TECNICHE DI PROGRAMMAZIONE

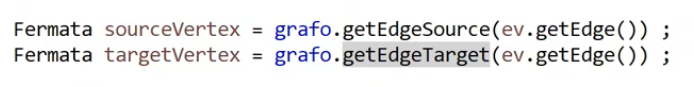
07/05/19

Continuazione albero di visita (esercizio) e cammini minimi nei grafi.

Abbiamo deciso di creare una classe “EdgeTraversedGraphListener” in cui ci interessava andare ad osservare quando l’algoritmo attraversava un arco. Dovevamo però fare dei controlli per verificare se, ogni volta che l’algoritmo visitava un arco, se questa visita aveva portato alla scoperta di qualcosa di nuovo. Questo lo sintetizziamo in una mappa (back) che per ogni vertice appena scoperto mi restituisci il vertice da cui è stato scoperto. Quindi è una mappa che punta dall’alto verso il basso. L’arco che ricevo, affinché sia una nuova relazione interessante, deve avere il vertice il vertice padre (quello da cui parto) noto mentre il vertice figlio non ancora noto. Noi quindi dobbiamo estrarre il vertice di partenza e il vertice di arrivo. Per fare questo dal nostro evento “ev” del metodo “edgeTraversed” estraggo l’arco. Per estrarlo devo chiederlo al grafo. Il problema è che il TraversalListener non sa qual è il grafo. Nel costruttore quindi, dovrò accettare un altro parametro il più (grafo su cui sto lavorando). Quindi:

Immagine che contiene screenshot

Descrizione generata automaticamenteOra andiamo nel metodo “EdgeTraversed” e troviamo il vertice di partenza e quello di arrivo mediante il seguente codice:



Ora se il grafo è orientato, il source sarà il padre mentre il target sarà il figlio. Se invece il grafo non è orientato, potrebbe essere anche il contrario.

Noi dobbiamo essere sicuri che il figlio non sia stato ancora usato come chiave della mappa e che il padre esista. Nel caso in cui questa condizione venga rispettata, aggiungo alla mappa l’informazione che quel determinato figlio si raggiunge a partire da quel determinato padre. Poi ho l’altro caso in cui la condizione è la stessa ma si invertono il vertice padre e il vertice figlio per tenere conto anche del caso di grafo non orientato. Il codice quindi sarà il seguente:

Immagine che contiene screenshot

Descrizione generata automaticamente

Quindi man mano che l’iteratore va avanti, salva nella mappa gli archi che vengono attraversati. Ora dobbiamo fare in modo che l’iteratore comunichi con il Listener (per salvare gli archi che ci interessano). Ritorniamo quindi nel Model e agganciamo il Listener. Dobbiamo sempre agganciarlo dopo aver creato l’iteratore e prima di farlo lavorare con il .hasNext() del while. Per fortuna abbiamo un metodo “addTraversalListener” che fa proprio quello che vogliamo noi. Aggiungo un ascoltatore all’iteratore (non al grafo!) creando un’istanza della classe “EdgeTraversedGraphListener” a cui passo il grafo e la mappa. Come ultima cosa, dopo il while, inseriamo un System.out.println per visualizzare la mappa.

Immagine che contiene screenshot

Descrizione generata automaticamente

Se runniamo il programma notiamo che la mappa è vuota. Questo perché ci siamo dimenticati di scrivere il vertice di partenza. Torniamo quindi nel Model e prima di iniziare l’iterazione, aggiungiamo alla mappa il primo valore sorgente.



Se runniamo il programma ora, tutto va bene e ci viene restituita la mappa. Noi vogliamo però che la mappa venga conservata anche dopo il processo di creazione e quindi non la dichiaro come variabile privata ma come variabile all’interno del Model cambiando il nome in backVisit. Di conseguenza andiamo a cambiare il nome a tutte le righe di codice in cui è presente la mappa.

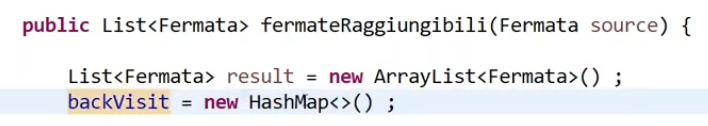
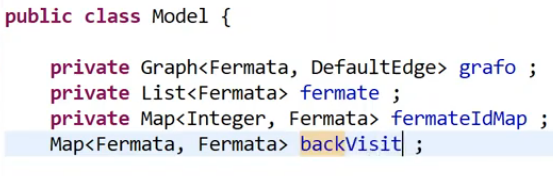


Immagine che contiene screenshot

Descrizione generata automaticamente

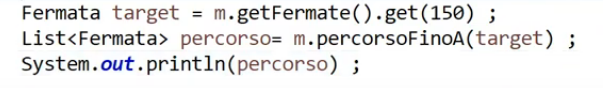
Così facendo possiamo creare un metodo che ci possa restituire il percorso fino ad una certa fermata. Per prima cosa dobbiamo chiederci se la destinazione a cui vogliamo arrivare è compresa nella mappa.

In caso negativo restituiamo “null” mentre in caso positivo restituiamo una lista con tutte le fermate che dobbiamo attraversare. In essa aggiungeremo sicuramente il target finale e poi dovremo passare a suo “padre” per potere descrivere il percorso da fare usando “backVisit”. Il nostro metodo finale sarà quindi il seguente:

Immagine che contiene screenshot

Descrizione generata automaticamente

Andiamo a testare il nostro programma dal “TestModel” inserendo come target una fermata a caso (150).



Quindi il nostro TestModel, dato un vertice sorgente, calcola tutte le fermate raggiungibili e data una fermata di arrivo, calcolo il percorso da svolgere partendo dal vertice sorgente:

Immagine che contiene screenshot

Descrizione generata automaticamente

Avessimo usato la visita in profondità avremmo ottenuto lo stesso risultato ma con più fermate attraversate, in quanto la visita in ampiezza è l’unica visita che ci permette di trovare il cammino minimo; quindi non sarebbe cambiato molto nel risultato, in quanto saremmo comunque riusciti a trovare il percorso per arrivare ad un target, solamente attraverso un altro percorso probabilmente più lungo.

In realtà tutto questo lavoro fatto per trovare le fermate attraversate tra 2 vertici (sorgente-destinazione), può essere svolto da un metodo “getSpanningTreeEdge.

Mediante questo metodo risparmieremmo la costruzione della mappa in quanto il metodo si occupa già di questo. Il corrispondente della nostra mappa “backVisit” è il metodo “getParent” che, dato un vertice figlio, restituisce il vertice padre o “null” nel caso in cui il vertice sia il sorgente.

La classe “EdgeTraversedGraphListener” ci serve solo per essere chiamata nel Model. Per questo motivo, noi non dobbiamo per forza creare un nuovo file con la nostra nuova classe, ma possiamo creare una classe Privata all’interno del Model. Così facendo non abbiamo neanche bisogno di passare tutte le variabili (grafo, backVisit ecc) in quanto esse sono già presenti nel Model (perché sarebbe una sottoclasse del Model). Le soluzioni sono entrambe valide (preferibile classe privata all’interno del Model).

Questo programma è terminato (ci sarebbe ancora da collegare il programma alla pressione del bottone corrispondente ma si può fare da soli).

Nel caso particolare in cui abbiamo un grafo non pesato e lo analizziamo con una visita in ampiezza, il percorso che otteniamo da un vertice di partenza a un vertice di arrivo è il percorso che ha la minima lunghezza.

Ma se non ci troviamo in questo caso (Visita non in ampiezza o grafico pesato)?

Come prima cosa dobbiamo definire il cammino minimo: il percorso più corto tra due vertici “v” e “u” e tale da avere il peso minore possibile.

Il cammino minimo tra due vertici “u” e “v” si rappresenta con il simbolo: 

Nel caso in cui il vertice “v” non sia raggiungibile dal vertice “u” il valore del cammino minimo è infinito.

In base ai problemi ci può essere chiesto quale sia il valore del peso del cammino minimo (es: 46) oppure quale sia il vero e proprio cammino minimo (la sequenza dei vertici per i grafi semplici o la sequenza di archi per i multigrafi).

Un algoritmo che mi calcola i cammini minimi, mi deve restituire la mappa dei padri (backVisit nell’esercizio di prima) e il peso del vertice (cioè il peso del cammino minimo che arriva fino a quel punto).

La mappa dei padri di un vertice “v” è indicata con il simbolo:  .

Se ci vengono dati tutti i cammini minimi, grazie al “Lemma sui cammini minimi”, possiamo dire che: conoscendo il cammino minimo tra “A” e “B”, prendendo un punto intermedio tra “A” e “B” e lo chiamiamo “C”, sappiamo immediatamente qual è il cammino medio tra “A” e “C” e tra “C” e “B”.

Quindi, in modo più operativo, tutti i cammini minimi da Torino a qualunque città che passano da Genova avranno come tratta comune Torino-Genova (poi dopo si divideranno).

Fare attenzione ai cammini minimi negativi (ultimo caso in figura). In questo caso il cammino minimo non esiste! (Raramente ci capiteranno negli esercizi).

Immagine che contiene cielo, sedendo

Descrizione generata automaticamente

Nel package org.jgrapht.alg.shortestpath troviamo l’elenco dei metodi che implementano algoritmi di ricerca di cammini minimi.

FINE.