ICS0005: Algorithms and Data Structures

Report for Homework 6

Task 14: Construct the deep clone of a given graph

Author: MD Tanveer Hasan (184069IVSB)

Description of the problem:

The task is to construct a deep clone of a given graph.

This means traversing the graph in the order in which its vertices and arcs were inserted, extracting the data, then inserting those data into a new graph in the same order of the original.

Note that we must construct a clone of a graph, so here the order of vertices and arcs are important. For example, lets take string representation of 2 graphs (the string representation method takes the order of vertices and arcs into account)

V1: av1->v2, av1->v3

V2: av2->v1

V3: av3->v1

V1: av1->v3, av1->v2

V2: av2->v1

V3: av3->v1

Although technically graphs are equal, one is not a clone of another because ordering of arcs is different for V1. So, when we must be mindful of the order in which vertices and arcs were inserted into the graph.

Description of the Solution:

Before we start the solution, let's look as to how the Graph class is implemented (we are only looking at the member variables of each class).

```
class Graph {
	private String id;
	private Vertex first;
	private int info = 0;
}
```

As you can see A graph object holds a reference to a Vertex object named first.

The we shall inspect Vertex class and Arc class.

```
class Vertex {
	private String id;
	private Vertex next;
	private Arc first;
	private int info = 0;
}
```

```
class Arc {
	private String id;
	private Vertex target;
	private Arc next;
	private int info = 0;
}
```

Vertex and Arc Class holds a reference to itself. From this we can deduce that They both can be chained together to form a LinkedList of Vertex objects and Arc objects respectively.

For all intents and purposes, a LinkedList is an array. To create a LinkedList, we create multiple objects, and the first objects holds a reference to the second object, and the second holds a reference to the third and so on. And to access the LinkedList, we simply store the reference to the first element (first element holds reference to 2^{nd}, 2^{nd} holds 3^{rd} and so on).

And each time we when we append or remove the first element of a LinkedList, it essentially becomes a stack. Because the last inserted element is currently the first element of the list.

So, each graph object holds a reference to the first Vertex of a LinkedList of Vertex.

Each vertex in turn, contains the reference to the first Arc of a LinkedList of arcs.

Each arc itself contains the destination Vertex it is connected to.

Next, we look at how Graphs are created.

Graph class provides createVertex and createArc method, which essentially modifies the LinkedList of vertex and arcs respectively to insert new elements to the list, more specifically, the new element is inserted into the first position of the list, i.e. it behaves like a stack. In order to the the oldest/bottom element in the stack. We must pop() or remove an element from the top of the stack, push/insert the element in another stack. Using this process, the 2^{nd} stack will contain all the element of the first, but in the reverse (or the original insertion order).

Equipped with this information, we can now investigate the actual steps of creating clone of a graph.

1)Traverse the LinkedList of vertex in the graph and put them in a stack for vertices, we also keep multiple copies of the stack. It is important to remember that the vertex will contain reference to 'old' vertex objects. In other words, we cannot directly use any element from the stack, otherwise, integrity of the (deep) clone will be compromised

2) The we pop off vertex from the stacks one by one. And use the vertex id to create new vertices in the new graphs, using createVertex method in the new graph. createVertex returns a reference to the newly created vertex, we store the reference inside a map so we can retrieve the vertex by id

3)Then again, we pop off vertex from another copy of the stack one by one. Each time we pop a vertex off, we traverse its LinkedList of arcs and store the arcs in another stack. Then, in that same iteration, we pop off arc from the arc-stack one by one, each time we pop off an Arc object, we create an arc in the new Graph object (the clone). We use the map in step 2 to get reference to the newly created vertex objects.

User guideline:

The user creates a new graph and uses the graph.clone() method to create the clone. Since the clone method is overriding the object.clone() method, the return value must be cast to (Graph) before usage.

Test Plan:

Inside the run method, there are multiple graphs and trees created, on which the clone method is applied, and their equality is tested.

1) A random tree is generated, cloned and tested against its parent for equality, then another random tree is generated and tested against the clone for inequality.
2) A single node with multiple loops is tested for inequality against a single node with one loop and no loop.
3) 4 random graphs (where node < edge count) are cloned and tested. Graphs are either simple graph with no loop or with loop and multigraph with no loop and with loop.
4) 1 simple graph where edge count == node count is cloned and tested.

For each kind of graph, a graph is created, cloned and tested so that they are equal.

If they are equal, then then cloning process was successful. However, the clone must also be a deep copy, that is, modifying g1 should not affect g2 (the clone). So g1 is modified, tested for equality against itself (which should output true) then g1 is tested against its previous clone (g2) which should output false.

Test output:

Below is the output from all the tests that are run inside the run() method

Begin assertion marks the begging of a test, where a graph is checked if they are equal or not.

```
      *******Begin assertion*******

assert G1 == G2: true

G
v1 --> av1_v3 (v1->v3)
v2 --> av2_v3 (v2->v3)
v3 --> av3_v1 (v3->v1) av3_v2 (v3->v2)


G
v1 --> av1_v3 (v1->v3)
v2 --> av2_v3 (v2->v3)
v3 --> av3_v1 (v3->v1) av3_v2 (v3->v2)

      *******Begin assertion*******

assert G1 == G2: true

G
VX -->
v1 --> av1_v3 (v1->v3)
v2 --> av2_v3 (v2->v3)
v3 --> av3_v1 (v3->v1) av3_v2 (v3->v2)


G
VX -->
v1 --> av1_v3 (v1->v3)
v2 --> av2_v3 (v2->v3)
v3 --> av3_v1 (v3->v1) av3_v2 (v3->v2)

      *******Begin assertion*******

assert G1 == G2: false

G
VX -->
v1 --> av1_v3 (v1->v3)
```

```
v2 --> av2_v3 (v2->v3)
v3 --> av3_v1 (v3->v1) av3_v2 (v3->v2)


G
v1 --> av1_v3 (v1->v3)
v2 --> av2_v3 (v2->v3)
v3 --> av3_v1 (v3->v1) av3_v2 (v3->v2)

      *******Begin assertion*******

assert G1 == G2: false

G
VX -->
v1 --> av1_v3 (v1->v3)
v2 --> av2_v3 (v2->v3)
v3 --> av3_v1 (v3->v1) av3_v2 (v3->v2)


G
v1 --> av1_v3 (v1->v3)
v2 --> av2_v3 (v2->v3)
v3 --> av3_v1 (v3->v1) av3_v2 (v3->v2)
v1 --> av1_v3 (v1->v3)
v2 --> av2_v3 (v2->v3)
v3 --> av3_v1 (v3->v1) av3_v2 (v3->v2)

      *******Begin assertion*******

assert G1 == G2: true

G
v1 --> av1_v4 (v1->v4) av1_v5 (v1->v5) av1_v2 (v1->v2) av1_v3 (v1->v3)
v2 --> av2_v3 (v2->v3) av2_v1 (v2->v1) av2_v4 (v2->v4) av2_v5 (v2->v5)
v3 --> av3_v5 (v3->v5) av3_v2 (v3->v2) av3_v1 (v3->v1) av3_v4 (v3->v4)
v4 --> av4_v1 (v4->v1) av4_v2 (v4->v2) av4_v3 (v4->v3) av4_v5 (v4->v5)
v5 --> av5_v3 (v5->v3) av5_v1 (v5->v1) av5_v2 (v5->v2) av5_v4 (v5->v4)


G
v1 --> av1_v4 (v1->v4) av1_v5 (v1->v5) av1_v2 (v1->v2) av1_v3 (v1->v3)
v2 --> av2_v3 (v2->v3) av2_v1 (v2->v1) av2_v4 (v2->v4) av2_v5 (v2->v5)
v3 --> av3_v5 (v3->v5) av3_v2 (v3->v2) av3_v1 (v3->v1) av3_v4 (v3->v4)
v4 --> av4_v1 (v4->v1) av4_v2 (v4->v2) av4_v3 (v4->v3) av4_v5 (v4->v5)
v5 --> av5_v3 (v5->v3) av5_v1 (v5->v1) av5_v2 (v5->v2) av5_v4 (v5->v4)

      *******Begin assertion*******

assert G1 == G2: true

G
V_NEW -->
```

```
v1 --> A_NEW (v1->V_NEW) av1_v4 (v1->v4) av1_v5 (v1->v5) av1_v2 (v1->v2)
av1_v3 (v1->v3)
v2 --> av2_v3 (v2->v3) av2_v1 (v2->v1) av2_v4 (v2->v4) av2_v5 (v2->v5)
v3 --> av3_v5 (v3->v5) av3_v2 (v3->v2) av3_v1 (v3->v1) av3_v4 (v3->v4)
v4 --> av4_v1 (v4->v1) av4_v2 (v4->v2) av4_v3 (v4->v3) av4_v5 (v4->v5)
v5 --> av5_v3 (v5->v3) av5_v1 (v5->v1) av5_v2 (v5->v2) av5_v4 (v5->v4)


G
V_NEW -->
v1 --> A_NEW (v1->V_NEW) av1_v4 (v1->v4) av1_v5 (v1->v5) av1_v2 (v1->v2)
av1_v3 (v1->v3)
v2 --> av2_v3 (v2->v3) av2_v1 (v2->v1) av2_v4 (v2->v4) av2_v5 (v2->v5)
v3 --> av3_v5 (v3->v5) av3_v2 (v3->v2) av3_v1 (v3->v1) av3_v4 (v3->v4)
v4 --> av4_v1 (v4->v1) av4_v2 (v4->v2) av4_v3 (v4->v3) av4_v5 (v4->v5)
v5 --> av5_v3 (v5->v3) av5_v1 (v5->v1) av5_v2 (v5->v2) av5_v4 (v5->v4)

      *******Begin assertion*******

assert G1 == G2: false

G
V_NEW -->
v1 --> A_NEW (v1->V_NEW) av1_v4 (v1->v4) av1_v5 (v1->v5) av1_v2 (v1->v2)
av1_v3 (v1->v3)
v2 --> av2_v3 (v2->v3) av2_v1 (v2->v1) av2_v4 (v2->v4) av2_v5 (v2->v5)
v3 --> av3_v5 (v3->v5) av3_v2 (v3->v2) av3_v1 (v3->v1) av3_v4 (v3->v4)
v4 --> av4_v1 (v4->v1) av4_v2 (v4->v2) av4_v3 (v4->v3) av4_v5 (v4->v5)
v5 --> av5_v3 (v5->v3) av5_v1 (v5->v1) av5_v2 (v5->v2) av5_v4 (v5->v4)


G
v1 --> av1_v4 (v1->v4) av1_v5 (v1->v5) av1_v2 (v1->v2) av1_v3 (v1->v3)
v2 --> av2_v3 (v2->v3) av2_v1 (v2->v1) av2_v4 (v2->v4) av2_v5 (v2->v5)
v3 --> av3_v5 (v3->v5) av3_v2 (v3->v2) av3_v1 (v3->v1) av3_v4 (v3->v4)
v4 --> av4_v1 (v4->v1) av4_v2 (v4->v2) av4_v3 (v4->v3) av4_v5 (v4->v5)
v5 --> av5_v3 (v5->v3) av5_v1 (v5->v1) av5_v2 (v5->v2) av5_v4 (v5->v4)

      *******Begin assertion*******

assert G1 == G2: true

G
v1 --> av1_v3 (v1->v3) av1_v4 (v1->v4) av1_v5 (v1->v5)
v2 --> av2_v3 (v2->v3) av2_v5 (v2->v5) av2_v4 (v2->v4)
v3 --> av3_v1 (v3->v1) av3_v2 (v3->v2) av3_v4 (v3->v4) av3_v5 (v3->v5)
v4 --> av4_v1 (v4->v1) av4_v3 (v4->v3) av4_v2 (v4->v2) av4_v5 (v4->v5)
v5 --> av5_v2 (v5->v2) av5_v1 (v5->v1) av5_v3 (v5->v3) av5_v4 (v5->v4)


G
v1 --> av1_v3 (v1->v3) av1_v4 (v1->v4) av1_v5 (v1->v5)
v2 --> av2_v3 (v2->v3) av2_v5 (v2->v5) av2_v4 (v2->v4)
v3 --> av3_v1 (v3->v1) av3_v2 (v3->v2) av3_v4 (v3->v4) av3_v5 (v3->v5)
```

```
v4 --> av4_v1 (v4->v1) av4_v3 (v4->v3) av4_v2 (v4->v2) av4_v5 (v4->v5)
v5 --> av5_v2 (v5->v2) av5_v1 (v5->v1) av5_v3 (v5->v3) av5_v4 (v5->v4)

      *******Begin assertion*******

assert G1 == G2: true

G
v1 --> av1_v2 (v1->v2) av1_v5 (v1->v5)
v2 --> av2_v5 (v2->v5) av2_v3 (v2->v3) av2_v1 (v2->v1) av2_v2 (v2->v2)
av2_v2 (v2->v2) av2_v4 (v2->v4)
v3 --> av3_v2 (v3->v2) av3_v4 (v3->v4) av3_v5 (v3->v5)
v4 --> av4_v3 (v4->v3) av4_v2 (v4->v2) av4_v5 (v4->v5)
v5 --> av5_v2 (v5->v2) av5_v1 (v5->v1) av5_v3 (v5->v3) av5_v4 (v5->v4)


G
v1 --> av1_v2 (v1->v2) av1_v5 (v1->v5)
v2 --> av2_v5 (v2->v5) av2_v3 (v2->v3) av2_v1 (v2->v1) av2_v2 (v2->v2)
av2_v2 (v2->v2) av2_v4 (v2->v4)
v3 --> av3_v2 (v3->v2) av3_v4 (v3->v4) av3_v5 (v3->v5)
v4 --> av4_v3 (v4->v3) av4_v2 (v4->v2) av4_v5 (v4->v5)
v5 --> av5_v2 (v5->v2) av5_v1 (v5->v1) av5_v3 (v5->v3) av5_v4 (v5->v4)

      *******Begin assertion*******

assert G1 == G2: true

G
v1 --> av1_v3 (v1->v3) av1_v3 (v1->v3) av1_v2 (v1->v2) av1_v4 (v1->v4)
v2 --> av2_v5 (v2->v5) av2_v5 (v2->v5) av2_v1 (v2->v1) av2_v5 (v2->v5)
v3 --> av3_v1 (v3->v1) av3_v1 (v3->v1) av3_v4 (v3->v4)
v4 --> av4_v1 (v4->v1) av4_v3 (v4->v3) av4_v5 (v4->v5)
v5 --> av5_v2 (v5->v2) av5_v2 (v5->v2) av5_v2 (v5->v2) av5_v4 (v5->v4)


G
v1 --> av1_v3 (v1->v3) av1_v3 (v1->v3) av1_v2 (v1->v2) av1_v4 (v1->v4)
v2 --> av2_v5 (v2->v5) av2_v5 (v2->v5) av2_v1 (v2->v1) av2_v5 (v2->v5)
v3 --> av3_v1 (v3->v1) av3_v1 (v3->v1) av3_v4 (v3->v4)
v4 --> av4_v1 (v4->v1) av4_v3 (v4->v3) av4_v5 (v4->v5)
v5 --> av5_v2 (v5->v2) av5_v2 (v5->v2) av5_v2 (v5->v2) av5_v4 (v5->v4)

      *******Begin assertion*******

assert G1 == G2: true

G
v1 --> av1_v4 (v1->v4) av1_v4 (v1->v4) av1_v5 (v1->v5)
v2 --> av2_v5 (v2->v5) av2_v5 (v2->v5)
v3 --> av3_v3 (v3->v3) av3_v3 (v3->v3) av3_v4 (v3->v4)
v4 --> av4_v5 (v4->v5) av4_v1 (v4->v1) av4_v1 (v4->v1) av4_v3 (v4->v3)
av4_v5 (v4->v5)
```

```
v5 --> av5_v4 (v5->v4) av5_v2 (v5->v2) av5_v1 (v5->v1) av5_v2 (v5->v2)
av5_v4 (v5->v4)


G
v1 --> av1_v4 (v1->v4) av1_v4 (v1->v4) av1_v5 (v1->v5)
v2 --> av2_v5 (v2->v5) av2_v5 (v2->v5)
v3 --> av3_v3 (v3->v3) av3_v3 (v3->v3) av3_v4 (v3->v4)
v4 --> av4_v5 (v4->v5) av4_v1 (v4->v1) av4_v1 (v4->v1) av4_v3 (v4->v3)
av4_v5 (v4->v5)
v5 --> av5_v4 (v5->v4) av5_v2 (v5->v2) av5_v1 (v5->v1) av5_v2 (v5->v2)
av5_v4 (v5->v4)

      *******Begin assertion*******

assert G1 == G2: true

G
V1 --> av1_v1 (V1->V1) av1_v1 (V1->V1) av1_v1 (V1->V1) av1_v1 (V1->V1)
av1_v1 (V1->V1)


G
V1 --> av1_v1 (V1->V1) av1_v1 (V1->V1) av1_v1 (V1->V1) av1_v1 (V1->V1)
av1_v1 (V1->V1)

      *******Begin assertion*******

assert G1 == G2: false

G
V1 --> av1_v1 (V1->V1) av1_v1 (V1->V1) av1_v1 (V1->V1) av1_v1 (V1->V1)
av1_v1 (V1->V1)


G
V1 --> av1_v1 (V1->V1) av1_v1 (V1->V1) av1_v1 (V1->V1) av1_v1 (V1->V1)
av1_v1 (V1->V1) av1_v1 (V1->V1)

      *******Begin assertion*******

assert G1 == G2: false

G
V1 -->


G
V1 --> av1_v1 (V1->V1) av1_v1 (V1->V1) av1_v1 (V1->V1) av1_v1 (V1->V1)
av1_v1 (V1->V1) av1_v1 (V1->V1)

      *******Begin assertion*******

assert G1 == G2: true
```

```
G
V1 -->


G
V1 -->
```

Full program text:

```
import java.util.*;

/**
 * Container class to different classes, that makes the whole set of
classes one
 * class formally.
 */
public class GraphTask {

   /** Main method. */
   public static void main(String[] args) {
      GraphTask a = new GraphTask();
      try {
         a.run();
      } catch (Exception e) {
         e.printStackTrace();
      }
      // throw new RuntimeException ("Nothing implemented yet!"); //
delete this
   }

   /** Actual main method to run examples and everything. */

   private void test_assertion(Graph g1, Graph g2, boolean cond) {
      String res = null;
      System.out.println("\t*******Begin assertion*******");
      res = (g1.isCloneOf(g2) == cond) ?  "" : "Assertion failed!!" ;

      System.out.println(res);
      System.out.println("assert G1 == G2: " + cond);
      System.out.println(g1);
      System.out.println(g2);
      if(res.isEmpty() == false) {
         System.exit(77);
      }
   }

   public void run() throws CloneNotSupportedException {

      // Test plan:
```

```
//// 1) Create a simple tree
//// 2) Create random graph (with edgecount == node count)
//// 3) Create random graph(edgecount > node count)
//// 4) Create simple graph with loops.
//// 5) Createmultigraph (multiple bidirectional arcs to few nodes)
//// 6) Createmultigraphwith loops
//// 7) create graph with one node and 1 or more loops.

Graph g1 = new Graph("G");
Graph g2 = null;
boolean args[][] = {
   {false,false}, // no loops no multigraph
   {true, false}, // loops but no multigraph
   {false, true}, // multigraph without loops
   {true,true}, //multigraph with loops
};


// test 1
g1.createRandomTree(3);
g2 = (Graph) g1.clone();
test_assertion(g1, g2, true);

g1.createVertex("VX");
test_assertion(g1, g1, true);
test_assertion(g1, g2, false);

g2.createRandomTree(3);
test_assertion(g1, g2, false);
g1.clearGraph();
g2.clearGraph();


//test 2
g1.createRandomGraph(5,10,false,false);
g2 = (Graph) g1.clone();
test_assertion(g1, g2, true);

g1.createVertex("V_NEW");
g1.createArc("A_NEW", g1.first.next, g1.first);
test_assertion(g1, g1, true);
test_assertion(g1, g2, false);
g1.clearGraph();
g2.clearGraph();;


// different number of arcs and edges
for(int i = 0; i < args.length; i++) {
   int node = 5;
   int edges = 9;
   g1.createRandomGraph(node,edges,args[i][0],args[i][1]);
   g2 = (Graph) g1.clone();
   test_assertion(g1, g2, true);
```

```
         g1.createVertex("V_NEW");
         g1.createArc("A_NEW", g1.first, g1.first.next);

         g1.clearGraph();
         g2.clearGraph();
      }

      //graph with one node and multiple loops
      g1.createGraphWithOneNode(5);
      g2 = (Graph) g1.clone();
      test_assertion(g1, g2, true);
      g2.createGraphWithOneNode(1);
      test_assertion(g1, g2, false);
      g1.clearGraph();
      test_assertion(g1, g2, false);
      g2.clearGraph();

      test_assertion(g1, g2, true);

   }

   class Vertex {

      private String id;
      private Vertex next; // ref to the first `Vertex` in a (linked)List
of vertex
      private Arc first;   // ref to the first `Arc` in a (linked)List of
arcs, Each arc in the list
                           // contains a `target` vertex, meaning
this(vertex) and target are connected.
      private int info = 0;
      // You can add more fields, if needed

      Vertex(String s, Vertex v, Arc e) {
         id = s;
         next = v;
         first = e;
      }

      Vertex(String s) {
         this(s, null, null);
      }

      @Override
      public String toString() {
         return id;
      }

      public boolean equals(Object obj) {
         // checks if this vertex and param vertex have the same id and
same arcs.
         Vertex v = (Vertex) obj;
         String thisID = this.id;
         String rhsID = v.id;
```

```
         if (thisID.equals(rhsID)) {
            // id's match, now check if arcs match or not
            Arc thisArc = this.first;
            Arc rhsArc = v.first;

            if(thisArc == null) {
               if(rhsArc == null) {
                  return true;
               }
               else {
                  return false;
               }
            }
            else {
            return thisArc.isClone(rhsArc);
            }
         }
         return false;
      }

      /**
       * Checks if a list of vertex is equal or not
       *
       * @param Vertex list to compare
       * @return true if they are equal, false if not.
       */
      public boolean isClone(Vertex obj) {
         Vertex rhs = obj;
         Vertex thisVertex = this;

         while (thisVertex != null && rhs != null) {
            if (!thisVertex.equals(rhs)) {
               return false;
            }
            thisVertex = thisVertex.next;
            rhs = rhs.next;
         }
         // both thisArc and rhs should be null at this point
         return thisVertex == rhs;
      }
   }

   /**
    * Arc represents one arrow in the graph. Two-directional edges are
represented
    * by two Arc objects (for both directions).
    */
   class Arc {

      private String id;
      private Vertex target;
      private Arc next;
      private int info = 0;
```

```
   // You can add more fields, if needed

   Arc(String s, Vertex v, Arc a) {
      id = s;
      target = v;
      next = a;
   }

   Arc(String s) {
      this(s, null, null);
   }

   @Override
   public String toString() {
      return id;
   }

   // TODO!!! Your Arc methods here!
   @Override
   public boolean equals(Object obj) {
      Arc rhs = (Arc) obj;
      String thisID = this.id;
      String rhsID = rhs.id;

      return thisID.equals(rhsID);
   }

   /**
    * Compares A linkedList of arc with another linkedeList of arc
    */
   public boolean isClone(Arc obj) {
      Arc thisArc = this;
      Arc rhs = obj;

      while (thisArc != null && rhs != null) {
         if (!thisArc.equals(rhs)) {
            return false;
         }
         thisArc = thisArc.next;
         rhs = rhs.next;
      }
      // both thisArc and rhs should be null at this point
      return thisArc == rhs;
   }
}

class Graph {

   private String id;
   private Vertex first;
   private int info = 0;
   // You can add more fields, if needed

   Graph(String s, Vertex v) {
```

```
         id = s;
         first = v;
      }

      Graph(String s) {
         this(s, null);
      }

      @Override
      public String toString() {
         String nl = System.getProperty("line.separator");
         StringBuffer sb = new StringBuffer(nl);
         sb.append(id);
         sb.append(nl);
         Vertex v = first;
         while (v != null) {
            sb.append(v.toString());
            sb.append(" -->");
            Arc a = v.first;
            while (a != null) {
               sb.append(" ");
               sb.append(a.toString());
               sb.append(" (");
               sb.append(v.toString());
               sb.append("->");
               sb.append(a.target.toString());
               sb.append(")");
               a = a.next;
            }
            sb.append(nl);
            v = v.next;
         }
         return sb.toString();
      }

      /**
       * Adds vertex with id = `vid` to the (linked)List of vertex(es)
       *
       * @param vid
       * @return
       */
      public Vertex createVertex(String vid) {
         Vertex res = new Vertex(vid);
         res.next = first;
         first = res;
         return res;
      }

      /**
       * Function to create an `Arc` (which represents a directed edge)
between `from`
       * and `to`
       *
       * @param aid
```

```
       * @param from
       * @param to
       * @return
       */
      public Arc createArc(String aid, Vertex from, Vertex to) {
         Arc res = new Arc(aid);
         // push new Arc `res` into Vertex `from` linkedList
         res.next = from.first;
         from.first = res;
         // res.target = destination vertex
         res.target = to;
         return res;

      }

      /**
       * Create a connected undirected random tree with n vertices. Each
new vertex is
       * connected to some random existing vertex.
       *
       * @param n number of vertices added to this graph
       */
      public void createRandomTree(int n) {
         if (n <= 0)
            return;
         Vertex[] varray = new Vertex[n];
         for (int i = 0; i < n; i++) {
            // "Push" new vertex to the linkedList of vertexes.
            varray[i] = createVertex("v" + String.valueOf(n - i));
            if (i > 0) {
               // Find an existing vertex between 0 and i
               int vnr = (int) (Math.random() * i);
               // Create (directed) arc from (existing) random vertex to i
               createArc("a" + varray[vnr].toString() + "_" +
varray[i].toString(), varray[vnr], varray[i]);
               // And in reverse
               createArc("a" + varray[i].toString() + "_" +
varray[vnr].toString(), varray[i], varray[vnr]);
            } else {
            }
         }
      }

      /**
       * Create an adjacency matrix of this graph. Side effect: corrupts
info fields
       * in the graph
       *
       * @return adjacency matrix
       */
      public int[][] createAdjMatrix() {
         info = 0;
         Vertex v = first;
         while (v != null) {
```

```
            v.info = info++;
            v = v.next;
         }
         int[][] res = new int[info][info];
         v = first;
         while (v != null) {
            int i = v.info;
            Arc a = v.first;
            while (a != null) {
               int j = a.target.info;
               res[i][j]++;
               a = a.next;
            }
            v = v.next;
         }
         return res;
      }

      /**
       * Create a connected simple (undirected, no loops, no multiple
arcs) random
       * graph with n vertices and m edges.
       *
       * @param n number of vertices
       * @param m number of edges
       */
      public void createRandomGraph(int n, int m, boolean withLoop,
boolean isMultigraph) {
         if (n <= 0)
            return;
         if (n > 2500)
            throw new IllegalArgumentException("Too many vertices: " + n);
         if (m < n - 1 || m > n * (n - 1) / 2)
            throw new IllegalArgumentException("Impossible number of
edges: " + m);
         first = null;
         createRandomTree(n); // n-1 edges created here
         Vertex[] vert = new Vertex[n];
         Vertex v = first;
         int c = 0;
         while (v != null) {
            vert[c++] = v;
            v = v.next;
         }
         int[][] connected = createAdjMatrix();
         int edgeCount = m - n + 1; // remaining edges
         while (edgeCount > 0) {
            int i = (int) (Math.random() * n); // random source
            int j = (int) (Math.random() * n); // random target
            if (i == j && !withLoop)
               continue; // no loops
            if ((connected[i][j] != 0 || connected[j][i] != 0) &&
!isMultigraph )
               continue; // no multiple edges
```

```
            Vertex vi = vert[i];
            Vertex vj = vert[j];
            createArc("a" + vi.toString() + "_" + vj.toString(), vi, vj);
            connected[i][j] = 1;
            createArc("a" + vj.toString() + "_" + vi.toString(), vj, vi);
            connected[j][i] = 1;
            edgeCount--; // a new edge happily created
         }
      }

      /**
       * creates graph with only one node and 1 or more loops
       * @param n number of loops
       */
      public void createGraphWithOneNode(int n) {
         // vertex already created
         while(n != 0) {
            createArc("av1_v1", this.first, this.first);
            n--;
         }
      }

      // TODO!!! Your Graph methods here! Probably your solution belongs
here.

      /**
       * Creates a clone (deep copy) of this graph
       * @return Graph object
       */
      @Override
      public Object clone() throws CloneNotSupportedException {
         Graph cln = null;
         Vertex thisVertex = null;
         Vertex cloneVertex = null;

         ArrayDeque<Vertex> vertexStack = null;
         ArrayDeque<Vertex> vStackCpy = null;
         // map will contains NEW copies of vertices along with their id.
         Map<String, Vertex> cloneVertexMap = null;

         // init datastrcutures
         cln = new Graph(this.id);
         cloneVertexMap = new HashMap<>();
         vertexStack = new ArrayDeque<>();
         thisVertex = this.first;

         // list all vertices of the graph
         while (thisVertex != null) {
            vertexStack.push(thisVertex);
            thisVertex = thisVertex.next;
         }

         try {
            vStackCpy = new ArrayDeque<>(vertexStack);
```

```
         } catch (Exception e) {
            e.printStackTrace();
         }

         // insert copies of vertices in the graph
         while (vStackCpy.isEmpty() == false) {
            Vertex n = null;
            String id = null;
            thisVertex = vStackCpy.pop();
            id = thisVertex.id;
            n = cln.createVertex(id);
            cloneVertexMap.put(id, n);
         }

         try {
            vStackCpy = new ArrayDeque<>(vertexStack);
         } catch (Exception e) {
            e.printStackTrace();
         }

         // now the graph contains copy of all the vertices of the
original.
         // now its time to copy arcs.
         while (vStackCpy.isEmpty() == false) {
            // Arc arc = null;
            Arc thisArc = null;
            Deque<Arc> arcStack = new ArrayDeque<>();

            // init
            thisVertex = vStackCpy.pop();
            thisArc = thisVertex.first;
            cloneVertex = cloneVertexMap.get(thisVertex.id);

            // insert all arcs into stack
            while (thisArc != null) {
               arcStack.push(thisArc);
               thisArc = thisArc.next;
            }

            // create arcs
            while (arcStack.isEmpty() == false) {
               String aid = null;
               Vertex from = null;
               Vertex to = null;

               // initalize
               thisArc = arcStack.pop();
               aid = thisArc.id;
               from = cloneVertex;
               to = cloneVertexMap.get(thisArc.target.id);

               cln.createArc(aid, from, to);
            }
```

```
         }
         return cln;
      }

      /**
       * Checks if 2 graphs are clones or not
       *
       * @param obj
       * @return
       */
      public boolean isCloneOf(Graph obj) {
         Graph rhs = obj;
         Vertex thisVertex = this.first;
         Vertex rhsVertex = rhs.first;

         return thisVertex.isClone(rhsVertex);

      }

      /**
       * Deletes the existing graph by assigining the graph.first to a new
(empty) vertex
       */
      public void clearGraph() {
         this.first = new Vertex("V1");
         this.info = 0;
      }

   }
}
```