Progettazione di Algoritmi - lezione 4

Discussione dell'esercizio [gradi]

L'algoritmo che vogliamo rendere efficiente è il seguente:

```
ORD(G: DAG)
    L <- lista vuota
    WHILE  c'è almeno un nodo in G  DO
        trova un nodo v senza archi entranti e rimuovilo da G
        L.append(v)
    RETURN L
```

Un'implementazione diretta di questo algoritmo è inefficiente perché ognuna delle n iterazione del `WHILE` potrebbe richiedere O($n + m$). L'idea è di mantenere durante l'esecuzione i gradi entranti dei nodi del DAG. All'inizio li possiamo calcolare facendo una passata dell'intero grafo e poi ad ogni iterazione quando rimuoviamo un nodo v aggiorniamo i gradi dei nodi della lista degli adiacenti uscenti di v. Inoltre, ci manteniamo uno stack dei nodi di grado entrante zero e quando durante l'aggiornamento dei gradi un nodo diventa a grado zero lo aggiungiamo allo stack.

```
ORD(G: DAG)
    indeg: array dei gradi entranti inizializzati con 0
    FOR ogni nodo v in G DO
        FOR ogni adiacente (uscente) w di v DO
            indeg[w] <- indeg[w] + 1
    S <- stack dei nodi con grado entrante zero, inizialmente vuoto
    FOR ogni nodo v in G DO
         IF indeg[v] = 0 THEN
             S.push(v)
    L <- lista vuota
    WHILE S non è vuoto DO
        v <- S.pop()
        L.append(v)
        FOR ogni adiacente (uscente) w di v DO
            indeg[w] <- indeg[w] - 1
            IF indeg[w] = 0 THEN
                S.push(v)
    RETURN L
```

La costruzione dell'array dei gradi entranti (`indeg`) ha costo O($n + m$) perché fa semplicemente una scansione dell'intero grafo. L'inizializzazione dello stack dei nodi con grado entrante zero costa O(n). Il `WHILE` esegue n iterazioni e complessivamente il numero di iterazioni del `FOR` interno è pari al numero di tutti gli archi, cioè m. Quindi la complessità totale è O($n + m$).

Esercizi

Questa lezione è interamente dedicata alla risoluzione di esercizi per acquisire una maggiore dimestichezza con le idee, concetti e algoritmi che abbiamo visto finora.

Esercizio [archi]

Vogliamo scrivere un algoritmo che esegue una DFS su un grafo diretto e ritorna il numero di archi dell'albero della DFS, il numero di archi all'indietro, il numero di archi in avanti e il numero di archi di attraversamento.

Esercizio [trasposto]

Il grafo trasposto di un grafo diretto $G = (V, E)$, e un grafo diretto $G^T = (V, E^T)$ che ha lo stesso insieme dei nodi ma tutti gli archi con direzione opposta, vale a dire $E^T = \{ (v, u) \mid (u, v) \text{ in } E \}$. Descrivere un algoritmo che dato un grafo diretto G, rappresentato tramite liste di adiacenza degli adiacenti uscenti, ritorna il grafo trasposto G^T rappresentato nello stesso modo. L'algoritmo deve avere complessita $O(n + m)$.

Esercizio [grado due]

Dimostrare che se tutti i nodi di un grafo non diretto G hanno grado almeno due allora c'è almeno un ciclo. Se il grado di ogni nodo è esattamente due, si puo affermare che G è un ciclo?

Esercizio [ponte]

Descrivere un algoritmo che, dato un grafo non diretto G e un arco $\{u, v\}$ del grafo, determina se G ha un ciclo che contiene $\{u, v\}$. Analizzare il tempo di esecuzione dell'algoritmo. E se, nel caso un ciclo esista, vogliamo anche trovare un ciclo che contiene l'arco?

Esercizio [pozzo]

In un grafo diretto, un nodo si dice *pozzo universale* se ha grado entrante $n - 1$ e grado uscente 0.

- Dimostrare che un grafo diretto non può avere più di un pozzo universale.
- Descrivere un algoritmo che preso un grafo diretto G, rappresentato tramite matrice di adiacenza, determina se G contiene o meno un pozzo universale. L'algoritmo deve avere complessità $O(n)$.
- Dimostrare che il problema non è risolvibile in tempo $O(n)$ se il grafo è rappresentato con liste di adiacenza.

Esercizio per casa [strade critiche]

La rete viaria di una cittadina non è stata progettata molto bene. Tutte le strade sono a doppio senso di marcia e da un qualsiasi incrocio è possibile arrivare ad un qualsiasi altro incrocio. Ma ci sono delle *strade critiche* che se interrotte (ad esempio per lavori) dividono la cittadina in due parti e non si può più andare da una parte all'altra. Vogliamo un algoritmo efficiente che analizzando la rete viaria trovi tutte le strade critiche.

Soluzioni

Di seguito presentiamo alcune possibili soluzioni degli esercizi proposti.

Discussione dell'esercizio [archi]

Si tratta di modificare l'algoritmo della DFS in modo da poter classificare durante la visita i vari tipi di archi. Ogni arco che porta ad un nodo non ancora visitato appartiene all'albero di visita, un arco che porta ad un nodo la cui visita è iniziata ma non ancora terminata è un arco all'indietro, un arco che porta ad un nodo la cui visita è terminata è un arco o in avanti se tale visita è iniziata dopo di quella del nodo corrente o di attraversamento se è iniziata prima. Per mantenere le informazioni sull'inizio delle visite e la loro terminazione possiamo usare un array `VIS` tale che `VIS[u] = 0` quando il nodo `u` non è stato ancora visitato, `VIS[u] = -k` quando la DFS da `u` è iniziata, al tempo `k`, e non è ancora terminata e `VIS[u] = k` quando la visita è terminata.

```
VIS: array inizializzato a 0
te, be, fe, ce: contatori degli archi inizializzati a 0
c <- 0          /* Contatore dei nodi visitati */

DFS_EDGES(G: grafo, u: nodo, VIS: array, c, te, be, fe, ce: contatori)
    c <- c + 1
    VIS[u] <- -c      /* La visita da u è iniziata */
    FOR ogni adiacente v di u DO
        IF VIS[v] = 0 THEN
            te <- te + 1        /* Arco dell'albero (tree edge) */
            DFS_EDGES(G, v, c, te, be, fe, ce)
        ELSE IF VIS[v] < 0 THEN
            be <- be + 1        /* Arco all'indietro (back edge) */
        ELSE IF VIS[v] > -VIS[u] THEN
            fe <- fe + 1        /* Arco in avanti (forward edge) */
        ELSE
            ce <- ce + 1        /* Arco di attraversamento (cross edge) */
    VIS[u] <- -VIS[u]  /* La visita da u è terminata */
```

La chiamata iniziale sarà `DFS_EDGES(G, u, VIS, c, te, be, fe, ce)`.

Discussione dell'esercizio [trasposto]

Dobbiamo semplicemente scorrere le liste di adiacenza di *G* e per ogni arco (*u*, *v*) che incontriamo aggiungiamo l'arco (*v*, *u*) a G^T, ovvero, aggiungiamo *u* agli adiacenti (uscenti) di *v*.

```
TRASP(G: grafo rappresentato tramite liste di adiacenza)
    GT: array delle liste di adiacenza di G trasposto, inizializzato con liste vuote
    FOR ogni nodo u di G DO
        FOR ogni adiacente (uscente) v di u DO
            GT[v].append(u)
    RETURN GT
```

Chiaramente la complessità è O(n + m).

Discussione dell'esercizio [grado due]

Un modo molto semplice di dimostrare che se ogni nodo di un grafo non diretto *G* ha grado almeno due allora il grafo contiene un ciclo e di considerare una DFS a partire da un nodo di *G*. La visita dovrà necessariamente incontrare un nodo *w* che è una foglia dell'albero della DFS e siccome *w* ha grado almeno due, *w* deve avere almeno un altro arco oltre a quello che appartiene all'albero. Sappiamo che tale arco, non appartenendo all'albero, non potrà che essere un arco all'indietro e questo dimostra l'esistenza di un ciclo.

Se ogni nodo di *G* ha grado esattamente due, non è detto che *G* sia un ciclo. Potrebbe infatti essere formato da due o più cicli disgiunti. Se invece è anche connesso, allora è necessariamente un ciclo. Perché?

Discussione dell'esercizio [ponte]

Se un arco {*u*, *v*} di un grafo non diretto *G* non è contenuto in nessun ciclo, allora nel grafo *G*' ottenuto rimuovendo l'arco da *G*, i nodi *u* e *v* non sono connessi. Infatti, se lo fossero vuol dire che ci sarebbe in *G*' un cammino che li collega e siccome tale cammino non contiene l'arco {*u*, *v*}, il cammino insieme a tale arco è un ciclo in *G* che contiene l'arco, contraddizione. Viceversa, se la rimozione dell'arco {*u*, *v*} sconnette i nodi *u* e *v* vuol dire che non ci poteva essere un ciclo che conteneva l'arco. Quindi abbiamo diamostrato

In un grafo non diretto e connesso *G*, un arco non è contenuto in nessun ciclo se e solo se la rimozione dell'arco sconnette il grafo.

Un arco la cui rimozione sconnette un grafo connesso è chiamato **ponte** (*bridge*). Un algoritmo molto semplice per determinare se un arco è un ponte di un grafo G non diretto e connesso consiste nel fare una visita del grafo G' ottenuto rimuovendo l'arco. Se G' è sconnesso, l'arco è un ponte, altrimenti non lo è. Se in generale il grafo non è connesso lo stesso ragionamento vale per la componente connessa che contiene l'arco da esaminare (ovvero la visita parte da uno dei due estremi dell'arco). Chiaramente, tale algoritmo ha complessità $O(n + m)$.

Nel caso vogliamo anche trovare un ciclo che contiene l'arco $\{u, v\}$ (se esiste) basterà fare una DFS a partire da u facendo in modo che il primo adiacente scelto sia proprio v. In questo modo la DFS troverà un arco all'indietro che arriva al nodo u e da qui possiamo ricostruire il ciclo come già sappiamo.

Discussione dell'esercizio [pozzo]

- Se un grafo ha un pozzo universale u allora per un qualsiasi altro nodo v c'è l'arco (v, u) che essendo un arco uscente da v impedisce che v possa essere un pozzo universale.
- Se consideriamo due nodi qualsiasi u e v e ci chiediamo se c'è un arco da u a v in base alla risposta possiamo escludere con certezza che uno dei due nodi sia il pozzo (se l'arco c'è escludiamo u altrimenti escludiamo v). A questo punto l'idea di un algoritmo per trovare il pozzo universale, se esiste, è molto semplice. Scegliamo due nodi, diciamo u e v, e vediamo se c'è l'arco da u a v. Per quanto detto almeno uno dei due nodi sarà scartato e quindi scegliamo un altro nodo w. Applichiamo la stessa procedura ai due nodi in esame scartandone almeno uno. Continuando così fino a considerare tutti i nodi, alla fine o rimarremo con un solo nodo oppure li avremo scartati tutti e il pozzo universale non c'è. Se rimaniamo con un nodo dobbiamo solamente verificare che sia il pozzo universale.

```
POZZO(A: matrice di adiacenza del grafo)
    p <- un qualunque nodo del grafo
    FOR ogni nodo u DO
        IF u <> p AND A[p][u] = 1 THEN    /* Se c'è l'arco (p, u), p non può */
            p <- u                        /* essere un pozzo universale */
    ENDFOR
    FOR ogni nodo u DO
        IF u <> p AND (A[u][p] = 0 OR A[p][u] = 1) THEN
            RETURN 0
    RETURN p
```

  Chiaramente la complessità dell'algoritmo è $O(n)$.
- Se il grafo è rappresentato tramite liste di adiacenza non è possibile risolvere il problema in tempo $O(n)$ perché per verificare che un nodo sia un pozzo universale bisogna controllare che abbia grado entrante $n - 1$ e questo richiede la scansione delle liste di adiacenza di tutti gli altri nodi.