GRAFI

In questo capitolo studieremo i grafi, che **rappresentano associazioni (*archi*) tra oggetti in cui ad ogni oggetto (*nodo*) sono associati più oggetti (*nodi adiacenti*)** e, al contrario degli alberi, **ogni oggetto può essere adiacente di più di un nodo**. Analizzeremo i concetti più significativi (*grafo orientato e non, cammino, connettività, ciclicità* e altri) e mostreremo le due modalità più comuni di rappresentazione concreta dei grafi, con *liste di adiacenza* e con *matrice di adiacenza*. Inoltre presenteremo i due principali algoritmi di *visita* di un grafo (a ventaglio o ampiezza e a scandaglio o in profondità) e utilizzeremo tali algoritmi per la risoluzione di alcuni classici problemi su grafi.

DEFINIZIONI FONDAMENTALI

Un *grafo* ***G*** è costituito da una coppia ***(N,A)*** dove N è l'insieme di *nodi* (o *vertici*) e A è un insieme di coppie di nodi, dette *archi*, Se gli archi sono coppie ordinate (v,w) allora il grafo è detto ***orientato***; se v e w coincidono l'arco è chiamato *anello*. Se gli archi sono invece coppie non ordinate (cioè insiemi) (v,w) allora il grafo è detto ***non orientato*** ed, ovviamente, v e w non possono coincidere. I grafi sono rappresentati come in Figura 1; ambedue i grafi nella figura hanno 7 nodi; il grafo orientato ha 9 archi mentre quello non orientato ne ha 8.

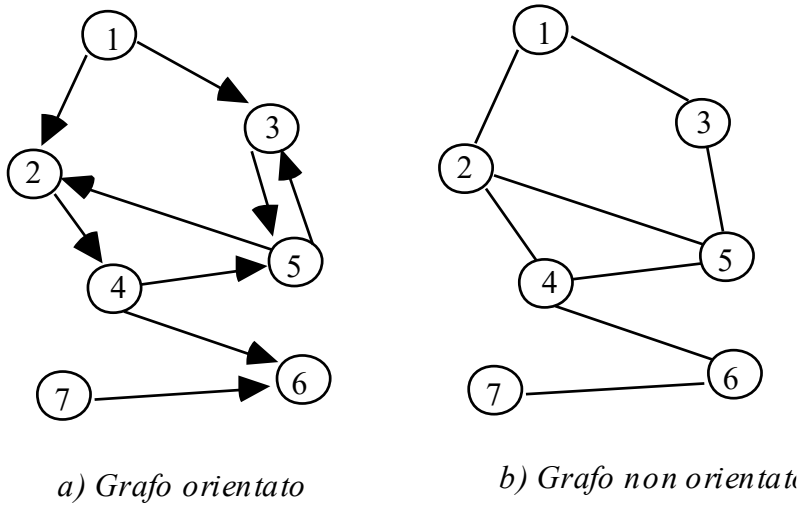
Nel seguito, dove non diversamente specificato, assumeremo che n sia il numero di nodi e m il numero di archi. Ovviamente il **numero massimo d'archi in un grafo orientato è n^2** mentre in un **grafo non orientato è $(n \times (n-1))/2$**. In generale quindi vale $m = O(n^2)$; se $m = \Omega(n^2)$ allora il grafo è detto ***denso*** mentre se $m = O(n)$ è detto ***sparso***.

Sia *G* un grafo orientato. Se esiste un arco (v,w) allora diremo che w è *adiacente* a v e che l'arco *esce* da v ed *entra* in w. Il ***grado di entrata*** di un nodo v è il numero di archi entranti in esso e il ***grado d'uscita*** è il numero di archi uscenti, cioè il numero di nodi adiacenti a v. Nella Figura 1a il nodo 2 ha grado di entrata 2 e grado di uscita 1.

Sia *G* un grafo non orientato. Se esiste un arco (v,w) allora diremo che v e w sono adiacenti e che quest'arco è lo stesso dell'arco (w,v). Il ***grado*** di un nodo v è il numero di nodi adiacenti a v. Nella Figura 1b il nodo 2 ha grado 3.

Un ***cammino*** da v a w in un grafo (orientato o non) è una sequenza di archi distinti (v_1,v_2), (v_2,v_3),..., (v_{k-1},v_k) tale che $v = v_1$ e $w = v_k$ ed ha *lunghezza* k-1. Per convenzione si assume che esiste sempre un cammino da un nodo a se stesso di lunghezza 0 (cammino *nullo*). Un cammino è

detto ***semplice*** se tutti i nodi v_1, v_2,..., v_k sono distinti tranne eventualmente v_1 e v_k che possono coincidere; in questo caso il cammino è detto ***ciclo***. Notiamo che (2,4),(4,5),(5,2) è un ciclo sia nel grafo di Figura 1a che in quello di Figura 1b mentre la sequenza (1,2),(2,4),(4,5),(5,3),(3,1) è un ciclo nel grafo di Figura 1b ms non in quello di Figura 1a.



a) Grafo orientato *b) Grafo non orientato*

Figura 1. *Rappresentazione di grafi*

La sequenza (1,2),(2,4),(4,5),(5,3),(3,1) è un ciclo nel grafo di Figura 1b ms non in quello di Figura 1a.

In un grafo orientato il più piccolo ciclo ha lunghezza 1 (caso di anello) mentre in un grafo non orientato il più piccolo ciclo ha lunghezza 3. La massima lunghezza di un cammino semplice è n; la massima lunghezza di un cammino semplice non ciclico è n-1. Un cammino non semplice può avere lunghezza illimitata e contiene al suo interno almeno un ciclo.

Un grafo, orientato o non, **è *ciclico*** se contiene almeno un ciclo di lunghezza maggiore di 1 ed ***aciclico*** altrimenti. I due grafi nella Figura 6 sono ambedue ciclici.

Un grafo non orientato è *connesso* se esiste un cammino che collega ogni coppia di nodi. Una ***componente connessa* è un insieme massimale di nodi tale che esiste un cammino che collega ogni coppia di nodi di esso**; la collezione delle componenti connesse costituisce una partizione dei nodi del grafo. Ovviamente se il grafo è connesso esso ha una sola componente connessa. Un nodo *isolato* (cioè non adiacente a nessun nodo) forma una componente connessa singoletta. Il grafo in Figura 1b è connesso; se eliminiamo l'arco (4,6), il grafo diventa non connesso ed ha due componenti connesse; una con i nodi da 1 a 5 e l'altra con i nodi 6 e 7. Ovviamente il sottografo corrispondente alla seconda componente è aciclico.

Un grafo orientato è *fortemente connesso* se per ogni coppia di nodi *v* e *w* esiste un cammino da *v* a *w* e da *w* a *v*; **una *componente fortemente connessa*** (o semplicemente *componente forte*) **è un insieme massimale di nodi tale che per ogni coppia di nodi *v* e *w* esiste un cammino da *v***

a *w* e da *w* a *v*. La collezione delle componenti forti costituisce una partizione dei nodi del grafo. Ovviamente il grafo è fortemente connesso se e solo se esso ha una sola componente forte. Se un grafo ha una componente forte con più di un elemento allora è ciclico. Un grafico è aciclico se e solo se tutte le componenti forti sono singoletti. Il grafo nella Figura 1a è ciclico ma non fortemente connesso; esso ha le seguenti 4 componenti forti: {1}, {2,3,4,5}, {6}, {7}.

Un grafo orientato è *debolmente connesso* se il grafo non orientato ottenuto da esso eliminando l'orientamento degli archi è connesso. Le componenti debolmente connesse coincidono con le componenti connesse del grafo non orientato corrispondente. Il grafo di Figura 1a è debolmente connesso in quanto la sua versione non orientata è connessa (vedi grafo di Figura 6b).

La ***chiusura transitiva*** di un grafo orientato G=(N,A) è un grafo orientato G^+=(N,A^+), tale che un arco (i,j) è in A^+ se e solo se esiste un cammino da i a j in G. Poichè se esiste un arco da i a j in G esiste anche un cammino da i a j per definizione, $A \subseteq A^+$. La chiusura transitiva del grafo di Figura 1a si ottiene aggiungendo gli archi (1,4), (1,5), (1,6), (2,5), (2,3), (2,6),(4,3), (4,2), (5,4), (5,6), (3,2), (3,4), (3,6) più gli anelli (1,1), (2,2),..., (7,7).

La definizione di chiusura transitiva di un grafo non orientato è poca significativa; infatti otterremmo un arco nella chiusura per ogni coppia di nodi nella stessa componente.

E' facile ora vedere che **un albero è un grafo orientato aciclico** tale che un solo nodo (la radice) non ha archi entranti, gli altri nodi hanno esattamente un arco entrante ed esiste un cammino (che è facile dimostrare essere unico) dalla radice ad ogni altro nodo. Ovviamente un albero è un grafo orientato debolmente ma non fortemente connesso. Eliminando gli archi (3,5) e (5,2) dal grafo di Figura 1a, il grafo diventa aciclico ma non un albero. Non è possibile rendere tale grafo un albero attraverso la sola eliminazione di archi. Infatti il nodo 6 ha due archi entranti per cui non può essere un nodo di un albero; questi due archi non possono essere eliminati pena la disconnessione dei nodi 7 e 8. Un albero può essere ottenuto eliminando gli archi (5,2), (3,5), (5,3), (7,6) e aggiungendo l'arco (6,7).

Un grafo orientato consistente di una collezione di alberi è detto una ***foresta***. e ogni componente debolmente connessa corrisponde ad un albero. Eliminando gli archi (5,2), (3,5), (5,3) e (4,6) il grafo di Figura 1a diventa una foresta di due alberi.

Un *albero non orientato* è un grafo non orientato connesso ed aciclico. Una volta scelto uno dei nodi come radice l'albero orientato diventa l'usuale albero che abbiamo studiato in quanto la radice induce un orientamento degli archi. Il grafo di Figura 1b non è un albero non orientato perchè pur essendo connesso non è aciclico.

Un albero non orientato G'=(N',A') è un *albero ricoprente* di un grafo non orientato G = (N,A) se $N = N'$ e $A' \subseteq A$. Un grafo non orientato ha almeno un albero ricoprente se e solo se è connesso. Eliminando gli archi (1,3) e (4,5) si ottiene un albero ricoprente del grafo di Figura 1b. Tale grafo ha altri 10 alberi ricoprenti: ad esempio uno di questi si ottiene eliminando gli archi (4,5) e (3,5).

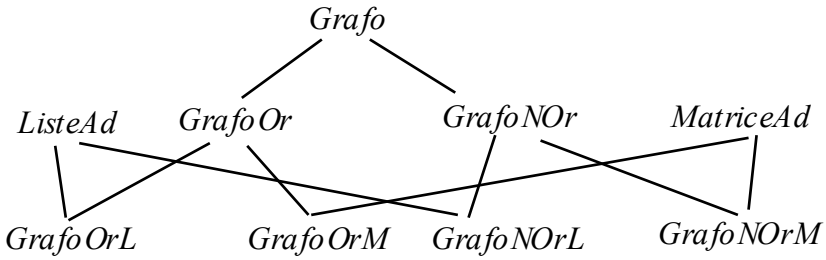
Un proprietà importante è la seguente: un grafo non orientato connesso è un albero non orientato se e solo se $m = n$-1. Questa condizione è necessaria ma non sufficiente nel caso di grafi orientati poichè per essi va anche verificata la direzione degli archi.

Una *foresta non orientata* è un grafo non orientato aciclico; ogni componente connessa del grafo corrisponde ad un albero. E' facile verificare che un grafo non orientato è una foresta non orientata se e solo se $m = n-k$, dove k è il numero di componenti connesse.

DEFINIZIONE DI GRAFI

Rappresentazione astratta di grafi

Presentiamo di seguito una gerarchia di classi che include la classe astratta di grafo, indipendente dal tipo di orientamento e di rappresentazione, le classi astratte di grafo orientato e non, indipendenti dal tipo di rappresentazione, e quattro classi concrete di grafo, due per i grafi orientati e due per quelli non orientati, che utilizzano le due modalità più comuni di rappresentazione: con *liste di adiacenza* (per ogni nodo viene memorizzata la lista dei nodi ad esso adiacenti) e con *matrice di adiacenza* (una matrice M di booleani indicizzata da una coppia di nodi i e j tale che $M[i][j]$ è VERO se e solo se esiste un arco da i a j). Di seguito presentiamo la gerarchia delle classi dei vari tipi di grafo:



Le classi *Grafo*, *GrafoO* e *GrafoNOr* sono astratte e permettono di definire funzioni che si applicano rispettivamente a grafi generali, grafi orientati e grafi non orientati indipendentemente dalla loro realizzazione e passati per riferimento come argomenti. Le classi *ListeAd* e *MatriceAd* sono private e sono utilizzate per definire le funzioni di manipolazione per le classi *GrafoOrL* e *GrafoNOrL* (grafi orientati e non con struttura a liste di adiacenza) e le classi *GrafoOrM* e *GrafoNOrM* (grafi orientati e non con struttura a liste di adiacenza). Queste ultime quattro sono le uniche classi utilizzabili per definire variabili .

Per semplicità di esposizione, introdurremo solo una classe Grafo che implementa le matrici di adiacenza. Si noti che, in un grafo, i nodi sono numerati da 0 a n-1. Il contenuto informativo di ciascun nodo può essere gestito tramite strutture dati esterne che associano a ciascun nodo (un indice tra 0 ed n-1) un valore.

Allo stesso modo, è possibile associare del contenuto informativo agli archi del grafo. In questo caso si parla di **grafo pesato**. Un modo semplice di implementare un grafo pesato è estendere la classe grafo, includendo una struttura dati (associata alla matrice di adiacenza o alla lista di

adiacenza) che contiene il “peso” (o etichetta) associato all’arco. Molte importanti applicazioni fanno uso di grafi pesati, quali problemi basati sui cammini di peso minimo, massimo flusso, ecc.

Di seguito una semplice implementazione della classe grafo orientato. La versione non orientata può essere ottenuta semplicemente ridefinendo gli operatori di accesso (). Eventuali ottimizzazioni possono essere ottenute utilizzando una struttura dati specifica per la rappresentazione delle matrici simmetriche.

```
#ifndef GRAFO_H_
#define GRAFO_H_

#include <cassert>
#include <vector>

using boolVec = std::vector<bool>;

// grafo orientato
// (i,j) != (j,i)
class Grafo {
protected:
    // numero di nodi, numero di archi
    unsigned vn = 0, vm = 0;

    // matrice di adiacenza
    std::vector<boolVec> archi;
      // vector<vector<bool>> (i,j) == true => esiste un arco da i verso j

    // inizializza la matrice di adiacenza
    void init(unsigned n) {
        this->vn = n;
        this->vm = 0;

        this->archi = std::vector<boolVec>(n);
        for (unsigned i = 0; i < n; i++)
            this->archi[i] = boolVec(n, false);
    }

public:
    Grafo(unsigned n) {
        assert(n >= 1);
        this->init(n);
    }

    // inserisce o elimina l'arco (i,j) (a seconda del valore di b)
    void operator()(unsigned i, unsigned j, bool b) {
        assert(i >= 0 && i < this->n() && j >= 0 && j < this->n());
        bool esisteArco = this->archi[i][j];
        if ((!esisteArco && b) || (esisteArco && !b)) {
            this->archi[i][j] = b;  // arco i->j

      //this->archi[j][i] = b; // arco j->i (rende il grafo non orientato)
            if (b)
                vm++;
            else
                vm--;
        }
    }
```

```
    // elimina tutti gli archi
    void svuota() {
        for (unsigned i = 0; i < this->n(); i++)
            for (unsigned j = 0; j < this->n(); j++) {
                archi[i][j] = false;
                //archi[j][i] = false;
            }
        vm = 0;
    }

    Grafo& operator=(const Grafo& g) {
        if (this == &g)
            return *this;
        this->init(g.n());
        for (unsigned i = 0; i < this->n(); i++)
            for (unsigned j = 0; j < this->n(); j++)
                if (g(i,j))
                    this->archi[i][j] = true;
                else
                    this->archi[i][j] = false;
        return *this;
    }


    unsigned n() const { return vn; }
    unsigned m() const { return vm; }

    // true se l'arco (i,j) esiste, altrimenti false
    bool operator()(unsigned i, unsigned j) const {
        assert(i >= 0 && i < this->n() && j >= 0 && j < this->n());
        return this->archi[i][j];
    }
};

#endif
```

Esempio 1. Scriviamo una funzione che, dato un grafo, restituisce per ogni nodo il numero di nodi adiacenti. Cioè questa funzione calcola il grado di ciascun nodo nel caso di grafi non orientati e il grado di uscita di ciascun nodo nel caso di grafi orientati.

```
vector<int> numAdiacenti( const Grafo& G )
{     vector<int> grado(G.n());
      for (int i =0; i < G.n(); i++ )
      {
            grado[i] = 0;
            for ( int j = 0; j < G.n(); j++ )
                  if ( G(i,j) )
                        grado[i]++;

      }
      return grado;
}
```

Esempio 2. Scriviamo una funzione che dato un grafo orientato restituisce il grafo non orientato ottenuto da esso eliminando l'orientamento degli archi.

```
void disOrienta( const Grafo& G, Grafo& G1)
{     assert ( G.n() == G1.n() );
      G1.svuota();
      for ( int i = 0; i < G.n(); i++ )
            for ( int j = 0; j < G.n(); j++ )
                  if ( i!=j && G(i,j) )
                  {
                        G1(i,j,true);
                        G1(j,i,true);
                  }
}
```

5.3 ALGORITMI SU GRAFI

5.3.1 Visite di Grafi

Un problema di base nei grafi è visitare tutti i nodi raggiungibili da un dato nodo *v*, cioè determinare tutti i nodi *w* per cui esiste un cammino da *v* a *w*. Ci sono due modalità fondamentali di effettuare una visita in un grafo:

(i) *visita a ventaglio*, anche detta *in ampiezza* o *BFS*, in cui sono visitati inizialmente tutti i nodi adiacenti al nodo di partenza e, successivamente, sono visitati gli adiacenti al nodo che è stato visitato per prima e i cui adiacenti non siano già stati tutti visitati (essa corrisponde alla visita per livelli);

(ii) *visita a scandaglio*, anche detta *in profondità* o *DFS*, in cui è visitato un nodo alla volta, scegliendolo tra gli adiacenti al nodo visitato per ultimo i cui adiacenti non siano già stati tutti visitati (essa corrisponde alla visita anticipata).

Esempio 3. Consideriamo il grafo di Figura 2. Nella visita a ventaglio a partire dal nodo 1 vengono visitati i nodi 2, 6 e 7, poi il nodo 3, poi il nodo 4 e infine il nodo 5. Nella visita a scandaglio invece vengono visitati i nodi nell'ordine: 1, 2, 3, 4, 5, 6 e 7.

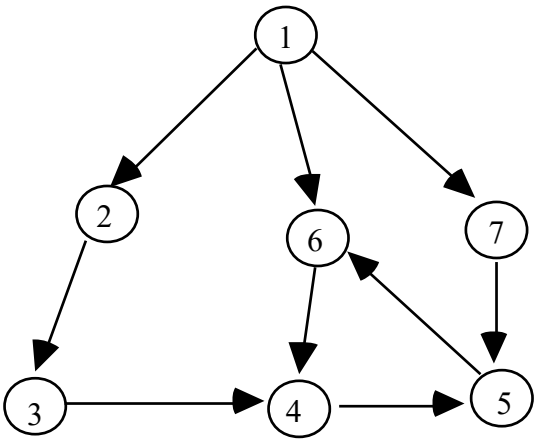


Figura 2. *Grafo orientato*

Scriviamo ora le funzioni di **visita**.

Cominciamo con la visita in profondità, o **DFS**. La funzione genera un vettore di booleani in cui un elemento *i* è 'true' se e solo se esso è raggiungibile dal nodo *s*.

```
void dfs(const unsigned& s, const Grafo& g, vector<bool>& visitati)
{
    // (1) segno come visitato i
    visitati[s] = true;
    // (2) itero su s nodi adiacenti di s e visito quelli non ancora visitati
    for (unsigned j = 0; j < g.n(); ++j)
        if (s != j && g(s,j) && !visitati[j])
            dfs(j, g, visitati);
}
```

La complessità della funzione *dfs* dipende dal numero di volte che è ripetuto il ciclo *while*. Nel caso di grafo a matrice di adiacenza è n^2. Pertanto la complessità è $\Theta(n^2)$.

Mostriamo ora un esempio d'uso della dfs per calcolare la chiusura transitiva di un grafo orientato.

```
Grafo chiusuraTransitiva(const Grafo& g) {
    Grafo gc(g.n());
      vector<bool> visitati(g.n());

      for (int i=0; i<g.n(); i++)
      {
            visitati.assign(visitati.size(),false);
            dfs(i,g,visitati);
            for (int j=0; j<g.n(); j++)
                  if (i!=j && visitati[j])
                        gc(i,j,true);
      }
      return gc;
}
```

La BFS è presentata di seguito. Ne presentiamo una versione in cui viene restituito il vettore dei "predecessori", che consente di ricostruire il cammino individuato tra *s* e ciascuno dei nodi raggiungi.

Si noti che, nel caso di grafi semplici (orientati e non), la BFS genera il cammino più breve, mentre per i grafi pesati (orientati e non) con il peso degli archi uguale per ogni arco, la BFS genera il cammino di peso minimo.

```
vector<int> bfs(const Grafo& g, const unsigned& s) {
      // visitati[i] == true se il nodo i è stato visitato a partire da s
    vector<bool> visitati(g.n(), false);

      // p[i] == -1 se il nodo i non è mai visitato a partire da s
      //(non esiste un cammino da s verso i)
    vector<int> p(g.n(), -1);

      // coda per la bfs
    queue<unsigned> q;

      // step 1: parto da s, lo inserisco nella coda e lo segno come visitato
    q.push(s);
    p[s] = s;
    visitati[s] = true;

      // itero fin quando la coda non è vuota
    while (!q.empty()) {
            // step 2: estraggo il nodo in testa alla coda
        unsigned u = q.front();
        q.pop();

            // step 3: itero sui nodi adiacenti di u e inserisco nella
            // coda quelli non visitati
        for (unsigned v = 0; v < g.n(); ++v)
            if (g(u,v) && u != v && !visitati[v]) {
                q.push(v);
                p[v] = u;     // il predecessore del nodo v è il nodo u
                              // (arrivo a visitare v passando da u)
                visitati[v] = true;
            }
    }

    return p;
}
```

La complessità della visita a ventaglio è $O(n^2)$.

Di seguito viene presentato l'algoritmo per **ricostruire il cammino tra due nodi**, una volta costruito il vettore dei pedecessori con la dfs appena introdotta:

```
void ricostruisciCammino(const int& s, const int& t, const std::vector<int>&
p, queue<int>& cammino) {
    // caso base
    if (s == t)
        cammino.push(s);
    else if (t == -1)
        return;
    else {
        // itero a ritroso fino a quando non arrivo ad s (caso base)
        ricostruisciCammino(s, p[t], p, cammino);
        cammino.push(t);
    }
}
```

Di seguito presentiamo una versione alternativa di ricostruisciCammino, che chiameremo trovaCammino, che utilizza la dfs. In particolare, tale versione cerca di costruire il cammino seguendo le adiacenze e andando in profondità. Se non si riesce a costruire il cammino secondo il percorso attuale, si cerca un'altra strada. N.B. questa versione non garantisce di trovare il cammino più breve, ma cerca di trovarlo nel più breve tempo possibile.

```
bool trovaCammino(const Grafo & g, const int &s, const int& t, vector<bool> &
visitati, list<int>& cammino)
{
      if (s==t){
            visitati[t]=true; return true;
      }
      else{
            visitati[s]=true;
            bool trovato=false;
            for (int j=0; j<g.n() && !trovato; j++)
                  if (g(s,j) && !visitati[j])
                  {
                        cammino.push_back(j);
                        trovato=trovaCammino(g,j,t,visitati,cammino);
                        if (!trovato) cammino.pop_back();
                  }
            return trovato;
      }
}
```

Si noti che, nel caso peggiore, la funzione si riconduce ad una visita in profondità di tutto il grafo a partire da s. Quindi la complessità di caso peggiore è la stessa della visita ($O(n^2)$). Tuttavia, nel caso in cui il cammino venga trovato subito, la funzione termina, quindi la complessità di caso migliore diventa O(l), dove l è la lunghezza del cammino. Al contrario, la ricostruisciCammino deve comunque eseguire la bfs, quindi sia nel caso migliore, che nel caso peggiore la sua complessità è $O(n^2)$.

Di seguito un main di prova:

Si ricorda che per leggere un input da un file, utilizzando sempre cin si può eseguire il codice con il comando windows:

```
type nomefile | ./programma.exe
```

oppure per linux:

```
cat nomefile | ./programma
```

```
#include <iostream>
#include <vector>
#include <list>
#include "Grafo.hpp"
using namespace std;

int main() {

	int n, u,v;
	cin>>n;	//Leggo il numero di nodi;
	Grafo g(n);

	while (cin.good()) {	//Leggo il grafo
		cin >> u >> v;
		g(u, v, true);  //indici dei nodi tra 0 ed n-1
	}

	vector<bool> visitati(g.n(),false);
	list<int> cammino;
	int s=0, t=2;
	if (trovaCammino(g,s,t,visitati,cammino))
	{
		cout<<s<<" ";
		while (!cammino.empty())
		{
			cout<<cammino.front()<<" ";
			cammino.pop_front();
		}
		cout<<endl;

	}
	else{cout <<"cammino non trovato"<<endl;}

	return 0;
}
```