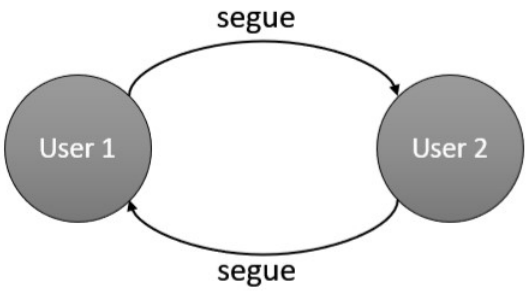
ANDREA TOSTI - CORSO B - Mat n. 518111

Relazione primo progetto, novembre 2016

Come da consegna, il tipo di dati astratto Graph<E> è una collezione di oggetti generici di tipo E.

Ho scelto un **grafo orientato** perchè poi nell'implementazione ho bisogno di gestire le relazioni tra nodi, con coppie ordinate, tali per cui (a, b) sia diversa da (b, a).

In una rete sociale come Twitter, un utente A può seguire un utente B e non è detto che l'utente B segua A. Infatti possono venire a crearsi quattro situazioni diverse:



A segue B, B non segue A

A non segue B, B segue A

A segue B, B segue A

A non segue B, B non segue A

Le prime due situazioni ci fanno capire che un grafo non orientato non è adatto allo scopo.

Nell'implementazione **TwitterGraph.java**, scelgo appunto **Twitter** come rete sociale.

Uso due strutture di tipo Map, una per associare l'identificativo utente al suo nodo,

un'altra per associare l'identificativo utente alla sua lista di persone seguite.

In particolare l'identificativo è univoco e potrebbe essere lo stesso identificativo che troviamo nell'url di una pagina Twitter. La struttura intrinseca di Map ci permette di evitare duplicati di chiavi (che sono le stringhe identificative). Ho scelto di implementare Map con TreeMap così da mantenere ordinate le chiavi degli id e sfruttare l'ordinamento lessicale delle stringhe.

Il file **Node.java** contiene l'implementazione del tipo Node con poche variabili di istanza, quali il Nickname (che è possibile cambiare in qualsiasi momento), il contatore dei followers e delle persone seguite. Forse sarebbe stato piu' efficiente richiamare il metodo size() sulle Map, ma volevo provare ad agire sulle variabili d'istanza del Nodo in fase di rimozione/aggiunta di archi/nodi.

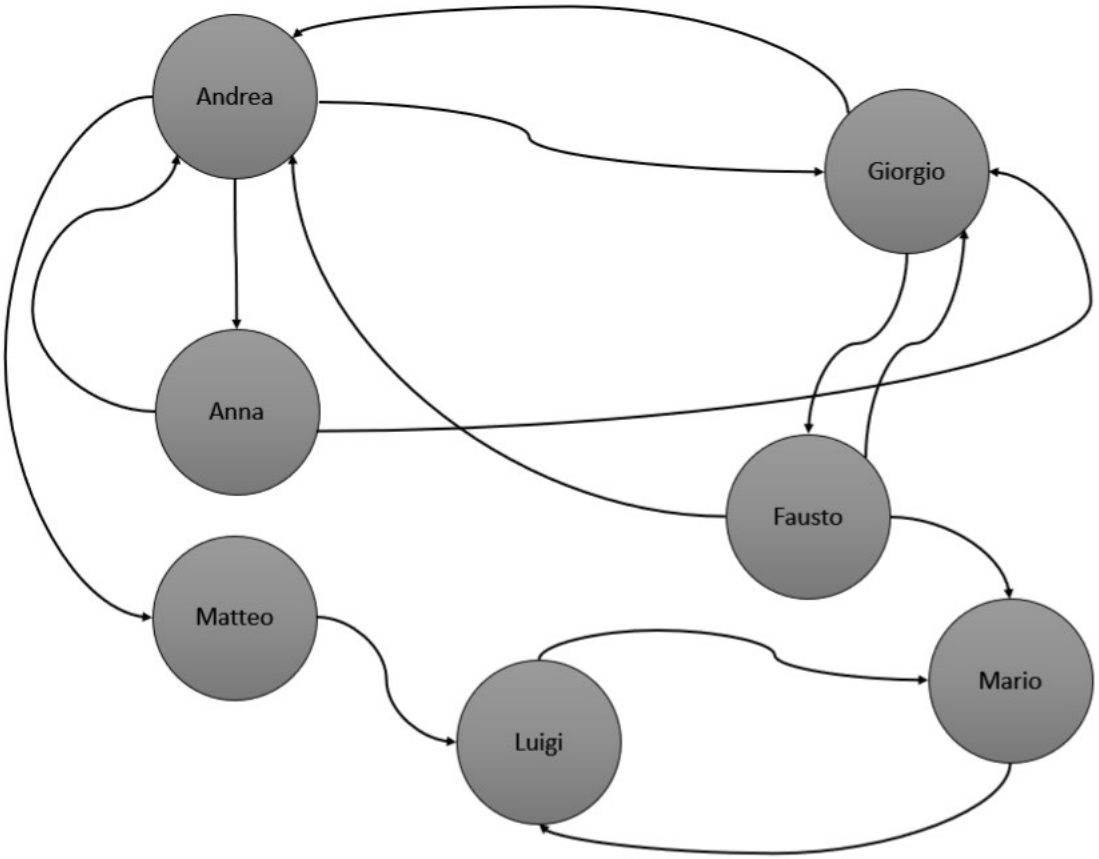
Dato che il grafo non è pesato, nell'implementazione del metodo **shortestPath** ho usato una coda che dà priorità ai livelli del grafo: mappo ciascun livello con la coda di stringhe per quel livello e vado avanti solo dopo averlo esplorato tutto.

La coda è implementata con una **ArrayDeque** che usa implicitamente una politica LIFO ed è preferibile allo Stack.

Nella **pagina seguente** mostro due esempi di grafo e ne descrivo il comportamento.

Basta quindi compilare ed eseguire **TestCaseOne.java** e **TestCaseTwo.java** per vederne gli effetti.

TestCaseOne.java : vengono simulate l'aggiunta di tutti gli archi e tutti I nodi di utenti, il metodo shortestPath, il metodo longestPath e la repOk.



TestCaseTwo.java : Oltre a ciò che è stato fatto nel TestCaseOne, vengono aggiunti altri nodi, rimossi diversi nodi, aggiunti archi e rimossi archi.