Soluzioni della settima esercitazione di Algoritmi 1

Beniamino Accattoli

19 dicembre 2007

1 Grafi

Un grafo è **non orientato** se descrivendo un arco come una coppia di vertici (i,j) l'ordine è ininfluente (ovvero gli archi sono sottoinsiemi di cardinalità 2 dell'insieme dei vertici). E' **orientato** se invece l'ordine conta. Gli archi dei grafi orientati sono rappresentati graficamente come frecce, mentre quelli dei grafi non orientati sono rappresentati come linee. In questa esercitazione lavoreremo sulle due principali rappresentazioni dei grafi.
Le **liste di adiacenza** rappresentano un grafo G usando un vettore di liste di dimensione n, dove n è il numero dei vertici di G. Le liste sono composte da elementi *vertice* che hanno due componenti, un campo *nome*, contente l'intero che rappresenta il vertice, e un campo *next*, contenente un puntatore a vertice. La lista i contiene i vertici adiacenti al vertice i (cioè collegati ad i tramite un arco). Nelle liste di adiacenza di un grafo orientato si rappresentano gli archi uscenti da un vertice: ogni arco (i,j) compare una sola volta: si ha un elemento *vertice* di nome j nella lista i. Nelle liste di un grafo orientato invece ogni arco compare due volte, nella lista i, come vertice j, e nella lista j, come vertice i.
L'altra rappresentazione è la **matrice di adiacenza**. Tale matrice M è binaria, ed ha un 1 nella locazione (i,j) (ovvero nella locazione alla riga i e colonna j) se vi è un arco tra i vertici i e j, altrimenti ha uno 0. La matrice di adiacenza di un grafo non orientato è simmetrica rispetto alla diagonale.

2 Esercizi

1. Scrivere un algortimo $ListeToMatrice(vettore\ vettListe) \rightarrow matrice$ che passi dalla rappresentazione in liste di adiacenza di un grafo alla rappresentazione come matrice di adiacenza, e che sia valido sia nel caso orientato che in quello non orientato. Scrivere poi un algoritmo che faccia il passaggio inverso, dalla matrice alle liste, valido sia nel caso orientato che in quello non orientato. Qual'è la complessità di questi algoritmi?

   risposta: gli algoritmi sono i seguenti, ed hanno entrambi complessità $\Theta(n^2)$, perché entrambi usano una matrice di n^2 elementi, nel primo caso la complessità è data dall'allocazione e inizializzazione della matrice a 0 (implicità nella dichiarazione) e nel secondo caso dallo scorrere di tutte le locazioni della matrice.

Algoritmo ListeToMatrice(vettore vettListe) → matrice di booleani
1. n ← dim(vettListe)
2. M matrice booleana di dimensione nxn
3. **for** i ← 1 **to** n **do**
4. **foreach** vertice v **in** vettListe[i]
5. M[i][v.nome] ← 1

Algoritmo MatriceToListe(matrice di booleani M) → vettore di liste
1. n ← dim(M[1])
2. vettListe vettore di n liste di vertici
3. **for** i ← 1 **to** n **do**
4. **for** j ← 1 **to** n **do**
5. **if** M[i][j]=1 **then**
6. nodoCorrente ← **new** vertice(j)
7. nodoCorrente.next ← vettListe[i]
8. vettListe[i] ← nodoCorrente

2. Dato un grafo $G = (V(G), E(G))$ si definisce il **grafo complementare** $\overline{G}$ come il grafo avente lo stesso insieme di vertici e avente un arco e tra due vertici u e v se e solo se e non è in G, ovvero tale che $e = (u, v) \in E(\overline{G}) \Leftrightarrow e \notin E(G)$ (cioè il grafo complementare è ottenuto aggiungendo gli archi che non sono in G e levando quelli che sono in G).

   (a) descrivere come passare da un grafo al suo complementare se il grafo è rappresentato in una matrice di adiacenza. Qual'è la complessità di un tale algoritmo?

   risposta: basta trasformare ogni zero della matrice di adiacenza in un uno e ogni uno in zero. La complessità di quest'algoritmo è quella della dimensione della matrice, quindi $\Theta(n^2)$.

   (b) Supponiamo di voler scrivere un algoritmo per passare da un grafo al suo complementare usando le liste di adiacenza. Quale complessità dobbiamo aspettarci da un simile algoritmo? Quale sarà il caso peggiore? Esiste un caso migliore?. Suggerire poi una semplice implementazione ispirandosi ai punti precedenti.

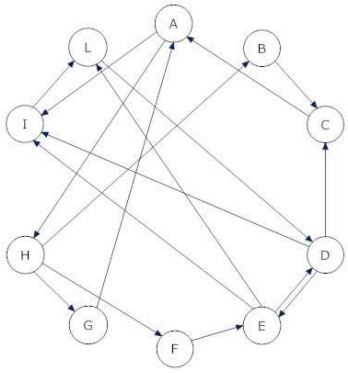
   risposta: Si consideri il caso estremo in cui il grafo è vuoto, ovvero non ha archi. Per passare al grafo complementare dobbiamo costruire una lista di adiacenza con tutti gli archi possibili e quindi tale caso avrà complessità $\Omega(n^2)$, poiché tale è il numero di archi possibili in un grafo su n vertici. Si consideri ora un grafo con qualche arco. La situazione non è diversa: per capire che arco non si deve inserire nel complementare bisogna scorrere quelli presenti in G, e poi si devono inserire tutti quelli assenti. Dunque per ogni arco, presente o meno, bisogna fare almeno un'operazione. La complessità sarà quindi $\Omega(n^2)$ su ogni input. Ma n^2 è anche una limitazione superiore della complessità infatti per ottenere il grafo complementare su liste

di adiacenza si possono combinare insieme gli algoritmi precedenti, tutti di complessità $\Theta(n^2)$:

- Si passa alla rappresentazione matriciale
- Si passa al grafo complementare su matrice
- Si torna alla rappresentazione con liste

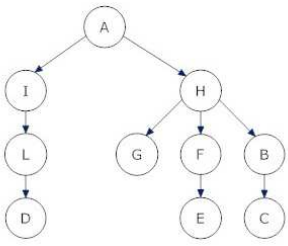
La morale dell'esercizio: le liste di adiacenza sono spesso preferibili alle matrici di adiacenza perché sono una rappresentazione più compatta di un grafo, rappresentante solo l"informazione positiva' in un grafo, cioè 'ciò che c'è'. Le matrici invece rappresentano anche l"informazione negativa', ovvero anche 'ciò che non c'è'. Il passaggio al grafo complementare è un esempio di problema nel quale l'informazione negativa è tanto importante quanto quella positiva, e quindi le liste di adiacenza non offrono un vantaggio rispetto alle matrici.

3. Si consideri il seguente grafo orientato:

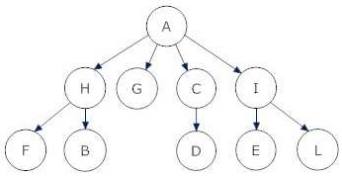


si produca l'albero BFS prodotto dalla visita in ampiezza a partire dal vertice A. Si consideri poi il grafo come fosse non orientato (si dimentichino le frecce sugli archi) e se ne dia nuovamente l'abero BFS prodotto dalla visita in ampiezza.

risposta: albero prodotto dalla visita in ampiezza del grafo orientato a partire da A:



albero prodotto dalla visita in ampiezza del grafo non orientato a partire da A:



In realtà l'albero prodotto da una visita in ampiezza a partire da un fissato vertice non è univocamente determinato. La forma dell'albero può variare a seconda dell'ordine con cui si inseriscono gli adiacenti di un vertice nella coda per la visita. Ad esempio se nella visita non orientata il primo dei figli di A ad essere visitato fosse stato il vertice I l'albero prodotto sarebbe stato diverso (D sarebbe stato un figlio di I). Tuttavia tutti questi alberi hanno su ogni livello lo stesso insieme di vertici (infatti D è comunque sarebbe stato allo stesso livello nell'albero, sia come figlio di I che come figlio di C). Ciò in cui possono differire sono le parentele.

4. (a) In un grafo non orientato G la **distanza** tra due vertici x e y è la lunghezza (in termini di archi) del cammino minimo tra x e y. Il **diametro** di G è invece la massima distanza tra due vertici in G. Essendo gli alberi dei particolari grafi si può considerare il diametro di un albero. Sia T un albero di altezza k. Quale può essere la massima lunghezza di un cammino semplice (ovvero che non ripete né archi né vertici) in T? E il massimo diametro di T?

   Risposta: In un albero di altezza k i cammini massimali (cioè non estendibili) si ottengono partendo da un foglia di altezza k, risalendo fino alla radice e poi scendendo fino ad un'altra foglia. Quindi se esistono due foglie di altezza k il cammino semplice massimo è lungo $2k$, altrimenti è più corto. Ogni albero è aciclico quindi vi è un solo cammino tra ogni coppia di vertici, quindi tale cammino è anche il cammino minimo. Dunque la lunghezza del cammino semplice massimo in un albero e il diametro di un albero sono uguali.

   (b) La visita in ampiezza di un grafo G a partire da un vertice x ha la proprietà di produrre un albero T tale che il livello di un nodo y in T è pari alla distanza tra y e x in G. E' possibile che un grafo di diametro 10 abbia un albero BFS di altezza 4? Giustificare la risposta. E' possibile limitare il diametro di un grafo conoscendo l'altezza di un suo albero BFS?

   Risposta: Se il grafo G coincide con l'albero BFS prodotto dalla visita allora il diametro di G è il diametro dell'albero BFS. Se invece non coincidono allora in G ci sono degli archi in più, che quindi accorciano lo distanze, e di conseguenza il diametro di G. Allora il diametro del grafo è sempre minore o uguale del diametro di un suo albero BFS, il quale a sua volta è minore o uguale del doppio dell'altezza dell'albero. Quindi non può esistere un grafo di diametro 10 con un albero BFS di altezza 4, perché tale altezza implica che il diametro del grafo sia minore o uguale a 8.

5. Un vertice di un grafo orientato è un **pozzo** se ha $n-1$ archi entranti e

nessun arco uscente. Quanti pozzi possono esserci in un grafo? Dimostrarlo. Scrivere poi un algoritmo $trovaPozzo(vettore\ vettListe) \rightarrow intero$ che restituisce, se esiste, l'indice di un pozzo in un grafo orientato, e se non esiste restituisce -1. Il grafo in input è rappresentato con liste di adiacenza. Si può immaginare di usare una funzione $trovaIndice(lista\ l, intero\ i) \rightarrow boolean$ che ritorna vero se trova l'indice i nella lista l, e falso altrimenti.

risposta: in un grafo può esserci al massimo un pozzo. Si supponga che non sia vero. Dunque ve ne saranno almeno due, entrambi senza archi uscenti. Ma allora entrambi possono avere al massimo $n-2$ archi entranti, e nessuno dei due è un pozzo. Assurdo.
L'algoritmo scorre anzitutto la lista di adiacenza per trovare un vertice che abbia la lista vuota. Se non lo trova o se ne trova più d'uno, abbandona la ricerca: il pozzo non può esserci. Altrimenti controlla in ogni lista di adiacenza (tranne quella del candidato pozzo) se il candidato compare. Se compare in tutte le liste allora il candidato è effettivamente un pozzo, altrimenti no.

```
Algoritmo trovaPozzo(vettore di liste vettListe) → intero 1. pozzo ← -1
2. for i ← 1 to n do
3.    if (vettListe[i]=null) then
4.       if (pozzo = -1) then pozzo ← i
5.       else return -1
6. if (pozzo ≠ -1) then
7.    foreach i in {1,..,n} diverso da pozzo
8.       if (not trovaIndice(vettListe[i], pozzo) then return -1
9.    return pozzo
```