Progettazione di Algoritmi - lezione 6

Discussione dell'esercizio [sensi unici]

La rete viaria della cittadina può essere rappresentata facilmente tramite un grafo diretto in cui i nodi sono gli incroci di due o più strade e ogni arco corrisponde a una strada (tra due incroci). Allora la proprietà che vorrebbe il sindaco equivale alla forte connessione del grafo. Si osservi che non fa differenza se due punti A e B che si vogliono connettere sono sugli incroci o sulle strade.

Come possiamo verificare se un grafo è fortemente connesso? Un modo semplice consiste nel fare una visita a partire da ogni nodo del grafo. Se tutte le visite raggiungono tutti i nodi del grafo, allora il grafo è fortemente connesso. Altrimenti non lo è. Ma questo algoritmo è piuttosto inefficiente, infatti richiede n visite e quindi tempo $O(n(n + m))$.

Ma non è necessario effettuare così tante visite, cioè una per ogni nodo del grafo. Se un grafo G è fortemente connesso, fissando un nodo u, sappiamo che tutti i nodi di G sono raggiungibili da u e che da ogni nodo si può raggiungere u. Inoltre la validità di questa proprietà per uno specifico nodo, implica che il grafo è fortemente connesso. Infatti, dati due qualsiasi nodi v e w, si può ottenere un cammino che va da v a w concatenando un cammino da v a u con uno da u a w e tali cammini esistono in virtù della proprietà suddetta; in modo simmetrico si ottiene un cammino da w a v. Riassumendo abbiamo un algoritmo per verificare se un grafo G è fortemente connesso: facciamo una visita da un nodo fissato u per verificare che tutti i nodi sono raggiungibili da u e poi, per verificare che da ogni nodo u è raggiungibile, basta fare una visita da u del grafo trasposto.

```
DFS_SC(G: grafo diretto)
    VIS: array di bool dei nodi visitati, inizializzato a false
    DFS(G, u, VIS)           /* Visita da un nodo arbitrariamente scelto u */
    FOR ogni nodo v DO
        IF VIS[v] = false THEN    /* Se v non è raggiungibile da u, */
            RETURN false          /* allora G non è fortemente connesso */
    GT <- TRASP(G)           /* Ritorna il grafo trasposto di G */
    FOR ogni nodo v DO
        VIS[v] <- false
    DFS(GT, u, VIS)          /* Visita da u del grafo trasposto */
    FOR ogni nodo v DO
        IF VIS[v] = false THEN   /* Se u non è raggiungibile da v, */
            RETURN false         /* allora G non è fortemente connesso */
    RETURN true
```

La trasposizione del grafo costa $O(n + m)$ e sono effettuate solamente due visite, per cui la complessità dell'algoritmo è $O(n + m)$.

Se però il grafo non è fortemente connesso e vogliamo trovare le componenti fortemente connesse? Ad esempio, nel caso del problema da cui siamo partiti, sarebbe utile conoscere le componenti connesse per sapere se possiamo garantire la forte connessione invertendo alcuni sensi unici. L'estensione dell'algoritmo appena visto per trovare le componenti fortemente connesse, non è efficiente.

Componenti fortemente connesse

Un algoritmo efficiente per le componenti fortemente connesse di un grafo diretto non è facile quanto quello per le componenti connesse di un grafo non diretto. Ci sono molti algoritmi efficienti per risolvere tale problema e quelli più noti sono tutti basati sulla DFS. L'algoritmo che vedremo si deve a Robert Endre Tarjan (1972).

Sia G un grafo diretto. Per ogni nodo u, sia $C(u)$ l'insieme dei nodi della componente fortemente connesse di u. Sappiamo che le componenti fortemente connesse partizionano l'insieme dei nodi di G. Consideriamo una DFS di G a partire da un nodo qualsiasi. Per ogni nodo u, denotiamo con $Tree(u)$ l'insieme dei nodi del sottoalbero della

DFS da u. Diciamo che un nodo u è una *c-radice* (rispetto alla DFS) se u è il primo nodo di $C(u)$ a venir visitato dalla DFS. In altri termini, u è una c-radice se per ogni v in $C(u)$, $t(v) > t(u)$; dove come al solito con $t(x)$ denotiamo il tempo d'inizio visita di x. Chiaramente, ogni componente contiene uno e un solo nodo che è una c-radice.

Proprietà 1

Se u è una c-radice, allora valgono le seguenti proprietà.

a. **$C(u)$ è contenuto in $Tree(u)$.**
b. **Se $u_1,\ldots,u_k$ sono tutte le c-radici in $Tree(u)$, allora $Tree(u) = C(u_1) \cup \ldots \cup C(u_k)$.**

Infatti, se u è una c-radice, quando la DFS da u inizia tutti i nodi in $C(u)$ non sono stati ancora visitati e quindi lo saranno durante la DFS da u dato che per ogni v in $C(u)$ c'è un cammino orientato da u a v. Vediamo ora la proprietà (b). Siano $u_1,\ldots,u_k$ tutte le c-radici in $Tree(u)$. Per la proprietà (a), $C(u_i)$ è contenuto in $Tree(u_i)$. Siccome u_i è in $Tree(u)$, si ha che $Tree(u_i)$ è contenuto in $Tree(u)$ e quindi $C(u_i)$ è contenuto in $Tree(u)$. Perciò $C(u_1) \cup \ldots \cup C(u_k)$ è contenuto in $Tree(u)$. Consideriamo ora un qualsiasi v in $Tree(u)$. Se per assurdo v non fosse in nessuna $C(u_i)$, allora la c-radice w di $C(v)$ non è in $Tree(u)$. C'è un cammino da v a w (perchè w e v appartengono alla stessa componente) e ovviamente c'è anche un cammino da u a v (perchè v è in $Tree(u)$). Ne segue che c'è un cammino da u a w. Siccome w non è in $Tree(u)$ e $Tree(w)$ e $Tree(u)$ non sono disgiunti, deve essere che w è un antenato di u. Ciò implica che c'è un cammino da w a u per cui u e w apparterebbero alla stessa componente ma questo è in contraddizione con l'ipotesi che u è una c-radice.

Se avessimo un test per sapere se un nodo u è una c-radice o meno, quando la DFS da u termina, avremmo un algoritmo che determina le componenti. Ogni volta che un nuovo nodo è visitato lo inseriamo in uno stack. In questo modo, per ogni nodo v, tutti i nodi di $Tree(v)$ saranno inseriti nello stack dopo v. Quindi se v è una c-radice, in base alla proprietà 1, al termine della DFS da v tutti i nodi di $C(v)$ sono nello stack dopo v. Se tutte le volte che verifichiamo, al termine della DFS da un nodo v, che v è una c-radice estraiamo dallo stack tutti i nodi fino al nodo v (compreso) siamo sicuri che questi sono proprio i nodi di $C(v)$. Perché? Due casi sono possibili.

- $Tree(v)$ non contiene altre c-radici oltre a v. Allora, quando la DFS da v termina, in base alla Proprietà 1 (b), nello stack ci saranno solamente i nodi di $C(v)$, dato che $Tree(v) = C(v)$.
- $Tree(v)$ contiene anche altre c-radici oltre a v. Sia w una di queste altre c-radici. La DFS da w terminerà prima di quella da v e quando accadrà i nodi di $C(w)$ saranno estratti dallo stack. Perciò, quando la DFS da v termina, saranno rimasti nello stack solamente i nodi di $C(v)$.

Ecco la descrizione di questo (schema di) algoritmo.

```
SCC(G: grafo diretto)
    S <- stack vuoto
    FOR ogni nodo u di G che non è visitato DO
        DFS_SCC(G, u, S)

DFS_SCC(G: grafo diretto, u: nodo, S: stack)
    marca u visitato
    S.push(u)
    FOR ogni adiacente v non visitato di u DO
        DFS_SCC(G, v, S)
    IF u è una c-radice THEN
        C <- lista vuota
        DO                      /* I nodi della componente sono quelli nello */
            w <- S.pop()        /* stack fino a u */
            C.append(w)
        WHILE w <> u
        OUTPUT C                /* I nodi della componente trovata */
```

Osserviamo che ad ogni passo della DFS da un nodo u i nodi che sono stati visitati fino a quel passo possono

essere partizionati nelle seguenti classi.

- Nodi in *Tree*(*u*).
- Nodi non in *Tree*(*u*) la cui componente è già stata determinata, cioè nodi non in *Tree*(*u*) che non sono nello stack.
- Nodi non in *Tree*(*u*) la cui componente non è stata ancora determinata, cioè nodi non in *Tree*(*u*) che sono nello stack.

La seguente proprietà fornisce un test per riconoscere le c-radici.

Proprietà 2

Un nodo *u* non è una c-radice se e solo se durante la DFS da *u* viene attraversato un arco (*v*, *w*) tale che *w* è stato visitato, la componente di *w* non è stata ancora determinata e *t*(*w*) < *t*(*u*).

Se *u* non è una c-radice, allora la c-radice *z* di *C*(*u*) non è in *Tree*(*u*). Siccome *u* e *z* sono nella stessa componente, esiste un cammino da *u* a *z*. Sia $u_0, u_1,\ldots,u_k$ un tale cammino con $u_0 = u$ e $u_k = z$. Sia $w = u_i$ il primo nodo del cammino che non è in *Tree*(*u*) (esiste perchè u_k non è in *Tree*(*u*)). Siccome *w* non può essere u_0 dato che u_0 è in *Tree*(*u*), esiste $v = u_{i-1}$ ed è in *Tree*(*u*). Non essendo *w* in *Tree*(*u*) ma esistendo un cammino da *u* a *w*, quando la DFS da *u* è iniziata *w* doveva essere già stato visitato, altrimenti apparterebbe a *Tree*(*u*). Quindi deve essere *t*(*w*) < *t*(*u*). Inoltre la componente di *w* non è determinata durante la DFS da *u* perchè è la stessa della componente di *u*. Ricapitolando, durante la DFS da *u* è attraversato l'arco (*v*, *w*), tale che *t*(*w*) < *t*(*u*) e la componente di *w* non è stata ancora determinata.

Viceversa, sia *u* una c-radice. Supponiamo per assurdo che durante la DFS da *u* sia attraversato un arco (*v*, *w*) con *t*(*w*) < *t*(*u*) e tale che la componente di *w* non è ancora determinata. Sia *z* la c-radice di *C*(*w*). Siccome *w* non è in *Tree*(*w*) (dato che *t*(*w*) < *t*(*u*)), *z* non è in *Tree*(*u*). Siccome durante la DFS da *u* la componente di *w* non è ancora determinata, questo significa che la DFS da *z* è iniziata (dato che *w* è visitato e per la Proprietà 1 (a), *C*(*z*) è contenuto in *Tree*(*z*) e *w* è in *C*(*z*)) ma non è ancora terminata. Perciò *z* è un antenato di *u*. Quindi c'è un cammino da *z* a *u* (nell'albero della DFS) e anche un cammino da *u* a *z* dato da un cammino da *u* a *v*, l'arco (*v*, *w*) e un cammino da *w* a *z* dovuto al fatto che *w* è nella componente di *z*. Questo implica che *u* e *z* sono nella stessa componente, in contraddizione con l'ipotesi che *u* è una c-radice (*z* sarebbe la c-radice di *C*(*u*)).

Si osservi che l'arco (*v*, *w*) della Proprietà 2 può essere o un arco all'indietro o un arco di attraversamento.

Grazie a questa proprietà abbiamo un test per riconoscere le c-radici. Basterà che durante la DFS da *u* ci manteniamo il tempo minimo di visita *back*(*u*) tra quello di tutti i nodi toccati la cui componente non è stata ancora determinata. Così *u* è una c-radice se solo se *back*(*u*) = *t*(*u*). Per mantenere le informazioni relative ai tempi d'inizio visita e alla determinazione delle componenti, usiamo un array `CC` tale che

1. se *u* non è stato ancora visitato, `CC[u] = 0`;
2. se *u* è stato visitato ma la componente non è stata ancora determinata, `CC[u] = -c`, dove `c` è il tempo di inizio visita di *u*;
3. se la componente di *u* è stata determinata, `CC[u] = cc`, dove `cc` è l'indice della componente.

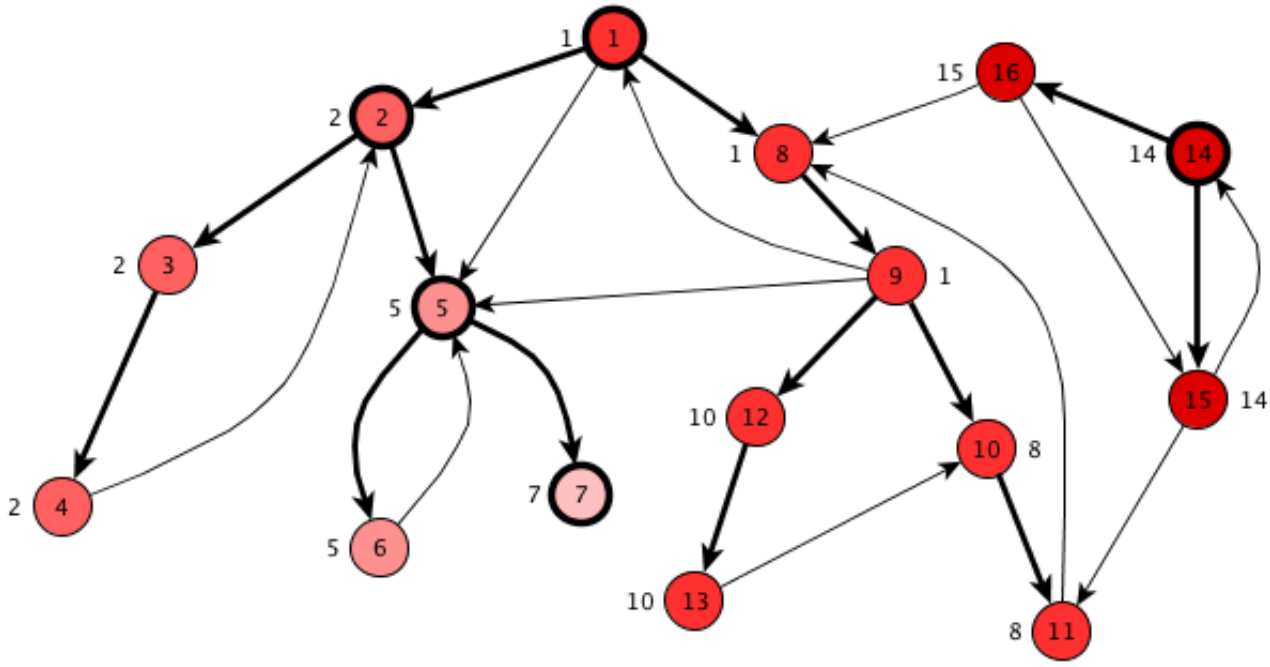
Inoltre, per calcolare *back*(*u*), facciamo in modo che ogni chiamata ricorsiva della DFS su un nodo *u* ritorni *back*(*u*). Ecco quindi l'algoritmo completo:

```
SCC(G: grafo diretto)
    CC: array che darà l'indice della componente di ogni nodo, inizializzato a 0
    nc <- 0       /* Contatore componenti */
    c <- 0        /* Contatore nodi visitati */
    S <- stack vuoto
    FOR ogni nodo u di G DO
        IF CC[u] = 0 THEN
            DFS_SCC(G, u, CC, S, c, nc)
    RETURN CC
```

```
DFS_SCC(G: grafo diretto, u: nodo, CC: array, S: stack, c: cont. nodi, nc: cont. comp.)
    c <- c + 1
    CC[u] <- -c   /* Il tempo di inizio visita in negativo per */
                  /* distinguerlo dall'indice di una componente */
    S.push(u)
    back <- c
    FOR ogni adiacente v di u DO
        IF CC[v] = 0 THEN
            b <- DFS_SCC(G, v, CC, S, c, nc)
            back <- min(back, b)
        ELSE IF CC[v] < 0 THEN   /* La componente di v non è ancora stata determinata */
            back <- min(back, -CC[v])
    IF back = -CC[u] THEN        /* u è una c-radice */
        nc <- nc + 1
        DO                       /* I nodi della componente sono quelli nello */
            w <- S.pop()         /* stack fino a u */
            CC[w] <- nc
        WHILE w <> u
    RETURN back
```

Osserviamo che il numero di operazioni effettuate sullo stack sono O(*n*) dato che ogni nodo del grafo è inserito nello stack una e una sola volta. Quindi la complessità dell'algoritmo è O(*n* + *m*).

Ecco un esempio di esecuzione dell'algoritmo. I nodi sono numerati con i tempi d'inizio visita e accanto ad ognuno c'è il valore di *back*() di quel nodo. Gli archi della DFS sono marcati e anche le c-radici sono marcate. Inoltre, i nodi sono colorati con diverse gradazioni di rosso per indicare gli indici delle componenti (più la gradazione è chiara più basso è l'indice).



Il *grafo delle componenti fortemente connesse* è un grafo diretto i cui nodi corrispondono alle componenti e c'è un arco da un nodo *u* a *v* se c'è almeno un arco dalla componente che corrisponde ad *u* e la componente che corrisponde a *v*. Ovviamente il grafo è un DAG. L'ordine con cui le componenti sono numerate (o se si vuole, determinate) dall'algoritmo è interessante. Infatti, se una componente ha un indice *k* maggiore dell'indice *h* di un'altra componente, allora non ci sono archi dalla componente *h* alla componente *k*. La dimostrazione è lasciata come esercizio. Quindi se ordiniamo le componenti per ordine di indice decrescente (o equivalentemente per ordine di tempo di determinazione decrescente), otteniamo un ordinamento topologico del DAG delle componenti.

Esercizio [broadcast]

Una rete di stazioni radio è costituita da *n* stazioni disposte in un territorio. Una stazione *s* può trasmettere direttamente ad una stazione *t* solo se *s* ha una potenza di trasmissione tale che i segnali radio di *s* raggiungono *t* con sufficiente energia. Le stazioni non sono tutte uguali e hanno potenze differenti. Così una stazione *s* per trasmettere un messaggio a una stazione *t* molto lontana deve trasmettere ad una stazione più vicina che funge da ponte la quale ritrasmette il messaggio o direttamente a *t* (se può) o a un'altra stazione ponte fino a che, eventualmente, il messaggio arriva a *t*. Una stazione e detta *stazione broadcast* se può trasmettere direttamente o indirettamente a tutte le stazioni della rete. Vogliamo un algoritmo che date le specifiche della rete (per ogni stazione *x*, le stazioni alle quali *x* può trasmettere direttamente) trovi, se esistono, tutte le stazioni broadcast.