TECNICHE DI PROGRAMMAZIONE

06/05/19

Visite nei grafi. Albero di vista.

Nella prima parte della lezione viene risolto il problema riscontrato 2 lezioni prima nel quale al posto del nome dei vertici veniva visualizzato “null”. Bisogna dimenticarsi di quel “trucchetto” usato per risolvere il problema degli oggetti parzialmente inizializzati. Questo perché non funziona quando dobbiamo usare i grafi (spiegato nella documentazione del sito di Jgrapht). Nella spiegazione si può leggere che l’informazione sugli archi viene memorizzata nei vertici ma anche che **l’informazione sui vertici viene memorizzata negli archi**!

I vertici incidenti vengono presi dai parametri della funzione e non dal grafo stesso. Per questo motivo i parametri della funzione devono essere dei vertici propriamente formati (completamente formati).

**Quindi, la soluzione migliore da usare quando lavoriamo con i grafi è quella vista nella lezione precedente: l’uso dell’Identity Map.**

Su GitHub è stata caricata la versione dello stesso programma con l’uso della mappa. Prima, infatti, la funzione “stazioniArrivo” prendeva come parametro una “Fermata partenza” e restituiva un elenco di fermate di arrivo. La modifica è stata quella di passare come parametro anche l’Identity Map che corrisponde alle fermate. Così facendo, quando dal Database ricavo l’id della stazione di arrivo, quest’ultimo viene usato immediatamente per fare una ricerca nella Mappa che mi restituisce l’oggetto “Fermata” corrispondente.  
**IMPORTANTE: ogni volta che, da un valore di un elemento, voglio trovare l’elemento stesso, non devo MAI scandire la lista ma devo usare l’Identity Map.**

Noi dobbiamo gestire in parallelo due strutture dati: l’elenco di fermate, (la lista) e la mappa che ci permette di avere l’accesso diretto ad un oggetto conoscendone la chiave.

La differenza tra la lista e la mappa è che la lista la posso ricercare solo in modo sequenziale mentre la mappa la posso indirizzare solo in modo diretto (con la chiave).

La lista la uso se ho bisogno delle Fermate in un certo ordine. La collecton .values della mappa non ci garantisce l’ordine degli oggetti.

Quando creo l’elenco delle fermate con il metodo del DAO “getAllFermate”, spendo qualche secondo per trasferire tutte le informazioni delle fermate sulla mappa.

**Immagine che contiene interni, screenshot

Descrizione generata automaticamente**

Quindi con il metodo “getAllFermate” io creo tutti gli oggetti di tipo “Fermata” e poi non ne creo più altri.

Quando ho bisogno dei vertici (mi servono degli oggetti di tipo “Fermata”) non sono oggetti nuovi, ma sono quelli che già avevo e che sono ripescati dalla Mappa. Nel metodo “stazioniArrivo” nel quale dovevo recuperare degli oggetti “Fermata” faccio il “get” dalla Mappa!

Immagine che contiene screenshot

Descrizione generata automaticamente

Finita la parentesi su questo esercizio.

Una volta creato il grafo, quali sono le operazioni che posso eseguire su di esso?

In questa lezione impareremo a visitare questa struttura dati ed a elencare i suoi elementi in un certo ordine che tenga conto delle connessioni.

Mentre una Lista si può visitare in un modo solo (si itera dal primo elemento all’ultimo), in un grafo l’ordino con cui visito il grafo può essere diverso. Come **VISITA** noi intendiamo un’esplorazione sistematica degli elementi di un grafo. In esso possiamo scegliere un vertice sorgente e da esso dobbiamo decidere dove andare.

Se abbiamo come obiettivo quello di toccare tutti i vertici del grafo, dobbiamo trovare un meccanismo per evitare di passare due volte dagli stessi vertici e di dimenticarmi dei pezzi di grafo.

Abbiamo 2 casi estremi di Visite (in mezzo ci sono gli altri tipi di visite):

1. **Visita in ampiezza**: partendo da un vertice sorgente dobbiamo trovare tutti i vertici raggiungibili da questo attraversando gli archi del grafo. Vengono costruiti una serie di livelli corrispondenti ai vertici che si trovano ad una certa distanza rispetto al nodo sorgente. Il nodo sorgente è di livello 0 mentre tutti i nodi che si trovano, per esempio, a distanza 3 dal nodo sorgente, saranno di livello 3. Ogni volta che ho un livello, posso trovare i vertici che si trovano al livello successivo. I vertici di livello 6 sono quindi l’insieme dei vertici non ancora visitati e adiacenti ad almeno un vertice di livello 5. Abbiamo un ciclo while che si ripete fino a quando non si finiscono i vertici. Grazie a questo metodo ho calcolato, dato un vertice sorgente di partenza, qual è il numero minimo di archi per raggiungere qualsiasi nodo raggiungibile dell’arco. Ogni volta che trovo un nuovo vertice, devo memorizzare anche l’arco compreso tra i due, in quanto è un arco importante. L’insieme di questi archi che ho usato per scoprire nuovi vertici, costituisce un **ALBERO**. Un albero è un insieme connesso e non ciclico. L’output di questa visita è composto da due informazioni:
   1. Quali sono i vertici raggiungibili.
   2. Come sono raggiungibili questi vertici.

La procedura in ampiezza mi permette quindi di trovare i Cammini Minimi (un tipo particolare dove la lunghezza del cammino coincide con il numero di archi).

Per esempio, nell’immagine seguente, il cammino minimo per raggiungere “u” partendo da “s” è (s-w-t-u) e non (s-w-x-y-u). L’albero è formato dalle righe rosse!

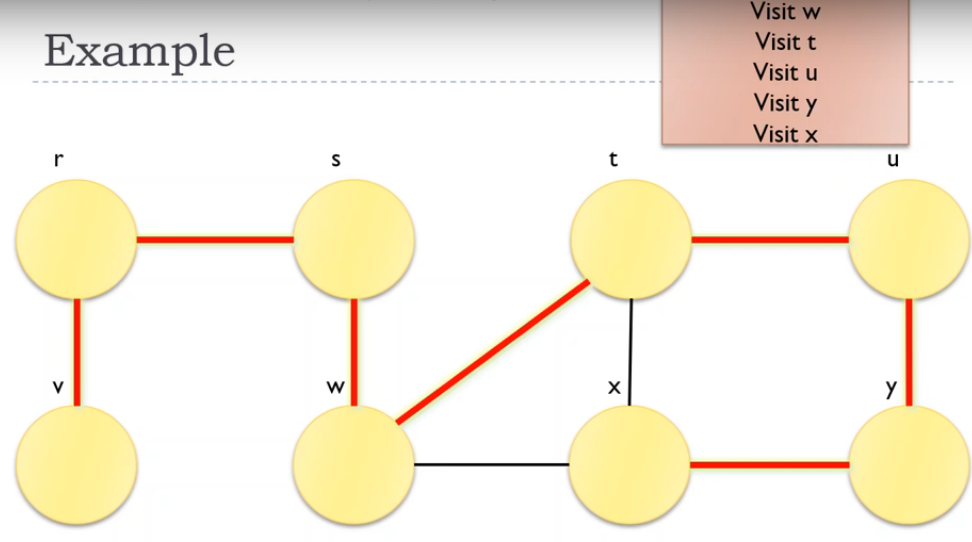
Immagine che contiene cielo, tavolo

Descrizione generata automaticamente

Qual è la complessità di questo grafo qui?

Dipende da come è implementato il metodo del calcolo dei vertici adiacenti (descritto in precedenza). Se io avessi, infatti, un’implementazione del grafo che, dato un insieme di vertici, mi restituisce un insieme di vertici, allora il numero di iterazioni del ciclo è uguale al numero di livelli del grafo (profondità del grafo). Ad ogni livello, il tempo che spendo è proporzionale al numero di vertici di quel livello.

1. **Visita in Profondità**: in essa avrò sempre un vertice sorgente. Da questo vertice certo un suo adiacente. Una volta che mi trovo sul vertice adiacente, cerco il vertice a lui adiacente e così via. Vado avanti così fino a quando non trovo un vertice che non abbia più connessioni verso vertici non ancora visitati. A questo punto faccio backtracking e controllo se nell’andare avanti così avessi scartato delle vie laterali che mi portano a vertici nuovi. L’algoritmo consiste quindi nel prendere, dato un vertice sorgente, tutti i vertici ad esso adiacenti (lavora su un vertice alla volta e non su un insieme) e chiamo la procedura di visita ricorsiva su ciascuno dei vertici adiacenti (a meno che non siano stati già visitati). Questa visita si conclude quando trovo un vertice che non ha più nessun vertice adiacente e quando non ha più vertici adiacenti che portano a vertici nuovi non ancora visitati. Se un vertice possiede più di un vertice adiacente, viene scelto il vertice che viene prima in ordine alfabetico. Un esempio di visita in profondità è il seguente:



Il primo percorso che viene fatto è il (s-r-v) mentre il secondo è il (s-w-t-u-y-x). Quindi, man mano che uso un arco per scoprire un vertice, me lo segno e ottengo così un albero connesso e non ciclico che è appunto l’albero della visita in profondità (che non c’entra niente con i cammini minimi). Logicamente, in base al criterio usato per scegliere l’ordine dei vertici adiacenti, l’albero cambierà di conseguenza.

La complessità degli alberi ottenuti da queste visite è sempre O(n).

Ma come sono stati implementati questi algoritmi di visita dentro la libreria JgraphT?

Dal punto di vista della libreria, una visita è implementata attraverso un iteratore. Quest’ultimo è una classe Java che ci permette di iterare sugli elementi di una Lista. Quindi attraversare un grafo attraverso un particolare algoritmo di visita, non è per niente diverso rispetto ad iterare una Lista fino all’ultimo elemento. Ho molti iteratori in quanto cambiano rispetto al tipo di visita che voglio fare.

Nel package org.jgrapht.traverse ho un’interfaccia generale (GraphIterator) e una serie di classi concrete che contengono tutto le diverse iterazioni che dipendono dalla visita che vogliamo eseguire.

Queste classi avranno sicuramente un metodo next() e un metodo hasNext() che ci permettono di navigare nel grafo.

Vediamo ora un’applicazione nel nostro programma della metropolitana di Parigi.

Nel Model del programma creiamo un nuovo metodo che ci restituisce una lista di “Fermata” e che ci permettere di conoscere tutte le stazioni raggiungibili da una stazione scelta.

Questo metodo lo implemento creando un iteratore (ad esempio quello in ampiezza). Aggiungiamo quindi l’interfaccia GraphIterator. “BreadthFirstIterator” è la classe concreta che riceve come parametro il grafo su cui iterare.

Immagine che contiene screenshot

Descrizione generata automaticamente

Questo comando quindi crea un nuovo iteratore,lo associa al nostro grafo e lo inizializza con un punto di partenza. Creiamo una nuova lista di risultati “result” e in essa inseriamo tutti i risultati ottenuti dal while. Quest’ultimo ha come obiettivo quello di iterare il grafo con il metodo hasNext(). Il risultato finale è:

Immagine che contiene screenshot

Descrizione generata automaticamente

Il metodo. next() fa due cose:

1. Restituisce il prossimo elemento.
2. Avanza l’iteratore all’elemento successivo.

Il metodo hasNext(), invece, ci dice se c’è un elemento successivo oppure no.

Questo metodo dobbiamo testarlo. Lo andiamo a richiamare nella classe TestModel. Prima visualizziamo i vertici e gli archi totali del nostro grafo e poi andiamo a vedere, dato un vertice, le fermate raggiungibili.





Però io devo far sì che il Modello mi dica o mi permetta di ricercare le fermate che abbiamo caricato nel Database. Praticamente, mi servono dei metodi di get() in quanto l’elenco di tutte le fermate noi ce l’abbiamo già. Notiamo che il Getter delle Fermate ce l’abbiamo già. Decidiamo quindi di prendere come esempio per testare il nostro metodo, la prima fermata che il metodo “getFermate” trova. Il codice da inserire nel nostro TestModel è quindi il seguente:

Immagine che contiene screenshot

Descrizione generata automaticamente

Nella classe Iteratore vengono definiti anche una serie di eventi che vengono generati dall’iteratore stesso. A questi eventi io posso agganciare un “Gestore di Eventi”. Quindi, per esempio, quando l’algoritmo trova un nuovo vertice, chiamo il mio metodo corrispondente che mi informa.

Abbiamo un’interfaccia che si chiama “TraversalListener” e dalla quale è possibile chiamare un insieme di metodi come “EdgeTraversed” che ci informa che uno specifico arco è stato attraversato. A noi non interessano tanto i vertici che vengono attraversati (ci vengono via via ritornati), ma gli archi attraversati. Posso definire quindi una classe Listener che va ad intercettare questo evento e ogni volta ci dice quale arco stiamo attraversando.

Andiamo quindi su Eclipse e creiamo la classe “EdgeTraversedGraphListener” (package Model) ma prima di premere “finish” premiamo su “Add…” a fianco del riquadro “Interfaces” e aggiungiamo l’interfaccia “TraversalListener”. Quando creiamo questa classe, conterrà subito degli errori in quanto dobbiamo definire ancora i vertici (Fermata) e gli archi (DefaultEdge) e dobbiamo implementare i metodi che vuole la classe (consigliati). Andiamo ora ad implementare in particolare il metodo “edgeTraversed”. Quest’ultimo dovrà restituirci un arco. Di questo arco dobbiamo estrarre i 2 vertici a cui è collegato e questi 2 vertici dobbiamo usarli per inserire un elemento in una mappa (rappresenta il grafico di visita). Ciò vuol dire che la nostra classe “EdgeTraversedGraphListener”, o il nostro chimante(cioè il metodo “fermateRaggiungibili” nel Model), dovrà contenere una mappa nella quale andrò a mettere, ogni volta che trovo un nuovo arco, le informazioni sui vertici raggiunto e il vertice sorgente. Decidiamo di costruire questa mappa nel Model e di usarla come parametro al costruttore della classe “EdgeTraversedGraphListener”.

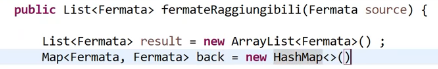
 Nel Model nel metodo in figura.

Immagine che contiene screenshot

Descrizione generata automaticamente In “EdgeTraversedGraphListener”

L’esercizio verrà concluso nella lezione successiva.   
FINE