TECNICHE DI PROGRAMMAZIONE

29/04/19

Introduzione ai grafi. Libreria JGraphT.

Un grafo è l’insieme di punti (nodi o vertici) e di arcihi che collegano tra di loro questi vertici. Bisogna fare attenzione però in quanto il termine “Graph” in inglese può rappresentare 3 cose differenti:

1. Il grafo in senso matematico.
2. Il grafico di una funzione (plot).
3. Il grafico di Excell tipo istogramma (Businnes Chart).

La definizione di grafo non è univoca, infatti esistono molte sottospecie di grafi. Quando vedremo le librerie, vedremo che ne supporta 16 differenti.

Le diverse sottospecie di grafi si differenziano tra di loro per le regole con cui gli archi si collegano ai diversi vertici del grafo stesso.

Abbiamo un grafo “semplice” quando, data una coppia di vertici, può esistere un arco che li collega o può non esistere. Inoltre, i vertici collegati ad un arco devono essere diversi tra di loro.

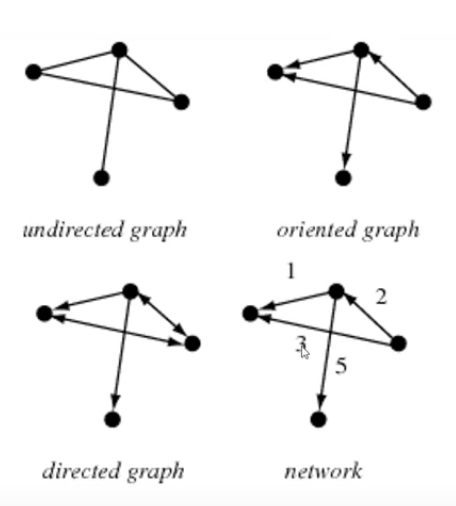
Eliminando la regola per cui per ogni coppia di vertici è presente solo un arco, otteniamo i “multigrafi”.

Eliminando la regola per cui il vertice di partenza e quello di arrivo siano distinti, otteniamo gli “pseudografi” in cui sono ammessi dei “loop” (archi che partono e arrivano allo stesso nodo”.

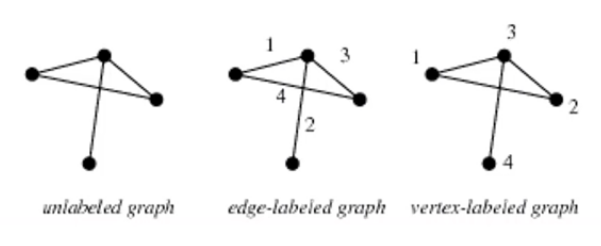
Per ciascuna delle possibilità precedenti, l’arco può essere concepito come entità ordinata (orientata) o non ordinata (non orientata). Nel primo caso, l’arco ha un punto di partenza e un punto di arrivo, mentre nel secondo caso non abbiamo questa specificazione e quindi, dire che un arco parte dal vertice A e arriva al vertice B, è la stessa cosa di dire che l’arco parte dal vertice B e arriva al vertice A.

Nel caso in cui sia presente per una coppia di vertici un arco che ha una doppia freccia, il grafo si dice direzionato.

Se invece ogni arco presenta un “peso”, il grafo si chiamerà “network”.

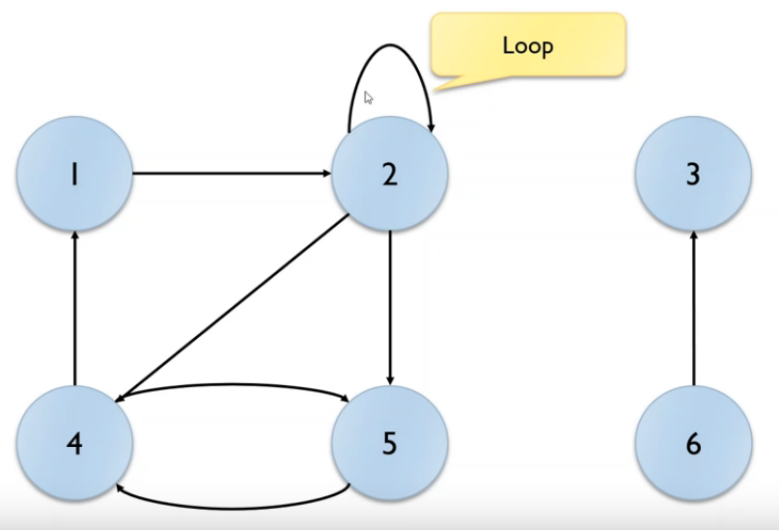
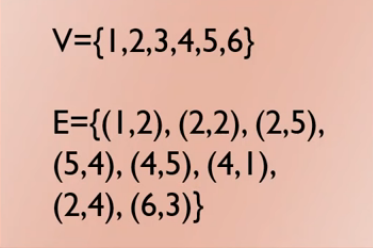


Ogni arco inoltre può essere etichettato. Nel caso in cui non sia etichettato, il grafo si chiamerà “unlabeled graph”. Questa etichettatura può avvenire o sugli archi stessi oppure sui vertici.



Quindi, riassumendo tutto quello detto fino ad adesso in termini formali, un grafo è rappresentato da due insiemi: un insieme di vertici V e un insieme di archi E. Un arco è un sottoinsieme del prodotto cartesiano del vertice per il vertice. 

Questa però è la definizione di arco orientato, quindi per ogni arco è possibile riconoscere il suo punto di partenza (sorgente) e il suo punto di arrivo (destinazione).

Notiamo che “E” è un insieme di coppie! In un grafo non orientato, queste coppie dell’insieme “E” vengono interpretate come coppie “non ordinate”, quindi non vanno definite tra parentesi tonde da tra parentesi graffe.Possono esistere anche vertici isolati.

Ma cosa possiamo calcolare in un grafo?

In un grafo non orientato, posso calcolare il numero di archi incidenti per ogni vertice (**grado** di un vertice).

In un grafo orientato, invece, avrò sia un “grado in entrata” che un “grado in uscita”, cioè sia il numero di archi che entrano in un vertice che il numero di archi che escono. Per avere il grado “finale” di un vertice, basta sommare il grado in entrata e quello in uscita.

Un vertice che ha un grado totale uguale a zero, è un **vertice isolato**.

Quindi, dato un vertice, noi possiamo conoscere tutti i vertici a lui adiacenti, muovendoci sugli archi adiacenti. Compiendo questo procedimento molte volte posso ottenere il “**cammino**” di un grafo. Esso è una sequenza di vertici in cui “tocco” più vertici e passo da uno all’altro attraversando un arco.

Abbiamo un cammino semplice se non passiamo più di una volta da uno stesso vertice, cioè se non usiamo più di una volta uno stesso arco.

Introducendo il concetto di cammino, posso introdurre anche il concetto di **raggiungibilità**. Un vertice è raggiungibile da un altro se esiste almeno un cammino che porta dal primo nodo al secondo.

Se il punto di arrivo è uguale al punto di partenza, avremo un cammino chiuso (**ciclo**).

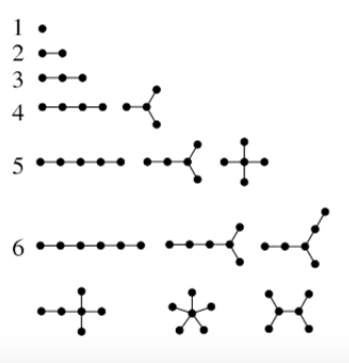
Possono esserci dei grafi che ammettono dei cicli e dei grafi che non li ammettono; anhe questa è una proprietà del grafo (**ciclico o aliclico**).

Affinchè un grafo sia ciclico basta che sia presente un solo ciclo.

Il concetto di ciclicità e di connettività sono molto collegati tra di loro. Nel caso in cui ogni vertice è raggiungibile da ogni altro vertice, il grafo è detto **connesso.**

Un qualsiasi grafo non connesso può essere scomposto e ricomposto in modo da formare tanti piccoli sottografi connessi tra di loro.

Un **albero** è un grafo connesso aciclico (non orientato). Esso quindi è uno spartiacque tra i grafi connessi e i grafi ciclici.

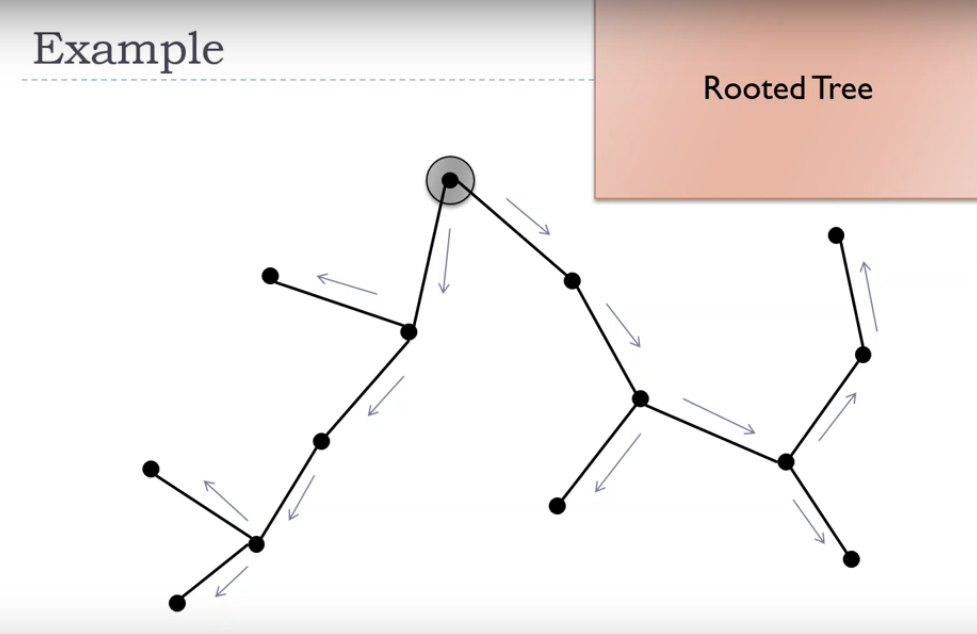


Questi sono tutti i tipi di alberi possibili fino a 6 vertici. Per ognuno di essi è impossibile aggiungere un arco senza creare un ciclo. Quindi un albero è un grafo che ha il numero massimo di archi che gli permette di non essere ciclico.

Lo stesso ragionamento può essere fatto con la connettività.

Un albero è un grafo che ha il numero minimo di archi per renderlo connesso.

Qualsiasi grafo che è connesso e che è aciclico, lo chiamiamo albero. Per ogni albero noi possiamo scegliere un nodo speciale che chiamiamo “**radice**” (a caso). Nel momento in cui scelgo una radice, creo automaticamente un orientamento naturale nei vertici e negli archi dell’albero.



Sui grafi pesati, noi attribuiamo un valore numerico (reale,positivo,non nullo), che è associato al concetto di peso (o costo) nell’usare un determinato arco. Attenzione alla differenza tra peso di un arco e all’ etichettatura dello stesso.

Ma nei nostri programmi, cosa possiamo usare questi concetti?

Nel nostro corso abbiamo scelto la libreria JGraphT che si può scaricare sotto forma di .zip nel sito apposito.

Di questo file .zip a noi interessa la “javadoc” e la “lib” che contiene i jar già compilati della libreria.

La libreria che ci interessa di più a noi (useremo sempre quella nei programmi) è la “jgrapht-core-1.3.0.jar”.

Creiamo un nuovo progetto su eclipse per vedere il funzionamento di questa libreria. Aggiungiamo a questo progetto una cartella “lib”. In essa ci copiamo il nostro file “j-grapht-core” e lo aggiungiamo al nostro “Build-Path” come sempre.

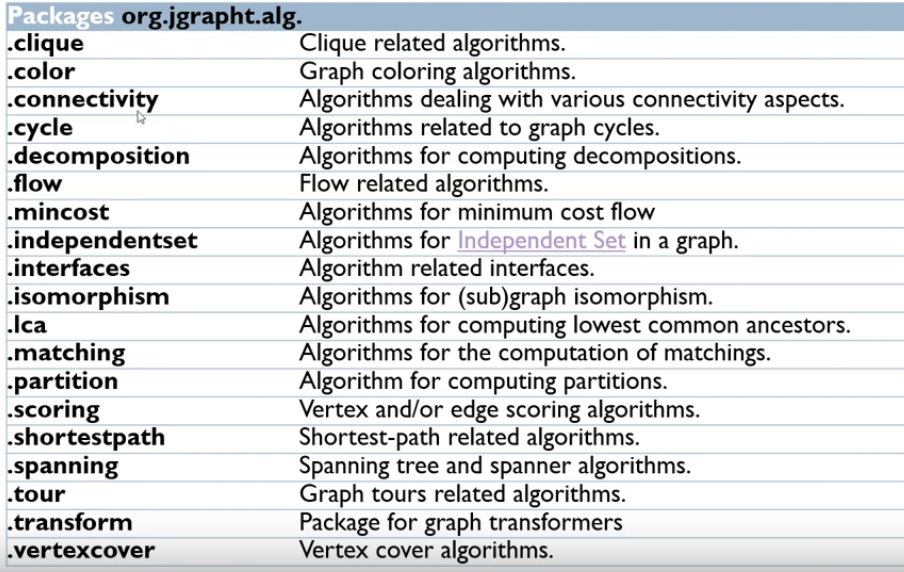
Creiamo infine una classe main “Esempio” nel package “it.polito.tdp.grafi”.

Il nostro file “core” presenta al suo interno diversi package che a sua volta conterranno delle classi o delle interfacce. Tra i diversi package è presente “org.jgrapht”, il principale, che contiene la classe principale di tutto il package che si chiama “Graph”. Da questa classe deriva tutto il resto.

È presente poi il package “org.jgrapht.graph” che contiene le classi di implementazione dei grafi (grafo orientato, non orientato, pesato, ecc…).

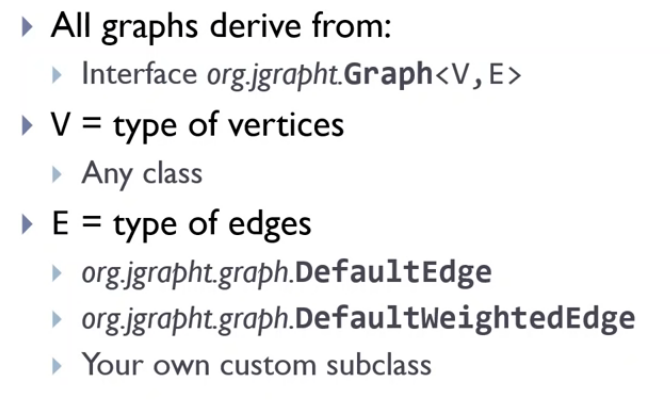
Il package “org.jgrapht.generate” ci permette di generare grafi automaticamente.

Infine c’è un altro package importante che è “org.jgrapht.alg” che contiene altri package che contengono a loro volta l’implementazione di molti algoritmi che potremmo volere utilizzare sui grafi.



Tutti i grafi devono essere istanze di una sottoclasse di Graph. Così come nelle collection, un grafo è parametrizzato. Noi abbiamo un grafo che ha due generici: “V” che è il tipo di vertice e “E” che è il tipo di arco. Quindi se io voglio mappare i collegamenti tra le varie città allora come vertice mi conviene usare l’oggetto città.

Per quanto riguarda invece il tipo dell’arco, ci interessa meno in quanto le informazioni importanti spesso sono all’interno dei vertici. La libreria ci mette già a disposizione 2 classi di arco: “DefaultEdge” e “DefaultWeightedEdge”. La prima crea un arco di default non pesato (la usiamo per i grafi non orientati), mentre la seconda ce lo crea pesato (grafi orientati). Quindi per riassumere:



Ricordarsi che “Graph” è un’interfaccia e non una classe.

Un requisito che devono avere “V” ed “E” è quello di implementare HashCode e Equals, altrimenti non funzionerebbe nulla.

Quali operazioni posso fare su un grafo?

Tutti i metodi che mi fornisce la libreria sono disponibili sul sito <https://jgrapht.org/javadoc/>.

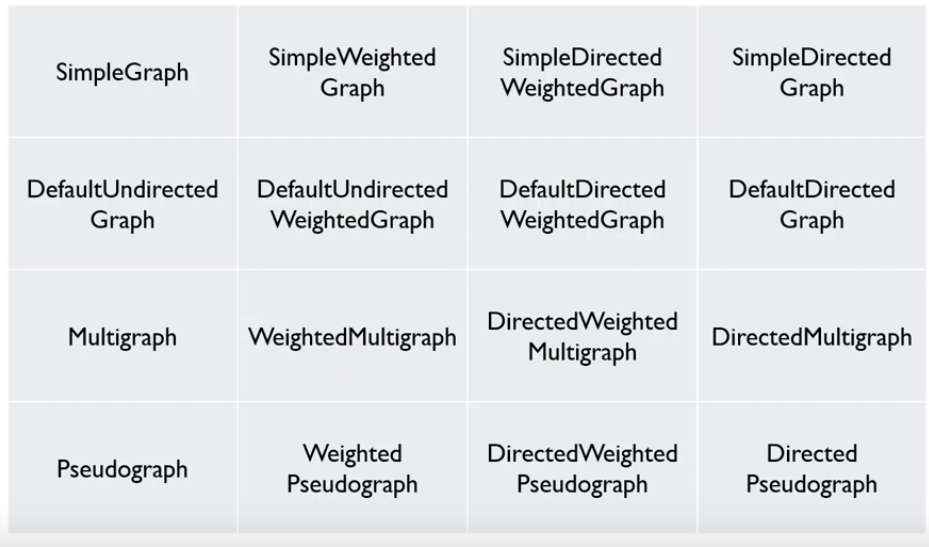
Avremo sicuramente un “addEdge” e un “addVertex” che utilizzeremo per aggiungere un nuovo arco o un nuovo vertice (passato come parametro). Una volta che li ho aggiunti, posso interrogarli. Posso chiedere al grafo quali sono gli archi che contiene (“edgeSet” che è un Set di oggetti “arco”).

Quindi se per esempio ho un grafo e voglio iterare su tutti i suoi vertici, basta iterare su “grafo.vertexSet()” che è l’insieme dei vertici stessi.

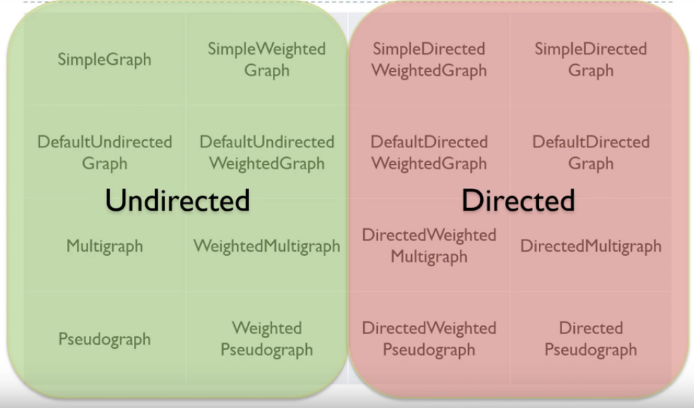
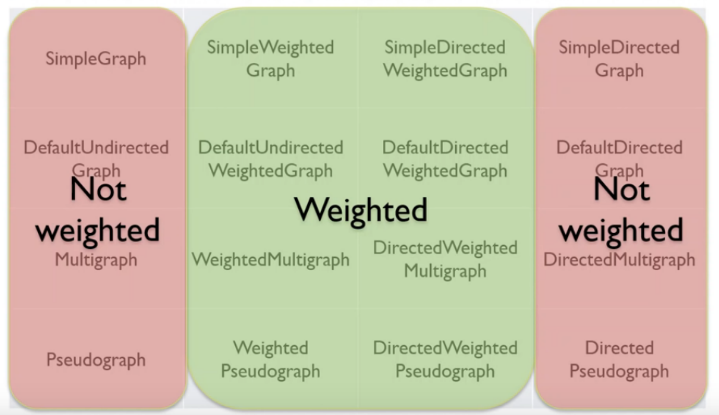
Il professore consiglia di leggere i metodi presenti sul sito per informazione personale.

Sono anche presenti metodi per impostare il peso di un arco nel grafo, per ottenere l’arco che collega due vertici, ecc…

Quindi, ricapitolando, abbiamo un’interfaccia “Graph” che definisce i metodi di lavoro, ma che definisce anche tutta una serie di classi di implementazione.



Queste sono le 16 classi di implemetazione dell’interfaccia “Graph”. In base al nostro problema, possiamo capire quale classe usare.

Mi chiedo se il grafo che dovrò sviluppare è orientato o non orientato e se è pesato o non pesato. Dopo aver risposto a queste 2 domande, mi rimarrà solamente una colonna della 4 disponibili (quindi mi rimarranno solamente 4 classi).

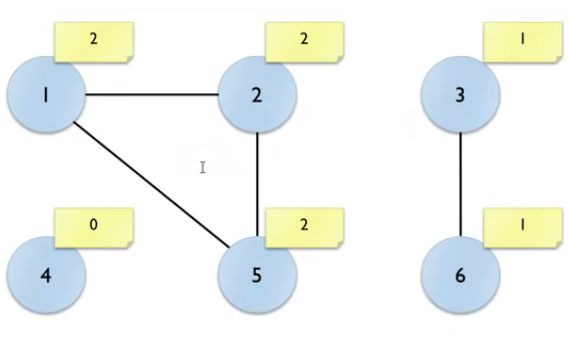


Come ultima cosa mi chiedo quale flessibilità voglio avere sugli archi. Posso avere un grafo semplice (per ogni coppia distinta di vertici esiste un solo arco), semplice con loop, multigrafo ( per ogni coppia distinta di vertici posso avere anche più archi) o uno pseudografo che sarebbe un multigrafo con i loop.

Dopo aver risposto a queste 3 domande, ottengo la classe tra le 16 che mi servirà per costruire il mio grafo.

Come facciamo a creare un grafo?

Usiamo i metodi delle librerie, naturalmente. Proviamo a costruire il seguente grafo.



Questo è un grafo semplice, non orientato e non pesato.

Come prima cosa andiamo nel nostro Main e dichiariamo una variabile di tipo Graph.

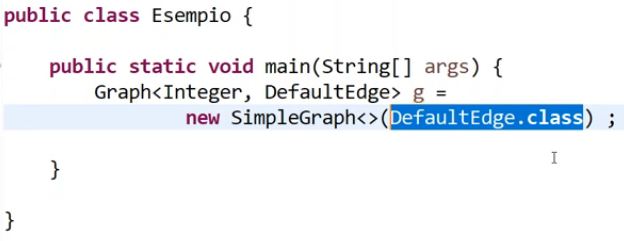
Il vertice è rappresentato dall’oggetto “Integer” in quanto nel nostro grafo i vertici sono rappresentati da dei numeri. Il grafo non è pesato, quindi per quanto riguarda gli archi posso usare “DefaultEdge”. Chiamiamo il nostro grafo “g”. Come ultima cosa, ci poniamo le 3 domande viste in precedenza e capiamo che ci serve un SimpleGraph.

Il costruttore dei grafi richiede un parametro strano; richiede un oggetto di tipo classe che corrisponde alla classe dell’arco, quindi “DefaultEdge.class”. L’attributo class è un attributo statico che restituisce un oggetto di tipo “Class” che descrive com’ è fatta una classe. Quindi non sto istanziando un arco, non sto facendo una “new”, sto prendendo il riferimento che descrive qual è l’oggetto che rappresenta quell’arco.

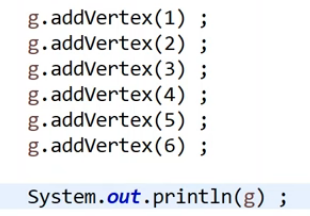
Perché gli serve? Gli serve per creare il grafo. Infatti, quando creo un vertice, lo passo io, ma quando aggiungo un arco, gli do il vertice di partenza e il vertice di arrivo e l’arco viene creato dal metodo. È lui che crea l’arco, fare la “new” di un oggetto di tipo arco. Come fa a sapere qual è il tipo di quell’oggetto? Ce lo chiede quando creiamo il grafo.

Ci chiede qual è la classe degli oggetti che deve creare tutte le volte che abbiamo bisogno di un arco.

Questa classe è appunto “**DefaultEdge**.class”.

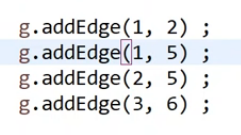


Però ora ho creato un grafo vuoto. Aggiungiamo quindi i vertici e stampiamo il risultato.

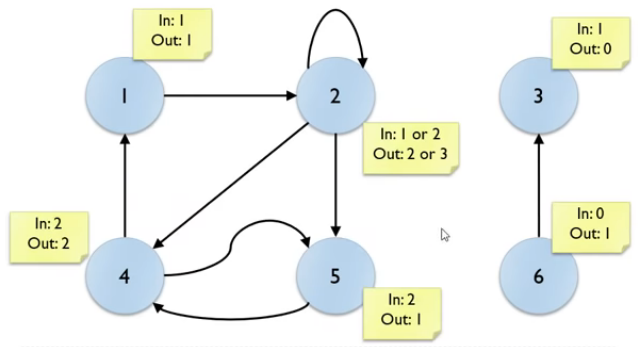
 

Come possiamo notare, il risultato ci viene stampato in modo “matematico”, quindi come due insiemi: quello dei vertici e quelli degli archi.

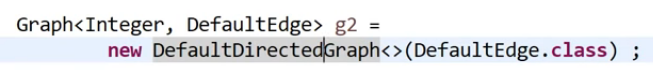
Ora posso aggiungere gli archi utilizzando “addEdge” che chiede come parametri il vertice di partenza e quello di arrivo (Nel nostro caso di tipo Integer). Posso mettere come oggetti gli stessi oggetti messi in “addVertex” (1,…,6) e altri oggetti che però sono “equals” a quelli precedenti.

Nel caso in cui avessimo voluto rappresentare il seguente grafo:



Il codice sarebbe stato il seguente:



Notiamo che in questo caso abbiamo uasto “DefaultDirectedGraph” in quanto il grafo è orientato, non pesato e semplice con loop. Anche in questo caso dobbiamo aggiungere tutti i vertici e gli archi. Però dobbiamo aggiungere gli stessi vertici.

Per evitare di riscrivere sempre lo stesso codice, notiamo che sul sito di jgrapht, in javadoc, c’è una classe chiamata “Graphs” (noi prima abbiamo usato l’interfaccia “Graph”).

Questa classe è una “collection di utilities” (un contenitore di metodi che ti possono aiutare a risolvere determinati problemi) ed è formata solamente da metodi static.

Usiamo il metodo “addAllVertices”. Il primo parametro di questo metodo è il grafo su cui voglio lavorare, mentre il secondo è la collection (i vertici di g che voglio aggiungere).



Ora devo aggiungere gli archi. Il risultato finale è il seguente:

…

Notiamo che in questo caso gli archi hanno le parentesi tonde in quanto sono rappresentati domce coppie di vertici (ordinati!!).