# Capitolo 5

# 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 quali verificare la ciclicità di un grafo, determinare il numero di componenti connesse, ecc.

## 5.1 DEFINIZIONI FONDAMENTALI

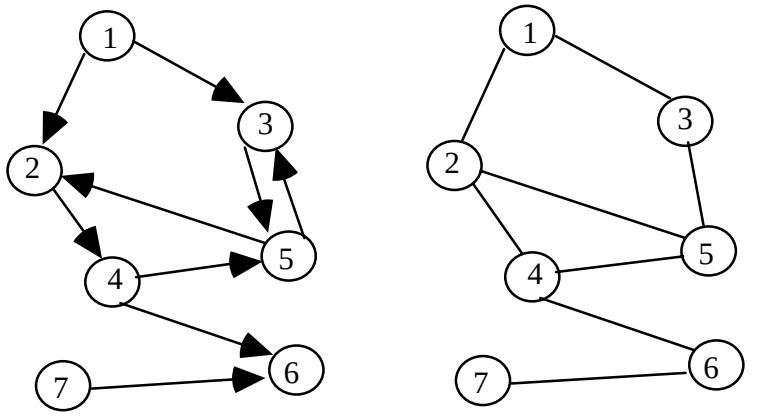
Un *grafo G* è costituito da una coppia (*N,A*) dove *N* è l'insieme di *nodi* (o *vertici*) e *A* è un insieme di coppie di nodi, dette *archi*, Se gli archi sono coppie ordinate (*v,w*) allora il grafo è detto *orientato*; se *v* e *w* coincidono l'arco è chiamato *anello*. Se gli archi sono invece 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*; di più 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).

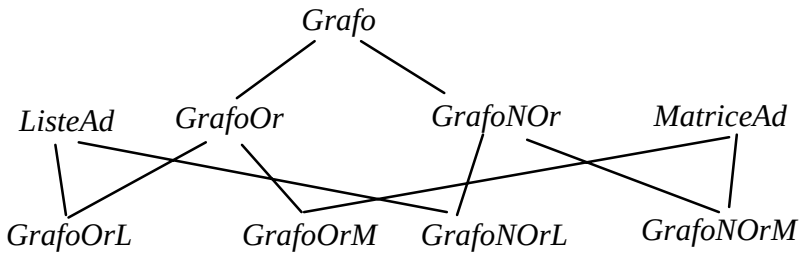
Un proprietà importante è la seguente: un grafo non orientato connesso è un albero non orientato se e solo se $m = n-1$. Questa condizione è necessaria ma non sufficiente nel caso di grafi orientati poichè per essi va anche verificata la direzione degli archi.

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.

## 5.2 DEFINIZIONE DI GRAFI

### 5.2.1 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 .

Abbiamo previsto due modalità per accedere agli archi: una chiedendo direttamente se esiste un arco tra due nodi e l'altra scandendo uno dopo l'altro gli archi adiacenti di un nodo $i$ nell'ordine lessicografico, cioè $(i_1,j_1)$ precede $(i_2,j_2)$ se $i_1<i_2$ oppure $i_1=i_2$ e $j_1<j_2$. Pe una tale scansione

definiamo la classe *Arco* in modo che il primo arco *e* di un grafo *G* possa essere restituito con il comando '*e=G.primoArco*()' e i successivi archi siano ottenuti facendo iterativamente eseguire il comando *e=G.succArco*(*e*) fino a che l'arco *e* non diventi nullo; è possibile anche accedere direttamente al primo arco adiacente a un nodo *i* con il comando '*e=G.primoAd*(*i*)' e i successivi adiacenti sono ottenuti facendo iterativamente eseguire il comando '*e=G.succAd*(*e*)' fino a che l'arco *e* non diventi nullo. Le due scansioni possono essere realizzate implicitamente attraverso le due macro '*for_each_arc*(*e,G*)' e '*for_each_ad*(*e,i,G*)'. La classe *Arco*, pur riferendo a una classe astratta, non è essa stessa astratta nonostante il suo funzionamento, dipendente dal tipo di realizzazione adottato per il grafo di riferimento, non sia specificato. Questa genericità è ottenuta attraverso un campo *pos* definito come puntatore a *void*, che è poi concretamente gestito all'interno del grafo di riferimento per il quale, all'atto della dichiarazione della variabile arco, è ormai nota l'effettiva realizzazione.

```
// file grafo.h
#ifndef GRAFOH
#define GRAFOH
#include "utilita.h"
class grafo; // definita successivamente
class Arco
{     public:
//    FUNZIONI NON COSTANTI
      Arco ( ): pgrafo(0), pos(0) {} // costruttore
      ~Arco ( ) {delete pos;} // distruttore
//    FUNZIONI COSTANTI
//    restituisce il grafo a cui appartiene l'arco
//    ecc   {(i<1||i>n||j<1||j>n)→Err(IndNodo);}
      const Grafo& grafo() const  throws (Grafo::ArcoNullo)
      {     if ( !(*this) ) throw Grafo::ArcoNullo();
            return *pgrafo;
      }
//    restituisce 'false' se l'arco è nullo
      operator bool() const {return pgrafo!=0;}
//    restituisce il nodo iniziale dell'arco
//    ecc   {(i<1||i>n||j<1||j>n)→Err(IndNodo);}
      friend card in ( const Arco& e )   throws (Grafo::ArcoNullo)
      {     if ( !e ) throw Grafo::ArcoNullo();
            return e.i;
      }
//    restituisce il nodo finale dell'arco
//    ecc   {(i<1||i>n||j<1||j>n)→Err(IndNodo);}
      friend card fin ( const Arco& e ) throws (Grafo::ArcoNullo)
      {     if ( !e ) throw Grafo::ArcoNullo();
            return e.j;
      }

      protected:
      friend class grafo;
      const Grafo* pgrafo; // grafo di appartenenza
      card i, j; // nodo iniziale e nodo finale
//    posizione dell'arco nella lista di adiacenza
      void* pos;
};
#define for_each_arc(e,G) for (e = G.primoArco();e; e = G.succArco(e))
#define for_each_ad(e,i,G) for (e = G.primoAd(i);e; e = G.succAd(e))
```

Siamo ora pronti a definire la classe *Grafo*. Come già anticipato, essa è astratta e, quindi, indipendente dalla possibile realizzazione.

```
class Grafo
{     public:
//    FUNZIONI NON COSTANTI
//    ecc   {NOMEM → ErrMem;}
      Grafo( card nNo = 0 ) throws (ErrMem): vn(nNo),vm(0) {}
//    inserisce o elimina (i,j) a secondo se b è true o false
//    ecc   {(i<1||i>n||j<1||j>n)→ Err(IndNodo);} + {NOMEM→ ErrMem;}
      virtual void operator()( card i, card j, bool b )
                                          throws (IndNodo,ErrMem) = 0;
//    elimina tutti gli archi
      virtual void svuota ()throws () = 0;
//    assegnamento
//    ecc   {G.n()>0 && G.n()!=GC.n())→ Err(DiffNodi);} + {NOMEM→ ErrMem;}
      Grafo& operator= ( const Grafo& GC ) throws (DiffNodi,ErrMem)
      {     if ( this == &GC ) return *this;
            if ( n()>0 && n() != GC.n() ) throw DiffNodi();
            if ( n() == 0 )
                  init( GC.n() );
            else
                  svuota();
            Arco e;
            for_each_arc(e,GC)
                  operator() (in(e),fin(e),true);
            return *this;
      }
//    fissa il numero di nodi per un grafo nullo
//    ecc   {G.n()>0)→ Err(NonNullo);} + {NOMEM→ ErrMem;}
      void init ( card nNo)    throws (NonNullo,ErrMem)
      {     if ( vn > 0 ) throw NonNullo();
            vn = nNo; if ( vn > 0 ) iniz();
      };
//    FUNZIONI COSTANTI
      card n() const {return vn;}; // restituisce il numero di nodi
      card m() const {return vm;} // restituisce il numero di archi
//    restituisce vero o falso se l'arco (i,j) è nel grafo o meno
//    ecc   {(i<1||i>n||j<1||j>n)→ Err(IndNodo);}
      virtual bool operator() ( card i, card j ) const throws (IndNodo) = 0;
//    restituisce il primo arco nell'ordine lessicografico
//    ecc   {NOMEM→ ErrMem;}
      virtual Arco primoArco ( ) const throws (ErrMem) = 0
//    restituisce l'arco successivo a 'e' nell'ordine lessicografico
//    ecc   {!e→ErrG(ArcoNullo); (&e.grafo() != &G)→ Err(DiffGrafo);}
            + {NOMEM→ ErrMem;}
      virtual Arco succArco ( const Arco& e ) const
                        throws (ArcoNullo, DiffGrafo,ErrMem) = 0
//    restituisce il primo arco adiacente al nodo 'i' nell'ordine lessicograf.
//    ecc   {(i<1||i>n||j<1||j>n)→ Err(IndNodo);} + {NOMEM→ ErrMem;}
      virtual Arco primoAd ( card i ) const
                         throws (IndNodo,ErrMem) = 0
//    restituisce l'arco adiacente al nodo i successivo a 'e' nell'ordine
//    ecc   {!e→ErrG(ArcoNullo); (e.grafo() != G)→ Err(DiffGrafo);
            + {NOMEM→ ErrMem;}
      virtual Arco succAd ( const Arco& e ) const
                        throws (ArcoNullo, DiffGrafo,ErrMem) = 0
```

```
//    TIPI DI ERRORI
      class ArcoNullo: public Err<Grafo> {}; // arco nullo
      class IndNodo: public Err<Grafo> {}; // indice nodo errato
      class NonNullo: public Err<Grafo>  {}; // grafo non nullo
      class GrafoNullo: public Err<Grafo>  {}; // grafo non nullo
      class DiffGrafo: public Err<Grafo> {}; // arco non del grafo
      class DiffNodi: public Err<Grafo> {}; // numero di nodi differente
      protected:
      card vn, vm; // numero di nodi e numero di archi
      virtual void iniz () throw (ErrMem) = 0; // inizializzazione
};
```

**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.

```
Vettore<card> numAdiacenti( const Grafo& G )
{     Vettore<card> grado(1,G.n());
      for ( card i =1; i <= G.n(); i++ )
      {     Arco e; grado[i] = 0;
            for_each_ad(e,i,G) grado[i]++;
      }
      return grado;
}
```

La stessa funzione può essere riscritta nel seguente modo*:*

```
Vettore<card> numAdiacenti( const Grafo& G )
{     Vettore<card> grado(1,G.n());
      for ( card i =1; i <= G.n(); i++ )
      {     grado[i] = 0;
            for ( card j = 1; j <= G.n(); j++ )
                  if ( G(i,j) )
                        grado[i]++;
      }
      return grado;
}
```

Le due funzioni sono utilizzabili con qualunque tipo di grafo, orientato o non, e qualunque realizzazione, liste o matrice di adiacenza come vedremo tra poco. Tuttavia non è possibile effettuare un'analisi di complessità senza far riferimento a una particolare realizzazione.

Suddividiamo ora il concetto di grafo nelle sue due versioni, orientato e non, attraverso la definizione di due corrispondenti sottoclassi:

```
class GrafoOr: public virtual Grafo { };

class GrafoNOr: public virtual Grafo { };
```

La clausola "virtual" è stata introdotta poichè essa sarà necessaria allorquando introdurremo i grafi pesati. Anche queste due classi sono astratte per cui non possono essere usate come tipo di una variabile se non negli argomenti di una funzione passati per riferimento; esse sono state definite per permettere di scrivere funzioni che siano esclusive di uno dei due tipi di grafo.

**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 GrafoOr& G, GrafoNOr& G1) throws (Grafo::DiffNodi)
{     if ( G.n() != G1.n() ) throw Grafo::DiffNodi();
      G1.svuota();
      for ( card i = 1; i <= G.n(); i++ )
            for ( card j = 1; j <= G.n(); j++ )
                  if ( G(i,j) )
                        G1(i,j,true);
}
```

Poichè *GrafoNOr* è una classe astratta non è possibile definire il grafo *G1* nel corpo della funzione e poi restituirlo come argomento di ritorno.

La seguente versione della funzione "disOrienta" non fa uso dell'operatore *G(i,j)*:

```
void disOrienta( const GrafoOr& G, GrafoNOr& G1) throws (Grafo::DiffNodi)
{     if ( G.n() != G1.n() ) throw Grafo::DiffNodi();
      G1.svuota();
      for ( card i = 1; i <= G.n(); i++ )
      {     Arco e;
            for_each_arc(e,G)
                  G1(in(e),fin(e),true);
      }
}
```

In effetti la procedura precedente è già compresa nella definizione dell'operatore di assegnamento per cui, per "disorientare" un grafo, è sufficiente scrivere: *G1 = G*.

La seguente funzione determina il grado di entrata di ciascun nodo:

```
Vettore<card> gradoEntrata( const GrafoOr& G )
{     Vettore<card> grado(1,G.n()); Arco e;
      impostaVettore(grado,0) // azzera tutti gli elementi di grado
      for_each_arc(e,G) grado[fin(e)]++;
      return grado;
}
```

### 5.2.2 Rappresentazione di Grafi con Liste di Adiacenza

Un primo modo di rappresentare un grafo è attraverso le liste di adiacenza, cioè per ogni nodo viene memorizzata la lista dei nodi adiacenti. Questo modo risulta particolarmente efficace soprattutto nel caso di grafi sparsi, dove $m = O(n)$, in quanto, essendo rappresentati solo gli archi presenti, la complessità spaziale è $\Theta(m) = \Theta(n)$. Per grafi densi la complessità spaziale della rappresentazione con liste di adiacenza è $\Theta(m) = \Theta(n^2)$.

Cominciamo col definire la struttura a lista di adiacenza. In essa introduciamo il vettore adiacenti di liste di adiacenza. Le varie funzioni sono realizzate attraverso opportune operazioni su liste. Abbiamo scelto le liste ordinate per poter scandire gli archi in ordine lessicografico e per gestire più efficacemente la rimozione di eventuali duplicati. Per rendere più efficiente le funzioni di accesso agli archi tramite gli operatori '()', abbiamo introdotto un riferimento per ciascuna lista *i*

che memorizza l'ultimo elemento adiacente del nodo *i* utilizzato dagli operatori. Per poter aggiornare tale riferimento nella funzione costante arco, dobbiamo definire un alias non costante per il riferimento.

```
// file listead.h
#ifndef LISTAADH
#define LISTAADH
#include "listaord.h"
#include "grafo.h"
#include "vettore.h"
class ListeAd
{     protected:
      typedef ListaOrd<card> Adiacenti;
      Vettore<Adiacenti> adiacenti; // vettore di liste di adiacenza
      typedef ElemLista<card> ELA
      Vettore<ELA> eLA; // elementi correnti delle liste di adiacenza
      void primoAd( card i, card& j, void* & pos ) const throws (ErrMem)
      {     if ( !adiacenti[i].vuota() )
            {     j = adiacenti[i].primo().info();
                  // riferimento al primo adiacente
                  ELA* pre = new ELA; *pre = adiacenti[i].inizio();
                  // pre viene trasformato in un puntatore a vuoto
                  pos = (void*) pre;
            }
      }
      void succAd( card i, card& j, void* & pos ) const
                              throws (ArcoNullo, DiffGrafo,ErrMem)
      {     // pos viene trasformato in un puntatore a riferimento di lista
            ELA* pre = (ELA *) pos;
            if ( &(pos->lista()) != & adiacenti[i] ) DiffGrafo();
            *pre = succ(*pre);
            if ( !pre->fine() )
            {     j = pre->info();
                  // pre viene trasformato in un puntatore a vuoto
                  pos = (void*) pre;
            }
            else
                  pos = 0;
      }
      void modArco( card i, card j, bool b )
      {     if ( b )
                  adiacenti[i].ins(j,eLA[i]);
            else
                  adiacenti[i].canc(j,eLA[i]);
      }
      bool arco( card i, card j ) const
      {     ELA* ei_alias = (ELA *) (&eLA[i]);
            return adiacenti[i].cerca(j,ei_alias);
      }
      void svuota()
      {     for ( card i = 1; i <=  adiacenti.n(); i++ )
            {     adiacenti[i].svuota(); eLA[i] = adiacenti[i].inizio();
            }
      }
      void init( card nNo ) throws (ErrMem):
      {     adiacenti.init(1,nNo); eLA.init(1,nNO);
      {     for ( card i = 1; i <= nNo; i++ )
                  eLA[i] = adiacenti[i].inizio();
      }
      ListeAd( ) {};
};
```

Presentiamo ora la classe dei grafi orientati con rappresentazione a liste di adiacenza:

```
// file grafoorl.h
#ifndef GRAFOORLH
#define GRAFOORLH
#include "grafo.h"
#include "listead.h"
class GrafoOrL: public GrafoOr, private  ListeAd
{     public:
      Arco primoArco ( ) const throws (ErrMem)
      {     Arco et;
            for ( et.i = 1; et.i<=n() && et.pos==0; et.i++ )
                  ListeAd::primoAd(et.i, et.j, et.pos);
            if ( et.pos != 0 ) et.pgrafo = this;
            return et;
      }
      Arco succArco( const Arco& e ) const throws(ArcoNullo, DiffGrafo,ErrMem)
      {     if ( !e ) throw ArcoNullo();
            if ( e.pgrafo != this ) throw DiffGrafo();
            Arco et = e; ListeAd::succAd(et.i,et.j,et.pos);
            if ( et.pos == 0 )
                  for ( et.i = e.i+1; et.i<=n() && et.pos==0; et.i++ )
                        ListeAd::primoAd(et.i, et.j, et.pos);
            if ( et.pos != 0 ) et.pgrafo = this;
            return et;
      }
      Arco primoAd( card i ) const throws (IndNodo,ErrMem)
      {     if ( i < 1 || i > n() ) throw IndNodo();
            Arco et; et.i = i; ListeAd::primoAd(et.i, et.j, et.pos);
            if ( et.pos != 0 ) et.pgrafo = this;
            return et;
      }
      Arco succAd( const Arco& e ) const throws (ArcoNullo, DiffGrafo,ErrMem)
      {     if ( !e ) throw ArcoNullo();
            if ( e.pgrafo != this ) throw DiffGrafo();
            Arco et = e; ListeAd::succAd(et.i, et.j, et.pos);
            if ( et.pos != 0 ) et.pgrafo = this;
            return et;
      }
      void operator()( card i, card j, bool b ) throws (IndNodo)
      {     if ( i <1 || i>n() || j<1 || j>n() ) throw IndNodo();
            bool esistearco = arco(i,j);
            if ( (!esistearco && b) || (esistearco && !b) )
            {     modArco(i,j,b);
                  if ( b ) vm++ else vm--;
            }
      }
      bool operator() ( card i, card j ) const throws (IndNodo)
      {     if ( i <1 || i>n() || j<1 || j>n() ) throw IndNodo();
            return arco(i,j);
      }
      void svuota() {ListeAd::svuota(); vm = 0;}
      GrafoOrL operator= ( const Grafo& G ) throws (DiffNodi,ErrMem)
      {     Grafo::operator=(G); return *this;
      }
      GrafoOrL( card nNo = 0 ) throws (ErrMem): Grafo(nNo) { iniz(); }
      protected:
      void iniz () throws (ErrMem) {ListaAd::init(n());}
};
```

Poichè la classe di base *Grafo* è stata dichiarata "virtual", il suo costruttore può essere chiamato direttamente anche se *GrafoOrL* è derivata da essa indirettamente attraverso *GrafoOr*. Il costruttore di *GrafoOrL* ha complessità $\Theta(n)$. La funzione *svuota* e l'operator "=" hanno complessità $\Theta(\max(n,m))$. Poichè in generale $m = \Omega(n)$, $\Theta(\max(n,m))$ è uguale a $\Theta(m)$. D'ora in poi assumeremo che $m = \Omega(n)$; in effetti un grafo che non rispetti tale vincolo è ben raro — significa che ha un numero costante o sublineare di archi. Le due versioni dell'operatore "()" hanno complessità costante nel caso migliore e complessità $\Theta(n)$ nel caso peggiore. Nel caso medio la complessità varia da costante per i grafi sparsi a $\Theta(n)$ per i grafi densi. Tutte le altre funzioni hanno complessità costante.

E' interessante ora valutare una volta per tutte la complessità della seguente porzione di codice che scandisce gli adiacenti di un nodo *i* in un grafo orientato con struttura a liste di adiacenza:

```
for ( card j = 1; j <= G.n(); j++ )
      ... G(i,j)...// o G(i,j,b)
```

Poichè $G(i,j)$ ha complessità $\Theta(n)$ nel caso peggiore e il ciclo è ripetuto $n$ volte, sembrerebbe che la complessità nel caso peggiore sia $\Theta(n^2)$. Questo non è vero! Infatti, grazie alla presenza dei riferimenti interni delle liste di adiacenza, la ricerca parte dall'ultimo adiacente trovato senza ritornare mai all'inizio della lista. Pertanto la complessità nel caso peggiore (e ovviamente anche nei casi migliore e medio) è $\Theta(n)$. La scansione degli adiacenti di un nodo può essere effettuata anche con la seguente porzione di codice:

```
for_each_ad(e,i,G)
      ...e... // utilizzo dell'arco uscente dal nodo i
```

che ha una complessità $\Theta(k)$ dove k è esatttamente il numero di adiacenti di *i*, per cui tale tipo di scansione risulta più efficiente. Ripetendo la scansione per ogni nodo,riusciamo a scandire tutti gli archi in tempo $\Theta(\max(n,m))$. La stessa complessità si ha utilizzando l'iteratore:

```
for_each_arc(e,G)
```

Per scandire i nodi entranti in un nodo non è possibile utilizzare l'iteratore del grafo per cui bisogna ricorrere a questa porzione di codice:

```
for ( card j = 1; j <= G.n; j++ )
      ... G(j,i)...
```

Poichè la ricerca avviene sulle liste di adiacenze dei nodi *j*, la complessità di $G(j,i)$ va moltiplicata per *n*, cioè è $\Theta(n^2)$ nel caso peggiore, $\Theta(n)$ nel caso migliore, $\Theta(n)$ nel caso medio per grafi sparsi e $\Theta(n^2)$ nel caso medio per grafi densi.

**Esempio 3.** Consideriamo la seguente funzione nel caso sia chiamata passando come argomento un grafo orientato con struttura a lista:

```
Vettore<card> XX( const GrafoOr& G )
{     Vettore<card> x(1,G.n());
      for ( card i =1; i <= G.n(); i++ )
      {     x[i] = 0;
            for ( j = 1; j <= G.n(); j++ )
```

```
                  if ( G(j,i) )
                        x[i]++;
      }
      return x;
}
```

Per quanto detto precedentemente il ciclo più interno ha complessità $\Theta(n^2)$ nel caso peggiore; per cui moltiplicando per $n$ il numero di iterazioni del ciclo più esterno, si ottiene che la complessità della funzione è $\Theta(n^3)$ nel caso peggiore. Nel caso migliore è $\Theta(n^2)$ e nel caso medio varia da $\Theta(n^2)$, grafi sparsi, a $\Theta(n^3)$, grafi densi.

Consideriamo ora la seguente funzione:

```
Vettore<card> YY( const GrafoOr& G )
{     Vettore<card> x(1,G.n());
      for ( card i =1; i <= G.n(); i++ )
            x[i] = 0;
      for ( card i =1; i <= G.n(); i++ )
            for ( j = 1; j <= G.n(); j++ )
                  if ( G(i,j) )
                        x[j]++;
      }
      return x;
}
```

Questa volta il ciclo più interno ha una complessità $\Theta(n)$ per cui moltiplicando per il numero di iterazione si ottiene che la funzione ha complessità $\Theta(n^2)$ nei tre casi.

Le due funzioni calcolano i gradi di entrata dei nodi di un grafo così come la funzione riportata nell'Esempio 2. La complessità di quest'ultima funzione per il caso di un grafo orientato con liste di adiacenza è $\Theta(m)$, cioè essa è l'algoritmo ottimo.

Passiamo ora alla rappresentazione dei grafi non orientati attravero liste di adiacenza. La realizzazione è del tutto uguale a quella dei grafi orientati tranne che nel costruttore in cui bisogna indicare che si tratta di grafo non orientato. La complessità delle funzioni non varia.

```
// file grafonorl.h
#ifndef GRAFONORLH
#define GRAFONORLH
#include "grafo.h"
#include "listead.h"
class GrafoNOrL: public GrafoOr, private ListeAd
{     public:
      Arco primoArco( ) const throws (ErrMem)
      {     Arco et;
            for ( et.i = 1; et.i<=n() && et.pos==0; et.i++ )
                  ListeAd::primoAd(et.i, et.j, ec.pos);
            if ( et.pos != 0 ) et.pgrafo = this;
            return et;
      }
      Arco succArco( const Arco& e ) const throws(ArcoNullo, DiffGrafo,ErrMem)
      {     if ( !e ) throw ArcoNullo();
            if ( e.pgrafo != this ) throw DiffGrafo();
            Arco et = e; ListeAd::succAd(et.i,et.j,et.pos);
            if ( et.pos == 0 )
```

```
                for ( et.i = e.i+1; et.i<=n() && et.pos==0; et.i++ )
                      ListeAd::primoAd(et.i, et.j, ec.pos);
          if ( et.pos != 0 ) et.pgrafo = this;
          return et;
    }
    Arco primoAd( card i ) const throws (IndNodo,ErrMem)
    {     if ( i < 1 || i > n() ) throw IndNodo();
          Arco et; et.i = i; ListeAd::primoAd(et.i, et.j, et.pos);
          if ( et.pos != 0 ) et.pgrafo = this;
          return et;
    }
    Arco succAd( const Arco& e ) const throws (ArcoNullo, DiffGrafo,ErrMem)
    {     if ( !e ) throw ArcoNullo();
          if ( e.pgrafo != this ) throw DiffGrafo();
          Arco et = e; ListeAd::succAd(et.i, et.j, et.pos);
          if ( et.pos != 0 ) et.pgrafo = this;
          return et;
    }
    void operator()( card i, card j, bool b ) throws (IndNodo)
    {     if ( i <1 || i>n() || j<1 || j>n() ) throw IndNodo();
          bool esistearco = arco(i,j);
          if ( (!esistearco && b) || (esistearco && !b) )
          {     modArco(i,j,b); modArco(j,i,b);
                if ( b ) vm++ else vm--;
          }
    }
    void operator()( card i, card j, bool b ) throws (IndNodo)
    {     if ( i <1 || i>n() || j<1 || j>n() ) throw IndNodo();
          return arco(i,j);
    }
    void svuota() {ListeAd::svuota(); vm = 0;}
    GarfoNOrL operator= ( const Grafo& G ) throws (DiffNodi,ErrMem)
    {     Grafo::operator=(G); return *this;
    }
    GrafoNOrL( card nNo = 0 ) throws (ErrMem): Grafo(nNo) { iniz();}
    protected:
    void iniz () throws (ErrMem) {ListaAd::init(n());}
};
```

**Esempio 4.** Consideriamo le due funzioni *numAdiacenti* dell'Esempio 1. Nel caso di grafi con struttura a liste di adiacenza, la prima versione di *numAdiacenti* ha complessità $\Theta(m)$ mentre la seconda versione ha complessità $\Theta(n^2)$.

### 5.2.3 Rappresentazione di Grafi con Matrici di Adiacenza

Un secondo modo di rappresentare un grafo è attraverso matrice di adiacenza; cioè per ogni coppia di nodi $(i,j)$ viene memorizzato vero o falso a secondo se esista l'arco $(i,j)$ o meno. La complessità spaziale della rappresentazione con matrice di adiacenza è $\Theta(n^2)$.

Definiamo per prima la classe privata *MatriceAd*. La rappresentazione è notevolmente semplificata rispetto a quella con liste di adiacenza.

```
// file matrad.h
#include "matrsimm.h"
class MatriceAd
{     protected:
```

```
      const bool eOrientato;
      Matrice<bool>* parchi; // puntatore a matrice di adiacenza
      void primoAd( card i, card& j ) const
      {     for ( card k = 1; k<=parchi->n() && j == 0; k++ )
                  if ( (*parchi) (i,k) )
                        j = k;
      }
      void succAd( card i, card& j ) const
      {     card k=j+1;  j=0;
            for ( ; k<=parchi->n() && j == 0; k++ )
                  if ( (*parchi)(i,k) )
                        j = k;
      }
      void modArco( card i, card j, bool b )
      {     (*parchi)(i,j) = b;
      }
      bool arco( card i, card j ) const
      {     return (*parchi)(i,j);
      }
      void svuota()
      {     for ( card i = 1; i <= parchi->n(); i++ )
            {     card k = eOrientato? parchi->n(): i;
                  for ( card j = 1; j <= k; j++ )
                        (*parchi)(i,j) = false;
            }
      }
      void init( card nNo ) throws (ErrMem):
      {     if ( eOrientato )
                  pArchi = new Matrice<bool>(1,nNo,1,nNo);
            else
                  pArchi = new MatriceSimm<bool>(1,nNo);
            svuota();
      }
      MatriceAd( card nNo = 0, bool or = true ): eOrientato(or), parchi(0) {}
      ~MatriceAd() {delete parchi;}
};
```

Presentiamo ora la classe dei grafi orientati con rappresentazione a matrice di adiacenza:

```
// file grafoorm.h
#ifndef GRAFOORMH
#define GRAFOORMH
#include "grafo.h"
#include "matrad.h"
class GrafoOrM: public GrafoOr, private MatriceAd
{     public:
      Arco primoArco( ) const throws (ErrMem)
      {     Arco et; et.j=0;
            for ( card i = 1; i<=n() && et.j==0; i++ )
            {     et.i=i;
                  MatriceAd::primoAd(et.i, et.j);
            }
            if ( et.j != 0 ) et.pgrafo = this;
            return et;
      }
      Arco succArco( const Arco& e ) const throws(ArcoNullo, DiffGrafo,ErrMem)
      {     if ( !e ) throw ArcoNullo();
            if ( e.pgrafo != this ) throw DiffGrafo();
            Arco et; et.i = e.i; et.j = e.j;
            MatriceAd::succAd(et.i,et.j);
            if ( et.j == 0 )
```

```
                for ( card i = e.i+1; i<=n() && et.j==0; i++ )
                {     et.i=i;
                      MatriceAd::primoAd(et.i, et.j);
                }
          if ( et.j != 0 ) et.pgrafo = this;
          return et;
    }
    Arco primoAd( card i ) const throws (IndNodo,ErrMem)
    {     if ( i < 1 || i > n() ) throw IndNodo();
          Arco et; et.i = i; et.j = 0;
          MatriceAd::primoAd(et.i, et.j);
          if ( et.j != 0 ) et.pgrafo = this;
          return et;
    }
    Arco succAd( const Arco& e ) const throws (ArcoNullo, DiffGrafo,ErrMem)
    {     if ( !e ) throw ArcoNullo();
          if ( e.pgrafo != this ) throw DiffGrafo();
          Arco et; et.i = e.i; et.j = e.j;
          MatriceAd::succAd(et.i, et.j);
          if ( et.j != 0 ) et.pgrafo = this;
          return et;
    }
    void operator()( card i, card j, bool b )  throws (IndNodo)
    {     if ( i <1 || i>n() || j<1 || j>n() ) throw IndNodo();
          bool esistearco = arco(i,j);
          if ( (!esistearco && b) || (esistearco && !b) )
          {     modArco(i,j,b);
                if ( b ) vm++ else vm--;
          }
    }
    bool operator() ( card i, card j ) const throws (IndNodo)
    {     if ( i <1 || i>n() || j<1 || j>n() ) throw IndNodo();
          return arco(i,j);
    }
    void svuota() {MatriceAd::svuota(); vm=0;}
    GrafoOrL operator= ( const Grafo& G ) throws (DiffNodi,ErrMem)
    {     Grafo::operator=(G); return *this;
    }
    GrafoOrM( card nNo = 0 ) throws(ErrMem): Grafo(nNo), MatriceAd(true)
    {iniz(); }
    protected:
    void iniz () throw (ErrMem) {MatriceAd::init(n());}
};
```

Anche in questo caso è richiamato direttamente il costruttore di *Grafo*. Tutte le funzioni hanno complessità costante tranne:

- il costruttore, l'operatore "=", *m* e *svuota*, che hanno complessità $\Theta(n^2)$;
- le funzioni *primoArco* e *succArco*, che hanno complessità costante nel caso migliore, e complessità $\Theta(n^2)$ nel caso peggiore; nel caso medio la complessità varia da costante per i grafi densi a $\Theta(n^2)$ per i grafi sparsi.
- le funzioni *primoAd* e *succAd*, che hanno complessità costante nel caso migliore, e complessità $\Theta(n)$ nel caso peggiore; nel caso medio la complessità varia da costante per i grafi densi a $\Theta(n)$ per i grafi sparsi.

Consideriamo le varie modalità di scandire gli adiacenti di un nodo *i* di un grafo con rappresentazione a lista:

```
for ( card j = 1; j <= G.n(); j++ )
      ... G(i,j)...

for_each_ad(e,i,G)
      ...e... // utilizzo dell'arco uscente dal nodo i

for ( card j = 1; j <= G.n(); j++ )
      ... G(j,i)...
```

La complessità delle tre modalità con la rappresentazione a matrice di adiacenza è la stessa nei tre casi, cioè $\Theta(n)$ . Ciò significa che nella scansione degli archi di un grafo è la stessa cosa dal punto di vista dell'efficienza accedere a tutti i possibili archi o utilizzare l'iteratore. Poichè quest'ultima modalità è più efficiente nel caso di uso di liste di adiacenza, adotteremo questa modalità nel descrivere algoritmi generici in cui non si fa riferimento alla particolare realizzazione.

**Esempio 5.** Consideriamo le due funzioni *XX*e *YY* dell'Esempio 3. Nel caso di grafi con struttura a matrice di adiacenza, ambedue le funzioni hanno complessità $\Theta(n^2)$.

Concludiamo l'analisi dei grafi con la rappresentazione dei grafi non orientati realizzati con matrice di adiacenza. La simmetria degli archi permette di adoperare la rappresentazione della matrice simmetrica descritta nel Capitolo 2. Nonostante il risparmio di quasi metà dello spazio, la complessità spaziale rimane $\Theta(n^2)$.

```
// file grafonom.h
#ifndef GRAFONORMH
#define GRAFONORMH
#include "grafo.h"
#include "matrad.h"
class GrafoNOrM: public GrafoNOr, private MatriceAd
{     public:
      Arco primoArco( ) const throws (ErrMem)
      {     Arco et; et.j=0;
            for ( card i = 1; i<=n() && et.j==0; i++ )
            {     et.i=i;
                  MatriceAd::primoAd(et.i, et.j);
            }
            if ( et.j != 0 ) et.pgrafo = this;
            return et;
      }
      Arco succArco( const Arco& e ) const throws(ArcoNullo, DiffGrafo,ErrMem)
      {     if ( !e ) throw ArcoNullo();
            if ( e.pgrafo != this ) throw DiffGrafo();
            Arco et; et.i = e.i; et.j = e.j;
            MatriceAd::succAd(et.i,et.j);
            if ( et.j == 0 )
                  for ( card i = e.i+1; i<=n() && et.j==0; i++ )
                  {     et.i=i;
                        MatriceAd::primoAd(et.i, et.j);
                  }
            if ( et.j != 0 ) et.pgrafo = this;
            return et;
      }
      Arco primoAd( card i ) const throws (IndNodo,ErrMem)
```

```
      {     if ( i < 1 || i > n() ) throw IndNodo();
            Arco et; et.i = i; et.j = 0;
            MatriceAd::primoAd(et.i, et.j);
            if ( et.j != 0 ) et.pgrafo = this;
            return et;
      }
      Arco succAd( const Arco& e ) const throws (ArcoNullo, DiffGrafo,ErrMem)
      {     if ( !e ) throw ArcoNullo();
            if ( e.pgrafo != this ) throw DiffGrafo();
            Arco et; et.i = e.i; et.j = e.j;
            MatriceAd::succAd(et.i, et.j);
            if ( et.j != 0 ) et.pgrafo = this;
            return et;
      }
      void operator()( card i, card j, bool b ) throws (IndNodo)
      {     if ( i <1 || i>n() || j<1 || j>n() ) throw IndNodo();
            bool esistearco = arco(i,j);
            if ( (!esistearco && b) || (esistearco && !b) )
            {     modArco(i,j,b); // stessa di non orientato
                  if ( b ) vm++; else vm--;
            }
      }
      bool operator() ( card i, card j ) const throws (IndNodo)
      {     if ( i <1 || i>n() || j<1 || j>n() ) throw IndNodo();
            return arco(i,j);
      }
      void svuota() {MatriceAd::svuota(); vm = 0;}
      GrafoOrL operator= ( const Grafo& G ) throws (DiffNodi,ErrMem)
      {     Grafo::operator=(G); return *this;
      }
      GrafoNOrM( card nNo = 0 ) throws (ErrMem):Grafo(nNo),MatriceAd(false)
      { iniz(); }
      protected:
      void iniz () throw (ErrMem) {MatriceAd::init(n());}
};
```

La complessità delle varie funzioni non varia rispetto a quella delle omonime funzioni per i grafi orientati con matrice di adiacenza.

**Esempio 6.** Consideriamo le due versioni di *numAdiacenti* dell'Esempio 1. Nel caso di grafi con struttura a matrice di adiacenza, ambedue le versioni hanno complessità $\Theta(n^2)$.

## 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*, 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*, 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 7.** 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.

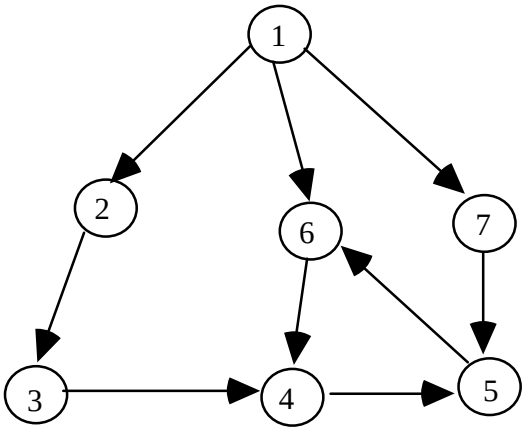



Figura 2. *Grafo orientato*

Scriviamo ora le due funzioni di visita in modo che siano utilizzabili per un grafo generico, orientato e non, con rappresentazione a liste o a matrice di adiacenza. Per come abbiamo organizzato le varie classi di grafo, è sufficiente utilizzare la classe astratta *Grafo*.

Cominciamo con la visita a ventaglio. La funzione restituisce un vettore di booleani in cui un elemento *i* è 'true' se e solo se esso è raggiungibile dal nodo *v*.

```
VetBool visitaVentaglio( const Grafo& G, card v ) throws (Grafo::GrafoNullo)
{     if ( G.n() == 0 ) throw Grafo::GrafoNullo();
      VetBool visitato(1,G.n(),false);
      Coda<card> C;
      // visita e incoda v
      visitato[v] = true; C.inCoda(v);
      while ( !C.vuota() )
      {     card i = C.fuoriCoda(); Arco e;
            // visita e incoda tutti gli adiacenti di i
            for_each_ad (e,i,G)                                          //*
                  if ( !visitato[fin(e)] )                               //*
                  {     C.inCoda(fin(e)); visitato[fin(e)] = true;       //*
                  }                                                      //*
      }
      return visitato;
}
```

La complessità del problema della visita a ventaglio ha complessità $\Omega(m)$ poichè dal nodo iniziale potrebbero essere raggiungibili tutti i nodi e trascurare di prendere in considerazione un arco potrebbe impedire di raggiungere qualche nodo. La complessità della funzione dipende dal blocco di istruzioni indicate con asterisco. Esse sono eseguite $O(n)$ volte, dove $n$ è il numero di nodi.

Nel caso di grafo con matrice di adiacenza, il blocco asteriscato ha complessità $\Theta(n)$; pertanto la complessità della funzione è $O(n^2)$ che corrisponde a $O(m)$ solo nel caso di grafi densi; in particolare la complessità è $\Theta(n^2)$ nel caso peggiore e medio e $\Theta(n)$ nel caso migliore.

Nel caso di grafo con liste di adiacenza, il blocco asteriscato è eseguito $\Theta(k_x)$ volte dove $k_x$ è il numero di adiacenti del nodo $x$. Poichè il blocco è ripetuto al più per ogni nodo del grafo, la complessità è

$$O(\sum_x k_x) = O(m)$$

cioè l'algoritmo è ottimo in questo caso. In particolare abbiamo che la complessità nel caso peggiore e medio è $\Theta(m)$ mentre nel caso migliore è $\Theta(1)$.

L'algoritmo può essere riscritto sostituendo al blocco di istruzioni indicate da "//*" dal seguente blocco:

```
for ( card j = 1; j <= G.n(); j++ )
      if ( G(i,j) && !visitato[j] )
      {     C.inCoda(j); visitato[j] = true;
      }
```

Con questa variazione, la complessità rimane $O(n^2)$ nel caso di grafo a matrice di adiacenza mentre passa da $O(m)$ a $O(n^2)$ nel caso di grafo a liste di adiacenza.

Di seguito presentiamo una funzione di visita a ventaglio che restituisce anche la lista dei nodi visitati nella sequenza di visita:

```
VetBool visitatiVentaglio( const Grafo& G, card v, Lista<card>& L )
                                    throws (Grafo::GrafoNullo)
{     if ( G.n() == 0 ) throw Grafo::GrafoNullo();
      L.svuota();
      VetBool visitato(1,G.n(),false);
      Coda<card> C;
      C.inCoda(v); visitato[v] = true;
      while ( !C.vuota() )
      {     card i = C.fuoriCoda(); Arco e;
            for_each_ad(e,i,G)
                  if ( !visitato[fin(e)] )
                  {     C.inCoda(fin(e)); visitato[fin(e)] = true;
                        L.appendi(fin(e));
                  }
      }
      return visitato;
}
```

Scriviamo ora la funzione della visita a scandaglio. In essa la coda va sostituita da una pila per realizzare il diverso ordine di visita. Inoltre in essa gli adiacenti di un nodo vanno scanditi uno alla volta e non in blocco come per la visita a ventaglio.

```
VetBool visitaScandaglio( const Grafo& G, card v ) throws (Grafo::GrafoNullo)
{     if ( G.n() == 0 ) throw Grafo::GrafoNullo();
      VetBool visitato(1,G.n(),false);
      Pila<Arco> P;
      visitato[v] = true; // visita v
      // inpila il primo adiacente di v
      P.inPila(G.primoAd(v));
      while ( !P.vuota() )
      {     // estrae il primo arco dalla pila
            Arco e = P.fuoriPila();
            if ( e )
            {     // inpila il successivo adiacente
                  P.inPila(G.succAd(e));
                  if ( !visitato[fin(e)] )
                  {     visitato[fin(e)] = true; // visita fin(e)
                        // inpila il primo adiacente di fin(e)
                        P.inpila(G.primoAd(fin(e));
                  }
            }
      }
      return visitato;
}
```

La complessità della funzione *visitaScandaglio* dipende dal numero di volte che è ripetuto il ciclo *while*. Nel caso di grafo a matrice di adiacenza è $n^2$ mentre nel caso di grafo a liste di adiacenza è $m$. Pertanto le rispettive complessità sono $\Theta(n^2)$ e $\Theta(m)$.

Di seguito presentiamo una versione ricorsiva della visita a scandaglio:

```
void visitaScandaglioRic1( const Grafo& G, card u, VetBool& visitato )
{     visitato[u] = true; Arco e;
      for_each_ad(e,u,G)
            if ( !visitato[fin(e)] )
                  visitaScandaglioRic1(G,fin(e),visitato);
}
VetBool visitaScandaglioRic( const Grafo& G, card v )
                                          throws (Grafo::GrafoNullo)
{     if ( G.n() == 0 ) throw Grafo::GrafoNullo();
{     VetBool visitato(1,G.n(),false);
      visitaScandaglioRic1(G,v,visitato);
      return visitato;
}
```

Utilizzando uno dei due tipi di visita è possibile risolvere un problema rilevante in un grafo orientato e non: la ricerca di un cammino tra due nodi. Questo problema è facile nel senso che è risolvibile in tempo polinomiale ma attenzione che il problema di ricercare tutti i cammini tra due nodi non lo è poichè il numero di cammini potrebbe essere uguale a

$$\sum_{i=2}^{n-1}(n-i)!$$

che è un numero esponenziale in quanto $n! = \Theta(n^n)$ per la nota formula di Stirling:

$$n! \approx \sqrt{2\pi n}\, n^n e^{-n}.$$

Per ricercare un cammino dal nodo $v$ al nodo $w$ viene effettuata una visita a partire da $v$ che viene arrestata non appena si arriva al nodo $w$. Durante la visita, viene memorizzato per ogni nodo $u$ visitato, il nodo $x$ di cui esso è adiacente (nodo precedente); una volta giunti su $w$, si passa al suo

precedente, da questo al suo precedente fino a ritornare a v, costruendo così un cammino tra i due nodi.

```
bool trovaCammino( const Grafo& G, card v, card w, Lista<card>& cammino )
                                                  throw (Grafo::GrafoNullo)
{     if ( G.n() == 0 ) throw Grafo::GrafoNullo();
      cammino.svuota();
      Pila<Arco> P;
      Vettore<card> precedente(1,G.n());
      impostaVettore(precedente,0);
      precedente[v] = v;
      P.inPila(G.primoAd(v));
      bool trovato = v == w;
      while ( !P.vuota() && !trovato )
      {     Arco e = P.fuoriPila();
            if ( e )
            {     trovato = fin(e) == w;
                  if ( !trovato ) P.inPila(G.succAd(e));
                  if ( precedente[fin(e)] == 0 )
                  {     precedente[fin(e)] = in(e);
                        if ( !trovato ) P.inPila(G.primoAd(fin(e));
                  }
            }
      }
      if ( trovato )
      {     card u;
            do
            {     cammino.intesta(w);  u = w;
                  w = precedente[w];
            }
            while ( u != v );
      }
      return trovato;
}
```

La funzione *trovaCammino* ha complessità $\Theta(n^2)$ e $\Theta(m)$ rispettivamente nei casi di grafo a matrice e a lista di adiacenza. Infatti la parte aggiuntiva di codice rispetto alla funzione *visitaScandaglio* non è altro che la creazione di una lista con al più $n$ elementi che è ovviamente effettuata in tempo $\Theta(n)$ per cui la complessità delle due funzioni non varia.

La seguente funzione verifica semplicemente l'esistenza di un cammino tra due nodi:

```
bool esisteCammino ( const Grafo& G, card v, card w )
                                 throw (Grafo::GrafoNullo)
{     if ( G.n() == 0 ) throw Grafo::GrafoNullo();
      Pila<Arco> P;
      VetBool visitato(1,G.n(),false);
      visitato[v] = true;
      P.inPila(G.primoAd(v));
      bool trovato = v == w;
      while ( !P.vuota() && !trovato )
      {     Arco e = P.fuoriPila();
            if ( e )
            {     trovato = fin(e) == w;
                  if ( !trovato )
                  {     P.inPila(G.succAd(e));
                        if ( !visitato[fin(e)] )
                        visitato[fin(e)] = true;
                        P.inPila(G.primoAd(fin(e));
```

```
                    }
              }
        }
        return trovato;
}
```

La funzione ha ovviamente la stessa complessità della funzione *visitaScandaglio*, cioè $\Theta(n^2)$ e $\Theta(m)$ rispettivamente nei casi di grafo a matrice e a lista di adiacenza.

### 5.3.2 Algoritmi su Grafi Non Orientati

Un utilizzo importante della visita di un grafo non orientato è nella risoluzione del problema di verificare se il grafo sia connesso o meno. A tale scopo è sufficiente visitare il grafo a partire da un nodo arbitrario, ad esempio il nodo 1, e verificare che siano stati visitati tutti i nodi del grafo.

```
bool eConnesso( const GrafoNOr& G ) throws (Grafo::GrafoNullo)
{     if ( G.n() == 0 ) throw Grafo::GrafoNullo();
      VetBool raggiunto = visitaVentaglio(G,1);
      bool connesso = true;
      for ( card i = 1; i <= G.n() && connesso; i++ )
            if ( !raggiunto[i] )
                  connesso = false;
      return connesso;
}
```

Ovviamente avremmo potuto usare la funzione *visitaScandaglio* al posto di *visitaVentaglio*. L'analisi di complessità coincide con quella di *visitaVentaglio*, cioè $O(n^2)$ e $O(m)$ rispettivamente nei casi di grafo a matrice e a lista di adiacenza.

La seguente funzione verifica se un grafo non orientato sia un albero o meno :

```
bool eAlbero( const GrafoNOr& G ) throws (Grafo::GrafoNullo)
{     if ( G.n() == 0 ) throw Grafo::GrafoNullo();
      return eConnesso(G) && G.m() == G.n()-1;
}
```

Ancora una volta la complessità è $\Theta(n^2)$ e $\Theta(m)$ rispettivamente nei casi di grafo a matrice e a lista di adiacenza.

Vediamo ora come sia possibile determinare un albero ricoprente un grafo non orientato segnalando con true o false a secondo che un albero ricoprente effettivamente esista, cioè se il grafo sia connesso o meno. Ancora una volta si tratta di una visita a partire da un nodo arbitrario, ad esempio il nodo 1: l'albero ricoprente è costruito con gli archi utilizzati nella visita.

```
bool alberoRicoprente ( const GrafoNOr& G, GrafoNor& A )
                              throws (Grafo::GrafoNullo, Grafo::DiffNodi)
{     if ( G.n() == 0 ) throw Grafo::GrafoNullo();
      if ( G.n() != A.n() ) throw Grafo::DiffNodi();
      A.svuota();
      Coda<card> C;
      VetBool visitato(1,G.n(),false);
      C.inCoda(1); visitato[1] = true; card nArchi = 0;
      while ( !C.vuota() )
      {     card x = C.fuoriCoda(); Arco e;
```

```
            for_each_ad(e,x,G)
                  if ( !visitato[fin(e)] )
                  {     C.inCoda(fin(e));
                        visitato[fin(e)] = true;
                        A(x,fin(e),true); nArchi++;
                  }
      }
      return nArchi == (A.n() - 1);
}
```

E' superfluo dire che la complessità è $O(n^2)$ e $O(m)$ rispettivamente nei casi di grafo a matrice e a lista di adiacenza.

Vediamo adesso come determinare il numero di componenti connesse di un grafo non orientato. A tale scopo, visitiamo il grafo a partire da un nodo iniziale arbitraro, ad esempio il nodo 1 in modo da determinare una prima componente connessa attraverso la visita del grafo a partire dal nodo prescelto; quindi selezioniamo uno dei nodi non in questa componente e determiniamo una seconda componente e così via.

```
card numCompConnesse ( const GrafoNOr& G ) throws (Grafo::GrafoNullo)
{     if ( G.n() == 0 ) throws Grafo::GrafoNullo();
      VetBool visitato(1,G.n(),false);
      card nComp = 0; card nodoScelto = 1; // prima scelta
      do
      {     VetBool raggiunto = visitaVentaglio(G,nodoScelto);
            nComp++; nodoScelto = 0;
            for ( card u = 1; u <= G.n(); u++ )
            {     visitato[u] = visitato[u]||raggiunto[u];
                  if ( !visitato[u] )
                        nodoScelto = u; // successiva scelta
            }
      }
      while ( nodoScelto > 0 );
      return nComp;
}
```

La complessità della funzione è $O(n^2)$ nei casi di grafi con rappresentazione sia a matrice sia con lista di adiacenza.

La seguente funzione, oltre a calcolare il numero di componenti connesse, restituisce in *LC* la lista delle componenti, ciascuna delle quali è rappresentata come lista di interi.

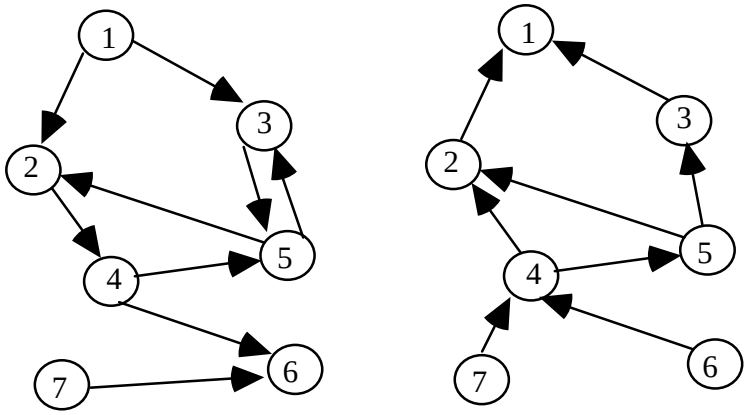
```
typedef Lista<card> Componente
card compConnesse ( const GrafoNOr& G, Lista<Componente>& LC )
                                    throws (Grafo::GrafoNullo)
{     if ( G.n() == 0 ) throw Grafo::GrafoNullo();
      LC.vuota();
      VetBool visitato(1,G.n(),false);
      card nComp = 0; card nodoScelto = 1;
      do
      {     Componente C;  nComp++;
            VetBool raggiunto = visitatiVentaglio(G,nodoScelto,C);
            LC.appendi(C); nComp++;
            nodoScelto = 0;
            for ( card u = 1; u <= G.n(); u++ )
            {     visitato[u] = visitato[u] || raggiunto[u];
                  if ( !visitato[u] ) nodoScelto = u;
```

```
            }
        }
        while ( nodoScelto > 0 );
        return nComp;
}
```

La complessità è $\Theta(n^2)$ nei casi di grafo a matrice e a lista di adiacenza.

La seguente funzione verifica se un grafo non orientato sia una foresta di alberi non orientati. La complessità è $\Theta(n^2)$ nei casi di grafo a matrice e a lista di adiacenza.

```
bool eForesta( const GrafoNOr& G ) throws (Grafo::GrafoNullo)
{       if ( G.n() == 0 ) throw Grafo::GrafoNullo();
        return G.m() == (G.n() - numCompConnesse(G));
}
```




*a) Grafo ciclico* *b) Grafo aciclico*

Figura 3. *Grafi orientati*

### 5.3.3 Algoritmi su Grafi Orientati

Una prima e fondamentale applicazione della visita per il caso di grafi orientati è nella verifica se esso sia aciclico o meno. Sappiamo che un grafo orientato è aciclico se e solo se non ci sono cicli di lunghezza maggiore di uno. Definiamo la seguente relazione "«" tra i nodi del grafo: dati due nodi distinti $v$ e $w$, $v$ « $w$ se esiste un cammino da $v$ a $w$. Allora possiamo affermare che un grafo orientato è aciclico se e solo la relazione "«" è antisimmetrica cioè non esistono due nodi $v$ e $w$ tali che $v$« $w$ e $w$ « $v$. Quindi il grafo è aciclico se e solo se è possibile determinare una permutazione $<v_1,..., v_n>$ dei nodi tale che non esistono due nodi $v_i$ e $v_j$ con $i < j$ per cui $v_j$ « $v_i$ . Una permutazione siffatta è chiamata un *ordinamento topologico* del grafo orientato; ogni grafo aciclico ha almeno un ordinamento topologico mentre i grafi ciclici non ne hanno nessuno.

La verifica di aciclicità può essere effettuata cercando un qualsiasi ordinamento topologico. A tale scopo notiamo che il primo nodo $v_1$ nell'ordinamento non deve avere archi entranti diversi

dagli anelli altrimenti esisterebbe un nodo $v_i$ tale che $v_i \ll v_1$. Pertanto scegliamo come primo nodo un qualsiasi nodo con grado d'entrata, al netto degli anelli, uguale a 0. Togliamo questo nodo dal grafo e tutti gli archi che entrano o escono da esso e aggiorniamo il grado d'entrata dei nodi restanti. Anche il secondo nodo nell'ordinamento topologico deve avere grado d'entrata uguale a 0. Continuiamo così fino alla eliminazione di tutti i nodi. Se ciò non risultasse possibile, cioè a un certo punto non troveremo alcun nodo con grado d'entrata 0, allora il grafo è ciclico.

**Esempio 7.** Consideriamo il grafo orientato ciclico di Figura 3a; esso non ha alcun ordinamento topologico in quanto, ad esempio, se mettiamo prima 4 e poi 5 o viceversa la condizione di ordinamento topologico viene violata poichè $4 \ll 5$ e $5 \ll 4$. In effetti, nella determinazione di un ordinamento topologico, scegliamo il nodo 1 come nodo con grado d'entrata uguale a 0 e aggiorniamo i gradi di entrata dei nodi 2 e 3 che passano da 2 a 1. Prendiamo ora il nodo 7 che è l'unico ad avere grado di entrata uguale a 0 e aggiorniamo il grado di entrata del nodo 6 che passa da 2 a 1. A questo punto dobbiamo arrestarci poichè non c'è alcun nodo con grado di entrata 0, cioè il grafo è ciclico.

Consideriamo ora il grafo orientato aciclico di figura 3b. Un ordinamento topologico del grafo aciclico è <6, 7, 4, 5, 2, 3, 1> ed è ottenuto nel seguente modo:

— scegliamo il nodo 6 che ha grado di entrata 0 (sarebbe possibile scegliere al suo posto il nodo 7) e aggiorniamo il grado d'entrata del nodo 4 da 2 a 1;

— preleviamo il nodo 7 che è l'unico ad avere grado di entrata 0 e aggiorniamo ulteriormente il grado d'entrata del nodo 4 da 1 a 0;

— preleviamo il nodo 4 che è l'unico ad avere grado di entrata 0 e aggiorniamo i gradi d'entrata dei nodi 2 e 5 rispettivamente da 2 a 1 e da 1 a 0;

— preleviamo il nodo 5 che è l'unico ad avere grado di entrata 0 e aggiorniamo i gradi d'entrata dei nodi 2 e 3 ambedue da 1 a 0;

— scegliamo il nodo 2 che ha grado di entrata 0 (sarebbe possibile scegliere al suo posto il nodo 3) e aggiorniamo il grado d'entrata del nodo 1 da 2 a 1;

— preleviamo il nodo 3 che è l'unico ad avere grado di entrata 0 e aggiorniamo il grado d'entrata del nodo 1 da 1 a 0;

— preleviamo il nodo 1 che è l'unico ad avere grado di entrata 0 e a questo punto il grafo non ha più nodi: l'ordinamento topologico è stato determinato.

Il grafo di Figura 3b ha i seguenti ulteriori 3 ordinamenti topologici: <7, 6, 4, 5, 2, 3, 1>, <6, 7, 4, 5,3, 2, 1> e <7, 6, 4, 5, 3, 2, 1>.

Possiamo ora scrivere la funzione che verifica se un grafo orientato è aciclico e restituisce, nel caso di verifica positiva, anche uno degli ordinamenti topologici.

```
bool eAciclico( const GrafoOr& G, Lista<card>& ordTop )
```

```
                                              throws (Grafo::GrafoNullo)
{     if ( G.n() == 0 ) throw Grafo::GrafoNullo();
      ordTop.svuota();
      Vettore<card> gradoIn = gradoEntrata(G);
      card u,v;
      Pila<card> P;
      for ( u = 1; u <= G.n(); u++ )
      {     if ( G(u,u) )
                  gradoIn[u]--; // eliminazione contributo anelli
            if ( gradoIn[u] == 0 )
                  P.inPila(u);
      }
      while ( !P.vuota() )
      {     u = P.fuoriPila();
            ordTop.appendi(u); Arco e;
            for_each_ad(e,u,G)
            {     gradoIn[fin(e)]--;
                  if ( gradoIn[fin(e)] == 0 )
                        P.inPila(fin(e));
            }
      }
      bool aciclico = true;
      for ( u = 1; u <= G.n() && aciclico; u++ )
            if ( gradoIn[u] > 0 ) aciclico = false;
      return aciclico;
}
```

Per la definizione della funzione *gradoEntrata* si può usare quella descritta nell'Esempio 2. La complessità della funzione è $O(n^2)$ e $O(m)$ rispettivamente nei casi di grafo a matrice e a lista di adiacenza. Nel caso del grafo di Figura 3b, la funzione *eAciclico* restituisce il seguente ordinamento topologico: <6, 7, 4, 5,2, 3, 1>.

**Esempio 8.** Abbiamo *n* attività da far svolgere a un unico esecutore; ogni attività è caratterizzata dalla chiave *nome* e dal tempo di esecuzione espresso in minuti. Esistono delle precedenze tra le attività, ad esempio l'attivita *i* precede l'attività *j* significa che per iniziare l'attività *j* è necessario aver prima completato l'attività *i*. Vogliamo determinare una schedulazione delle attività, cioè un ordine di esecuzione di esse, che rispetti il vincolo delle precedenze. Le attività possono essere rappresentate come nodi di un grafo orientato aciclico in cui gli archi rappresentano le precedenze tra attività. Un nodo fittizio indica la fine dell'esecuzione di tutte le attività ed è preceduto da tutte le attività che non ne precedano altre. Una schedulazione delle attività corrisponde a uno degli ordinamenti topologici del grafo. Mentre possono esistere differenti schedulazioni, il tempo totale di esecuzione è unico.

Un esempio di grafo con 8 attività più quella fittizia finale è mostrato in Figura 4. In ogni cerchio, in alto è indicato il numero del nodo, al centro il nome e in basso il tempo di esecuzione. Vi sono diversi ordinamenti topologici; quello determinato dalla funzione *eAciclico* è il seguente: <2,4,1,3,6,8,5,7,9>. In questa schedulazione l'attività B (nodo 2) parte al tempo 0, l'attività D (nodo 4) parte al tempo 3, l'attività A (nodo 1) parte al tempo 5, l'attività C (nodo 3) parte al tempo 7, l'attività F (nodo 6) parte al tempo 11, l'attività H (nodo 8) parte al tempo 12, l'attività E (nodo 5) parte al tempo 16, l'attività G (nodo 7) parte al tempo 18, e l'attività FINE (nodo 9) parte al tempo 19, cioè il tempo totale di esecuzione è 19. Di seguito presentiamo il programma che determina la schedulazione sopra indicata:

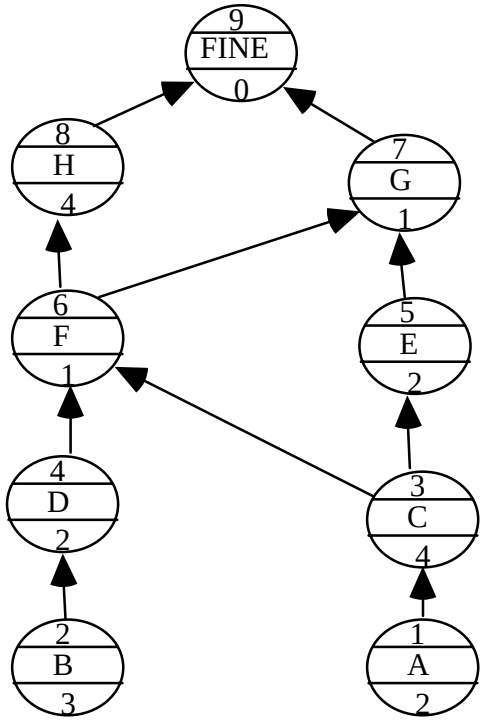



Figura 4. *Grafo di attività*

```
#include "grafoorl.h"
#include "stringa.h"
struct Att
{     stringa<20> nome; card tempo;
      bool operator== ( const Att& att )
      {     if ( nome == att.nome )
                  return true;
            else
                  return false;
      }
};
void letturaGrafo( GrafoOr&, Vettore<Att>& );
void stampaSchedule (   const GrafoOr& G,
            const Lista<card>&, const Vettore<Att>& );

void main ()
{     card nAtt;
      cout << "No attivita?:"; cin >> nAtt;
      GrafoOrL G(nAtt+1);
      Vettore<Att> attivita(1,nAtt+1);
      letturaGrafo(G, attivita);
      Lista<card> schedule;
      if ( eAciclico( G, schedule) )
            stampaSchedule(G,schedule, attivita);
      else
            cout << "Errore: attività cicliche";
}
```

La funzione *letturaGrafo* legge le attività e le organizza come i nodi di un grafo, verificando che non ci siano attività con lo stesso nome e che nessuna attività sia chiamata "*FINE*" che è nome riservato; infine legge le precedenze, e le rappresenta come archi verificando che non ci siano precedenze superflue o cicli.

```
void letturaGrafo( GrafoOr& G, Vettore<Att>& attivita )
{     Att att;
//    inserimento attività
      for ( card i = 1; i <= (G.n() - 1); i++ )
      {     cout << "\nAttivita " << i <<':';
            do
            {     cout << "Nome (diverso da FINE)?:";
                  cin >> att.nome;
            }     while ( att.nome == "FINE" );
            cout << "Tempo?:"; cin >> att.tempo;
            attivita[i] = att;
      }
//    inserimento attività fittizia finale
      att.nome = "FINE"; att.tempo = 0;
      attivita[G.n()] = att;
//    inserimento precedenze
      char risp; card j;
      cout << "\nInserisci prima precedenza?(S/N):"; cin >> risp;
      while ( risp == 's' )
      {     cout << "\nAttivita che precede?:"; cin >> i;
            cout << "\nAttivita che segue?:"; cin >> j;
            G(i,j,true);
            cout << "\nAltra precedenza?(S/N):"; cin >> risp;
      }
//    tutte le attività che non precedono nessun'altra vengono
//    rese precedenti dell'attività finale.
      for ( i = 1; i <= G.n()-1; i++ )
            if ( !G.primoAd(i) )
                  G(i,G.n(),true);
}
```

La funzione *stampaSchedule* non fa altro che scandire i nodi secondo l'ordinamento topologico determinato:

```
void stampaSchedule (   const GrafoOrL& G,
      const Lista<card>& sc, const Vettore<Att>& attivita )
{     card totTempo = 0; ElemLista<card> e;
      cout << "\nSchedulazione";
//    non viene stampata l'attività fittizia finale
      for_each_elemL(e,sc)
      {     Att att = attivita[e.info()];
            cout << "\nAttivita:" << att.nome;
            cout << " inizia al tempo:" << totTempo;
            totTempo += att.tempo;
      }
      cout << "Tempo Totale di esecuzione :" << totTempo;
}
```

Per effettuuare l'analisi di complessità del programma di schedulazione, consideriamo per prima la funzione *letturaGrafo* ; l'inserimento delle *n* attività e l'inserimento dell'attività finale fittizia e dei suoi precedenti richiede tempo $\Theta(n)$; nell'inserimento delle precedenze, sono inseriti $\Theta(m)$ archi; pertanto la funzione *letturaGrafo* ha complessità $\Theta(m)$ nell'ipotesi ragionevole che $m$ =

$\Omega(n)$. La complessità della funzione *eAciclico* è $\Theta(m)$. La funzione *stampaSchedule* è una semplice scansione di una permutazione dei nodi per cui ha complessità $\Theta(n)$. Pertanto il programma ha complessità $\Theta(m)$.

Determiniamo ora la chiusura transitiva di un grafo orientato. A tale scopo è sufficiente effettuare la visita di tutti i nodi del grafo.

```
void chiusuraTransitiva( const GrafoOr& G, GrafoOr& GP )
                  throws (Grafo::GrafoNullo, Grafo::DiffNodi)
{     if( G.n() == 0 ) throw Grafo::GrafoNullo();
      if( G.n() != GP.n() ) throw Grafo::DiffNodi();
      GP.svuota();
      for ( card v = 1; v <= G.n; v++ )
      {     VetBool visitati = visitaVentaglio(G,v);
            for ( card w = 1; w <= G.n(); w++ )
                  if ( visitati[w] )
                        GP(v,w,true);
      }
}
```

La funzione *visitaVentaglio* è richiamata $n$ volte mentre *GP(v,w,true)* $n^2$ volte. Nel caso di grafo orientato con matrice di adiacenza, *visitaVentaglio* e *GP(v,w,true)* hanno complessità $\Theta(n^2)$ e $\Theta(1)$ rispettivamente; pertanto l'algoritmo ha complessità $\Theta(n^3)$. Nel caso di grafo orientato con liste di adiacenza, *visitaVentaglio* e *GP(v,w,true)* hanno complessità $\Theta(m)$ e $\Theta(1)$ rispettivamente; pertanto l'algoritmo ha complessità $\Theta(n\ m)$. Sottolineamo che *GP(v,w,true)* ha complessità costante in quanto, grazie alla presenza degli iteratori interni, la funzione *cerca* delle liste ordinate riconosce in tempo costante che l'arco va sempre posto a fine lista.

Viene confermato che la struttura a liste di adiacenza si comportano meglio in quanto sostituiscono $n^2$ con $m$ nelle misure di complessità e $m = n^2$ solo nel caso peggiore. E' bene precisare che però nel caso della chiusura transitiva l'uso delle liste di adiacenza non porta alla scrittura di un algoritmo ottimo come accadeva per i problemi precedenti sui grafi in cui il lower bound $\Omega(m)$ coincideva con la complessità $O(m)$ dell'algoritmo. Nella chiusura transitiva l'algoritmo è ottimo solo per quei grafi in cui in numero di archi nella chiusura è $\Omega(n\ m)$; cioè per grafi densi o per grafi sparsi la cui chiusura diventa un grafo denso. Nei casi di grafi sparsi la cui chiusura rimane un grafo sparso esistono algoritmi più efficienti.

Per quanto riguarda i grafi con archi a matrice di adiacenza, esistono algoritmi per il calcolo della chiusura transitiva più efficienti, cioè con complessità $\Theta(n^k)$ dove $k$ è minore di 3 ma comunque non minore di 2 poichè la sola scrittura della matrice del grafo della chiusura richiede tempo $\Theta(n^2)$. Infatti il problema della chiusura transitiva è legato al problema della moltiplicazione di due matrici quadrate che con l'algoritmo usuale è effettuata in tempo $\Theta(n^3)$ ma che può essere eseguita da alcuni algoritmi in tempo $\Theta(n^k)$ dove $2 < k < 3$ (l'algoritmo di Strassen è uno di questi algoritmi). Ritorneremo a parlare della chiusura transitiva e della relazione di essa con la moltiplicazione di matrici nel Capitolo 8.

E' interessante notare che il calcolo della chiusura transitiva di un grafo non orientato è più semplice e può essere effettuato in tempo $O(n^2)$ nel seguente modo: (i) utilizziamo la funzione *compConnesse* per determinare le componenti connesse e (ii) inseriamo un arco nella chiusura

per ogni coppia di nodi che siano nella stessa componente. Il passo (i) è effettuato in tempo $\Theta(n^2)$ nel caso di rappresentazione a matrice di adiacenza e in tempo $\Theta(m)$ nel caso di rappresentazione a liste di adiacenza; il secondo passo è eseguito in tempo $\Theta(m')$ dove $m'$ è il numero di archi nella chiusura. Poichè $m' = O(n^2)$ e $m = O(n^2)$, l'algoritmo ha complessità $\Theta(n^2)$.

La chiusura transitiva può essere adoperata per calcolare le componenti fortemente connesse di un grafo orientato.

```
typedef Lista<card> Componente;
card compForti ( const GrafoOr& G, Lista<Componente>& LC )
{     LC.svuota();
      card nComp = 0;
      GrafoOrM GP (G.n()); // viene scelta la rappresentaz. a matrice
      chiusuraTransitiva(G,GP);
      VetBool inserito(1,G.n(),false),
      for ( v = 1; v <= G.n(); v++ )
            if ( !inserito[v] )
            {     Componente C; nComp++; // nuova componente
                  C.appendi(v); // si inserisce v nella componente
                  for ( card w = v+1; w <= G.n(); w++ )
                        if ( !inserito[w]&& GP(v,w) && GP(w,v) )
                        {     C.appendi(w); inserito[w] = true;
                        }
                  LC.appendi(C);
            }
      return nComp;
}
```

La scelta di rappresentare la chiusura *GP* con matrice di adiacenza non richiede che il grafo *G* debba avere la stessa rappresentazione — come abbiamo definito i grafi, le diverse rappresentzazioni possono essere utilizzate in maniera trasparente. La complessità di questo funzione dipende da *chiusuraTransitiva* poichè le restanti istruzioni (in particolare i due cicli di *for* innestati) sono eseguite in tempo $O(n^2)$. Pertanto la complessità è $\Theta(n^3)$ poichè il grafo della chiusura è stato dichiarato con struttura a matrice di adiacenza. Nel caso lo avessimo dichiarato con struttura a liste di adiacenza, la complessità non sarebbe cambiata in quanto i due cicli di for innestati hanno questa volta complessità $\Theta(n^3)$ in quanto la funzione GP(w,v) ha complessità lineare e non più costante. Esiste un algoritmo, pubblicato da Tarjan nel 1972, che calcola le componenti forti in tempo $O(m)$, che ovviamente non utilizza la chiusura e che è basato sulla visita a scandaglio.