---

## 9.2 Componenti fortemente connesse

**Definizione 9.2** (grafo fortemente connesso). *Un grafo orientato  $G = (V, E)$  è **fortemente connesso** sse ogni suo nodo è raggiungibile da ogni altro suo nodo.*

**Definizione 9.3** (componente fortemente connessa). *Un grafo  $G' = (V', E')$  è una **componente fortemente connessa** di  $G$  sse  $G'$  è un sottografo connesso e massimale di  $G$ .*

### 9.3 Algoritmo di Kosaraju

---

```
// effettua una visita in profondità del grafo, esaminando i nodi nell'ordine inverso di tempo  
di fine della prima visita.  
int[] scc(GRAPH G)  
    STACK S ← topSort(G) // otteniamo i nodi in ordine decrescente di fine  
     $G^T \leftarrow \text{transpose}(G)$  // inverte il senso degli archi  
    ritorna cc( $G^T$ , S) // esegue una visita dfs sul grafo trasposto  
  
// parte iterativa  
// cc modificato in 88/101  
int[] cc(GRAPH G, STACK S)  
    int[] id ← new int[1...G.size]  
    per ciascun  $u \in G.V$  fai  
        id[u] ← 0  
    int counter ← 0  
    finché not S.isEmpty fai  
        u ← S.pop // estrazione in tempo inverso di tempo di fine  
        se id[u]==0 allora  
            counter++  
            ccdfs(G, counter, n, id)  
    ritorna id  
  
// parte ricorsiva  
ccdfs(GRAPH G, int counter, NODE n, int[] id)  
    id[u] ← counter // assegna il contatore a tutti i nodi che incontro  
    per ciascun  $u \in G.\text{adj}(u)$  fai  
        se id[u]==0 allora  
            ccdfs(G, counter, u, id)
```

---