Algoritmica 07/08
19. GRAFI

Un grafo G=(V,E) consiste in un insieme di *vertici* V e un insieme di *spigoli* E. La notazione e=(u,v) indica che lo spigolo e connette i vertici u,v: si dice allora che u,v sono *adiacenti*. In un grafo *orientato* uno spigolo e=(u,v) è diretto da u a v, a indicare che si può percorrere la struttura spostandosi direttamente da u a v ma non viceversa, a meno che non vi sia anche lo spigolo e'=(v,u). In un grafo *non orientato* gli spigoli non hanno direzione e possono essere percorsi nelle due direzioni. Nella rappresentazione grafica standard di G i vertici sono indicati come punti e gli spigoli che li connettono sono indicati come frecce (G orientato) o segmenti (G non orientato).

Nel seguito porremo |V|=n, |E|=m. Notiamo che $m \le n(n-1)/2$ per un grafo non orientato e $m \le n(n-1)$ per un grafo orientato, ove il limite superiore corrisponde alla presenza di tutti i possibili spigoli. Un grafo è *connesso* se da ogni vertice è possibile raggiungere tutti gli altri percorrendo gli spigoli. Un albero è un grafo connesso non orientato con m=n-1 spigoli. Per m<n-1 il grafo non è connesso. Per m=O(n) il grafo è *sparso* (un albero è un grafo sparso); per $m=\Theta(n^2)$ il grafo è *denso*.

1. **Rappresentazione di grafi in memoria**

Vi sono due modi standard di rappresentare un grafo in memoria: attraverso un insieme di *liste di adiacenza*, o una *matrice di adiacenza*. Tali rappresentazioni, e gli algoritmi che costruiremo su di esse, sono indipendenti dal fatto che il grafo sia orientato o no e prevedono che i vertici siano indicati con gli interi da 1 a n.

Nel primo metodo si impiega un array A[1..n] che contiene in ogni posizione A[i] il puntatore a una lista (di adiacenza) dei vertici adiacenti al vertice i in un ordine qualsiasi. Indicheremo tale lista con L_i. Un grafo non orientato G di cinque vertici che ha la forma pittorica di una casetta (il vertice 1 è il culmine del tetto) è rappresentato con le liste di adiacenza dell'esempio seguente: per esempio il vertice 1 è adiacente a 2 e 3.

Esempio 1a. Liste di adiacenza

```
            vertice :   1   2   3   4   5           1
                  A :   ↓   ↓   ↓   ↓   ↓          / \  G
liste (in verticale):   2   1   1   2   4         2---3
                        3   3   2   5   3         |   |
                            4   5                 4---5
```

La matrice di adiacenza è invece una matrice booleana quadrata M[1..n,1..n], ove per ogni posizione i,j si pone M[i,j]=1 se esiste lo spigolo (i,j), M[i,j]=0 se tale spigolo non esiste. Se il grafo non è orientato si ha M[i,j] = M[j,i], cioè la M coincide

con la sua trasposta. Si noti che per ogni i si ha M[i,i]=0 perché non vi è vertice connesso con sé stesso. La matrice per il grafo sopra visto è indicata nell'esempio 1b. Si noti che essa coincide con la propria trasposta: per esempio M[2,4]=M[4,2] a indicare la presenza dello spigolo non orientato (2,4).

Esempio 1b. Matrice di adiacenza

M	1	2	3	4	5
1	0	1	1	0	0
2	1	0	1	1	0
3	1	1	0	0	1
4	0	1	0	0	1
5	0	0	1	1	0

Per quanto riguarda l'occupazione di memoria la rappresentazione con liste di adiacenza è più compatta per i grafi sparsi, quella con matrice di adiacenza per i grafi densi. Se economizzare lo spazio non è importante si sceglie di volta in volta la rappresentazione più adatta al problema da risolvere. Spesso le due rappresentazioni si equivalgono.

2. **Visita in ampiezza**

Come per gli alberi, visitare un grafo significa percorrerlo *esaminando* tutti i vertici, ove l'esame corrisponde a qualsiasi operazione si desideri eseguire sui vertici: contarli (se n non è noto), registrare l'informazione a essi associata, cercare un vertice particolare e così via. Rispetto agli alberi il problema è molto più complesso perché i grafi non hanno struttura ricorsiva. Un metodo è la *visita in ampiezza* che ora descriviamo.

Presi due vertici u,v tali che v sia raggiungibile da u, la *distanza* d(u,v) è data dal numero minimo di spigoli che si devono percorrere per spostarsi da u a v. Nella visita in ampiezza si parte da un vertice arbitrario s detto *sorgente* e si esaminano tutti i vertici raggiungibili da s, incontrando prima quelli a distanza 1, poi quelli a distanza 2 ecc. Diremo che questi insiemi di vertici si trovano nello strato 1, 2 ecc. All'interno di ogni strato l'ordine in cui si incontrano i vertici dipende da come è memorizzato il grafo: in particolare dall'ordine scelto nelle liste di adiacenza, o dalla numerazione dei vertici nella matrice.

L'algoritmo di visita in ampiezza, detto BFS da "breadth-first search", è definito su un grafo G=(V,E), orientato o no, memorizzato con liste di adiacenza (ricordiamo che L_v è la lista di adiacenza del vertice v). BFS impiega un array binario I[1..n], inizialmente azzerato, per indicare che un vertice v non è stato ancora incontrato (I[v]=0), o è stato già incontrato (I[v]=1). BFS impiega anche una coda Q, inizialmente vuota, su cui sono definite le operazioni:

x ←Q il primo elemento di Q è estratto e posto in x,
x →Q un nuovo elemento x è inserito in Q.

Quando si incontra un nuovo vertice v con I[v]=0 esso viene "esaminato", marcato con I[v]=1 e posto in attesa in Q; quando è il suo turno per essere estratto da Q si cercano tutti i vertici adiacenti a v, e il vertice stesso non è più considerato. L'esame di v è eseguito con una procedura ESAME(v) che può essere definita a piacere. L'algoritmo può impiegare anche un array D[1..n] che, pur non essenziale per la visita, fornisce senza ulteriori costi la distanza D[v]=d(s,v) per ogni vertice v.

```
BFS(G,s)
  for (i=1; i≤n; i=i+1) {I[i]=0; D[i]=∞};
  ESAME(s); I[s]=1; D[s]=0; s→Q;
  while (Q non vuota) {
    u ←Q;
    for (∀ v ∈Lu)
      {if (I[v]==0) {ESAME(v); I[v]=1; D[v]=D[u]+1; v→Q;}}}
```

Si noti che nel secondo ciclo for l'algoritmo esegue una scansione di lista che deve essere opportunamente programmata.

Il lettore è invitato a convincersi per proprio conto che l'algoritmo "esamina" una sola volta tutti i vertici raggiungibili da s. Applicato al grafo dell'esempio 1a con s=1, BFS esamina i vertici nell'ordine 1,2,3,4,5 ponendo D: 0 1 1 2 2.

Per quanto riguarda la complessità dell'algoritmo, l'iniziale ciclo for richiede tempo $\Theta(n)$. Il ciclo while si esegue tante volte per quanti elementi entraranno nella coda: in essa ogni vertice entra e esce esattamente una volta, e per ciascuno di essi si esaminano tutti i vertici contenuti nella sua lista di adiacenza. Il tempo richiesto dal ciclo while è quindi proporzionale al numero di vertici contenuti complessivamente in tutte le liste di adiacenza del grafo, quindi al numero di spigoli m (è il caso di massima complessità in cui tutti i vertici sono raggiungibili da s). BFS richiede quindi tempo complessivo $\Theta(n+m)$, e questo tempo è ottimo perché coincide con la dimensione complessiva del grafo che deve chiaramente essere completamente esaminato.

3. **Visita in profondità**

La *visita in profondità* è simile alle visite degli alberi nel senso che, partendo da una sorgente s, si cammina nel grafo lungo il percorso più profondo possibile: quando da un vertice non se ne possono raggiungere di nuovi si torna sull'ultimo vertice incontrato, di cui si devono ancora esaminare dei successori. Ciò conduce a una spontanea definizione ricorsiva dell'algoritmo che non richiede l'impiego di una coda: ma non si creda che per questo la visita sia più elementare della BFS, perché per gestire la ricorsività essa richiede una pila, definita dal compilatore del

linguaggio e invisibile all'utente. La visita è detta DFS per "depth-first search".

Impiegando ancora le liste di adiacenza, DFS ha la struttura seguente. L'algoritmo impiega un modulo ricorsivo VPROF definito per un vertice generico u, e chiamato inizialmente su s. L'array I ha il solito significato.

```
DFS(G,s)
  for (i=1; i≤n; i=i+1) I[i]=0;
  VPROF(s);

VPROF(u)
  ESAME[u]; I[u]=1;
  for (∀ v ∈Lu){if (I[v]==0) VPROF(v);}
```

Applicato al grafo dell'esempio 1a con s=1, DFS esamina i vertici nell'ordine 1,2,3,5,4 con un unico percorso in profondità.

Per quanto riguarda la complessità dell'algoritmo, l'iniziale ciclo for richiede tempo $\Theta(n)$. La procedura ricorsiva VPROF è chiamata esattamente n volte (posto che tutti i vertici siano raggiungibili da s) perché è chiamata solo per vertici x con I[x]=0 e si pone subito I[x]=1. Il ciclo for all'interno della procedura VPROF è ripetuto complessivamente per tutti gli elementi in tutte le liste di adiacenza, cioè per m iterazioni complessive (si noti che per molte di queste non si effettuerà la successiva chiamata di VPROF perché risulterà I[v]≠0). DFS richiede quindi tempo complessivo $\Theta(n+m)$, e anche in questo caso questo tempo è ottimo.

4. Percorsi tra vertici

Dati due vertici distinti x,y, un problema di base è individuare, se esiste, un percorso da x a y. Risolviamo questo problema mediante un algoritmo ispirato alla visita DFS, valido per un grafo orientato o no. Costruiremo il percorso in una pila P che, dalla testa alla coda, conterrà i vertici incontrati ordinatamente da x a y (P=∅ se non c'è percorso). Poiché i percorsi possono essere più d'uno l'algoritmo si arresta quando viene individuato il primo di essi senza che si possa prevedere quale sia. Se si cerca il (o un) percorso minimo si deve impiegare un algoritmo ispirato alla visita BFS (vedi sotto).

```
PERC(x,y)
  P=∅; T=false;
  for (i=1; i≤n; i=i+1) I[i]=0;
  VPERC(x);

VPERC(u)
  I[u]=1;
  for (∀ v ∈Lu : I[v]=0){
       if (v=y) {v→P; T=true; return;} else VPERC(v); }
       if (T==true) {u→P; return; }
```

Esaminare attentamente questo algoritmo, in particolare per quanto riguarda l'uscita dalle chiamate ricorsive della procedura PERC.

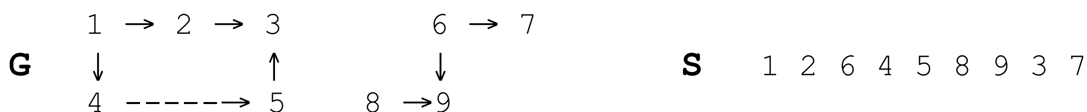
Se si desidera individuare un percorso più breve tra x e y si può usare un'estensione di BFS definendo un nuovo array G ove G[i] indica il predecessore del vertice i nella visita. Il lettore sviluppi per proprio conto l'algoritmo.

5. Ordinamento topologico

Consideriamo un grafo G=(V,E) orientato e privo di cicli: in gergo **dag**, per "directed acyclic graph". In esso inevitabilmente esistono uno o più vertici di *ingresso* senza spigoli entranti (*sorgenti*), e uno o più vertici di *uscita* senza spigoli uscenti (*pozzi*). I dag rappresentano tipicamente insiemi di azioni tra cui sono definite delle propedeuticità, cioè alcune di esse devono essere eseguite prima di altre: come per esempio avviene in una catena di produzione, o in un gruppo di processi software, o in un insieme di esami universitari. Le azioni sono rappresentate nei vertici e lo spigolo orientato (x,y) rappresenta la *dipendenza* di y da x, cioè non si può eseguire y se prima non si è eseguita x. Questa dipendenza è ovviamente transitiva, cioè se y dipende da x e z dipende da y (per esempio perché esistono gli spigoli (x,y) e (y,z)) allora z dipende da x anche se non esiste lo spigolo (x,z).

Un *ordinamento topologico* di G è una sequenza ordinata S dei vertici che rispetti tutte le dipendenze tra azioni: tale cioè che se y dipende da x deve apparire in S dopo x (ma non vale strettamente il contrario: se y appare dopo x può dipendere o no da x). In tal modo un singolo esecutore può eseguire tutte le azioni di G secondo l'ordine di S, rispettando tutte le propedeuticità. Se G è una semplice catena di vertici in cui ciascuno è connesso con uno spigolo al successivo, esiste un solo ordinamento topologico (banale) per G. Altrimenti gli ordinamenti sono più di uno ed esistono anche se il grafo è costituito da diverse componenti non connesse tra loro.

Ecco un dag G con due componenti, tre sorgenti (1, 6, 8) e tre pozzi (3, 7, 9); e un ordinamento topologico S di G:



Si noti che in S, per esempio, 5 è preceduto da 4 e 4 è preceduto da 1; 3 è preceduto da 2 e 5; 9 è preceduto da 6 e 8.

Per determinare un ordinamento topologico per un dag arbitrario si può utilizzare il seguente algoritmo TOP, semplice estensione dell'algoritmo DFS. La sequenza S è gestita come una pila inizialmente vuota: la notazione x →S indica che l'elemento x è inserito in testa alla sequenza.

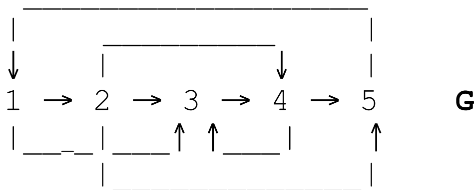
```
TOP(G)
  for (i=1; i≤ n; i=i+1) I[i]=0;
  for (i=1; i≤ n; i=i+1) {if (I[i]==0) VTOP(i);}

VTOP(u)
  I[u]=1;
  for (∀ v ∈L_u){if (I[v]==0) VTOP(v);}
   u →S;
```

Il lettore potrà dimostrare personalmente che l'algoritmo TOP è corretto e richiede tempo $\Theta(n+m)$; e che, per l'esempio visto sopra, costruisce l'ordinamento topologico S = 8 6 9 7 1 4 5 2 3.

6. Matrici di adiacenza

La matrice d'adiacenza M di un grafo G, orientato o no, indica gli spigoli presenti in G, quindi i percorsi di lunghezza uno tra vertici. Ammettendo che gli elementi di M siano zeri e uni numerici, ove ogni 1 indica uno spigolo, la matrice quadrato M^2= M×M indica i percorsi di lunghezza due, il cubo di M indica i percorsi di lunghezza tre, ecc. (si ricordi che ogni elemento del prodotto M×M si calcola come prodotto riga-colonna). Il lettore potrà dimostrare questa proprietà per proprio conto (esercizio 4) aiutandosi con il seguente esempio riferito a un grafo orientato.



M	1	2	3	4	5
1	0	1	1	0	0
2	0	0	1	1	1
3	0	0	0	1	0
4	0	0	1	0	1
5	1	0	0	0	0

M^2	1	2	3	4	5
1	0	0	1	2	1
2	1	0	1	1	1
3	0	0	1	0	1
4	1	0	0	1	0
5	0	1	1	0	0

Per esempio M^2[2,1]=1 indica che vi è un percorso lungo due dal vertice 2 al vertice 1: infatti M^2[2,1]=1 è il risultato del prodotto tra la seconda riga e la prima colonna di M che hanno entrambe un 1 in posizione 5, e il percorso corrispondente è costituito dagli spigoli (2,5)-(5,1). Si ha poi M^2[1,4]=2, che indica che vi sono due percorsi lunghi due tra 1 e 4, costituiti infatti da (1,3)-(3,4) e (1,2)-(2,4). E così via.

Esercizi

1. Applicare gli algoritmi BFS e DFS a grafi a piacere, orientati e non, per comprenderne bene il funzionamento. L'insieme dei vertici esaminati dipende dalla scelta della sorgente s?
2. Riformulare gli algoritmi BFS e DFS impiegando la matrice di adiacenza anziché le liste, e discutere la complessità di questi nuovi algoritmi.
3. Dimostrare che, nell'algoritmo BFS, l'array D contiene le minime distanze tra s e gli altri vertici. Suggerimento: vedere come i vertici dei diversi strati entrano nella coda.
4. Dimostrare che il quadrato della matrice di adiacenza M di un grafo, orientato o no, rappresenta tutti i percorsi di lunghezza due nel grafo. Stabilire cosa rappresenta la matrice somma M+M^2. Estendere il ragionamento alle potenze successive di M.

Bibliografia per approfondimenti

T.H.Cormen, C.E.Leiserson, R.L.Rivest, C.Stein. Introduzione agli algoritmi e strutture dati. Cap.22. McGraw-Hill 2005.