Progettazione di algoritmi

Discussione dell'esercizio [broadcast]

Possiamo rappresentare il problema tramite un grafo diretto G in cui i nodi sono le stazioni e c'è un arco dal nodo u a v se la stazione che corrisponde a u può direttamente trasmettere alla stazione che corrisponde a v. Il problema chiede di trovare un algoritmo che ritorna, se esistono, i *nodi broadcast*, cioè nodi da cui si possono raggiungere tutti i nodi del grafo.

Un algoritmo molto semplice consiste nel controllare nodo per nodo se è un nodo broadcast. Per controllare se u è un nodo broadcast basta fare una visita a partire da u e verificare che vengono visitati tutti i nodi del grafo. Però questo algoritmo è piuttosto inefficiente perchè richiede n visite con una complessità totale di $O(n(n + m))$.

Osserviamo che se u e v sono due nodi broadcast, devono appartenere alla stessa componente fortemente connessa perchè ci sono cammini da u a v e da v a u. Inoltre, un nodo w che appartiene alla componente in cui c'è un nodo broadcast u deve essere anch'esso un nodo brodcast dato che un qualsiasi nodo z essendo raggiungibile da u è anche raggiungibile da w perché c'è un cammino da w a u. Quindi se nel grafo ci sono nodi broadcast, allora c'è una componente fortemente connessa che è formata dall'insieme di tutti i nodi broadcast. Inoltre, se ci sono anche altre componenti, ci sono cammini che vanno dai nodi della componente broadcast verso tutte le altre componenti. Questo implica che in un qualsiasi ordinamento topologico del DAG delle componenti, la componente broadcast (se esiste) deve necessariamente essere la prima componente.

Queste osservazioni ci portano al seguente algoritmo. Eseguiamo l'algoritmo per determinare le componenti fortemente connesse. L'algoritmo di Tarjan numera le componenti secondo un ordinamanto topologico (in ordine di indice decrescente). Così la componente B con indice più alto è la prima componente dell'ordinamento. Scegliamo un nodo u da B e verifichiamo tramite una visita se tutti i nodi sono raggiungibili da u, In caso affermativo, B è la componente broadcast, altrimenti non ci sono nodi broadcast nel grafo.

Ecco quindi lo pseudo-codice dell'algoritmo che ritorna in una lista tutti i nodi broadcast (se non ci sono nodi broadcast, ritorna la lista vuota):

```
DFS_BROAD(G: grafo diretto)
    CC <- SCC(G)            /* calcola l'array delle componenti con l'algoritmo di Tarjan */
    nc <- 0
    u <- 0
    FOR ogni nodo v DO      /* cerca la prima componente, cioè quella di indice più alto */
        IF CC[v] > nc THEN
            nc = CC[v]
            u <- v
    vis <- DFS(G, u)        /* esegue la DFS da u e ritorna il numero di nodi visitati */
    B <- lista vuota
    IF vis = n THEN             /* se da u si raggiungono tutti i nodi */
        FOR ogni nodo v DO      /* salva in B i nodi della componente */
            IF CC[v] = nc THEN
                B.append(v)
    RETURN B
```

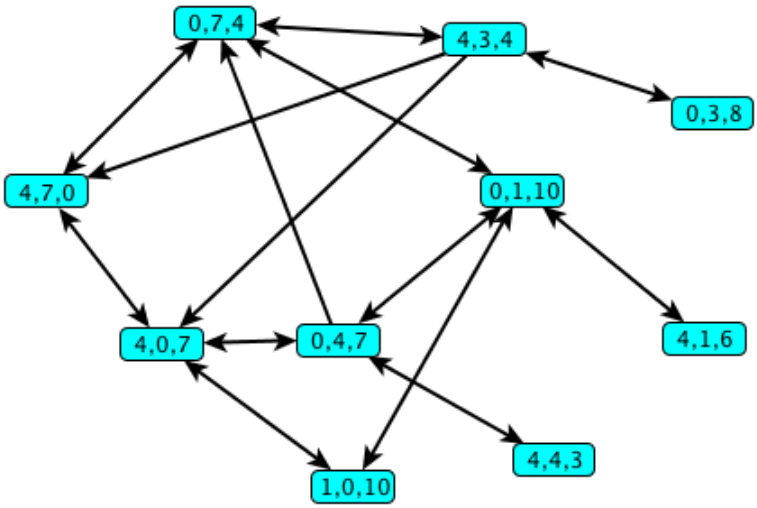
Siccome la complessità dell'algoritmo di Tarjan è lineare come quella della visita DFS, la complessità dell'algoritmo è $O(n + m)$.

Il problema delle distanze

Nelle prime lezioni abbiamo visto alcuni problemi come il 9-puzzle

```
-------------                              -------------
| 7 | 3 | 9 |                              | 1 | 2 | 3 |
|---|---|---|                              |---|---|---|
| 2 | 5 | 1 |     --->. . . . . . . --->   | 4 | 5 | 6 |
|---|---|---|                              |---|---|---|
| 4 | 8 | 6 |                              | 7 | 8 | 9 |
-------------                              -------------
```

o il problema dei tre contenitori d'acqua



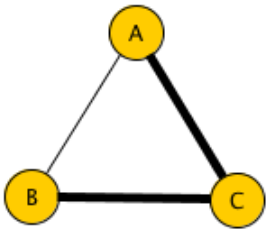
in cui ci si chiedeva se da una data configurazione fosse possibile raggiungere, effettuando mosse ammissibili, delle configurazioni target. Per risolvere tali problemi è stato sufficiente fare una visita a partire dalla data configurazione nel grafo di tutte le configurazioni connesse dalle mosse ammissibili e controllare se si raggiunge una delle configurazioni target. Ma se non ci accontentiamo di conoscere che una configurazione target è raggiungibile e vogliamo trovare il numero *minimo* di mosse per raggiungerla? Lo stesso problema si presenta, ad esempio, se vogliamo trovare il percorso più breve (inteso come minimo numero di bivi che devono essere attraversati) per uscire da un labirinto.

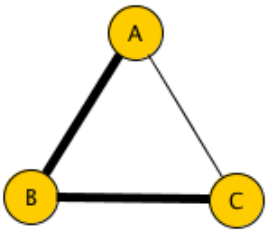
In termini generali, il problema può essere formulato così: dato un grafo (diretto o non diretto) e dati due nodi u e v si vuole trovare un *cammino di lunghezza minima* da u a v, se ne esiste almeno uno. Per *lunghezza di un cammino* (orientato o non orientato) si intende il numero di archi del cammino. Spesso la lunghezza minima di un cammino da u a v è detta *distanza* da u a v ed è denotata $d(u, v)$. Se non ci sono cammini da u a v, si conviene che $d(u, v) = \infty$. Ovviamante, risulta sempre $d(u, u) = 0$. Se il grafo non è diretto le distanze sono simmetriche, cioè, per qualsiasi u e v, $d(u, v) = d(v, u)$. Invece, per grafi diretti le distanze non sono simmetriche, in generale. Una proprietà che vale per ogni tipo di grafo è la *disuguaglianza triangolare*:

$$\text{per qualsiasi } u, v \text{ e } w,\ d(u, v) \leq d(u, w) + d(w, v)$$

La validità di questa proprietà deriva semplicemente dal fatto che possiamo sempre concatenare un cammino da u a w con uno da w a v e ottenere un cammino da u a v.

Per determinare le distanze tra i nodi di un grafo la visita DFS non è adatta perchè i cammini che trova non sono in generale di lunghezza minima. Ad esempio, nel grafo qui sotto sono marcati i due possibili alberi della DFS a partire dal nodo A.





In entrambi i casi, per uno dei due nodi *B* o *C* il cammino trovato non è di lunghezza minima. Per determinare le distanze occorre un altro modo di visitare un grafo.

Visita in ampiezza (BFS)

Per cercare di visitare i nodi di un grafo partendo da un nodo *u* e trovando i cammini di lunghezza minima da *u*, dovremmo prima di tutto visitare i nodi a distanza 1 da *u*, che sono tutti gli adiacenti di *u*. Avendo determinato tutti i nodi a distanza 1, i nodi a distanza 2 sono quelli (diversi dai primi) che sono adiacenti ai nodi a distanza 1. In generale, avendo determinati i nodi a distanza *k*, quelli a distanza $k + 1$ sono i nodi (diversi da tutti quelli già classificati) che sono adiacenti ai nodi a distanza *k*. Per effettuare questo tipo di visita basterà mantenere in una coda i nodi visitati i cui adiacenti non sono stati ancora esaminati. Ad ogni passo viene prelevato il primo nodo della coda e sono esaminati i suoi adiacenti e se si scopre un nuovo nodo viene aggiunto alla coda. Grazie all'uso della coda siamo sicuri che verranno considerati tutti gli adiacenti dei nodi a distanza *k* prima di considerare gli adiacenti dei nodi a distanza $k + 1$. Questo tipo di visita si chiama **visita in ampiezza** (in inglese *Breadth First Search*, abbreviato *BFS*).

Ecco una implementazione della BFS che ritorna sia l'array delle distanze che l'albero di visita rappresentato tramite vettore dei padri. Assumiamo per semplicità che tutti i nodi del grafo siano raggiungibili da *u*.

```
BFS(G: grafo, u: nodo)
    P: vettore dei padri, inizializzato a 0
    Dist: array delle distanze
    P[u] <- u           /* u é la radice del'albero della BFS */
    Dist[u] <- 0
    Q <- coda vuota
    Q.enqueue(u)
    WHILE Q non è vuota DO
         v <- Q.dequeue()
         FOR ogni adiacente w di v DO
             IF P[w] = 0 THEN
                 P[w] <- v
                 Dist[w] <- Dist[v] + 1
                 Q.enqueue(w)
   RETURN P, Dist
```

Il vettore dei padri rappresenta l'albero di visita della BFS e quindi per ogni nodo *v* contiene un cammino di lunghezza minima da *u* a *v*. Per ricostruire un tale cammino basta partire da *v* e percorrerlo di padre in padre tramite `P` fino alla radice *u*:

```
L <- lista vuota
L.head(v)     /* aggiunge v in testa alla lista */
WHILE v <> u
    v <- P[v]
    L.head(v)
RETURN L     /* ritorna in L il cammino da u a v */
```

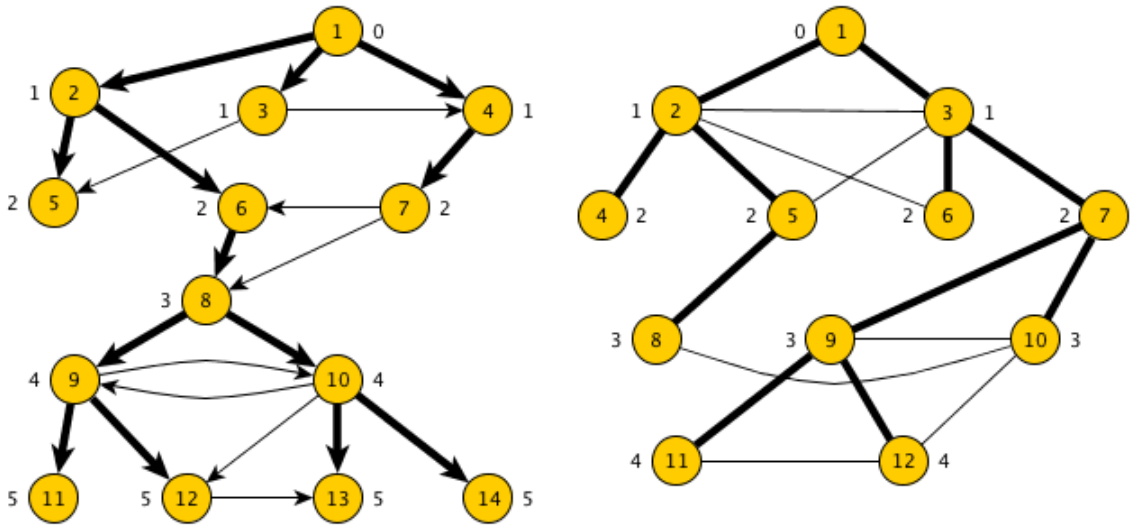
Per quanto riguarda la complessità della BFS facciamo le seguenti osservazioni. Le inizializzazione degli array `P` e `Dist` costano O(n). La coda `Q` può essere implementata in modo tale che le operazioni su di essa hanno costo costante. Il `WHILE` esegue esattamente n iterazioni perchè ogni nodo viene aggiunto alla coda `Q` una sola volta e ad ogni iterazione un nodo è estratto da `Q`. In ogni iterazione del `WHILE` il `FOR` interno fa un numero di iterazioni pari al numero degli adiacenti del nodo estratto in quella iterazione del `WHILE`. Perciò il numero complessivo di iterazioni del `FOR` è la somma dei gradi (per un grafo diretto, i gradi uscenti) dei nodi e quindi O(m). In totale, la complessità è dunque O(n + m).

La dimostrazione della correttezza dell BFS può essere fatta per induzione dimostrando che l'algoritmo della BFS partendo da un nodo u soddisfa il seguente enunciato:

> Per ogni k = 0,1,2,… c'è un passo dell'algoritmo della BFS in cui
>
> a. ogni nodo v con $d(u, v) \leq k$ è stato visitato ed ha `Dist[v]` = $d(u, v)$;
> b. la coda `Q` contiene esattamente i nodi a distanza k da u.

La dimostrazione è per induzione su k. Per k = 0, prima di iniziare il `WHILE` le proprietà (a) e (b) sono soddisfatte perchè u, l'unico nodo a distanza 0, ha `Dist[u] = 0` ed è l'unico nodo nella coda. Supponiamo per ipotesi induttiva che le ci sia un passo per cui le proprietà (a) e (b) valgono per k e dimostriamole per k + 1. Se l'insieme dei nodi a distanza k è vuoto, le proprietà (a) e (b) valgono perchè la coda è vuota e anche l'insieme dei nodi a distanza k + 1 è vuoto. I nodi a distanza k + 1, se ci sono, sono esattamente quelli adiacenti a nodi a distanza k che non sono a distanza $\leq k$. Ogni volta che un nodo v a distanza k è estratto dalla coda ogni adiacente w di v è esaminato e se non è stato già visitato, `P[w] <> 0`, e quindi, per l'ipotesi induttiva, la sua distanza non è $\leq k$, la distanza `Dist[w]` è impostata a k + 1 (cioè, $d(u, w)$) ed è aggiunto alla coda. Ci sarà un passo in cui tutti i nodi a distanza k sono stati estratti dalla coda e i loro adiacenti sono stati esaminati. In quel passo, le proprietà (a) e (b) valgono per k + 1.

Qui sotto ci sono due esempi di visite BFS, una su un grafo diretto e una su un grafo non diretto. I nodi sono numerati in ordine di visita, gli archi dell'albero (o arborescenza) della BFS sono marcati e accanto ad ogni nodo c'è la distanza dal nodo di partenza (il nodo 1).



Esercizio [labirinto]

In un labirinto ci possono essere diversi percorsi minimi, cioè che attraversano un numero minimo di bivi, per andare dall'entrata all'uscita. Trovare un algoritmo efficiente che calcola il numero di percorsi minimi in un labirinto.