[PR2] Progetto 2016/2017 — Corso A

Marco Pampaloni

28/11/2016

Abstract

La seguente relazione contiene una descrizione del lavoro che è stato fatto e le motivazioni che hanno causato le principali scelte di progetto. Il testo è stato redatto al termine della fase di sviluppo ed è quindi esposto il risultato di tale periodo lavorativo nel suo complesso. Un workflow più dettagliato è comunque disponibile e analizzabile nella repository GitHub che ho creato appositamente per ospitare i sorgenti del progetto: https://github.com/crybot/urban-guacamole.
(La copertina non conta per il limite massimo delle pagine)

1 Specifica di progetto

Il testo del progetto richiedeva la descrizione e la specifica dettagliata di un'interfaccia Java che definisse il tipo di dato astratto generico Graph, oltre l'implementazione di quest'ultimo. In matematica un grafo è una coppia $\langle V, E \rangle$ dove V è un insieme di oggetti chiamati nodi, ed E è un insieme di oggetti chiamati archi che descrivono le relazioni di connessione tra oggetti di V.

Node ed Edge: Sono stati quindi definiti i tipi Edge e Node, entrambi generici. Il tipo di dato Node è stato definito come astratto in modo da separare i dettagli implementativi riguardanti la scelta della collezione da utilizzare per mantenere le informazioni relative ai nodi ad esso adiacenti, dalla specifica delle funzionalità.
Un oggetto Node, infatti, contiene al suo interno un valore che ne identifica l'unicità (un etichetta, o un campo informativo se vogliamo vedere un nodo come un "contenitore") e un insieme di valori del medesimo tipo che elenca i nodi con cui è direttamente connesso. Tali connessioni sono da intendersi strettamente direzionali, è compito dell'utilizzatore decidere se utilizzare connessioni di tipo dirette o non dirette (in tal caso dovrà essere sua premura inserire anche le connessioni opposte qualora sia necessario). Il tipo Node possiede i metodi di base per operare su di esso:

- *getLabel*(): restituisce al chiamante il contenuto informativo del nodo;
- *addConnection*(*E v*): aggiunge una connessione tra il nodo this e il nodo v;
- *removeConnection*(*E v*): similmente, rimuove una connessione;
- *getAdjacency*(): restituisce una collezione contenente i nodi adiacenti.

Il tipo Node è esteso dal tipo di dato HashNode, che reimplementa alcuni dei suoi metodi ed utilizza un HashSet (collezione inclusa nel JCF) per memorizzare i nodi adiacenti. È la soluzione più efficiente per eliminare la possibilità di avere elementi duplicati all'interno della collezione.

Graph: Il tipo di dato astratto Graph è stato definito come un'interfaccia e non presenta quindi variabili di istanza, la definizione delle quali è delegata all'implementatore. Sono presenti i metodi descritti all'interno della specifica di progetto (che utilizzano i tipi di dato Node ed Edge), oltre ai metodi *size*(), *containsNode*(*E v*) e *getAdjacency*(*E v*) che sono auto esplicativi. È stato definito il tipo di dato HashGraph (con parametro di tipo E) che estende Graph e che implementa l'interfaccia Iterable. Tale scelta è stata dettata dalla necessità di esporre il contenuto informativo del grafo (cioè l'insieme dei valori associati ai nodi) all'esterno della classe, ma senza esporre riferimenti ad oggetti di tipo

Node.
Internamente il grafo è rappresentato da una variabile di istanza nodeMap di tipo HashMap (JCF) che associa oggetti di tipo E a oggetti di tipo $Node\langle E\rangle$. Il methodo $iterator()$ è implementato semplicemente restituendo $nodeMap().keySet().iterator()$.

$$AF: \quad [\langle k_1, f(k_1)\rangle, \langle k_2, f(k_2)\rangle, ..., \langle k_n, f(k_n)\rangle]$$
$$where \quad f(k) : nodeMap.keySet() \longrightarrow nodeMap.values()$$

SocialNetwork: Il tipo di dato SocialNetwork implementa, come da richiesta, la rete dei contatti di un social network (per esempio Facebook). Il grafo implementato è di tipo non diretto e quindi per ogni arco aggiunto in una direzione viene aggiunto l'equivalente nella direzione opposta. La struttura dati interna è implementata attraverso l'utilizzo di un $HashGraph\langle String\rangle$, dove i nodi sono identificati dai nomi degli utenti della rete. Sono implementati i metodi necessari a creare la rete da zero:
$addUser(String)$, $addFriendship(String, String)$. Il metodo $randomUser()$ restituisce invece il nome di un utente presente nella rete in modo casuale (tale funzione viene utilizzata nella classe Main per generare casualmente un grafo data una lista contenente circa 9000 nomi italiani).
È stato implementato un metodo per calcolare la shortest path tra due utenti utilizzando l'algoritmo *BFS*, che nel caso di grafi non pesati è più efficiente dell'algoritmo di Dijkstra. È implementato infine il metodo per calcolare il diametro della rete, che sfrutta il metodo $shortestPath()$ e le sue informazioni accumulate per effettuare della memoizzazione. Con la rete di test, che conta circa 9000 nodi, è risultato impraticabile calcolare il diametro con l'algoritmo proposto (che risulta essere ottimale). È stimabile un upper bound per la complessità computazionale di $O(|V|^3)$ dove V è l'insieme dei nodi appartenenti al grafo.