Una breve introduzione all'implementazione in C di algoritmi su grafo

A cura di Gianmaria Leo

Introduzione

La lezione è un'introduzione a concetti e strumenti che permettono l'implementazione di algoritmi su grafo, con riferimento al linguaggio C. Implementare un algoritmo significa scrivere in un linguaggio di programmazione il codice che esegue le operazioni di una metodologia o di un algoritmo. Per conseguire questo risultato è innanzitutto necessario rappresentare l'istanza del problema (l'input dell'algoritmo da implementare) nella memoria del calcolatore. Questo significa scegliere un'opportuna *struttura dati* e allocare le *risorse di memoria* necessarie per acquisirla o costruirla. Ad esempio, supponiamo di dover implementare una procedura che ordina una lista $\mathcal{L}$ di un numero fissato N di interi. Innanzitutto, scegliamo una struttura dati che consente di rappresentare una lista, poi decidiamo come allocare la memoria per acquisirla, infine progettiamo il codice che implementa un algoritmo di ordinamento. Intuitivamente, possiamo rappresentare $\mathcal{L}$ con un array (struttura dati elementare) di N variabili di di tipo intero. Nel codice sorgente occorre: (*i*) definire la costante N; (*ii*) dichiarare le variabili dell'array; (*iii*) allocare la memoria. In questo caso la lunghezza dell'array è fissata, quindi procediamo con un'allocazione statica della memoria.

```
#define N 100
int main()
{
  int l[N];

  /* procedura per acquisire L */

  /* implementazione algoritmo */
}
```

Dato un grafo $G = (V, E)$ con $|V| = n$ e $|E| = m$, quali sono le strutture dati che permettono di rappresentarlo nella memoria di un calcolatore? Come possiamo codificarle in linguaggio di programmazione?

Matrici di adiacenza

La *matrice di adiacenza* A di un grafo G è una matrice binaria $n \times n$ tale che il generico elemento $a_{ij} = 1$ se e solo se $(i, j) \in E$, $a_{ij} = 0$ altrimenti. Se il grafo G è pesato sugli archi (abbiamo una funzione $w : E \to \mathbb{R}$), possiamo costruire la matrice A in modo tale che $a_{ij} = w(i, j)$ per ogni $(i, j) \in E$, $a_{ij} = 0$ altrimenti. Se G è pesato sui nodi (abbiamo una funzione $w : V \to \mathbb{R}$), possiamo modificare la matrice di adiacenza impostando $a_{ii} = w(i)$ per ogni $i \in V$. Se G è diretto, la matrice di adiacenza occupa n^2 celle di memoria; se G è non diretto, la matrice A è simmetrica rispetto alla diagonale principale, quindi per memorizzarla sono necessarie $\frac{n(n-1)}{2}$ celle ($\frac{n(n+1)}{2}$ se G è pesato sui nodi). In entrambi i casi, possiamo affermare che la matrice di adiacenza occupa uno spazio pari a un $O(n^2)$. In allocazione statica della memoria, il codice per implementare una matrice di adiacenza è il seguente:

```
#define N 20
int main()
{
  int a[N][N];

  /* acquisizione/costruzione matrice */
}
```

L'allocazione statica assegna le risorse di memoria in modo definitivo. Questo significa che le risorse destinate a una variabile non potranno essere assegnate ad altre variabili fino al termine dell'esecuzione del programma. Inoltre, la modifica della struttura, come l'aggiunta/eliminazione di un nodo dal grafo, potrebbe causare violazioni di accesso alla memoria o errori di esecuzione. Al contrario, l'allocazione dinamica consente di assegnare la stessa risorsa a più variabili durante l'esecuzione del programma, quindi garantisce maggiore efficienza nella gestione della memoria e maggiore versatilità per modifiche/manipolazioni delle strutture che implementano l'istanza. Per ottenere una matrice dinamica possiamo usare il seguente codice:

```
#include <stdlib.h>
int main()
{
  int **a ; /* dichiarazione puntatore bidimensionale */
  int n ; /* dimensione della matrice */
```

```
  int i ; /* variabile contatore */

  /* allocazione array di puntatori alle righe */
  a = (int **)malloc(n * sizeof(int *));

 /* allocazione riga per ogni puntatore dell'array */
 for(i=0; i<n; i++) {
   a[i] = (int *)malloc(n * sizeof(int)); }

 /* acquisizione/costruzione del grafo */
}
```

Assumiamo che l'allocazione/riallocazione/deallocazione di blocchi di memoria richiede $O(1)$ operazioni. Per eliminare un vertice $u \in V$ dal grafo G, nella struttura dati è necessario:

1. per ogni $i \neq u$, eliminare `a[i][u]` dalla riga puntata da `a[i]`;
2. eliminare l'intera riga puntata da `a[u]`;
3. eliminare il puntatore `a[u]`.

Per eliminare il j-esimo elemento da un array `c[]` con k posizioni è necessario:

1. effettuare l'assegnamento `c[i-1] = c[i]` per ogni $i = j + 1, \ldots, k$
2. rilasciare il blocco di memoria assegnato a `c[k]` (oppure riallocare la memoria da destinare a `c`).

La rimozione di un elemento da un array richiede $O(n)$ operazioni, quindi la rimozione di un vertice richiede $O(n^2)$ operazioni. Per aggiungere un vertice al grafo, è necessario riallocare il blocco di memoria destinato ai puntatori e riallocare per ogni riga il blocco di memoria destinato alle colonne per una dimensione pari a $n + 1$. Quindi, l'aggiunta di un vertice impiega $O(n)$ operazioni.

Liste di adiacenza

Le *liste di adiacenza* sono delle strutture dati ricorsive che consentono un'altra rappresentazione del grafo G. L'elemento base della lista, detto *nodo*, è un record che contiene delle informazioni relative a un vertice o a un arco del grafo e un puntatore all'elemento successivo della lista. Il codice che serve per implementare questa struttura dati può essere il seguente:

```
#include <stdlib.h>
struct node {
        int node_id ; /* ID del nodo */
        int list_id ; /* ID della lista */
        struct node *next ; /* puntatore al prossimo nodo */
};
typedef struct node Node;
typedef Node *NodePtr

int main()
{
 NodePtr adj ; /* dichiarazione puntatore alle liste */

 NodePtr nodeList ; /* puntatore a nodo di lista concatenata */
 int n, i ;

 /* costruzione dell'array di liste concatenate */
 for(i=0; i<n; i++) {
   adj[i] = malloc(sizeof(Node)) ;
   adj[i]-> node_id = i ;
   adj[i]-> list_id = i ;
   adj[i]-> next = NULL ; }

 /* acquisizione/costruzione del grafo */
}
```

Per rappresentare un grafo $G = (V, E)$ con $|V| = n$ mediante liste di adiacenza, utilizziamo un array `adj[]` di n variabili puntatore di tipo `NodePtr`. L'elemento i-esimo dell'array, indicato con `adj[i]`, è associato al vertice $i \in V$ del grafo e ogni nodo j della lista concatenata a cui punta `adj[i]` (i.e. un nodo della lista tale che `nodeList`$\rightarrow$ `node_id == j && nodeList`$\rightarrow$ `list_id == i`) è associato a un vertice $j \in N(i)$, dove $N(i)$ è l'intorno del vertice i. Se G è non diretto, la rappresentazione in liste di adiacenza occupa $n + 2m$ blocchi di memoria (vedere l'*handshaking lemma*); se G è diretto sono necessari solo $n + m$ blocchi. Se G è pesato sugli archi o sui vertici possiamo aggiungere il campo `weight` alla struttura nodo: il campo `weight` del nodo j appartenente alla lista a cui punta `adj[i]` potrebbe contenere il peso dell'arco (i, j); il campo `weight` del nodo `adj[i]` potrebbe contenere il peso del vertice i.

La rimozione di un vertice $u \in V$ dal grafo comporta nella struttura dati:

1. per ogni $v \in N(u)$, l'eliminazione del nodo u dalla lista a cui punta `adj[v]`;

2. l'eliminazione dell'intera lista a cui punta `adj[u]`;

3. l'eliminazione di `adj[v]`.

L'eliminazione di un nodo j da una lista avviene nel seguente modo:

1. accedere sequenzialmente al nodo j vistando tutti i nodi della lista che lo precedono, tenendo traccia del nodo corrente (`currentNode`), del predecessore (`prevNode`) e del successore (`nextNode`);

2. quando `currentNode`$\rightarrow$ `node_id == j`, eseguire `prevNode`$\rightarrow$ `next = nextNode`;

3. rilasciare la memoria occupata dal nodo j (con l'istruzione `free(currentNode)`).

L'eliminazione di un nodo da una lista richiede $O(n)$ operazioni, quindi concludiamo che l'eliminazione di un vertice dal grafo impiega ancora una volta $O(n^2)$ operazioni.

Osservazioni

Infine, riportiamo alcune osservazioni sulle strutture dati descritte:

• La matrice di adiacenza occupa n^2 blocchi di memoria. Le liste di adiacenza occupano $n + 2m$ blocchi (se G è non diretto), quindi la memoria occupata dipende anche dal numero di archi: l'efficienza nell'utilizzo della memoria aumenta tanto più il grafo è sparso.

• L'eliminazione di un vertice $u \in V$ richiede la rimozione degli archi $(u, v) \in E$ che hanno un estremo in u. Con le matrici di adiacenza, l'operazione critica è lo shift degli elementi di $n - 1$ array di taglia n. Con liste di adiacenza, l'operazione critica è l'accesso sequenzialmente al nodo u di $|N(v)|$ liste concatenate. In quest'ultimo caso, il tempo di esecuzione non è fissato come accade per le matrici di adiacenza, ma dipende dalla densità del grafo: lo sforzo computazionale diminuisce tanto più il grafo è sparso.