Relazione `Graph` primo progetto (PR2B)

Matteo Giorgi

Novembre 2016

Il progetto `Graph` definisce ed implementa un grafo di oggetti generici omogenei non orientato senza cappi.

1 Interfaccia `Graph`⟨`E`⟩

`Graph`⟨`E`⟩ definisce il tipo di dato astratto grafo come una collezione modificabile di oggetti generici omogenei di tipo `E` priva di ripetizioni, univocamente definito dalla coppia ⟨insieme vertici, insieme archi⟩, dove ogni arco é identificato dalla coppia dei vertici da lui collegati.

Il tipo astratto prevede 15 metodi per eseguire operazioni di

- manipolazione: `addVertex`, `removeVertex`, `addEdge`, `removeEdge`;
- ispezione&info: `existsVertex`, `existsEdge`, `numVertex`, `numEdge`, `degreeVertex`, `distanceInBetween`, `graphDiameter`;
- altri: `listVertex`, `adjacentVertex`, `toArray` e `equals`.

2 Classe `GraphMap`⟨`E`⟩

`GraphMap`⟨`E`⟩ fornisce una implementazione dell'interfaccia `Graph`⟨`E`⟩ usando una mappa chiave-valore `TreeMap`⟨`E, TreeSet`⟨`E`⟩⟩, dove ad ogni vertice del grafo é associato l'insieme dei vertici a lui adiacenti. Con questa scelta la *nested-class* `Map.Entry`⟨`K,V`⟩ e l'insieme ordinato `TreeSet`⟨`E`⟩, permettono efficienti ricerche di vertici ed archi all'interno del grafo.

`GraphMap`⟨`E`⟩ impone che il tipo generico `E` estenda `Comparable`⟨`E`⟩ in modo da confrontare i vertici secondo il loro naturale ordinamento e raggrupparli in insiemi ordinati per una migliore visualizzazione; inoltre implementa `Iterable`⟨`E`⟩ per fornire al cliente un iteratore sull'insieme ordinato dei vertici.

2.1 Invariante di rappresentazione

Per controllare che la struttura concreta del grafo non infranga l'invariante di rappresentazione, é presente il metodo privato `repOk` che testa la validitá di ciascuna frase logica dell'`IR` e lancia eventualmente una `RepInvariantException`.
`repOk` viene invocato all'interno di ogni "metodo scrittore", prima della terminazione dello stesso: questa scelta é frutto della natura didattica del progetto, cosí da mostrare la correttezza semantica dei metodi forniti, costruttori compresi.

2.2 Costruttori

`GraphMap`⟨`E`⟩ é fornito di due costruttori:

- `GraphMap()` che istanzia un grafo vuoto
- `GraphMap(Set<E> vertexes, List<TreeSet<E>> adjoints)` che invoca il precedente costruttore, controlla la validitá degli argomenti passatigli ed associa, nell'ordine ricevuto, ogni elemento dell'insieme dei vertici `vertexes` con il corrispettivo elemento della lista degli insiemi degli adiacenti `adjoints`, cosí da generare una mappa vertice/lista-adiacenti completa.

2.3 Nested-class `Path`

In alcuni metodi quali `distanceInBetween`, `pathInBetween`, `graphDiameter` é necessario esplorare il grafo in ampiezza: `GraphMap`⟨`E`⟩ usa una *nested-class* privata per l'implementazione dell'algorimo *Breadth-First-Search*.

```
private class Path{
    Map<E, Integer> distMap = new TreeMap<E, Integer>();
    Map<E, E> prevMap = new TreeMap<E, E>();

    Path(TreeMap<E, TreeSet<E>> st, Comparable ... vrt)
    {
        @SuppressWarnings("unchecked")
        E[] _vrt = (E[]) vrt;

        Queue<E> queue = new ArrayDeque<E>();
        E head = null;
        queue.add(_vrt[0]);
        distMap.put(_vrt[0], 0);
        if( _vrt.length>1 && _vrt[0].compareTo(_vrt[1])==0 )
            return;
        while( (head=queue.poll())!=null )
            for(E neighbour: st.get(head))
                if(!distMap.containsKey(neighbour)){
                    distMap.put(neighbour, distMap.get(head)+1);
                    prevMap.put(neighbour, head);
                    if( _vrt.length>1 && _vrt[1].compareTo(neighbour)==0 )
                        return;
                    queue.add(neighbour);
                }
    }
}
```

La classe é dotata solamente di una mappa delle distanze `distMap`, una dei percorsi `prevMap` (che assieme simulano un *Minimun-Spanning-Tree*) ed un unico costruttore che, quando invocato, esegue i passi del *BFS* che esplora il grafo e riempie le mappe.

Il costruttore puó ricevere come argomento un solo *variadic*, esplorando l'intero grafo e generando l'*MST* completo con radice l'unico vertice fornito; se invece dovesse ricevere due *variadic*, interromperebbe la costruzione delle mappe creando un *MST* parziale, sufficente peró a dare informazioni sui due vertici forniti.

2.4 Metodi extra

`GraphMap⟨E⟩` é fornita, oltre che dei metodi ereditati da `Graph⟨E⟩` anche di altri che facilitano la manipolazione e l'esplorazione delle strutture del grafo. Se ne citano alcuni: `addSetVertex`, `addAttachedVertex`, `removeSetVertex`, `addSetEdge`, `removeSetEdge`, `isolateVertex`, `pathInBetween`, `commonNeighbours`, `toString`.

2.5 Esposizione della rappresentazione

Il problema di esporre la rappresentazione al cliente ricorre piú volte nel codice, causa l'uso del tipo generico `E`: non é possibile fare una copia sicura degli oggetti generici perché non se ne conosce la strutura interna.

Tuttavia in alcuni casi, come nei metodi `listVertex` e `adjacentVertex`, si é ovviato in parte al problema usando dei `NavigableSet` inmodificabili.

2.6 Eccezioni

La politica seguita per la gestione delle eccezioni é difensiva: vengono lanciate eccezioni in ogni situazione in cui gli argomenti dei metodi non siano consoni alla struttura.

In particolare, per evitare fastidiose ripetizioni di codice, sono presenti: `existsEdge`, metodo pubblico che controlla l'esistenza di un vertice e `checkVertex`, metodo privato usato internamente per testare un vertice con la precedente `existsEdge`.

Per gestire problemi riguardanti la mancata correttezza dell'`IR` é stata creata una nuova classe di eccezioni `RepInvariantException`, sottoclasse di `RuntimeException`.

Tutte le eccezioni usate nella classe sono sottoclassi di `RuntimeException`.

3 Classe `Graphs`

`Graphs` é una classe di metodi statici utili per una pratica gestione logica e grafica di grafi: la classe intende rappresentere per i grafi quello che `Arrays` e `Collections` rappresentano per array

e collezioni.

Di particolare importanza sono i metodi `readAndFill` che legge un file di configuazione e riempie il grafo passatogli, `asGraph` che analogamente legge un file di configuazione e restituisce un nuovo grafo, `drawGraph` che prima scrive un file html, poi disegna il grafo usando una libreria javascript, `printInfo` che stampa le informazioni sul grafo passatogli.

3.1 File di configurazione `friends.txt`

L'idea é quella di mettere a disposizione un file di configurazione facilmente modificabile in fase di test, che il metodo privato `fillSets` possa leggere per creare l'insieme dei vertici e la lista degli adiacenti da passare poi a `readAndFill` o `asGraph` che li useranno per riempire o creare un grafo.

Il file di configurazione é un file di testo che contiene, per ogni riga, il nome di un vertice (prima stringa dalla riga) e l'insieme degli adiacenti a quel vertice (rimanenti stringhe nella riga).

Si noti che per ottenere un vertice senza archi é sufficiente scrivere una riga con il solo nome del vertice. Possono essere usati spazi ovunque si voglia anche righe completmente vuote.

4 Classe `SocialNetTest`

`SocialNetTest` é la simulazione di una rete sociale di contatti realizzata usando la classe `GraphMap`⟨`E`⟩.

Il `main` della classe istanzia due grafi e fornisce una interfaccia testuale interattiva per il test di tutti i metodi della classe `GraphMap`⟨`E`⟩.

Sará quindi sufficiente compilare la classe con il comando `javac SocialNetTest.java` ed eseguire con `java SocialNetTest`.