UFR ST - Besançon- M1 ISL - 2024/25

GRAPHS Practical Works Project: Graph API in Java

Representation by Adjacent Edge Lists

Foreword. For all the practical work to be submitted, respect the following rules:

- **Precisely follow the topic statement.** Evaluation is based on this requirements specification document.
- **Variable and method names** cited in the topic statement must strictly be respected.
- **The submission date for your work is a firm deadline.** *Submission is to be done on Moodle*, before the PW session indicated. The exact deadline will be given by your teacher. If a report is to be provided, you can either join it as a PDF document to your submission, or alternatively print it and have it dated by the department secretary.
- If your submission is made of a single file, give it the name of both the students with the appropriate extension. If your submission is made of several files, group them as a ZIP or TGZ archive, then name it the same way.
- Pay careful attention to indentation and comments in your code. In particular, each function has to be *specified* in the javadoc format, by precisely and comprehensively indicating: what the function computes, what it returns, what are the input parameters and what are the output parameters (if any).

1 Work Objectives

The objective of this first project is to develop in the java language a package providing useful classes for describing a *graph* data structure, with methods implementing some usual operations on graphs. Your package shall not be restricted to simple graphs, but should allow for the storage and manipulation of multigraphs with possible self-loops.

N.B. This first project is important because, not only will it be evaluated for itself, but it will constitute the API with which to implement the second project at the second half of the semester.

2 Graph Representation by Adjacent Edge Lists

Let $G = (V, E)$ be a directed graph such as that of Fig. 1. It could be represented a a table of adjacency lists, such for example as seen in the Lecture Courses, or as shown in Fig. 2. The edges are implicitly stored in such a representation: we know for example that there is an edge from node 1 to node 2 because node 2 appears in the list of nodes adjacent to 1. Although a table of adjacency lists would be sufficient for storing the graph structure into memory, we will use an alternative representation.

Indeed, algorithms manipulating graphs will often benefit from having explicit representations of edges such as lists of edges, rather than lists of nodes. Although it is less compact, the advantage is to avoid reconstructing the edges each time they are physically required.

Thus, we propose in this project to implement graphs as lists of the edges (rather than the nodes) adjacent to the nodes. We call this structure *Adjacent Edge Lists*. It is illustrated by means of Fig. 3 where the left-hand column gives a node number, and the right-hand column gives the list of edges outgoing from it.

3 Basic Implementation Instructions

Your classes are to be placed in a package named `m1graphs2024`.

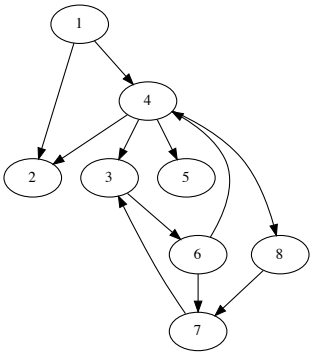

Figure 1: A directed graph

1	2, 4
2	
3	6
4	2, 3, 5, 8
5	
6	4, 7
7	3
8	7

Figure 2: Adjacency Lists representation of the graph of Fig. 1. N.B. *This is not the structure that we will implement.*

1	(1, 2), (1, 4)
2	
3	(3, 6)
4	(4, 2), (4, 3), (4, 5), (4, 8)
5	
6	(6, 4), (6, 7)
7	(7, 3)
8	(8, 7)

Figure 3: Adjacent Edge Lists representation of the graph of Fig. 1. **N.B. This is that structure that we will implement.**

3.1 Nodes

A class `Node` codes a node of a graph. A node has a number, possibly a name, and a *graph holder*, which is a reference to the graph to which the node belongs. The number is named `id` and it must act as an *identifier*, meaning that it uniquely identifies a node: not two different nodes can share the same `id` within a graph. Constructor for the class `Node` takes both the `id` and the graph holder as parameters. An overloaded version additionally takes the name of the node if any. Not mandatory, a constructor parameterized only by the graph holder can provide the `id` automatically if you consider it useful.

To deal with java collections, remember to properly override the `equals()` and `hashCode()` methods. Also, `Node` must implement `Comparable<Node>` for sorting the nodes by increasing order of `id`.

If you need afterwards to encapsulate more information inside a node, you will either complete the class or proceed by extension of the class `Node`.

3.2 Graphs

A class `Graph`, as its name implies, codes a graph.

We propose that the graph structure is coded as adjacent edge lists, meaning the list of its out edges is mapped to each of the graph nodes, in the following way:

```
Map<Node, List<Edge>> adjEdList;
```

Graphs are directed by default. Undirected graphs will be coded in a class named `UndirectedGraph` that *extends* the class `Graph`.

3.3 Edges

The class `Edge` aims at physically coding edges as objects. This will allow for physically disposing of the set E of edges of a graph $G = (V, E)$. Edges can be weighted: if so, we restrict the weights to integers, but values 0 or negative are allowed. A `null` value for the weight indicates that it is not weighted.

Constructor for the `Edge` class necessarily take the source node (`from`) and the target one (`to`) as parameters. An overloaded version takes ids instead of nodes' references. Other overloadings provide possibilities for assigning a weight. *Constructor constraints.* For successfully building an edge, both nodes `from` and `to` must be non-null and belong to the same graph. When this is not the case, the construction is refused and an `IllegalArgumentException` is thrown. N.B. As `IllegalArgumentException` is an unchecked exception, it not required to catch it or specify it.

Remember to override the `equals()` and `hashCode()` methods. Class `Edge` must implement interface `Comparable<Edge>` in order to be able to enumerate the edges in a predictable order. The edges are ordered first by source node number, then by target node number in case of source node equality, and then by increasing weight in case of source and target node equality.

For example, with this ordering, the edges of the graph of Fig. 1 enumerate as $(1,2)$, $(1,4)$, $(3,6)$, $(4,2)$, $(4,3)$, $(4,5)$ $(4,8)$, $(6,4)$, $(6,7)$, $(7,3)$ and finally $(8,7)$.

4 Graph Construction

At least two different ways should be provided to the user for building a graph: through dedicated constructors, or by reading a file.

4.1 Constructors

A graph can be constructed empty (default constructor). Or it can alternatively be described in the SA (*Successor Array*) formalism (see Lectures for a precise specification of the SA formalism). This can simply be obtained with a constructor accepting either an array of integers, or an unspecified number of integers.

For example, the graph of Fig. 1 could be obtained thanks to the line of code

```
Graph g = new Graph(2, 4, 0, 0, 6, 0, 2, 3, 5, 8, 0, 0, 4, 7, 0, 3, 0, 7, 0);
```

N.B. Only unweighted graphs can be built this way.

4.2 File Reading: the DOT format

Building a graph *via* a dedicated constructor is practicable only when it is small enough. The preferred method for building largest graphs will be to read them from a file.

You have been introduced to DOT in PW1. Additional resources are for example the Wikipedia page on DOT: DOT (graph description language), as well as the GraphViz Pocket Reference website.

You shall implement the possibility of inputting a graph by reading it from a DOT file. Actually, you are not asked to fully interpret the DOT language, but only a very simple subpart of it, as specified hereafter.

Simplified DOT Format Specification. The edges are enumerated one by one, one line at a time. The ending semicolon '`;`' may be provided but is not mandatory. The source and target nodes of an edge are separated by an *edge operator* being strictly either ' `--` ' or ' `->` ': pay attention to the explicit surrounding spaces. A typical unweighted (directed) edge appears for example as the line:

```
1 -> 2
```

The weight, if any, will be given thanks to *two* attributes `len` and `label`. Attribute `len` is the standard way of associating an edge with a weight, but DOT won't show the `len` value when it draws the graph. Thus duplicating the `len` value in attribute `label` is a way of getting DOT to explicitly show which weight goes with the edge. The two attributes are enumerated (in order `label` then `len`) between square brackets, separated by a comma and a space. The attribute values are given as numerical values, i.e. not surrounded by quotes. No space separates the equal sign that assigns its value to the attribute. A typical weighted (directed) edge appears for example as the line:

```
1 -> 2 [label=5, len=5]
```

For a complete example, the unweighted directed graph of Fig. 1 can be described in this restriction of the DOT format as shown in Fig. 4.

5 Minimum API to Implement

Feel free to implement all the methods you will find useful for the sake of graph manipulation! However and at the minimum, the following methods are required. Pay attention to implement the API strictly as indicated. Indeed, evaluation will be based on tests run automatically as a program that calls the methods of this API.

The name of your package must be `m1graphs2024`. For clarity, the methods are presented hereafter as related to nodes, edges, degrees, etc. Notice nonetheless that they all are methods of the `Graph` class, except where noted. Indeed, only the `Graph` class knows the graph structure. Individual instances of `Node` and `Edge` know nothing about the other instances.

```
# DOT representation for the directed graph of Fig. 1

digraph figureOne {
  1 -> 2
  1 -> 4
  3 -> 6
  4 -> 2
  4 -> 3
  4 -> 5
  4 -> 8
  6 -> 4
  6 -> 7
  7 -> 3
  8 -> 7
}
```

Figure 4: DOT file example for the graph of Fig. 1

5.1 Nodes

In addition to the constructors and collection utility methods specified in Sec. 3.1, the class `Node` will provide the following API.

1. `int getId()` for getting a node's `id`.
2. `Graph getGraph()` for getting the graph holder of node `this`.
3. `String getName()` for getting a node's name.
4. `List<Node> getSuccessors()` for getting a list *without duplicates* of the successors (or neighbours in the undirected case) of node `this`. "Without duplicates" means that each successor appears uniquely in the list returned, even in the case of a multigraph (as opposed to the next method).
5. `List<Node> getSuccessorsMulti()` for getting a list *with possible duplicates* of the successors (or neighbours in the undirected case) of node `this`. "With possible duplicates" means that in the case of a multigraph, and as opposed to the previous method, each successor (or neighbour) appears as many times as it is joined by an edge to node `this`.
6. `boolean adjacent(Node u)` for knowing whether node `u` is adjacent to node `this`. Overload with node id.
7. `int inDegree()` for knowing the in-degree of node `this`.
8. `int outDegree()` for knowing the out-degree of node `this`.
9. `int degree()` for knowing the degree of node `this`.
10. `List<Edge> getOutEdges()` for getting the list of all edges *leaving* node `this`.
11. `List<Edge> getInEdges()` for getting the list of all edges *entering* node `this`.
12. `List<Edge> getIncidentEdges()` for getting the list of all edges incident to node `this`. This is the union of the *out* and *in* edges. Notice that in the undirected case, all incident edges to a node are both *in* and *out* edges.
13. `List<Edge> getEdgesTo(Node u)` for getting the list of all edges going from node `this` to node `u`. N.B. Theoretically, this is the intersection of the out edges from *this* and the in edges to u; but computing this intersection is not efficient so it is better to simply get among the out edges from *this* the ones that lead to v.

5.2 Edges.

In addition to the constructors and collection utility methods specified in Sec. 3.3, the class `Edge` will provide the following API.

14. `Node from()` for returning the source node of a directed edge.
15. `Node to()` for returning the target node of a directed edge.
16. `Edge getSymmetric()` for getting the symmetric of an edge as a new `Edge` instance.
17. `boolean isSelfLoop()` for knowing if an edge is a self-loop or not.
18. `boolean isMultiEdge()` for knowing if edge `this` is a multi-edge, i.e. there is at least one other edge with the same endpoints as `this` in the graph holder.
19. `boolean isWeighted()` for knowing if an edge is weighted or not.
20. `Integer getWeight()` for getting the weight of an edge (or `null` in the unweighted case).

5.3 Graphs.

In addition to the constructors (see Sec. 4.1), the `Graph` class will provide the following API.

Methods related to the nodes.

21. `int nbNodes()` for knowing the number of nodes of the graph.
22. `boolean usesNode(Node n)` for knowing if node `n` has an id that is already used in `this` graph. Note that this method has to be overloaded with a version that directly takes an integer (the node id) as a parameter.
23. `boolean holdsNode(Node n)` for knowing if node `n` is a node of `this` graph, meaning its id is used and the graph holder of `n` is `this`.
24. `Node getNode(int id)` for getting the node instance held by `this` graph whose number is `id`. This method should return `null` in case `this` does not have a node with number `id`.
25. `boolean addNode(Node n)` for adding a node to the graph. This method also has to be overloaded to take a node id as a parameter. The returned boolean indicates whether the adding has been performed or not.
26. `boolean removeNode(Node n)` for removing a node from the graph, if it exists (otherwise *false* is returned). This consequently removes all edges incident to that node. An overloaded version must take an integer node id as parameter.
27. `List<Node> getAllNodes()` for getting the list of all the nodes of the graph.
28. `int largestNodeId()` for knowing the largest `id` used by a node in the graph.
29. `int smallestNodeId()` for knowing the smallest `id` used by a node in the graph.
30. `List<Node> getSuccessors(Node n)` for getting a list *without duplicates* of the successors (or neighbours in the undirected case) of node `n`. This should be overloaded with node id as usual. "Without duplicates" means that each successor appears uniquely in the list returned, even in the case of a multigraph (as opposed to the next method).
31. `List<Node> getSuccessorsMulti(Node n)` for getting a list *with possible duplicates* of the successors (or neighbours in the undirected case) of node `n`. This should be overloaded with node id as usual. "With possible duplicates" means that in the case of a multigraph, and as opposed to the previous method, each successor (or neighbour) appears as many times as it is joined by an edge to node `n`.

32. `boolean adjacent(Node u, Node v)` for knowing whether nodes `u` and `v` are adjacent in the graph. Overload with node ids.
33. `int inDegree(Node n)` for knowing the in-degree of node `n`. Overload.
34. `int outDegree(Node n)` for knowing the out-degree of node `n`. Overload.
35. `int degree(Node n)` for knowing the degree of node `n`. Overload.

Methods related to the edges.

36. `int nbEdges()` for knowing the number of edges of the graph.
37. `boolean existsEdge(Node u, Node v)` for knowing whether an edge exists in `this` graph between nodes `u` and `v`. Overloaded versions will take as parameters integer node ids, as well as edge reference.
38. `boolean isMultiEdge(Node u, Node v)` for knowing if edge (u, v) is a multi-edge, i.e. there is at least one other edge from u to v in graph `this`. Overloaded versions will take as parameters integer node ids, as well as edge reference.
39. `void addEdge(Node from, Node to)` for adding an edge from the node `from` towards the node `to`. These nodes have to be created first in case they don't already belong to the graph. Overload with node ids, edge weight, as well as with edge reference.
40. `boolean removeEdge(Node from, Node to)` for removing an edge between nodes `from` and `to`. Overload with node ids, edge weight, as well as with edge reference. The returned boolean indicates whether the removal has succeeded or not.
41. `List<Edge> getOutEdges(Node n)` for getting the list of all edges *leaving* node `n`. Overload with node id.
42. `List<Edge> getInEdges(Node n)` for getting the list of all edges *entering* node `n`. Overload as usual.
43. `List<Edge> getIncidentEdges(Node n)` for getting the list of all edges incident to node `n`. This is the union of the *out* and *in* edges. Notice that in the undirected case, all incident edges to a node are both *in* and *out* edges. Overload as usual.
44. `List<Edge> getEdges(Node u, Node v)` for getting the list of all edges going from node `u` to node `v`. N.B. Theoretically, this is the intersection of the out edges from u and the in edges to v; but computing this intersection is not efficient so it is better to simply get among the out edges from u the ones that lead to v.
45. `List<Edge> getAllEdges()` for getting the list of all the edges of the graph.

Methods related to the graph's representations and tranformations.

46. `int[] toSuccessorArray()` for obtaining a representation of the graph in the SA (*successor array*) formalism.
47. `int[][] toAdjMatrix()` for obtaining a representation of the graph as an adjacency matrix. Multigraphs are allowed, so the elements in the matrix may be greater than 1, indicating the number of edges between any two nodes. Also graphs with self-loops are allowed, thus allowing nonzero diagonal elements.
48. `Graph getReverse()` for computing in a new graph the reverse (G^{-1}) of the graph.
49. `Graph getTransitiveClosure()` for computing in a new graph the transitive closure of the graph.
50. `boolean isMultiGraph()` for knowing if `this` is a multi-graph (i.e. it has at least one multi-edge) or not.
51. `boolean isSimpleGraph()` for knowing if `this` is a simple graph (i.e. it has neither self-loop nor multi-edge) or not.
52. `boolean hasSelfLoops()` for knowing if `this` has self-loops or not.
53. `Graph toSimpleGraph()` for transforming the (possibly) multi-graph `this` into a simple one, by removing its self-loops and multi-edges.
54. `Graph copy()` to get a copy of `this` graph into a new graph.

5.4 Graph Traversal

By default, the graph traversals are launched from the lowest id node. They should not forget any node even in case the graph is disconnected.

55. `List<Node> getDFS()` for getting a *Depth-First Search* traversal of the graph.
56. `List<Node> getDFS(Node u)` for getting a *Depth-First Search* traversal of the graph, starting from node u. Overload with node id.
57. `List<Node> getBFS()` for getting a *Breadth-First Search* traversal of the graph.
58. `List<Node> getBFS(Node u)` for getting a *Breadth-First Search* traversal of the graph, starting from node u. Overload with node id.

Many properties of the graph can be deduced from its traversal, such as *is it connected?*, *strongly connected?*, *is it acyclic?*, etc. by taking advantage of a richer version of DFS as presented in the lecture on DFS. This allows for characterizing:

- the nodes by their colour (*white*, *gray*, *black*); their predecessor in the traversal; their discovery and finish timestamps;
- the edges by their type (*tree*, *backward*, *forward* or *cross* edge).

Thus "rich" versions of DFS are also expected by means of methods

59. `List<Node> getDFSWithVisitInfo(Map<Node, NodeVisitInfo> nodeVisit, Map<Edge, EdgeVisitType> edgeVisit)`
    (starting by default from lowest id node)
60. `List<Node> getDFSWithVisitInfo(Node u, Map<Node, NodeVisitInfo> nodeVisit, Map<Edge, EdgeVisitType> edgeVisit)`
    (starting from node u)

which map the nodes and edges to their characterization collected along the traversal.

`NodeVisitInfo` is a class that encapsulates the `colour` of a node (of type `enum NodeColour {WHITE, GRAY, BLACK}`), its predecessor (of type `Node`), its `discovery` and `finished` timestamps (of type `Integer`). N.B. The `time` variable used in the lecture on DFS can be coded as a `static int`.

`EdgeVisitType` is simply an enum: `enum EdgeVisitType {TREE, BACKWARD, FORWARD, CROSS}`.

5.5 Graph Import and Export

61. `static Graph fromDotFile(String filename)` for importing a file in the restricted DOT format as specified in Section 4.2. The argument `filename` specifies the absolute path to the DOT file *with no extension*. The extension is assumed to be '`.gv`'.
62. `static Graph fromDotFile(String filename, String extension)` allows for importing a file with a different extension (such for example as '`.dot`').

Export in the DOT format will also be provided, by means of at least three methods.

63. `String toDotString()` for exporting the graph as a String in the DOT syntax.
64. `void toDotFile(String fileName)` for exporting the graph as a file in the DOT syntax. The argument `filename` specifies the absolute path to the DOT file *with no extension*. The default extension '`.gv`' will be added to the file name.
65. `void toDotFile(String fileName, String extension)` for providing a different extension.

5.6 Reproducibility of the outputs

Despite it impacts negatively on the complexity, you are asked to sort the nodes and edges before you visit them during a traversal (DFS and BFS), export to the DOT format (file and String), or compute the successor array representation (`toSuccessorArray()`). *This is purely for the sake of reproducibility, and it is a non-functional requirement.*

A typical example looks like this:

```
List<Node> nodes = this.getAllNodes();
Collections.sort(nodes); // remove this line for efficiency
for (Node u: nodes) {
    List<Edge> edges = this.getOutEdges(u);
    Collections.sort(edges); //  remove this line for efficiency
    ....
```

This requirement should be easy to meet since classes `Edge` and `Node` have to implement the java `Comparable` interface. Thus, the java predefined `Collections.sort()` method applies to collections of such objects.

6 Documentation

As indicated in the foreword, your package must be fully documented and specified in the javadoc format. No additional written report is required for this project, instead the javadoc will serve as the documentation for your API.

7 Java Requirements

The project to submit must not be a Maven project but a standalone java package. It has to be developed in Java 11, with no need for libraries external to the Java Development Kit 11.

8 Random Graphs (optional)

Implementing the above API is the minimum required for this project. A very nice additional feature would be to offer the possibility of getting various kind of random graphs.

In the general sense, a random graph is a graph whose nodes and edges are chosen at random. It can nonetheless be computed so as to satisfy certain properties in order to get random examples for various kinds of graphs, such as:

- **connected graphs** (i.e. made of a single part),
- **dense graphs** (with density drawing near to 1),
- **sparse graphs** (with density drawing near to 0),
- more generally, **graphs parameterized** (by number of nodes, number of edges, edge probability distribution, ...)
- **DAG** (Directed Acyclic Graphs),
- **etc.** (whatever you can think of).

N.B. You can use Graphviz tools such as `gvgen` (or other) as a basis for random graph generation, and then enrich or modify the generated graphs according to your purpose.