15.9 Grafi

Un *grafo* è costituito da un insieme di nodi e un insieme di archi che uniscono coppie di nodi tali che non c'è mai più di un arco che unisce una coppia di nodi. Il grafo è *orientato* quando viene attribuito un senso di percorrenza agli archi stessi: in tal caso è ammesso che tra due nodi vi siano due archi purché orientati in sensi opposti; è anche possibile che un arco ricada sullo stesso nodo (Figura 15.6). È evidente come gli alberi siano un caso particolare di grafi, in quanto rispondono a un insieme di regole più restrittivo.

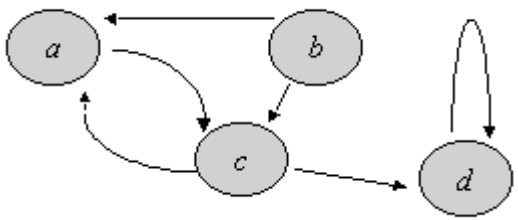

Figura 15.6 Rappresentazione di un grafo

Un modo per memorizzare i grafi, orientati e non, è quello di utilizzare una *matrice di adiacenza*, in cui ogni riga e ogni colonna rappresentano un nodo. Il grafo di Figura 15.6 viene memorizzato in una matrice di adiacenze come in Figura 15.7.

	a	*b*	*c*	*d*
a	0	0	1	0
b	1	0	1	0
c	1	0	0	1
d	0	0	0	1

Figura 15.7 Rappresentazione del grafo di Figura 15.6 con una matrice di adiacenze

Nel caso di grafi orientati la regola è la seguente: nella matrice appare un 1 se esiste un arco orientato dal nodo di riga al nodo di colonna. In particolare, si osservi il nodo *d*, che ha un arco orientato su se stesso.
La matrice di adiacenze provoca un notevole spreco di memoria nel caso, per esempio, che il numero *n* di nodi sia molto elevato in confronto al numero di archi: in ogni caso, infatti, si deve prevedere una matrice $n \times n$. Questa soluzione non è dunque molto flessibile.
Un'altra possibilità è rappresentata dalle liste di successori. In un array vengono memorizzati tutti i nodi e per ogni nodo è presente un puntatore a una lista che contiene i riferimenti ai nodi successori. In Figura 15.8 è presentato il grafo di Figura 15.6 memorizzato con liste di successori.

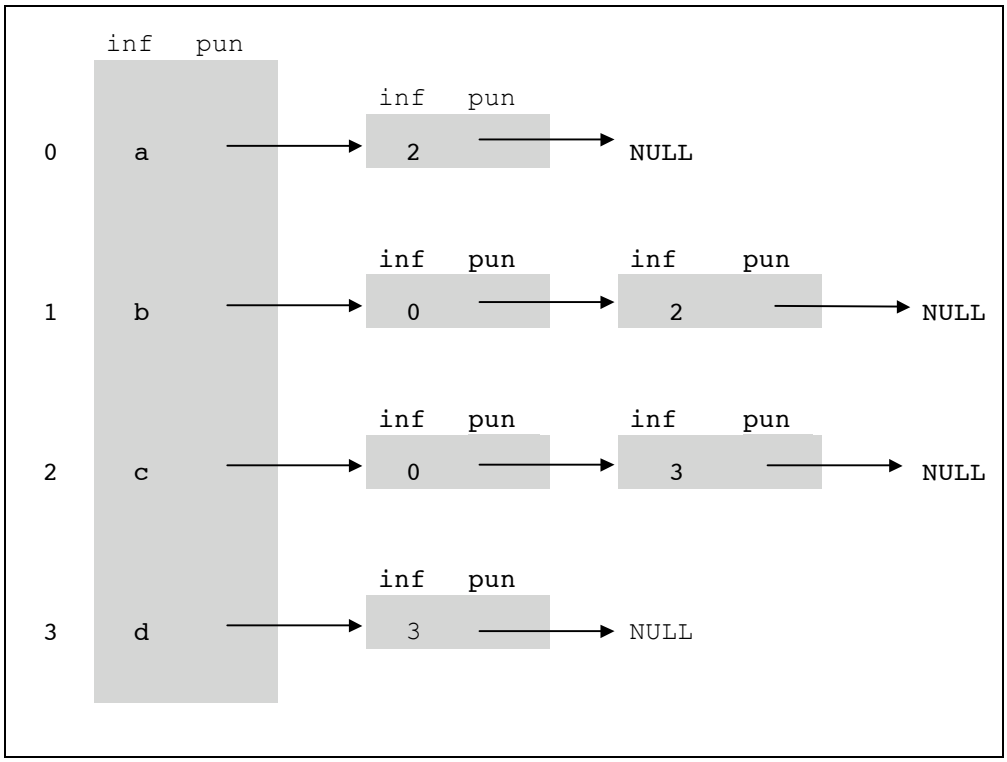

Figura 15.8 Grafo memorizzato mediante una lista di successori

Nel Listato 15.7 è presentato un programma che memorizza un grafo con una matrice di adiacenze, da cui ricava la rappresentazione in una lista di successori.
La funzione `mat_adiacenze` richiede all'utente i nodi e gli archi che conformano il grafo, memorizza le etichette dei nodi nel campo `inf` dell'array `s` e crea la matrice di adiacenze; successivamente la funzione `vis_mat_adiacenze` visualizza la matrice. La funzione `successori` percorre la matrice e per ogni suo elemento con valore 1 chiama la funzione `crea_succ`:

```
void crea_succ(int, int);
```

Questa funzione crea per ogni arco un elemento di lista inserendo nel campo informazione il riferimento al nodo di arrivo. La funzione `visita` percorre le liste di successori visualizzando tutti i nodi raggiungibili da un certo nodo.

✓ **NOTA**

Questa volta si è ritenuto utile mostrare un programma che fa uso essenzialmente di variabili globali cui tutte le funzioni fanno riferimento. Sappiamo che in generale è meglio far uso del passaggio di parametri come abbiamo mostrato in tutto il testo, ma non è detto che in casi particolari un'altra scelta non sia da apprezzare.

Abbiamo visto alcune delle strutture dati più importanti e abbiamo avuto modo di sperimentare la loro implementazione in linguaggio C. Nei problemi reali, pur rimanendo valide le logiche mostrate in queste pagine, le cose sono più complesse.
Il campo informazione, che abbiamo detto contenere l'etichetta di un nodo dell'albero, del grafo o di un'altra struttura, può contenere un insieme di dati molto ampio o essere un riferimento a essi. Per esempio, si pensi a un archivio di dati in memoria secondaria contenente un considerevole numero di record: per poter accedere a essi in tempi soddisfacenti si ha la necessità di utilizzare degli indici che permettano, dato il valore di una chiave (quale il codice di un articolo del magazzino o il conto corrente di un utente della banca), di ottenere tutte le informazioni a essa correlate.
In molti sistemi gli indici vengono realizzati con degli alberi, che consentono di accedere alle chiavi e da queste – tramite puntatori – al relativo record dell'archivio. Sono organizzati ad albero i file indici del linguaggio Cobol e di molti gestori di basi di dati. Gli indici stessi sono immagazzinati in file in memoria secondaria e vengono caricati in memoria centrale al momento della loro utilizzazione.
I grafi sono spesso usati nella soluzione di problemi di ricerca operativa che implicano, per esempio, lo studio di cammini minimi.

```
/* Trasformazione della rappresentazione di un grafo
   da una matrice di adiacenze a una lista di successori */

#include <stdio.h>
#include <malloc.h>

struct nodo {              /* Struttura di un nodo */
  char inf;
  struct successore *pun;
};

struct successore {       /* Elemento della lista di successori */
  int inf;
  struct successore *pun;
};

int a[10][10];            /* Matrice di adiacenze */
struct nodo s[10];        /* Array di nodi */
int n;                    /* Numero di nodi */

void mat_adiacenze(void);
void vis_mat_adiacenze(void);
void successori(void);
void crea_succ(int, int);
void visita(void);

main()
{
mat_adiacenze();      /* Creazione della matrice di adiacenze */
vis_mat_adiacenze();  /* Visualizzazione della matrice */
successori();         /* Creazione delle liste di successori */
visita();             /* Visual. dei successori di ogni nodo */
}


/* Crea la matrice di adiacenze */

void mat_adiacenze(void)
{
```

```
int i, j;
char invio;

printf("\nNumero di nodi: ");
scanf("%d", &n);
scanf("%c", &invio);

for(i=0; i<n; i++) {            /* Richiesta etichette dei nodi */
  printf("\nEtichetta del nodo: ");
  scanf("%c", &s[i].inf);
  scanf("%c", &invio);
  s[i].pun = NULL;
}

for(i=0; i<n; i++)              /* Richiesta archi orientati */
  for(j=0; j<n; j++) {
    printf("\nArco orientato da [%c] a [%c] (0 no, 1 si) ? ",
            s[i].inf, s[j].inf);
    scanf("%d", &a[i][j]);
  }
}


/* Visualizza la matrice di adiacenze */

void vis_mat_adiacenze(void)
{
int i, j;

printf("\nMATRICE DI ADIACENZE\n");
for(i=0; i<n; i++)               /* Visualizza i nodi (colonne) */
  printf("   %c", s[i].inf);

for(i=0; i<n; i++) {
  printf("\n%c  ", s[i].inf);    /* Visualizza i nodi (righe) */
  for(j=0; j<n; j++)
    printf("%d   ", a[i][j]);    /* Visualizza gli archi */
}
}


/* Crea le liste di successori. Per ogni arco rappresentato
   nella matrice di adiacenze chiama crea_succ()            */

void successori(void)
{
int i, j;

for(i=0; i<n; i++)
  for(j=0; j<n; j++) {
    if(a[i][j]==1)
      crea_succ(i, j);
   }
}


/* Dato un certo arco nella matrice di adiacenze crea
il rispettivo elemento di lista                        */
```

```
void crea_succ( int i, int j )
{
struct successore *p;

if(s[i].pun==NULL) {     /* Non esiste la lista dei successori */
  s[i].pun = (struct successore *)(malloc(sizeof(struct successore)));
  s[i].pun->inf = j;
  s[i].pun->pun = NULL;
}
else {                   /* Esiste la lista dei successori */
  p = s[i].pun;
  while(p->pun!=NULL)
    p = p->pun;
  p->pun = (struct successore *)(malloc(sizeof(struct successore)));
  p = p->pun;
  p->inf = j;
  p->pun = NULL;
}
}


/* Per ogni nodo del grafo restituisce i suoi successori.
   Lavora sulle liste di successori                      */

void visita(void)
{
int i;
struct successore *p;

printf("\n");

for(i=0; i<n; i++) {
  printf("\n[%c] ha come successori: ", s[i].inf);
  p = s[i].pun;
  while(p!=NULL) {
    printf(" %c", s[p->inf].inf);
    p = p->pun;
  }
}
}
```

Listato 15.4 Rappresentazione di grafi