

Algoritmi e Strutture Dati

Schifano S. Fabio `schifano@fe.infn.it`

Grafi Orientati

Un **grafo orientato** o **grafo diretto** é una coppia $G = (N, A)$, con N insieme finito di elementi detti **nodi** ed A insieme finito di coppie ordinate di nodi detti **archi**.

Esempio: $G = (N, A)$, $N = \{1, 2, 3, 4\}$, $A = \{(2, 1), (1, 3), (3, 1), \dots\}$

Essendo gli archi **coppie orientate** di nodi, per ogni coppia di nodi i, j esistono al piú due archi distinti: l'arco (i, j) e l'arco (j, i) .

In un grafo orientato $G = (N, A)$, $|N| = n$, un **cammino** é una sequenza di nodi u_0, u_1, \dots, u_k tali che per ogni coppia $u_i, u_{i+1} \in N$ esista un arco appartenente ad A che li unisca.

In un grafo G :

- ▷ la lunghezza di un cammino si definisce come il numero di archi attraversati dal cammino
- ▷ se nel cammino non ci sono nodi ripetuti allora il cammino é detto **semplice**
- ▷ se $u_0 = u_k$ allora il cammino é detto **chiuso**

Un cammino **semplice** e **chiuso** é detto **ciclo**

Un grafo G é detto **fortemente connesso** se per ogni coppia di nodi u e v esiste almeno un cammino da u a v e viceversa.

Grafi non Orientati

Se gli archi del grafo G non sono coppie ordinate, allora il grafo é detto **non orientato**.

In particolare, se il grafo **non é orientato** esiste al piú un solo arco $[i, j]$ che li unisce.

In un grafo non orientato G una sequenza di nodi u_0, u_1, \dots, u_k , tali che per ogni coppia u_i, u_{i+1} esista un arco appartenente di A che li unisca, é detta **catena**.

Se non ci sono nodi ripetuti allora la catena é **semplice**.

Se $u_0 = u_k$ la catena é detta **chiusa**.

Una catena **semplice** e **chiusa** é detta **circuito**.

Un grafo non orientato é detto **connesso** se per coppia di nodi u e v esista una catena da u a v .

Un grafo non orientato puó essere rappresentato mediante un grafo orientato in cui per ogni arco $[u, v]$ del grafo non orientato siano presenti entrambi gli archi (u, v) e (v, u) del grafo orientato.

Relazioni tra Grafi ed Alberi

Sotto opportune condizioni, la definizione di grafo coincide con quella di albero.

Un grafo non orientato G é un **albero**, detto **albero libero**, se il grafo é **connesso** e per ogni coppia di nodi esista una ed una sola catena semplice che li unisca.

La precedente definizione equivale ad una delle seguenti definizioni:

- ▷ il grafo é connesso e il numero di archi é minimo, cioè uguale al numero dei nodi meno uno,
- ▷ il grafo é connesso e non ci sono circuiti

In particolare:

- ▷ dato un albero libero, possiamo ottenere un **albero radicato** stabilendo un ordinamento **verticale** tra i nodi. In particolare, un nodo viene designato come **radice** ed i rimanenti sono ordinati per livelli, ovvero in base alla distanza dalla radice.
- ▷ dato un albero libero possiamo ricavare n alberi radicati distinti, ognuno dei quali é ottenuto designando un nodo diverso come nodo radice.
- ▷ dato un albero radicato possiamo ottenere un **albero ordinato** stabilendo un ordinamento **orizzontale** tra i nodi che sono allo stesso livello.

Grafi: Specifica

Poichè un grafo non orientato può essere rappresentato mediante un grafo orientato, definiamo la specifica solamente dei grafi orientati.

- ▷ CREAGRAFO: $() \longrightarrow \text{grafo}$
 $\text{CREAGRAFO}() = G$
Post: $G = (N, A)$, $N = \phi$ e $A = \phi$
- ▷ GRAFOVUOTO: $(\text{grafo}) \longrightarrow \text{boolean}$
 $\text{GRAFOVUOTO}(G) = b$
Post: $b = T$ se $N = \phi$ e $A = \phi$, $b = F$ altrimenti
- ▷ INSNODO: $(\text{nodo}, \text{grafo}) \longrightarrow \text{grafo}$
 $\text{INSNODO}(u, G) = G'$
Pre: $G = (N, A)$, $u \notin N$
Post: $G' = (N', A)$, $N' = N \cup \{u\}$
- ▷ INSARCO: $(\text{nodo}, \text{nodo}, \text{grafo}) \longrightarrow \text{grafo}$
 $\text{INSARCO}(u, v, G) = G'$
Pre: $G = (N, A)$, $u \in N$, $v \in N$, $(u, v) \notin A$
Post: $G' = (N, A')$, $A' = A \cup \{(u, v)\}$

Grafi: Specifica

- ▷ CANCNODO: (nodo, grafo) \longrightarrow grafo

$$\text{CANCNODO}(u, G) = G'$$

Pre: $G = (N, A), u \in N, \nexists v \in N | (u, v) \in A \text{ oppure } (v, u) \in A$

Post: $G' = (N', A), N' = N - \{u\}$

CANCNODO cancella un nodo **isolato**, cioè che non ha archi incidenti

- ▷ CANCARCO: (nodo, nodo, grafo) \longrightarrow grafo

$$\text{CANCARCO}(u, v, G) = G'$$

Pre: $G = (N, A), u, v \in N, (u, v) \in A$

Post: $G' = (N, A'), A' = A - \{(u, v)\}$

CANCARCO cancella un arco esistente tra una coppia di nodi

- ▷ ADIACENTI: (u, G) = A(u)

Pre: $G = (N, A), u \in N$

Post: $A(u) = \{v | v \in N, (u, v) \in A\}$

ADIACENTI restituisce l'insieme di tutti i nodi v raggiungibili da u con un arco (u, v) .

Grafi: Realizzazione con Matrice di Incidenza

Un grafo può essere rappresentato in memoria utilizzando una matrice detta matrice di **incidenza nodi-archi**.

Dato un grafo orientato $G = (N, A)$, con $|N| = n$ e $|A| = m$, e supponiamo di numerare gli archi con indici $0 \dots m - 1$.

La matrice di **incidenza nodi-archi** B é una matrice rettangolare $n \times m$, ove gli elementi sono definiti nel seguente modo:

$$b_{ik} = \begin{cases} -1 & \text{se l'arco } k\text{-esimo esce dal nodo } i \\ 0 & \text{se l'arco } k\text{-esimo non incide sul nodo } i \\ +1 & \text{se l'arco } k\text{-esimo entra nel nodo } i \end{cases}$$

Se il grafo non é orientato allora la matrice di incidenza é definita nel seguente modo:

$$b_{ik} = \begin{cases} 1 & \text{se l'arco } k\text{-esimo incide sul nodo } i \\ 0 & \text{se l'arco } k\text{-esimo non incide sul nodo } i \end{cases}$$

Grafi: Realizzazione con Matrice di Incidenza

Considerazioni sull'implementazione dei grafi tramite matrice di incidenza:

- ▶ lo spazio di memoria occupato é sempre $\Theta(nm)$, anche per grafi con pochi archi, ad esempio $m \in \mathcal{O}(n)$, causando un inefficiente uso della memoria
- ▶ la matrice di incidenza é utile per determinare la **stella uscente** o **entrante** di un nodo, cioè l'insieme degli archi che incidono sul nodo.

L'algoritmo che calcola la stella, uscente o entrante, di un nodo u deve scorrere la riga u della matrice ed individuare gli elementi diversi da zero.

Quindi, la complessità dell'algoritmo che calcola la stella uscente è $\Theta(m)$.

- ▶ utilizzando la rappresentazione tramite matrice di incidenza, non é agevole, dato un nodo u , determinare l'**insieme di adiacenza** $A(u)$, cioè l'insieme dei nodi raggiungibili da u con un solo arco.
L'algoritmo che calcola $A(u)$ deve scorrere la riga u della matrice, e per ogni arco uscente k deve scorrere la colonna k per individuare un nodo v per cui l'arco k risulta entrante.

Quindi, la complessità dell'algoritmo per il calcolo dell'insieme di adiacenza di un nodo è $\Theta(mn)$.

Esercizio: scrivere un algoritmo che calcola $A(u)$ supponendo che il grafo sia rappresentato tramite matrice di incidenza.

Grafi: Realizzazione con Matrice di Adiacenza

Dato un grafo orientato $G = (N, A)$, con $|N| = n$ e $|A| = m$, la matrice di **adiacenza nodi-nodi** E é una matrice quadrata $n \times n$ ove gli elementi sono definiti nel seguente modo:

$$e_{ij} = \begin{cases} 1 & \text{se l'arco } (i, j) \in A \\ 0 & \text{se l'arco } (i, j) \notin A \end{cases}$$

Se il grafo non é orientato, allora la matrice é **simmetrica**, cioè $e_{ij} = e_{ji}$

Considerazioni sull'implementazione dei grafi tramite matrice di adiacenza:

- ▷ l'appartenenza di un arco (i, j) al grafo si verifica in $\mathcal{O}(1)$, basta verificare se $e_{ij} = 1$
- ▷ la complessità per determinare l'insieme di adiacenza di un nodo u é sempre $\Theta(n)$. Infatti bisogna scandire sempre tutta la riga di indice u della matrice.
- ▷ lo spazio di memoria occupato é sempre $\Theta(n^2)$, anche per grafi **sparsi**, cioè se il numero degli archi m é dello stesso ordine di grandezza del numero dei nodi, cioè $m \in \mathcal{O}(n)$.
- ▷ il tempo per esaminare tutti gli archi é sempre $\Theta(n^2)$ anche per grafi sparsi.

Se ad ogni arco (i, j) del grafo é associato un **costo** o **peso**, allora gli elementi della matrice di adiacenza contengono i valori di tali costi o pesi al posto degli elementi binari 1 e 0.

Per gli archi mancati il costo o peso é pari a ∞ o $-\infty$ a seconda dell'uso che bisogna farne.

Grafi: Realizzazione con liste di insiemi di adiacenza

Al fine di ottimizzare l'occupazione di spazio per memorizzare un grafo, si utilizza una realizzazione del grafo tramite **insiemi di adiacenza**.

Si usa un vettore di dimensione $n = |N|$, ed ogni elemento i del vettore é un riferimento ad lista che contiene l'insieme di adiacenze $A(i)$ del nodo associato alla posizione i del vettore.

```
#define n |N|  
  
typedef lista insiemeadiacenza[n] grafo
```

Con questa implementazione:

- ▷ lo spazio di memoria occupato é $\Theta(n + \sum_{i \in N} |A(i)|) = \Theta(n + m)$
- ▷ se il grafo é sparso l'occupazione, cioè $m \in \mathcal{O}(n)$, l'occupazione di memoria é $\Theta(n + n) = \Theta(n)$
- ▷ la scansione della lista di adiacenza di un nodo i richiede tempo **ottimo** $\Theta(|A(i)|)$
- ▷ la scansione di tutti gli archi del grafo richiede tempo **ottimo** $\Theta(n + m)$
- ▷ la verifica di appartenenza di un arco (i, j) richiede tempo $\mathcal{O}(|A(i)|)$

Grafi: Realizzazione con vettori di insiemi di adiacenza

Nell'ipotesi di grafo **statico**, cioè in cui non si effettuano operazioni di cancellazione ed inserzioni di nodi e/o archi, è possibile rappresentare gli insiemi di adiacenza senza usare le liste.

In pratica, dato un grafo $G = (N, A)$, $|N| = n$, $|A| = m$, si usano due vettori **NODI** e **ARCHI**:

- ▷ il vettore **NODI** ha una dimensione n e per ogni nodo i , $NODI[i]$ contiene un indice del vettore **ARCHI** a partire dal quale è memorizzato l'insieme di adiacenza $A(i)$.

Se $A(i) = \phi$ per qualche nodo, allora $NODI[i] = NODI[i + 1]$.

- ▷ il vettore **ARCHI** ha dimensione m e memorizza in posizione contigue $A(n_1), A(n_2), \dots, A(n_n)$.

Considerazioni

- ▷ lo spazio di memoria occupato è $\mathcal{O}(n + m)$, quindi **ottimo**
- ▷ si risparmia la memoria necessaria a realizzare una lista lincata

Realizzazione C con vettori di insiemi di adiacenza

La seguente é una realizzazione C di un grafo rappresentato tramite vettori di insiemi di adiacenza.

```
struct grafostruct {  
    int NODI[n+1];  
    int ARCHI[m];  
};  
  
typedef grafostruct * grafo;  
  
typedef int          nodo;
```

L'elemento $NODI[n] = m$ e svolge il ruolo di sentinella. Risulta utile nella definizione della procedure di **scansione** dell'insieme di adiacenza di un nodo u :

```
for ( i = G->NODI[u]; i < G->NODI[u+1]; i++ ) {  
    v = G->ARCHI[i];  
}
```

L'uso della sentinella evita di introdurre un controllo in piú quando per il trattamento delle condizioni di bordo, ovvero quando $u = n$.

I nomi dei nodi sono supposti essere numeri interi $[0 \dots n - 1]$.

Grafi: Algoritmi di Visita

Gli algoritmi di visita di grafi piú usati sono due:

- ▷ **Depth-First-Search** (DFS) visita in profondità o scandaglio
- ▷ **Breadth-First-Search** (BFS) visita in ampiezza o ventaglio

Entrambi gli algoritmi visitano un grafo a partire da un generico nodo r in tempo $\Theta(n + m)$, ovvero visitano ogni arco ed ogni nodo una sola volta e quindi sono algoritmi **ottimi**.

L'algoritmo di visita **DFS** é una estensione dell'algoritmo di visita in ordine anticipato di un albero:

quando si visita un nodo u ci si allontana da esso il piú possibile lungo un cammino, visitando i nodi e gli archi del cammino, fino a raggiungere un nodo i cui adiacenti sono stati tutti visitati. A questo punto si torna indietro lungo il cammino di un arco e ci si ri-allontana percorrendo un cammino di nodi ed archi non ancora visitati.

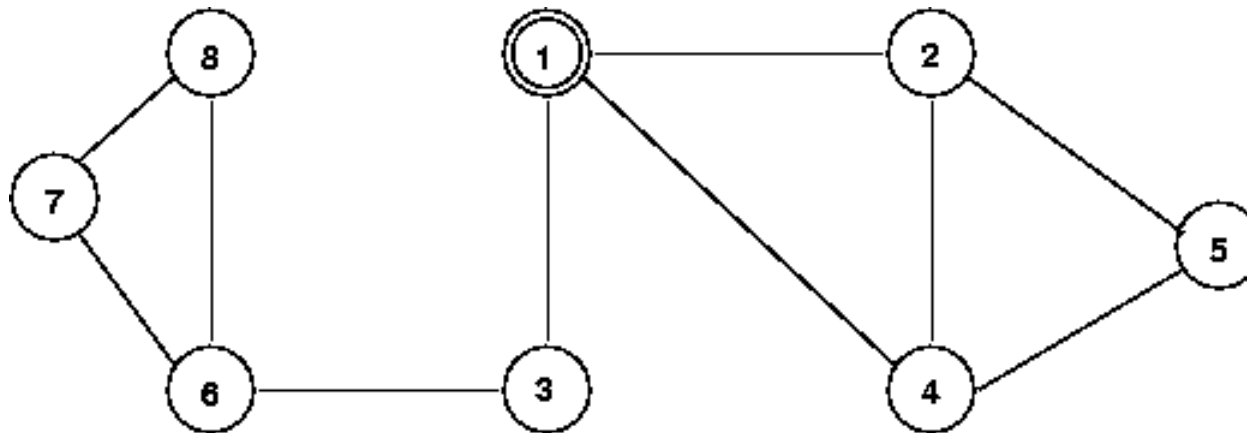
L'algoritmo di visita **BFS** é una estensione di una visita per livelli di un albero:

i figli di un nodo sono visitati dopo aver visitato tutti i fratelli del padre, ovvero, i nodi del grafo sono visitati in ordine di **distanza crescente** rispetto ad un nodo di partenza r . La distanza di un generico nodo u da r é definita come il **minimo** numero di archi in un cammino da r a u .

Grafi: Algoritmi di Visita

Entrambi i metodi di visita hanno bisogno di mantenere un elenco dei nodi visitati i cui nodi adiacenti non sono stati ancora visitati:

- ▶ nel caso dell'algoritmo DFS, poiché l'algoritmo torna indietro all'ultimo nodo visitato quando ha raggiunto un nodo i cui adiacenti sono stati tutti visitati, tale elenco si realizza con una pila oppure l'algoritmo si implementa con una procedura iterativa.
- ▶ nel caso dell'algoritmo BFS, poiché l'algoritmo visita prima tutti i nodi che hanno la medesima distanza da un nodo di partenza, e poi quelli più lontani, è naturale mantenere tale elenco in una coda.



- ▶ ordine di visita DFS: 1, 2, 5, 4, 3, 6, 7, 8
- ▶ ordine di visita BFS: 1, 2, 4, 3, 5, 6, 7, 8

Grafi: Implementazione Algoritmi di Visita

Per la realizzazione in C degli algoritmi di visita supponiamo di rappresentare un grafo

$$G = (N, A), \quad |N| = n, \quad |A| = m$$

mediante la seguente struttura dati:

```
struct structgrafo {  
    int NODI[n+1];    // vettore dei nodi  
    int ARCHI[m];      // vettore degli insiemi di adiacenza  
    int VISITATO[n];   // vettore booleano dei nodi visitati  
    int INCODA[n];     // vettore booleano dei nodi inseriti in coda Q  
};  
  
typedef struct structgrafo * grafo;  
  
typedef int nodo;
```

Grafi: Algoritmo di Visita DFS

La realizzazione C ricorsiva dell'algoritmo di visita DFS é la seguente:

```
void DFS ( grafo G, nodo u ) {  
    int i;  
    nodo v;  
  
    G->VISITATO[u] = 1;                // marca u visitato  
  
    for ( i = G->NODI[u]; i < G->NODI[u+1]; i++ ) { // scansione dei nodi adiacenti  
  
        v = G->ARCHI[i];                // nodo adiacente  
  
        visita_arco(u, v);              // visita l'arco  
  
        if ( ! G->VISITATO[v] )  
            DFS(G, v); // chiama ricorsivamente la procedura di visita su nodi adiacenti  
                        // non ancora visitati  
    }  
}
```

- ▷ il campo **VISITATO** é un vettore booleano di dimensione uguale al numero dei nodi del grafo. É usato per tenere traccia di quali nodi sono stati già visitati.
- ▷ la procedura **visita_arco** effettua una qualche elaborazione su un arco che unisce due nodi

Grafi: Algoritmo di Visita BFS

La realizzazione C iterativa dell'algoritmo di visita **BFS** é la seguente:

```
void BFS ( grafo G, nodo u ) {
    coda Q; // coda dei nodi visitati i cui adiacenti non sono stati ancora visitati
    nodo v;
    int i;

    Q = CREACODA(); // crea la coda Q vuota
    INCODA(u, Q); // inserisce nella coda il nodo u
    G->INCODA[u] = 1; // il nodo u e' stato inserito in Q

    while ( ! CODAVUOTA(Q) ) {
        u = LEGGICODA(Q); // leggi un nodo dalla coda Q
        FUORICODA(Q); // estrai l'elemento dalla coda
        G->INCODA[u] = 0; // il nodo u non e' in coda

        G->VISITATO[u] = 1; // marca u visitato

        for ( i = G->NODI[u]; i < G->NODI[u+1]; i++ ) { // scansione dei nodi adiacenti
            v = G->ARCHI[i]; // nodo adiacente
            visita_arco(u, v); // visita l'arco
            if ( ( ! G->VISITATO[v] ) && ( ! G->INCODA[v] ) ) { // se il nodo non e' stato visitato
                INCODA(v, Q); // e non e' in coda
                G->INCODA[v] = 1;
            }
        }
    }
}
```

Grafi: Algoritmo di Visita BFS

- ▷ il campo **INCODA** é un vettore di dimensione uguale al numero dei nodi del grafo
- ▷ ogni volta che un nodo v é inserito nella coda Q il corrispondente elemento $INCODA[v] = 1$
- ▷ ogni volta che un nodo v é estratto dalla coda Q il corrispondente elemento $INCODA[v] = 0$

La verifica di appartenenza di un nodo v alla coda Q é quindi realizzata in tempo $\mathcal{O}(1)$.

Tale verifica si rende necessaria per evitare che un nodo v , raggiungibile da due nodi diversi di cui uno risulta già visitato, possa essere inserito più volte nella coda Q (caso di un nodo raggiungibile da due nodi allo stesso livello).

Poichè:

- ▷ ogni nodo del grafo é marcato una sola volta
- ▷ per ogni nodo v viene visitato tutto l'insieme $A(v)$ degli archi incidenti una sola volta

possiamo dedurre che la complessità di entrambe gli algoritmi di visita é $\mathcal{O}(n + m)$ e quindi **ottima** (poiché il minimo necessario a visitare tutti gli archi e tutti i nodi)

Le procedure di visita DFS e BFS, opportunamente specializzate sono alla base di algoritmi di risoluzione di molti problemi sui grafi. Negli esempi che seguono si suppone che il grafo G sia rappresentato tramite vettore di insiemi di adiacenza, ma gli stessi algoritmi, opportunamente modificati, possono essere utilizzati con implementazione dei grafi differenti.

Grafi: Algoritmo per il Calcolo delle Componenti Connesse

Dato un grafo G non orientato si vogliono determinare tutte le componenti connesse di cui G é composto. L'algoritmo di risoluzione consiste nel richiamare iteritivamente la procedura di visita DFS su ogni nodo del grafo, in modo da marcare ogni nodo con il numero della componente connessa a cui esso appartiene.

```
void COMPONENTICONNESSE(grafo G, int n) {           // n = numero nodi del grafo G
    int COMP    = (int[]) malloc(n * sizeof(int)); // per ogni nodo i COMP[i] = num comp a cui appartiene
    int numcomp = 0;                                // numero della componente connessa

    for (i = 0; i < n; i++)
        COMP[i] = 0;                                // inizializza vettore COMP

    for (i = 0; i < n; i++)
        if ( COMP[i] == 0 ) { // nodo non ancora marcato
            numcomp++;
            DFS(G,i,numcomp);
        }
}

void DFS ( grafo G, nodo u, int numcomp ) {
    nodo v;
    COMP[i] = numcomp;                                // marca il nodo con il valore della componente connessa

    for ( i = G->NODI[u]; i < G->NODI[u+1]; i++ ) {
        v = G->ARCHI[i];                                // nodo adiacente
        if ( COMP[v] == 0 )                            // nodo non ancora marcato
            DFS(G, v, numcomp);
    }
}
```

Grafi: Albero di Copertura DFS

Dato un grafo G , tramite una visita DFS possiamo determinare un albero T , detto **albero di copertura DFS**.

Un albero di copertura T di un grafo G é un sotto-grafo di G **aciclico** che include tutti i nodi di G .

L'albero di copertura T partiziona gli archi di un grafo nel seguente modo:

- ▷ insieme degli archi che appartengono a T
- ▷ insieme di archi, non appartenenti a T , in avanti o all'indietro cioè congiungono coppie di nodi di T che sono o discendenti o antenati l'uno dell'altro.

```
void DFS ( grafo G, nodo u, albero * T ) {  
  
    G->VISITATO[u] = 1;           // marca u visitato  
    aggiungi_nodo(*T);           // aggiungi nodo u all'albero T  
  
    for ( i = G->NODI[u]; i < G->NODI[u+1]; i++ ) {  
        v = G->ARCHI[i];         // nodo adiacente  
        if ( ! G->VISITATO[v] ) {  
            aggiungi_arco(u,v, *T); // aggiungi arco (u,v) all'albero T  
            DFS(G, v);  
        }  
    }  
}
```

Grafi: Algoritmo per Determinare se un Grafo é Aciclico

Un grafo é **aciclico** se durante una visita DFS non si incontrano archi all'**indietro**, cioè se durante la visita di del nodo u non si incontra un arco (u, v) tale che v appartiene al cammino corrente che va dalla radice al nodo u .

```
void ACICLICO ( grafo G, nodo u, int * aciclico ) {
    G->CAMMINO[u] = 1;           // u appartiene al cammino corrente
    G->VISITATO[u] = 1;          // marca u visitato

    for ( i = G->NODI[u]; i < G->NODI[u+1]; i++ ) {
        v = G->ARCHI[i];         // nodo adiacente
        if ( ! G->VISITATO[v] )
            ACICLICO(G, v, aciclico);
        else
            if ( G->CAMMINO[v] == 1 )
                *aciclico = 0;
    }
    G->CAMMINO[u] = 0;
}
```

Il vettore CAMMINO é un vettore booleano, e per ogni nodo v , **CAMMINO** $[v] = 1$ se il nodo v appartiene al cammino corrente radice nodo u .

N.B.: l'insieme di tutti i nodi x per cui **CAMMINO** $[x] = 1$ costituisce un cammino che va dalla radice al nodo che si sta visitando.

Grafi: Algoritmo per Determinare se un Grafo é Bipartito

Un grafo non orientato G é **bipartito** se l'insieme dei nodi può essere partizionato in due parti tali che nessun arco connetta due nodi appartenenti alla stessa partizione.

Un algoritmo di risoluzione consiste nel colorare i nodi del grafo con due colori diversi, in modo tale che due nodi adiacenti siano colorati con due colori diversi.

L'algoritmo esegue una visita BFS ed ad ogni passo cerca di colorare i nodi adiacenti al nodo in visita con un colore diverso.

Il vettore **COLORE** memorizza per ogni nodo i valori 0, 1, 2 cioè:

- ▷ 0 = nodo non colorato,
- ▷ 1 = nodo colorato con colore 1
- ▷ 2 = nodo colorato con colore 2

Grafi: Algoritmo per Determinare se un Grafo é Bipartito

```
int BIPARTITO ( grafo G, nodo r ) {  
    int bipartito = 1;  
    coda Q = CREACODA();  
  
    G->COLORE[r]++; // coloro la radice con il colore 1  
    INCODA(r, Q);  
  
    while ( !CODAVUOTA(Q) && bipartito ) {  
        u = LEGGICODA(Q);  
        FUORICODA(Q);  
  
        for ( i = G->NODI[u]; i < G->NODI[u+1]; i++ ) {  
            v = G->ARCHI[i]; // nodo adiacente  
  
            if ( G->COLORE[v] == 0 ) { // nodo non colorato  
                G->COLORE[v] = (G->COLORE[u] == 1) ? 2 : 1; // coloro il nodo  
                INCODA (v, Q);  
            } else  
                if ( G->COLORE[u] == G->COLORE[v] ) // grafo non bipartito  
                    bipartito = 0;  
        }  
    }  
    return bipartito;  
}
```

Ordine Topologico di un Grafo Orientato Aciclico

L'ordine **topologico** o **linearizzazione** di un grafo orientato **aciclico** stabilisce una relazione di ordinamento $<$ tra i nodi del grafo, in modo tale che se (u, v) é un arco del grafo, allora il nodo $u < v$.

Supponiamo, per semplicità, che il grafo sia **connesso**¹, ovvero il grafo ha una unica componente connessa.

L'ordine topologico é calcolato tramite una visita DFS che costruisce un vettore **ORDINALE** di lunghezza $n = |N|$, che contiene gli indici dei nodi del grafo ordinati in modo crescente in base alla relazione $<$.

```
void TOPOLOGICALSORT ( grafo G, nodo u, int k ) {
    G->VISITATO[u] = 1; // marca u visitato
    for ( i = G->NODI[u]; i < G->NODI[u+1]; i++ ) { // per ogni nodo adiacente di u
        v = G->ARCHI[i]; // nodo adiacente
        if ( ! G->VISITATO[v] )
            TOPOLOGICALSORT(G, v);
    }
    G->ORDINALE[n-k] = u; // memorizza il nodo u in posizione n-k, riempito a ristroso
    k++;
}

int main () {
    TOPOLOGICALSORT( G, r, 1 ); // chiamata iniziale, r = nodo da cui si comincia la visita
}
```

¹per ogni coppia di nodi u, v o esiste il cammino da u a v o il cammino da v a u