TalTech

ICS0005
Algorithms and Data Structures

Report of Homework 6

Construct the deep clone of a given graph

Jaan Metlitski 179943IVSB

Description of the problem

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

What is meant by a deep clone is that in deep clone we create a clone which is independent of original object and making changes in the cloned object should not affect original object. Also it is important that order of data is the same as in the original object. Object in our case is graph and data is vertices and all members that belong to them and arcs.

So here is example of a graph in our case :

Graph_1
v42 -->
v1 --> av1_v6 (v1->v6) av1_v5 (v1->v5)
v2 --> av2_v6 (v2->v6) av2_v3 (v2->v3)
v3 --> av3_v2 (v3->v2) av3_v4 (v3->v4)
v4 --> av4_v5 (v4->v5) av4_v3 (v4->v3) av4_v6 (v4->v6)
v5 --> av5_v4 (v5->v4) av5_v1 (v5->v1) av5_v6 (v5->v6)
v6 --> av6_v2 (v6->v2) av6_v1 (v6->v1) av6_v4 (v6->v4) av6_v5 (v6->v5)

Graph_1 - is the name of graph.

v42 - is vertex, here we can see that it doesn't have an arc.

v1 - is also vertex, but with arcs

av1_v6 - is an arc

(v1->v6) - is the direction of an arc.

The goal was to create clone of a given graph, with the same vertices, arcs, directions and same order. After creating a deep clone of a given graph we need to compare original and clone graph to see if they are equal or not. To prove correctness of task we need to modify original graph and then compare them again to see that clone was not affected by modification of original and if before modification of original they were identical, after modification they are not.

Description of the solution

In order to create a clone of a given graph we need to traverse through all of it vertices and keep track of them.

```
public Object clone() throws CloneNotSupportedException {
    Graph clone = null;
    Vertex originalVertex = null;
    Vertex clonedVertex = null;

    // Turns out Stack is not optimal for this case
    // https://stackoverflow.com/questions/12524826/why-should-i-use-deque-over-stack
    ArrayDeque<Vertex> vertexStack = null;
    ArrayDeque<Vertex> vertexStackCopy = null;
    Map<String, Vertex> cloneVertexMap = null;
    clone = new Graph(this.id);
    cloneVertexMap = new HashMap<>();
    vertexStack = new ArrayDeque<>();
    originalVertex = this.first;
```

If we look at class Vertex it has reference to itself and to an arc, Arc class has also reference to itself and to vertex. We can use here linked list, in linked list each element points to the next.

```
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;
    }
```

```
class Vertex {

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

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

When we traverse through original graph we add it vertices to the vertexStack, then we declare vertexStackCopy and put there vertexStack this is needed because vertexStack will contain reference to original vertices, so we cannot use it directly otherwise deep clone criteria will not be matched. After that we remove and return first vertex from stack (pop) and using vertex id we create new vertices in the new (cloned) graph. While we create them we store reference in a map (cloneVertexMap) so we can retrieve them later.

```
while (originalVertex != null) {       //traverse through all the vertices we have
    vertexStack.push(originalVertex);
    originalVertex = originalVertex.next;
}

vertexStackCopy = new ArrayDeque<>(vertexStack);

while (!vertexStackCopy.isEmpty()) {    // inserting vertex copies
    Vertex v = null;
    String s = null;
    originalVertex = vertexStackCopy.pop();
    s = originalVertex.id;
    v = clone.createVertex(s);
    cloneVertexMap.put(s, v);
}
vertexStackCopy = new ArrayDeque<>(vertexStack);
```

After we storage vertices, we again pop vertices from vertexStackCopy and now we also need to traverse through arcs and store them in stack (Deque). After that, we pop arc from our arc stack copy (arcDeque) and setting arc direction from vertice.

```
while (!vertexStackCopy.isEmpty()) {     // inserting arc copies
    Arc a = null;
    Deque<Arc> arcDeque = new ArrayDeque<>();
    originalVertex = vertexStackCopy.pop();
    a = originalVertex.first;
    clonedVertex = cloneVertexMap.get(originalVertex.id);

    while (a != null) {     // inserting arc to deque
        arcDeque.push(a);
        a = a.next;
    }
```

Finally we create an arc in the cloned graph. We set the destination with the help of our map to get the right reference of created vertex.

```
        while (!arcDeque.isEmpty()) {
            String arcId = null;
            Vertex from = null;
            Vertex to = null;
            a = arcDeque.pop();
            arcId = a.id;
            from = clonedVertex;
            to = cloneVertexMap.get(a.target.id);
            clone.createArc(arcId, from, to);
        }
    }
    return clone;
}
```

User guidelines

To create and test the graphs user need to use run() method.

First of all user need to create graph and give it a name.

Beyond we declare a graph (g) and give it a name (Graph_1)

```
/**
 * Actual main method to run examples and everything.
 */
public void run() throws CloneNotSupportedException {
    Graph g = new Graph( s: "Graph_1");
```

After that we to actually create graph with vertices and edges.

Beyond we create a graph with createRandomSimpleGraph method, there we pass how many vertices and how many edges we want. In our example we create graph with 6 vertices and 8 edges.

```
g.createRandomSimpleGraph( n: 6,  m: 8);
```

Now to clone this graph, user need to declare another graph, g2 in our case and before call clone() method we need to cast g2 to Graph.

```
Graph g2 = null;
g2 = (Graph) g.clone();
```

Testing plan

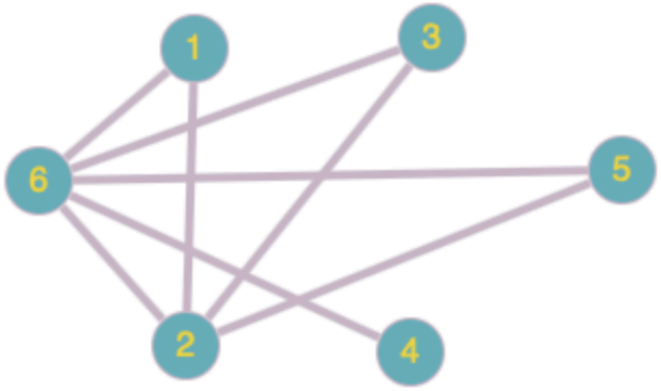
To test our clone() method we need to ensure that deep clone criteria is satisfied.

This means that if we clone original graph and after cloning, we modify original graph, the clone graph is should not be affected by this modification.

When we clone graph, we need to see that it is equal to original. If they are equal then our clone() method is working as it was intended.

We create 5 graphs.

One graph will be 8 vertices and 6 edges, this graph we also test that clone does not depend on original and that deep clone criteria is satisfied.



Second graph will have vertices more than edges.

Third graph will have zero vertices and zero edges.

Fourth graph will have exactly one vertices and zero edge.

Fifth graph will have 2000 vertices and 2000 edges. Running time we also be measured.

Used literature

https://enos.itcollege.ee/~japoia/algorithms/graphs1.html

https://www.geeksforgeeks.org/clone-an-undirected-graph-with-multiple-connected-components/

https://graphonline.ru

Appendix 1

Full text solution code

```
import java.util.ArrayDeque;
import java.util.Deque;
import java.util.HashMap;
import java.util.Map;


/**
 * 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();
        }
    }


    /**
     * Actual main method to run examples and everything.
     */
    public void run() throws CloneNotSupportedException {
```

```
Graph g = new Graph("Graph_1");
g.createRandomSimpleGraph(6, 8);
Graph g2 = null;
g2 = (Graph) g.clone();
System.out.println(g);
System.out.println(g2);
boolean test;
test = g.isItEqual(g2);
System.out.println("Are the graphs equal? ---> " + test);
g.createVertex("v42");
System.out.println(g);
System.out.println(g2);
test = g.isItEqual(g2);
System.out.println("Are the graphs equal? ---> " + test);
```

```
  Graph g3 = new Graph("Graph_2");
  g3.createRandomSimpleGraph(3,2);
  Graph g4 = null;
  g4 = (Graph) g3.clone();
  System.out.println(g3);
  System.out.println(g4);
  test = g3.isItEqual(g4);
  System.out.println("Are the graphs equal? ---> " + test);

  Graph g5 = new Graph("Graph_3");
  g5.createRandomSimpleGraph(0,0);
  Graph g6 = null;
  g6 = (Graph) g5.clone();
  System.out.println(g5);
  System.out.println(g6);
```

```
        test = g5.isItEqual(g6);
        System.out.println("Are the graphs equal? ---> " + test);

        Graph g7 = new Graph("Graph_4");
        g7.createRandomSimpleGraph(1,0);
        Graph g8 = null;
        g8 = (Graph) g7.clone();
        System.out.println(g7);
        System.out.println(g8);
        test = g7.isItEqual(g8);
        System.out.println("Are the graphs equal? ---> " + test);
        System.out.println();

        Graph g9 = new Graph("GRAPH_5");
        g9.createRandomSimpleGraph(2000,2000);
        Graph g10 = null;
        long stime = System.nanoTime();
        g10 = (Graph) g9.clone();
        long ftime = System.nanoTime();
        long diff = ftime - stime;
        System.out.println("Graph 5 is not printed here because it is
quite big print");
        test = g9.isItEqual(g10);
        System.out.println("Graph 5 clone time test in ms: " + diff /
1000000);
        System.out.println("Are the graphs equal? ---> " + test);


    }


    class Vertex {
```

```
        private String id;
        private Vertex next;
        private Arc first;
        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;
        }

        @Override
        public boolean equals(Object o) {  // check for  vertices id and
arc
            Vertex v = (Vertex) o;
            String originalId = this.id;
            String cloneId = v.id;
            if (originalId.equals(cloneId)) {
                Arc originalArc = this.first;
                Arc cloneArc = v.first;
```

```
            if (originalArc == null) {
                return cloneArc == null;
            } else return originalArc.isItEqual(cloneArc);
        }
        return false;
    }

    public boolean isItEqual(Vertex v) {   // check vertices
        Vertex cloneVertex = v;
        Vertex originalVertex = this;
        while (originalVertex != null && cloneVertex != null) {
            if (!originalVertex.equals(cloneVertex)) {
                return false;
            }
            originalVertex = originalVertex.next;
            cloneVertex = cloneVertex.next;
        }
        return originalVertex == cloneVertex;
    }

}


/**
 * 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;
}

@Override
public boolean equals(Object o) {   // check arc id
    Arc clone = (Arc) o;
    String originalId = this.id;
    String cloneId = clone.id;
    return originalId.equals(cloneId);
}

public boolean isItEqual(Arc a) {    // check arcs
    Arc cloneArc = a;
    Arc originalArc = this;
    while (originalArc != null && cloneArc != null) {
```

```
            if (!originalArc.equals(cloneArc)) {
                return false;
            }
            originalArc = originalArc.next;
            cloneArc = cloneArc.next;
        }
        return originalArc == cloneArc;
    }

}


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();
    }

    public Vertex createVertex(String vid) {
        Vertex res = new Vertex(vid);
        res.next = first;
        first = res;
        return res;
    }
```

```
        public Arc createArc(String aid, Vertex from, Vertex to) {
            Arc res = new Arc(aid);
            res.next = from.first;
            from.first = res;
            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++) {
                varray[i] = createVertex("v" + String.valueOf(n - i));
                if (i > 0) {
                    int vnr = (int) (Math.random() * i);
                    createArc("a" + varray[vnr].toString() + "_"
                            + varray[i].toString(), varray[vnr],
varray[i]);
                    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 createRandomSimpleGraph(int n, int m) {
        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)
                continue;  // no loops
            if (connected[i][j] != 0 || connected[j][i] != 0)
```

```
                    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
            }
        }

        /**
         * We have  original graph and need to make a deep clone of it.
         *
         * @return Graph of type object
         */
        @Override
        public Object clone() throws CloneNotSupportedException {
            Graph clone = null;
            Vertex originalVertex = null;
            Vertex clonedVertex = null;

            // Turns out Stack is not optimal for this case
            // https://stackoverflow.com/questions/12524826/why-should-i-
use-deque-over-stack
            ArrayDeque<Vertex> vertexStack = null;
            ArrayDeque<Vertex> vertexStackCopy = null;
            Map<String, Vertex> cloneVertexMap = null;
            clone = new Graph(this.id);
            cloneVertexMap = new HashMap<>();
```

```
            vertexStack = new ArrayDeque<>();
            originalVertex = this.first;
            while (originalVertex != null) {       //traverse through all
the vertices we have
                vertexStack.push(originalVertex);
                originalVertex = originalVertex.next;
            }

            vertexStackCopy = new ArrayDeque<>(vertexStack);

            while (!vertexStackCopy.isEmpty()) {    // inserting vertex
copies
                Vertex v = null;
                String s = null;
                originalVertex = vertexStackCopy.pop();
                s = originalVertex.id;
                v = clone.createVertex(s);
                cloneVertexMap.put(s, v);
            }
            vertexStackCopy = new ArrayDeque<>(vertexStack);

            while (!vertexStackCopy.isEmpty()) {     // inserting arc
copies
                Arc a = null;
                Deque<Arc> arcDeque = new ArrayDeque<>();
                originalVertex = vertexStackCopy.pop();
                a = originalVertex.first;
                clonedVertex = cloneVertexMap.get(originalVertex.id);

                while (a != null) {     // inserting arc to deque
                    arcDeque.push(a);
                    a = a.next;
```

```
                }

                while (!arcDeque.isEmpty()) {
                    String arcId = null;
                    Vertex from = null;
                    Vertex to = null;
                    a = arcDeque.pop();
                    arcId = a.id;
                    from = clonedVertex;
                    to = cloneVertexMap.get(a.target.id);
                    clone.createArc(arcId, from, to);
                }
            }
            return clone;
        }

        public boolean isItEqual(Graph graph) {    // check graphs
            Vertex originalVertex = this.first;
            Vertex cloneVertex = graph.first;
            if (originalVertex == null){
                return true;
            }
            return originalVertex.isItEqual(cloneVertex);
        }
    }

}
```

Appendix 2

Test results

```
Graph_1
v1 --> av1_v2 (v1->v2) av1_v6 (v1->v6)
v2 --> av2_v5 (v2->v5) av2_v1 (v2->v1) av2_v6 (v2->v6) av2_v3 (v2->v3)
v3 --> av3_v2 (v3->v2) av3_v6 (v3->v6)
v4 --> av4_v6 (v4->v6)
v5 --> av5_v2 (v5->v2) av5_v6 (v5->v6)
v6 --> av6_v2 (v6->v2) av6_v1 (v6->v1) av6_v3 (v6->v3) av6_v4 (v6->v4) av6_v5 (v6->v5)


Graph_1
v1 --> av1_v2 (v1->v2) av1_v6 (v1->v6)
v2 --> av2_v5 (v2->v5) av2_v1 (v2->v1) av2_v6 (v2->v6) av2_v3 (v2->v3)
v3 --> av3_v2 (v3->v2) av3_v6 (v3->v6)
v4 --> av4_v6 (v4->v6)
v5 --> av5_v2 (v5->v2) av5_v6 (v5->v6)
v6 --> av6_v2 (v6->v2) av6_v1 (v6->v1) av6_v3 (v6->v3) av6_v4 (v6->v4) av6_v5 (v6->v5)

Are the graphs equal? ---> true

Graph_1
v42 -->
v1 --> av1_v2 (v1->v2) av1_v6 (v1->v6)
v2 --> av2_v5 (v2->v5) av2_v1 (v2->v1) av2_v6 (v2->v6) av2_v3 (v2->v3)
v3 --> av3_v2 (v3->v2) av3_v6 (v3->v6)
v4 --> av4_v6 (v4->v6)
v5 --> av5_v2 (v5->v2) av5_v6 (v5->v6)
v6 --> av6_v2 (v6->v2) av6_v1 (v6->v1) av6_v3 (v6->v3) av6_v4 (v6->v4) av6_v5 (v6->v5)


Graph_1
v1 --> av1_v2 (v1->v2) av1_v6 (v1->v6)
v2 --> av2_v5 (v2->v5) av2_v1 (v2->v1) av2_v6 (v2->v6) av2_v3 (v2->v3)
v3 --> av3_v2 (v3->v2) av3_v6 (v3->v6)
v4 --> av4_v6 (v4->v6)
v5 --> av5_v2 (v5->v2) av5_v6 (v5->v6)
v6 --> av6_v2 (v6->v2) av6_v1 (v6->v1) av6_v3 (v6->v3) av6_v4 (v6->v4) av6_v5 (v6->v5)

Are the graphs equal? ---> false
```

```
Graph_2
v1 --> av1_v3 (v1->v3)
v2 --> av2_v3 (v2->v3)
v3 --> av3_v1 (v3->v1) av3_v2 (v3->v2)


Graph_2
v1 --> av1_v3 (v1->v3)
v2 --> av2_v3 (v2->v3)
v3 --> av3_v1 (v3->v1) av3_v2 (v3->v2)

Are the graphs equal? ---> true

Graph_3


Graph_3

Are the graphs equal? ---> true

Graph_4
v1 -->


Graph_4
v1 -->

Are the graphs equal? ---> true

Graph 5 is not printed here because it is quite big print
Graph 5 clone time test in ms: 3
Are the graphs equal? ---> true
```