Introduzione alla programmazione in C

Appunti delle lezioni di Tecniche di programmazione

Giorgio Grisetti
Luca Iocchi
Daniele Nardi
Fabio Patrizi
Alberto Pretto

Dipartimento di Ingegneria Informatica, Automatica e Gestionale
Facoltà di Ingegneria dell'Informazione, Informatica, Statistica
Università di Roma "La Sapienza"

Edizione 2018/2019

2019

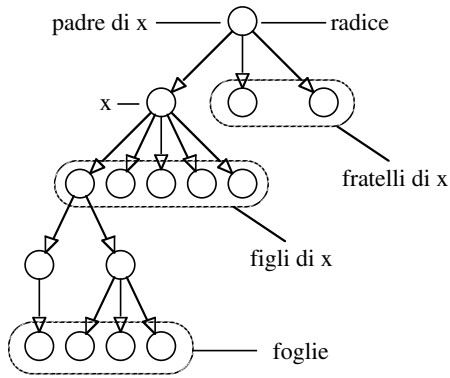
Indice

13. Alberi n-ari e grafi

In questo capitolo viene generalizzata la struttura *albero*, presentata nel capitolo precedente, inizialmente considerando alberi generici, ossia alberi nei quali ogni nodo può avere un numero arbitrario di figli. Successivamente, viene eliminato il vincolo che ogni nodo debba avere uno ed un solo nodo padre, ottenendo in questo modo una struttura dati denominata *grafo*. Di entrambe queste strutture dati vengono descritte diverse possibili implementazioni.

13.1. Alberi n-ari

Come visto nel capitolo precedente, l'albero n-ario è un tipo di dato non lineare utilizzato per memorizzare informazioni in modo gerarchico. Un esempio di albero è il seguente:



Naturalmente, la terminologia per denotare gli elementi dell'albero e le loro relazioni è la stessa del capitolo precedente.

13.1.1. Definizione induttiva di albero n-ario

Generalizziamo la definizione induttiva di albero binario:

- l'albero vuoto è un albero;
- se $T_1, \ldots, T_n$ $(n >= 1)$ sono alberi, allora l'albero che ha un nodo radice e $T_1, \ldots, T_n$ come sottoalberi è un albero;
- nient'altro è un albero.

I sottoalberi $T_1, \ldots, T_n$ $(n >= 1)$ di un albero non vuoto T vengono detti **sottoalbero1** ... **sottoalberon** di T.

13.1.2. Tipo astratto AlberoN

Il tipo di dato astratto AlberoN può essere definito come segue.

TipoAstratto AlberoN(T)

Domini

AlberoN : dominio di interesse del tipo

T : dominio dei valori associati ai nodi dell'albero

Funzioni

albVuoto() $\mapsto$ AlberoN
pre: nessuna
post: RESULT è l'albero vuoto (ossia senza nodi)

creaAlb(T r, AlberoN a1, ..., AlberoN an) $\mapsto$ AlberoN
pre: nessuna
post: RESULT è l'albero n-ario che ha r come radice, a1,..., an come sottoalberi

estVuoto(AlberoN a) $\mapsto$ Boolean
pre: nessuna
post: RESULT è true se a è vuoto, false altrimenti

radice(AlberoN a) $\mapsto$ T
pre: a non è vuoto
post: RESULT è il valore del nodo radice di a

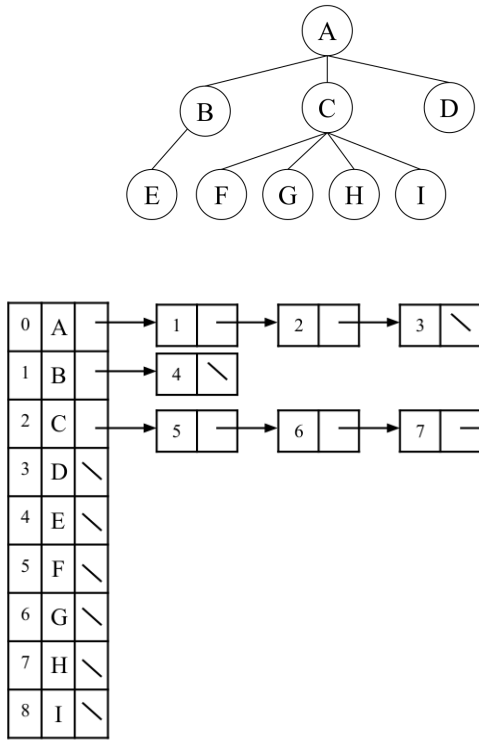
primofiglio(AlberoN a) $\mapsto$ AlberoN
pre: a non è vuoto
post: RESULT è il primo figlio di a

FineTipoAstratto

13.1.3. Rappresentazione tramite lista dei successori

La rappresentazione indicizzata mediante array si generalizza al caso degli alberi n-ari. Tuttavia, a causa del fatto che la struttura non è regolare, nel senso che il numero dei figli di un nodo è arbitrario, occorre memorizzare per ogni nodo dell'albero in quale posizione dell'array vengono memorizzati i nodi figli. Questa rappresentazione prende quindi il nome di rappresentazione tramite *lista di successori*.

Il seguente è un esempio di albero n-ario rappresentato tramite lista dei successori:



13.1.3.1. Caratteristiche della rappresentazione

Si noti che:

- Gli elementi dell'array dovranno contenere un campo per memorizzare l'informazione associata al nodo ed un campo per memorizzare il puntatore ad una lista dei figli.

- Normalmente la radice si fa corrispondere al nodo in posizione 0, altrimenti si può indicare esplicitamente la posizione del nodo radice.
- I nodi foglia sono caratterizzati da una lista di successori vuota.
- L'ordine in cui vengono memorizzati i nodi nell'array non è in genere significativo, anche se si può forzare l'ordinamento in modo che la scansione sequenziale dell'array corrisponda alla visita per livelli.

13.1.4. Algoritmi di visita

Per visitare un albero n-ario occorre considerare che per ogni nodo occorre fare tante chiamate ricorsive quanti sono i figli del nodo.

```
if T non è vuoto {
     analizza il nodo radice di T
     while (primofiglio(T) !=NULL) {
            visita primofiglio(T)
            passa al figlio successivo
     }
}
```

- La visita in post-ordine richiede di effettuare l'analisi del nodo radice dopo l'esecuzione del ciclo.
- La visita simmetrica non è definita.
- La visita per livelli (in ampiezza) richiede l'utilizzo di una coda.

Per l'implementazione dell'albero n-ario tramite lista dei successori si rimanda alla sezione successiva sui grafi.

13.1.5. Rappresentazione collegata di n-ari

Gli alberi n-ari possono essere rappresentati anche mediante la rappresentazione collegata. Tuttavia, a differenza del caso degli alberi binari, non si può definire a priori la dimensione del record, dato che il numero dei figli di un nodo non è definita a priori. Pertanto si utilizza una ulteriore struttura collegata per la memorizzazione dei figli.

La struttura del record per la memorizzazione di un nodo diventa semplicemente:

1. informazione associata al nodo;
2. riferimento ad una lista dei figli.

È possibile astrarre questo concetto in C definendo un tipo TipoAbero che è un puntatore ad un nodo di un albero:

```
typedef int TipoInfoAlbero;
struct StructNodoFiglio;
struct StructAlbero {
  TipoInfoAlbero info;
  struct StructNodoFiglio* figli;
};
struct StructNodoFiglio {
  struct StructAlbero* albero;
  struct StructNodoFiglio* next;
};
typedef struct StructAlbero* TipoAlbero;
typedef struct StructNodoFiglio* TipoFigli;
```

Osserviamo che:

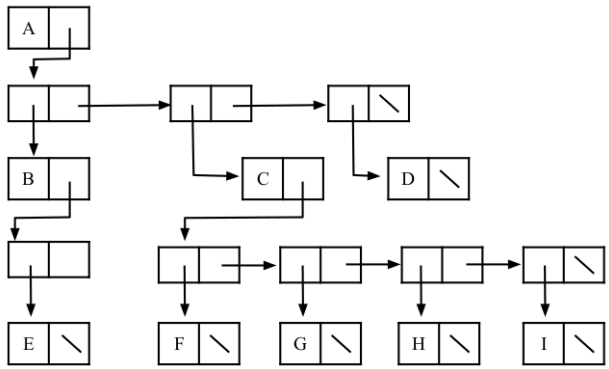
- il campo info memorizza il contenuto informativo del nodo. Il tipo TipoInfoAlbero verrà definito di volta in volta a seconda della natura delle informazioni che si vogliono memorizzare nei nodi dell'albero, analogamente a quanto visto per le strutture collegate lineari.
- il campi figli contiene un riferimento ad una lista dei figli del nodo. Il valore NULL di questo campo indica che si tratta di un nodo foglia.

Un albero n-ario nella rappresentazione collegata sarà rappresentato quindi nel seguente modo:

- L'albero *vuoto* viene rappresentato usando il valore NULL.
- Un albero *non vuoto* viene rappresentato mantenendo il riferimento al nodo radice dell'albero. Osserviamo che partendo dal riferimento alla radice dell'albero è possibile utilizzare i collegamenti contenuti nella lista dei figli per raggiungere ogni altro nodo dell'albero. In base a questa assunzione, *ogni valore di tipo puntatore a*

TipoNodoAlbero consente di accedere a tutto il sottoalbero alla cui radice esso punta.

Esempio: L'albero n-ario mostrato nella figura precedente, può essere rappresentato in C in modo collegato come illustrato.



La variabile albero rappresenta quindi l'albero e contiene un riferimento alla radice dell'albero.

13.1.5.1. Implementazione del tipo astratto AlberoN

Una realizzazione del tipo astratto AlberoN usando la rappresentazione collegata è mostrata di seguito. Il tipo astratto AlberoN viene implementato mediante la struttura C TipoAlbero, il tipo degli elementi T è definito in C con il tipo TipoInfoAlbero.

Costruttori e predicati sono analoghi al caso dell'albero binario.

```
TipoAlbero alberoVuoto() {
  return NULL;
}

TipoAlbero creaAlbero(TipoInfoAlbero infoRadice,
          TipoFigli f) {
  TipoAlbero a = (TipoAlbero) malloc(sizeof(struct
    StructAlbero));
  a->info = infoRadice;
  a->figli = f;
  return a;
}
```

```
bool estVuoto(TipoAlbero a) {
  return (a == NULL);
}
```

Per quanto riguarda i selettori, la selezione dell'informazione nel nodo radice è analoga al caso dell'albero binario.

```
TipoInfoAlbero radice(TipoAlbero a) {
  if (a == NULL) {
    printf ("ERRORE accesso albero vuoto \n");
    return ERRORE_InfoAlbBin;
  } else {
    return a->info;
  }
}
```

La scelta di rappresentare i figli tramite una lista, richiede invece una opportuna funzione che selezioni dal nodo la lista dei figli. Una volta acquisito il puntatore a questa lista si possono utilizzare le funzioni sulle liste per estrarre il primo figlio `primofiglio`, e, pi in generale gestire gli altri elementi della lista.

```
TipoFigli figli(TipoAlbero a) {
  if (a == NULL) {
    printf ("ERRORE accesso albero vuoto \n");
    return NULL;
  } else {
    return a->figli;
  }
}
```

Una volta realizzata la rappresentazione, la visita in pre-ordine dell'albero n-ario segue la formulazione generale dello schema di visita:

```
void stampaPreordine(TipoAlbero a) {
    if (a!=NULL) {
        printf(" %d ",a->info);
        TipoFigli p = a->figli;
        while (p!=NULL) {
            stampaPreordine(p->albero);
            p = p->next;
        }
    }
}
```

13.1.6. Rappresentazione parentetica di alberi n-ari

La **rappresentazione parentetica**, basata su stringhe di caratteri, si generalizza facilmente da quella dell'albero n-ario. Assumiamo che $f(T)$ sia la stringa che denota l'albero T; e che

1. l'albero vuoto è rappresentato dalla stringa `"()"`, cioè $f(\text{albero vuoto})$=`"()"`;
2. un albero con radice x e sottoalberi $T_1 \ldots T_n$ è rappresentato dalla stringa `"( "` $+ x + f(T_1) + \ldots + f(T_n) +$ `"() )"`.

Esempio: L'albero mostrato nella seguente figura può essere rappresentato mediante la stringa:

```
"( 15 ( 3 ( -5 ( 0 () ) () ) "  +
         "( 2 () ) "  +
         "( 10 () ) "  +
         "() ) "  +
     "( 1 ( 12 () ) ( 5 () ) () ) "  +
     "( 8 () ) "  +
     "() )".
```

Le differenze con la rappresentazione parentetica degli alberi binari sono:

- il numero di figli dei nodi è variabile ed essi sono rappresentati dalla concatenazione delle rispettive rappresentazioni;
- viene inserito `"()"` per indicare che non vi sono altri figli in un nodo;
- la foglia dell'albero non richiede la presenza di due sottoalberi vuoti, ma semplicemente `"()"`, che indica l'assenza di figli.

13.1.7. Costruzione di un albero n-ario da rappresentazione parentetica

La funzione per costruire un albero n-ario data una sua rappresentazione parentetica segue lo schema iniziale di lettura da file della funzione LeggiAlbero vista per gli alberi binari.

```
TipoAlbero LeggiAlbero(char *nome_file) {
  TipoAlbero result;
  FILE *file_albero;
  file_albero = fopen(nome_file, "r");
  result = LeggiSottoAlbero(file_albero);
  fclose(file_albero);
  return result;
}
```

La funzione ha come parametro in ingresso una stringa che rappresenta il nome del file che contiene l'albero e, oltre a gestire le operazioni di apertura e chiusura chiama la funzione ausiliaria LeggiSottoAlbero, che restituisce un puntatore al sottoalbero letto.

```
TipoAlbero LeggiSottoAlbero(FILE *file_albero) {
  char c;
  TipoInfoAlbero i;
  TipoAlbero r;
  LeggiParentesi(file_albero);   // parentesi aperta
  c = fgetc(file_albero);   // carattere successivo
  if (c == ')')
    return NULL;   // se () l'albero e' vuoto
  else { // altrimenti legge la radice
    fscanf(file_albero, "%d", &i);
    r = creaAlbero(i, LeggiFigli(file_albero));
    LeggiParentesi(file_albero); // parentesi chiusa
    return r;
  }
}
```

Le convenzioni per la lettura dell'albero sono analoghe a quelle viste per gli alberi binari. Si riportano di seguito per completezza:

- quando l'albero è vuoto, il carattere che segue la parentesi aperta '(' deve necessariamente essere la parentesi chiusa ')' (senza spazi);
- al contrario quando il nodo non è l'albero vuoto, l'informazione di tipo carattere da associare al nodo deve essere preceduta da uno spazio ' '.

- Se le informazioni sono di tipo int, o altro tipo primitivo numerico il tipo della variabile usata per leggere l'informazione ed il formato della istruzione fscanf devono essere aggiornate coerentemente;

- se le informazioni sono di altro tipo, in particolare di tipo non primitivo per leggere l'informazione associata diventa necessario progettare una specifica funzione.

Seguono le funzioni ausiliarie per la lettura delle parentesi e per la lettura della lista dei figli.

```
void LeggiParentesi(FILE *file_albero) {
  char c = ' ';
  while (c!='(' && c!=')')
    c = fgetc(file_albero);
}

TipoFigli LeggiFigli(FILE *file_albero) {
    TipoAlbero f = LeggiSottoAlbero(file_albero);
    if (f==NULL)
        return NULL;
    else {
        // printf("aggiungi figlio %d\n", f->info);
        return add(LeggiFigli(file_albero), f);
    }
}
```

Ed infine si riportano le funzioni ausiliarie per la costruzione della lista dei figli.

```
TipoFigli add(TipoFigli f, TipoAlbero a) {
    TipoFigli n = (TipoFigli)malloc(sizeof(struct
    StructNodoFiglio));
    n->albero = a;
    n->next = f;
    return n;

}

void aggiungiFiglio(TipoAlbero a, TipoAlbero f) {
  if (a == NULL) {
    printf ("ERRORE accesso albero vuoto \n");
  } else {
    a->figli = add(a->figli, f);
  }
}
```

13.1.8. Esercizi

La variante della funzione che calcola la profondità dell'albero binario è immediata:

```
/* calcola la profondita' di un albero n-ario */
int Profondita(TipoAlbero albero)
{
  int pmax,p;
  if (albero == NULL)
    return -1;   // l'albero vuoto ha profondita' -1
  else {
    // calcola la profondita' dei sottoalberi
    pmax=-1;
    TipoFigli l = a->figli;
    while (p!=NULL) {
            p=Profondita(l->albero);
            if (p>pmax) pmax=p;
            l = l->next;+
            }
  return pmax+1;
  }
}
```

Anche l'estensione della ricerca si ottiene sostituendo alle due chiamate corrispondenti a sottoalbero sinistro e destro, la sequenza di chiamate ricorsive relative agli elementi della lista dei figli.

```
bool RicercaAlbero(TipoAlbero A, TipoInfoAlbero elem,
    TipoAlbero *posiz) {
  bool trovato = false; *posiz=NULL;
  if (A == NULL)
    trovato = false;
  else if (elem == A->info) { // elemento trovato
    *posiz = A;  trovato = true;
  }
  else {
    /* cerca nel sottoalbero sinistro */
    TipoFigli l = A->figli;
    while (p!=NULL and !trovato) {
        trovato = RicercaAlbero(l->albero, elem, posiz
    );
            l = l->next;
            }
  }
  return trovato;
}
```

Infine, anche per quanto riguarda la copia di un albero n-ario, l'estensione consiste nuovamente nel realizzare l'operazione su ciascun

elemento della lista di figli.

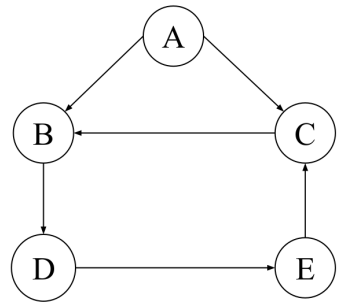
```
TipoFigli copiaFigli(TipoFigli f);

TipoAlbero copia(TipoAlbero a) {
    if (a==NULL)
        return NULL;
    else
        return creaAlbero(a->info,copiaFigli(a->figli)
    );
}

TipoFigli copiaFigli(TipoFigli f) {
    if (f==NULL)
        return NULL;
    else
        return add(copiaFigli(f->next),copia(f->albero
    ));
}
```

13.2. Grafi

Come anticipato, se rimuoviamo il vincolo che ogni nodo abbia uno ed un solo padre, la struttura dati che otteniamo prende il nome di *grafo*. In realtà, in molti testi si introduce prima il grafo, come struttura dati in grado di rappresentare *relazioni binarie* e successivamente si considera come caso particolare, l'albero n-ario.

In questa sezione, dopo una definizione della struttura dati grafo, faremo cenno ad alcune possibili implementazioni e rivisiteremo le nozioni di visita in profondità e visita in ampiezza nel caso del grafo.

Un esempio di grafo è il seguente:



13.2.0.1. Definizione di grafo

Un grafo orientato è definito da una coppia di insiemi $\langle N, A \rangle$, in cui:

- N è un insieme di nodi $N = \{n_0, \ldots, n_{s-1}\}$
- A è un insieme di archi $N = \{a_1, \ldots, a_{m-1}\}$, in cui ciascun arco è a sua volta costituito da una coppia di nodi $\langle n_i, n_j \rangle$, che sta ad indicare che il nodo n_i è collegato direttamente al nodo n_j.

Un grafo può essere visto coma la rappresentazione di una relazione NxN, con N dominio finito.

13.2.0.2. Un po' di nomenclatura

- *predecessore* è il primo nodo dell'arco $\langle n_i, n_j \rangle$;
- *successore* è il secondo nodo dell'arco $\langle n_i, n_j \rangle$;
- *sorgente* è un nodo che non ha archi entranti (e.g. la radice, ma in un grafo se ne possono avere più di una!);
- *pozzo* è un nodo che non ha archi uscenti (e.g. una foglia);
- *cammino* tra due nodi di un grafo è una sequenza di nodi $\{n_0, \ldots, n_k\}$, tale che per ogni coppia di elementi consecutivi della sequenza $\langle n_i, n_j \rangle$ vi è un arco nel grafo;
- *ciclo* è un cammino che contiene più volte lo stesso nodo del grafo.

La nozione di grafo si può generalizzare in varie maniere, considerando ad esempio che si può associare un peso agli archi o ammettendo più di un arco tra due nodi, ma questi argomenti esulano dagli scopi di questa dispensa, il cui scopo è quello di utilizzare delle rappresentazioni del grafo per lo svolgimento di esercizi relativi all'uso dei meccanismi di costruzione delle strutture dati resi disponibili dal linguaggio C.

13.2.0.3. Rappresentazione del grafo tramite matrice delle adiacenze

Questa rappresentazione fa uso di una matrice di dimensione NxN, i cui elementi sono definiti come segue:

- $[i, j] = 1$, se nel grafo è presente l'arco $\langle n_i, n_j \rangle$;
- $[i, j] = 0$, altrimenti

	A	B	C	D	E
A	0	1	1	0	0
B	0	0	0	1	0
C	0	1	0	0	0
D	0	0	0	0	1
E	0	0	1	0	0

L'implementazione di questa struttura si ottiene facilmente

```
#define NumeroNodi ...;
typedef int TipoInfoNodo;

struct Grafo{
      TipoInfoGrafo valori_nodi[NumeroNodi];
      int mat_adiacenza[NumeroNodi][NumeroNodi];
      };

typedef struct Grafo* TipoGrafo;
```

Si noti che la definizione della componente array `valori_nodi` non è necessaria, se l'informazione memorizzata nel nodo corrisponde all'etichetta numerica associata al nodo stesso.

Per esercizio si provi a verificare proprietà come:

- l'esistenza di nodi sorgente, nodi pozzo, presenza di nodi isolati (senza archi né ingresso né in uscita,
- esistenza di cammini tra coppie di nodi del grafo nodi del grafo,
- presenza di cicli etc. .

Questa rappresentazione offre una facile gestione delle operazioni fondamentali sulla struttura dati, ma ha l'inconveniente di richiedere molta memoria, che spesso viene usata solo in parte limitata. Infatti, dato che il numero degli archi è in genere paragonabile con il numero dei nodi la matrice delle adiacenze risulterà spesso *sparsa*, ossia avrà la maggioranza di valori pari a 0.

13.2.0.4. Rappresentazione del grafo tramite liste dei successori

Questa rappresentazione è del tutto analoga a quella vista nel caso degli alberi n-ari.

Il grafo viene rappresentato da un array di nodi a ciascuno dei quali è associata:

- l'informazione memorizzata nel nodo;
- il riferimento ad una lista che contiene i puntatori ai nodi figli.

L'implementazione di questa struttura si ottiene creando una struttura collegata per la lista dei successori e definendo il grafo come array di puntatori a liste di successori. Anche in questo caso, se l'informazione da memorizzare nel nodo è rappresentabile direttamente tramite le etichette numeriche dei nodi non occorre definire l'array `info`.

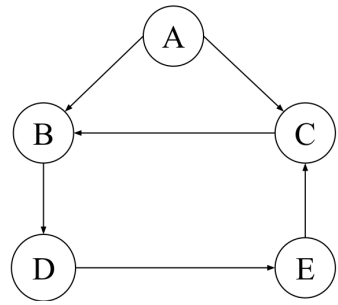
```
#define NumeroNodi ...;
typedef int TipoInfoNodo;
struct ListaSucc {
  TipoNodo successore;
  struct ListaSucc *next;
};
typedef struct ListaSucc * TipoLS;
struct Grafo {
  TipoInfoNodo info [NumNodi];
  TipoLS vett_lista_succ[NumNodi];
};
```

13.2.0.5. Rappresentazione del grafo tramite strutture collegate

La rappresentazione tramite strutture collegate non presenta alcuna differenza, rispetto a quanto visto per gli alberi n-ari, in quanto la lista degli archi uscenti da un nodo corrisponde esattamente alla lista dei figli del nodo di un albero n-ario. La proprietà che nell'albero i nodi hanno un solo padre non vincola infatti in alcun modo la struttura definita.

13.2.0.6. Tecniche di visita del grafo: in profondità

Consideriamo la visita in profondità del grafo mostrato in figura:



In questo caso, la presenza di un ciclo causa la non terminazione della visita, sia nel caso in cui questa venga realizzata tramite lo schema ricorsivo sia nel caso in cui essa sia realizzata usando la pila. Nel primo caso, la visita continuerà ad invocare se stessa a partire dallo stesso nodo, mentre nel caso della pila, il ciclo comporterà che lo stesso nodo rientrerà nella pilla un numero infinito di volte.

La tecnica per estendere i metodi di visita visti per gli alberi al caso del grafo consiste nel *marcare* i nodi durante la visita in modo che la visita prosegua per un nodo successore soltanto se esso non è già stato marcato (visitato in precedenza). In sintesi:

- si aggiunge all'informazione contenuta nel nodo un campo di tipo `bool`, che dovrà essere inizializzato a `false` prima dell'inizio della procedura di visita.
- prima di procedere con la visita in un nodo, si verifica se il suo campo marca è `false`: se non lo è si scarta e si passa al nodo successivo, se invece è `false` si visita il nodo, mettendo a `true` il suo campo marcatura.

13.2.0.7. Tecniche di visita del grafo: ampiezza

Consideriamo la visita in ampiezza del grafo mostrato nella figura precedente. Anche nel caso della visita del grafo in ampiezza si presenta il problema di gestire l'eventuale presenza di cicli. Ed, anche in questo caso, si utilizza la marcatura dei nodi.