TECNICHE DI PROGRAMMAZIONE

30/04/19

Costruzione di grafi. Esercizio “Metro Paris”.

In questa lezione proveremo a costruire un grafo a partire dai dati contenuti in un database. Prenderemo come spunto per la costruzione del grafo la mappa della metropolitana di Parigi. Importiamo il progetto da GitHub. Successivamente su Heidi eseguiamo il file .sql presente nella cartella db del nostro file importato.

In questo database abbiamo 3 tabelle: linea, connessione e fermata. In particolare, la tabella “connessione” rappresenta un legame molti e molti con le altre 2 tabelle. Concettualmente è formata da tutti i segmenti che collegano 2 fermate successive. Come attributi, quindi, avrà l’id delle due fermate collegate e l’id della linea che collega queste due fermate.

Se ordiniamo questa tabella per l’id della linea possiamo vedere il percorso che fa ogni linea.

Il nostro obiettivo è quello di creare un grafo che permetta la ricerca dei percorsi all’interno della Metropolitana di Parigi. Questo grafo dovrà avere le seguenti caratteristiche:

* Deve essere semplice orientato e non pesato.
* Ciascuna fermata dovrà corrispondere ad un vertice.
* Due vertici saranno collegati da un arco se e solo se esiste almeno una Connessione tra le due fermate.

Nel fare un grafo di questo genere stiamo buttando le informazioni che abbiamo sulle linee e sul cambio di linea.

Ci perdiamo cioè l’informazione che due o più fermate siano sulla stessa linea o su più linee differenti.

Prima di creare il nostro programma analizziamo il progetto.

Per quanto riguarda le librerie avremo:

1. JgraphT per gestire i grafi
2. Mysql Connector per leggere il database.
3. HikariCP come Connection Pooling
4. Slf4j che è la libreria di Logging portata dietro da Hikari.
5. SimplelatIng la quale è una libreria che contiene l’oggetto “LatLng” che rappresenta al suo interno già le coordinate di latitudine e longitudine e ha un paio di metodi che ci permettono di calcolare la distanza fra due punti.

All’interno del package “Model” ho già costruito le classi base per rappresentare gli oggetti. La classe “Fermata” modella la classe “Fermata” del database e quindi gestisce gli attributi idFermata, nome e coordinate. Inoltre, ha i Getter, Setter, HashCode ed equals (sull’ idFermata).

La stessa cosa vale per la classe “Linea”.

Un po’ più complicata è la tabella connessione. Al suo interno, infatti, non contiene informazioni ma solamente un suo id e le chiavi esterne delle altre 2 tabelle (idLinea, idFermataPart, idFermataArrivo).

Le “chiavi esterne” sono quindi riferimenti agli oggetti Linea e Fermata creati prima.

Il nostro obiettivo è quindi quello di costruire un grafo.

Come prima cosa creiamo l’oggetto grafo e creiamo un vertice per ogni fermata.

Dobbiamo quindi interrogare il database per avere l’elenco delle fermate. Queste fermate mi serviranno successivamente per popolare i vertici del grafo. Questo lavoro va fatto dentro al model. Creiamo quindi una classe “model” dentro al package “Model”. Essa gestirà le informazioni sul grafo. Quindi, internamente gestirà il grafo ed esternamente esporrà dei metodi per costruire, analizzare e ricercare informazioni all’interno del grafo.

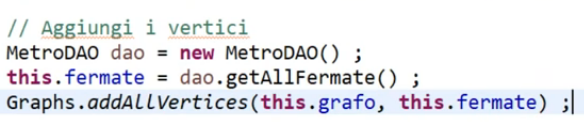
Creiamo quindi l’istanza del grafico che ha come tipo del vertice le diverse Fermate mentre come tipo di arco (grafo orientato e non pesato) DefaultEdge.

Aggiungiamo il metodo “creaGrafo” che creerà un oggetto grafo e lo riempirà con le informazioni che provengono dal database. Il nostro grafo sarà un “SimpleDirectedGraph”, con arco “DefaulEdge.class”.



Ora come ora abbiamo un grafo vuoto; dobbiamo riempirlo con le fermate. Quest’ultime sono contenute nel database, quindi dobbiamo interrogarlo. Per fortuna noi abbiamo già una classe MetroDAO che contiene il metodo “getAllFermate” che, interrogando il database, crea un oggetto fermata per ogni elemento della tabella.

Quindi creo un oggetto MetroDAO e ottengo le fermate chiedendole al DAO. Quest’ultime verranno memorizzate in una lista privata di “Fermata” nel model. Come ultima cosa aggiungo le fermate al grafo.

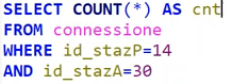
 

Ora dobbiamo pensare ad aggiungere gli archi. Vedremo 3 modi per fare questo.

Il primo modo è fare proprio quello che ci dice il testo, cioè creare un arco se e solo se esiste una connessione tra le due fermate. Per fare questo usiamo due cicli in cui prendiamo le fermate di partenza da tutte le possibili fermate (this.grafo.vertexSet) e ci chiediamo, andando a navigare di nuovo fra tutte le fermate del grafo, se queste possono essere possibili stazioni di arrivo. Dentro a questi due cicli ci chiediamo se esiste una connessione tra la stazione di partenza e quella di arrivo. Se si, aggiungo l’arco, altrimenti no. Il codice sarà il seguente:



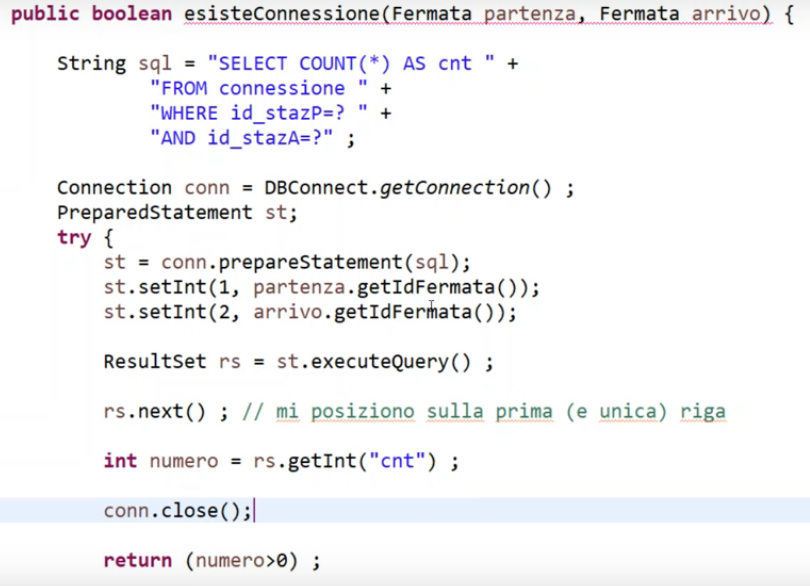
Naturalmente, il metodo “esisteConnessione” dobbiamo andarlo ancora a gestire. Esso sarà un metodo che dovrà interrogare il database per sapere se è presente la connessione fra 2 fermate. La query che utilizzeremo in questo metodo la andiamo a costruire su Heidi e sarà la seguente:



Questa query mi restituirà sempre una tabella formata da una riga e una colonna. A noi interessa sapere se il valore presente all’interno di quella casella è 0 (non è presente la connessione) o un numero >0 (è presente almeno una connessione tra le 2 fermate).

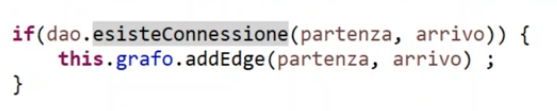
Una volta trovata la query, andiamo a gestire il metodo nella classe MetroDAO.

Esso sarà sicuramente un metodo “public boolean” che ci restituisce valore true se esiste la connessione e valore false se non esiste. Il codice sarà il seguente:

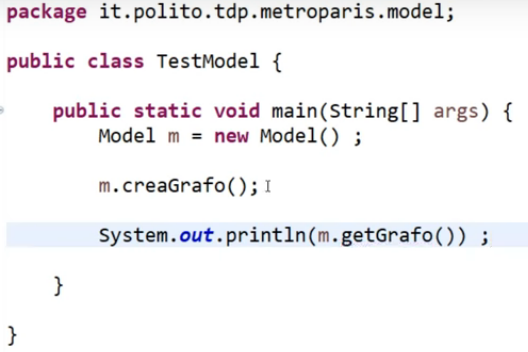


Copiamo la query da Heidi e la salviamo su una stringa ricordandoci di sostituire i punti di domanda ai valori casuali messi in precedenza. Stabiliamo successivamente la connessione con il Connection Pooling, creiamo lo statement e gestiamo i 2 punti interrogativi.

Salviamo il risultato della casella ottenuto dalla query in una variabile chiamata “numero”. Il metodo ritornerà valore true se il numero è maggiore di 0. Ricordarsi sempre di chiudere la connessione e di mettere “return false” dopo il catch (non in figura).  
Ora se torniamo nel Modello, notiamo quindi che “esisteConnessione” è un metodo del DAO e quindi bisogna scrivere:



Per vedere se il tutto funziona, possiamo aggiungere una classe di Test nel Model (con Main) che ha come obiettivo quello di creare un nuovo Modello e di creare un grafo.

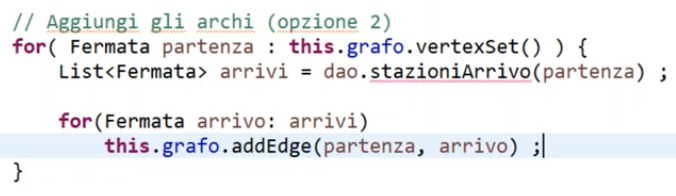


Notiamo anche che dobbiamo aggiungere nella classe “Model” i getter di “grafo” e “fermate”.

Se facciamo partire questo Test, possiamo vedere che il programma ci mette molto a restituire il risultato in quanto deve svolgere tantissime query (n° fermate al quadrato). Bisogna svolgere circa 400 mila query. Cosa possiamo fare per ridurre il tempo?

Possiamo agire sulla query di Heidi. Modificandola, infatti, possiamo farci dare, per ogni id di una stazione di partenza, tutti gli id delle stazioni di arrivo collegati a quella di partenza. Così facendo, a parità di tempo possiamo conoscere per ogni stazione (partenza), tutti gli archi uscenti da quel vertice.

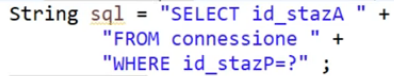
Aggiungiamo quindi un’opzione 2 per l’aggiunta di un arco.



Faccio un livello di iterazione in meno in Java. Per ogni fermata di partenza (iteriamo per tutti i vertici del grafo), creiamo una lista contenente tutte le sue stazioni di arrivo. Successivamente, itero su questa lista contenente tutti gli arrivi e, per ogni arrivo, creo un arco. Aggiungo in un colpo solo tutti gli archi uscenti da una determinata fermata, mentre con l’opzione 1 aggiungevo un arco alla volta.

Dobbiamo ancora implementare il metodo del DAO “stazioniArrivo” che ci permette di conoscere tutte le stazioni di arrivo di una determinata stazione di partenza.

Aggiungiamo quindi questo metodo alla classe MetroDAO. Esso ci restituirà una lista di “Fermata” contenente tutte le stazioni di arrivo. Anche per questo metodo dobbiamo creare la query corrispondente che sarà la seguente:



Anche in questo metodo, come in quello precedente, dobbiamo gestire la connessione, i punti di domanda, lo statement ecc…

A differenza del metodo di prima, in questo metodo dobbiamo creare una nuova arrayList di “Fermata” che conterrà tutte le stazioni di arrivo e che verrà restituita dal return del metodo. Scandendo il ResultSet, aggiungo a questa List tutti gli oggetti “Fermata” presi dal Database.

Il problema è che l’oggetto “Fermata” è caratterizzato da id, nome e coordinate mentre la nostra query ci restituisce solamente un id (un intero). Come posso fare? Abbiamo diversi modi:

1. Non restituisco una lista di “Fermata” ma una lista di interi. In questo caso il DAO “se ne frega” e il Model dovrà occuparsi, in base al numero ottenuto, di creare l’oggetto Fermata.
2. Ci facciamo restituire dal DAO l’oggetto Fermata (join nella query con la tabella Fermata per ottenere tutte le informazioni relative a quest’ultima).

Fare la join nella query sarebbe un po’ stupido, in quanto noi abbiamo già un metodo (“getAllFermate”) che ci crea un oggetto Fermata. Noi potremmo scandire la lista ottenuta da questo metodo e andare a vedere quale fermata ha l’id uguale a quello ottenuto nella query. ES per capire: nella nostro query del metodo “stazioniArrivo” ottengo che per la fermata numero 12 c’è come stazione di arrivo la fermata numero 36; vado a scorrere la Lista del metodo “getAllFermate” e vado a vedere per la fermata 36 qual è il suo nome e le sue coordinate.

Questo procedimento è anche un po’ stupido in quanto ogni volta dovremmo scorrere la Lista di “getAllFermate”. Sarebbe molto più comodo avere una **Mappa** che, dato l’id, mi dia l’oggetto Fermata.

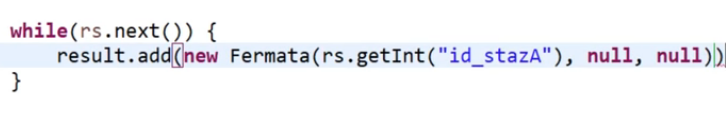
Dato che il metodo “getAllFermate” mi restituisce una lista di “Fermata”, non mi costa niente costruire anche una Mappa che, data la chiave primaria della tabella, mappi l’oggetto completo. Questo è lo stesso oggetto che metto nella Lista, però a volte mi può servire la Lista per scandirla, a volte mi può servire la Mappa per trovare un elemento al volo quando conosco l’id.

Questa tecnica che usa le Mappe per velocizzare queste operazioni prende il nome di “**Identity Map**“e lo vedremo un’altra volta. Essa ci permette di evitare di creare un oggetto più di una volta.

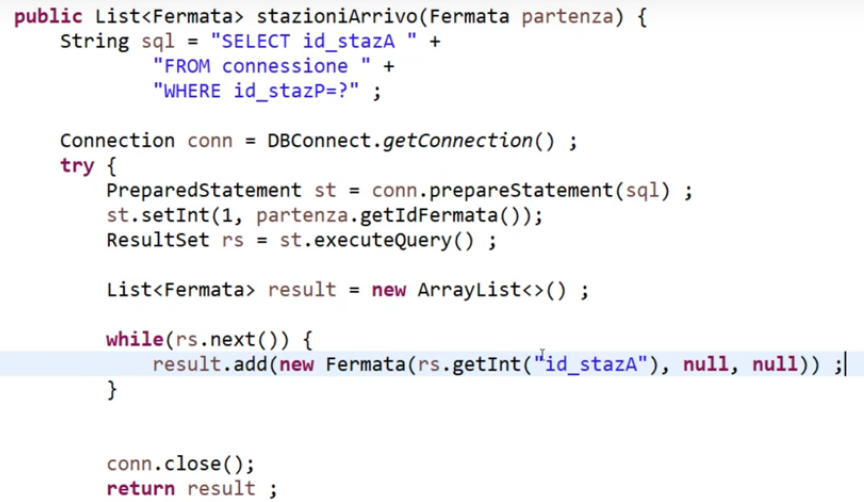
Dato che questa tecnica la vedremo un’altra volta e gli altri metodi non ci vanno bene, l’unica cosa che ci è rimasta da fare è barare. Creiamo un “finto” oggetto Fermata. Il nostro oggetto Fermata infatti ci serve solo per dire al grafo tra quali vertici creare l’arco. Il grafo non deve aggiungere il vertice partenza e il vertice arrivo che gli passo io; deve aggiungere un arco tra due vertici che sono già presenti nel grafo.

Noi dobbiamo solo dare un oggetto che sia equals al vertice di partenza e al vertice di arrivo per permettere di identificarli. L’equals dell’oggetto Fermata lavora sull’id, quindi l’unico campo veramente necessario in questo caso è l’id!!

Ricapitolando, l’oggetto Fermata vorrebbe l’id, il nome e le coordinate. Noi gli diamo l’id\_fermata, che sarà il rs.gxetInt della colonna “stazA”, mentre gli altri 2 parametri non glieli do.



Il codice finale di questo metodo sarà quindi il seguente (manca il catch alla fine):



Runnando il TestModel notiamo che nel risultato gli archi hanno vertice di arrivo “null”. Nella lezione successiva, il professore farà sapere qual era l’errore.

Il terzo modo per creare gli archi è quello di modificare la query nel seguente modo:



Così facendo, avrei già una coppia di stazioni tra le quali esiste una connessione. Questo però è un caso speciale, infatti non capita quasi mai di avere già nella tabella del database due colonne che ti danno il vertice di partenza e il vertice di arrivo.

Quindi è possibile a volte costruire in un’unica query un risultato in cui le righe siano esattamente gli archi che mi servono.

Il codice verrà analizzato nella prossima lezione.

La prima opzione fa lavorare molte volte il Database con una richiesta molto semplice, infatti devo solo decidere se i due vertici presi in considerazione sono collegati o no (oppure decidere il peso dell’arco che li collega nel caso di grafo pesato). Ha il difetto che per grafi molto grandi il numero di interrogazioni da fare cresce quadraticamente.

La terza opzione ci fa lavorare il più possibile con il Database per avere un risultato finale che sia il più vicino possibile all’informazione che mi serve per costruire il grafo. Avrò delle query complicate ma dovrebbero essere più veloci rispetto a fare tante query piccole.

La seconda opzione invece è una via di mezzo. Non faccio un arco per volta, non faccio tutto in un botto, ma faccio un vertice per volta e per ognuno di essi vado a trovare tutti gli archi.

La scelta di un’opzione rispetto ad un’altra dipende:

* Dalla dimensione del grafo (numero di archi/vertici). Se è piccolo vanno tutte bene quindi prendiamo la più semplice.
* Dalla nostra dimestichezza con Heidi (ci sono query più o meno complicate).

FINE.