Graph DS and Traversal

1. Objective

The objectives of Lab 5 are (1) to introduce an implementation of Graph DS and (2) to practice Traversal algorithms for this data structure.

2. Contents and references

You should read:

1. Chapter 14.1-3 of the following book (available on intranet of Ton Duc Thang University)

Michael T. Goodrich, Roberto Tamassia, Michael H. Goldwasser, [2014], Data Structures and Algorithms in Java, 6th Edition, Wiley.

2. E-Lectures for Graph DS and Traversal at site www.visualgo.net

3. Exercises

Suppose, we have an implementation of a Graph ADT. The code is presented to you in a zip file *LAB5.zip* provided to you with this file. You need to unzip the file. But the code in file GraphAlgorithms.java is uncompleted. Find the sentence "write your code here", then implement necessary Traversal algorithms. Then you need to write a **client class** which applies the Traversal algorithms to examples graphs. Submit your work to SAKAI.

4. Homework

4.1 Give answers for **10** questions with **medium** difficulty at visualgo.net.

4.2 Give answers for **10** questions with **hard** difficulty at visualgo.net.

The topic of the questions is Graph Traversal.

4.3 In the exercises you experience an implementation of a Graph ADT represented by Adjacency Map Structure. Your homework is to rewrite the code. You are required to use Adjacency **List** Structure to represent Graph ADT. Submit your homework solution to SAKAI.

Below are uncompleted code of class AdjacencyMapGraph and GraphExamples for your exercises. For your convenience, commands for compiling and running program in Cygwin environment (or Linux) are provided as follows:

Compilation: javac GraphExamples.java

Execution: java GraphExamples

Listing 1

```
/**
 * An implementation for a graph structure using an adjacency map for each
 * vertex.
 *
 * Every vertex stores an element of type V. Every edge stores an element of
 * type E.
 *
 * @author Michael T. Goodrich
 * @author Roberto Tamassia
 * @author Michael H. Goldwasser
 */
public class AdjacencyMapGraph<V, E> implements Graph<V, E> {
	private boolean isDirected;
	private PositionalList<Vertex<V>> vertices = new LinkedPositionalList<>();
	private PositionalList<Edge<E>> edges = new LinkedPositionalList<>();

	/**
	 * Constructs an empty graph. The parameter determines whether this is
	 * an undirected or directed graph.
	 */
	public AdjacencyMapGraph(boolean directed) {
		isDirected = directed;
```

```
}

/** Returns the number of vertices of the graph */
public int numVertices() {
        return vertices.size();
}

/** Returns the vertices of the graph as an iterable collection */
public Iterable<Vertex<V>> vertices() {
        return vertices;
}

/** Returns the number of edges of the graph */
public int numEdges() {
        return edges.size();
}

/** Returns the edges of the graph as an iterable collection */
public Iterable<Edge<E>> edges() {
        return edges;
}

/**
 * Returns the number of edges for which vertex v is the origin. Note
 * that for an undirected graph, this is the same result returned by
```

```
     * inDegree(v).
     *
     * @throws IllegalArgumentException
     *             if v is not a valid vertex
     */
    public int outDegree(Vertex<V> v) throws IllegalArgumentException {
        InnerVertex<V> vert = validate(v);
        return vert.getOutgoing().size();
    }

    /**
     * Returns an iterable collection of edges for which vertex v is the
     * origin. Note that for an undirected graph, this is the same result
     * returned by incomingEdges(v).
     *
     * @throws IllegalArgumentException
     *             if v is not a valid vertex
     */
    public Iterable<Edge<E>> outgoingEdges(Vertex<V> v) throws
IllegalArgumentException {
        InnerVertex<V> vert = validate(v);
        return vert.getOutgoing().values(); // edges are the values in
                                            // the adjacency map
    }
```

```
/**
 * Returns the number of edges for which vertex v is the destination.
 * Note that for an undirected graph, this is the same result returned
 * by outDegree(v).
 *
 * @throws IllegalArgumentException
 *          if v is not a valid vertex
 */
public int inDegree(Vertex<V> v) throws IllegalArgumentException {
    InnerVertex<V> vert = validate(v);
    return vert.getIncoming().size();
}


/**
 * Returns an iterable collection of edges for which vertex v is the
 * destination. Note that for an undirected graph, this is the same
 * result returned by outgoingEdges(v).
 *
 * @throws IllegalArgumentException
 *          if v is not a valid vertex
 */
public Iterable<Edge<E>> incomingEdges(Vertex<V> v) throws
IllegalArgumentException {
    InnerVertex<V> vert = validate(v);
    return vert.getIncoming().values(); // edges are the values in
```

```
                                                    // the adjacency map
        }


        /** Returns the edge from u to v, or null if they are not adjacent. */
        public Edge<E> getEdge(Vertex<V> u, Vertex<V> v) throws
IllegalArgumentException {
                InnerVertex<V> origin = validate(u);
                return origin.getOutgoing().get(v); // will be null if no edge
                                                    // from u to v
        }


        /**
         * Returns the vertices of edge e as an array of length two. If the
         * graph is directed, the first vertex is the origin, and the second is
         * the destination. If the graph is undirected, the order is arbitrary.
         */
        public Vertex<V>[] endVertices(Edge<E> e) throws IllegalArgumentException {
                InnerEdge<E> edge = validate(e);
                return edge.getEndpoints();
        }


        /** Returns the vertex that is opposite vertex v on edge e. */
        public Vertex<V> opposite(Vertex<V> v, Edge<E> e) throws
IllegalArgumentException {
                InnerEdge<E> edge = validate(e);
                Vertex<V>[] endpoints = edge.getEndpoints();
```

```
			if (endpoints[0] == v)

				return endpoints[1];

			else if (endpoints[1] == v)

				return endpoints[0];

			else

				throw new IllegalArgumentException("v is not incident to this 
edge");

		}


		/** Inserts and returns a new vertex with the given element. */

		public Vertex<V> insertVertex(V element) {

			InnerVertex<V> v = new InnerVertex<>(element, isDirected);

			v.setPosition(vertices.addLast(v));

			return v;

		}


		/**

		 * Inserts and returns a new edge between vertices u and v, storing

		 * given element.

		 *

		 * @throws IllegalArgumentException

		 *             if u or v are invalid vertices, or if an edge already

		 *             exists between u and v.

		 */

		public Edge<E> insertEdge(Vertex<V> u, Vertex<V> v, E element) throws 
IllegalArgumentException {
```

```
        if (getEdge(u, v) == null) {
            InnerEdge<E> e = new InnerEdge<>(u, v, element);
            e.setPosition(edges.addLast(e));
            InnerVertex<V> origin = validate(u);
            InnerVertex<V> dest = validate(v);
            origin.getOutgoing().put(v, e);
            dest.getIncoming().put(u, e);
            return e;
        } else
            throw new IllegalArgumentException("Edge from u to v exists");
    }

    /** Removes a vertex and all its incident edges from the graph. */
    public void removeVertex(Vertex<V> v) throws IllegalArgumentException {
        InnerVertex<V> vert = validate(v);
        // remove all incident edges from the graph
        for (Edge<E> e : vert.getOutgoing().values())
            removeEdge(e);
        for (Edge<E> e : vert.getIncoming().values())
            removeEdge(e);
        // remove this vertex from the list of vertices
        vertices.remove(vert.getPosition());
        vert.setPosition(null); // invalidates the vertex
    }
```

```
@SuppressWarnings({ "unchecked" })
/** Removes an edge from the graph. */
public void removeEdge(Edge<E> e) throws IllegalArgumentException {
	InnerEdge<E> edge = validate(e);
	// remove this edge from vertices' adjacencies
	InnerVertex<V>[] verts = (InnerVertex<V>[]) edge.getEndpoints();
	verts[0].getOutgoing().remove(verts[1]);
	verts[1].getIncoming().remove(verts[0]);
	// remove this edge from the list of edges
	edges.remove(edge.getPosition());
	edge.setPosition(null); // invalidates the edge
}

@SuppressWarnings({ "unchecked" })
private InnerVertex<V> validate(Vertex<V> v) {
	if (!(v instanceof InnerVertex))
		throw new IllegalArgumentException("Invalid vertex");
	InnerVertex<V> vert = (InnerVertex<V>) v; // safe cast
	if (!vert.validate(this))
		throw new IllegalArgumentException("Invalid vertex");
	return vert;
}

@SuppressWarnings({ "unchecked" })
private InnerEdge<E> validate(Edge<E> e) {
```

```
        if (!(e instanceof InnerEdge))
            throw new IllegalArgumentException("Invalid edge");
        InnerEdge<E> edge = (InnerEdge<E>) e; // safe cast
        if (!edge.validate(this))
            throw new IllegalArgumentException("Invalid edge");
        return edge;
    }

    // ---------------- nested Vertex class ----------------
    /** A vertex of an adjacency map graph representation. */
    private class InnerVertex<V> implements Vertex<V> {
        private V element;
        private Position<Vertex<V>> pos;
        private Map<Vertex<V>, Edge<E>> outgoing, incoming;

        /**
         * Constructs a new InnerVertex instance storing the given
         * element.
         */
        public InnerVertex(V elem, boolean graphIsDirected) {
            element = elem;
            outgoing = new ProbeHashMap<>();
            if (graphIsDirected)
                incoming = new ProbeHashMap<>();
            else
```

```
            incoming = outgoing; // if undirected, alias
                                  // outgoing map
    }

    /**
     * Validates that this vertex instance belongs to the given
     * graph.
     */
    public boolean validate(Graph<V, E> graph) {
        return (AdjacencyMapGraph.this == graph && pos != null);
    }

    /** Returns the element associated with the vertex. */
    public V getElement() {
        return element;
    }

    /**
     * Stores the position of this vertex within the graph's vertex
     * list.
     */
    public void setPosition(Position<Vertex<V>> p) {
        pos = p;
    }
```

```
        /**
         * Returns the position of this vertex within the graph's vertex
         * list.
         */
        public Position<Vertex<V>> getPosition() {
                return pos;
        }

        /**
         * Returns reference to the underlying map of outgoing edges.
         */
        public Map<Vertex<V>, Edge<E>> getOutgoing() {
                return outgoing;
        }

        /**
         * Returns reference to the underlying map of incoming edges.
         */
        public Map<Vertex<V>, Edge<E>> getIncoming() {
                return incoming;
        }
} // ------------ end of InnerVertex class ------------

// ---------------- nested InnerEdge class ----------------
/** An edge between two vertices. */
```

```
private class InnerEdge<E> implements Edge<E> {
	private E element;
	private Position<Edge<E>> pos;
	private Vertex<V>[] endpoints;

	@SuppressWarnings({ "unchecked" })
	/**
	 * Constructs InnerEdge instance from u to v, storing the given
	 * element.
	 */
	public InnerEdge(Vertex<V> u, Vertex<V> v, E elem) {
		element = elem;
		endpoints = (Vertex<V>[]) new Vertex[] { u, v }; // array
		                                                  // of
		                                                  // length
		                                                  // 2
	}

	/** Returns the element associated with the edge. */
	public E getElement() {
		return element;
	}

	/** Returns reference to the endpoint array. */
	public Vertex<V>[] getEndpoints() {
```

```
            return endpoints;
        }

        /**
         * Validates that this edge instance belongs to the given graph.
         */
        public boolean validate(Graph<V, E> graph) {
            return AdjacencyMapGraph.this == graph && pos != null;
        }

        /**
         * Stores the position of this edge within the graph's vertex
         * list.
         */
        public void setPosition(Position<Edge<E>> p) {
            pos = p;
        }

        /**
         * Returns the position of this edge within the graph's vertex
         * list.
         */
        public Position<Edge<E>> getPosition() {
            return pos;
        }
```

```
} // ------------ end of InnerEdge class ------------


/**
 * Returns a string representation of the graph. This is used only for
 * debugging; do not rely on the string representation.
 */
public String toString() {
        StringBuilder sb = new StringBuilder();
        // sb.append("Edges:");
        // for (Edge<E> e : edges) {
        // Vertex<V>[] verts = endVertices(e);
        // sb.append(String.format(" (%s->%s, %s)",
        // verts[0].getElement(), verts[1].getElement(),
        // e.getElement()));
        // }
        // sb.append("\n");
        for (Vertex<V> v : vertices) {
                sb.append("Vertex " + v.getElement() + "\n");
                if (isDirected)
                        sb.append(" [outgoing]");
                sb.append(" " + outDegree(v) + " adjacencies:");
                for (Edge<E> e : outgoingEdges(v))
                        sb.append(String.format(" (%s, %s)", opposite(v,
e).getElement(), e.getElement()));
                sb.append("\n");
```

```
                                if (isDirected) {

                                          sb.append(" [incoming]");

                                          sb.append(" " + inDegree(v) + " adjacencies:");

                                          for (Edge<E> e : incomingEdges(v))

                                                    sb.append(String.format(" (%s, %s)", opposite(v,
e).getElement(),

                                                                        e.getElement()));

                                          sb.append("\n");

                                }

                      }

                      return sb.toString();

            }

}
```

Listing 2

```
import java.util.HashMap;

import java.util.TreeSet;



/**

 * This class provides a utility to build a graph from a list of edges.

 *

 * It also contains specific factory methods to generate graph instances used as

 * examples within Data Structures and Algorithms in Java, 6th edition.

 */
```

```
public class GraphExamples {


	/**
	 * Constructs a graph from an array of array strings.
	 *
	 * An edge can be specified as { "SFO", "LAX" }, in which case edge is
	 * created with default edge value of 1, or as { "SFO", "LAX", "337" }, in
	 * which case the third entry should be a string that will be converted to
	 * an integral value.
	 */
	public static Graph<String, Integer> graphFromEdgelist(String[][] edges, boolean
directed) {
		Graph<String, Integer> g = new AdjacencyMapGraph<>(directed);


		// first pass to get sorted set of vertex labels
		TreeSet<String> labels = new TreeSet<>();
		for (String[] edge : edges) {
			labels.add(edge[0]);
			labels.add(edge[1]);
		}


		// now create vertices (in alphabetical order)
		HashMap<String, Vertex<String>> verts = new HashMap<>();
		for (String label : labels)
			verts.put(label, g.insertVertex(label));
```

```
        // now add edges to the graph
        for (String[] edge : edges) {
            Integer cost = (edge.length == 2 ? 1 : Integer.parseInt(edge[2]));
            g.insertEdge(verts.get(edge[0]), verts.get(edge[1]), cost);
        }
        return g;
    }


    /** Returns the unweighted, directed graph from Figure 14.3 of DSAJ6. */
    public static Graph<String, Integer> figure14_3() {
        String[][] edges = { { "BOS", "SFO" }, { "BOS", "JFK" }, { "BOS",
"MIA" }, { "JFK", "BOS" }, { "JFK", "DFW" },
                { "JFK", "MIA" }, { "JFK", "SFO" }, { "ORD", "DFW" },
{ "ORD", "MIA" }, { "LAX", "ORD" },
                { "DFW", "SFO" }, { "DFW", "ORD" }, { "DFW",
"LAX" }, { "MIA", "DFW" }, { "MIA", "LAX" }, };
        return graphFromEdgelist(edges, true);
    }


    /** Returns the unweighted, directed graph from Figure 14.8 of DSAJ6. */
    public static Graph<String, Integer> figure14_8() {
        String[][] edges = { { "BOS", "SFO" }, { "BOS", "JFK" }, { "BOS",
"MIA" }, { "JFK", "BOS" }, { "JFK", "DFW" },
                { "JFK", "MIA" }, { "JFK", "SFO" }, { "ORD", "DFW" },
{ "ORD", "MIA" }, { "LAX", "ORD" },
                { "DFW", "SFO" }, { "DFW", "ORD" }, { "DFW",
"LAX" }, { "MIA", "DFW" }, { "MIA", "LAX" },
```

```
                                        { "SFO", "LAX" }, };

                    return graphFromEdgelist(edges, true);

          }



          /**

           * Returns the unweighted, undirected graph from Figure 14.9 of DSAJ6. This

           * is the same graph as in Figure 14.10.

           */

          public static Graph<String, Integer> figure14_9() {

                    String[][] edges = { { "A", "B" }, { "A", "E" }, { "A", "F" }, { "B", "C" },
{ "B", "F" }, { "C", "D" },

                                        { "C", "G" }, { "D", "G" }, { "D", "H" }, { "E", "F" }, { "E",
"I" }, { "F", "I" }, { "G", "J" },

                                        { "G", "K" }, { "G", "L" }, { "H", "L" }, { "I", "J" }, { "I",
"M" }, { "I", "N" }, { "J", "K" },

                                        { "K", "N" }, { "K", "O" }, { "L", "P" }, { "M", "N" }, };

                    return graphFromEdgelist(edges, false);

          }



          /** Returns the unweighted, directed graph from Figure 14.11 of DSAJ6. */

          public static Graph<String, Integer> figure14_11() {

                    String[][] edges = { { "BOS", "JFK" }, { "BOS", "MIA" }, { "JFK",
"BOS" }, { "JFK", "DFW" }, { "JFK", "MIA" },

                                        { "JFK", "SFO" }, { "ORD", "DFW" }, { "LAX", "ORD" },
{ "DFW", "SFO" }, { "DFW", "ORD" },

                                        { "DFW", "LAX" }, { "MIA", "DFW" }, { "MIA",
"LAX" }, };

                    return graphFromEdgelist(edges, true);
```

```
	}

	/**
	 * Returns the unweighted, directed graph from Figure 14.12 of DSAJ6. This
	 * is the same graph as in Figure 14.13.
	 */
	public static Graph<String, Integer> figure14_12() {
		String[][] edges = { { "A", "C" }, { "A", "D" }, { "B", "D" }, { "B", "F" }, { "C", "D" }, { "C", "E" },
				{ "C", "H" }, { "D", "F" }, { "E", "G" }, { "F", "G" }, { "F", "H" }, { "G", "H" } };
		return graphFromEdgelist(edges, true);
	}

	/** Returns the weighted, undirected graph from Figure 14.14 of DSAJ6. */
	public static Graph<String, Integer> figure14_14() {
		String[][] edges = { { "SFO", "LAX", "337" }, { "SFO", "BOS", "2704" }, { "SFO", "ORD", "1846" },
				{ "SFO", "DFW", "1464" }, { "LAX", "DFW", "1235" }, { "LAX", "MIA", "2342" }, { "DFW", "ORD", "802" },
				{ "DFW", "MIA", "1121" }, { "ORD", "BOS", "867" }, { "ORD", "JFK", "740" }, { "MIA", "JFK", "1090" },
				{ "MIA", "BOS", "1258" }, { "JFK", "BOS", "187" }, };
		return graphFromEdgelist(edges, false);
	}

	/**
```

```
         * Returns the weighted, undirected graph from Figure 14.15 of DSAJ6. This
         * is the same graph as in Figures 14.16, 14.17, and 14.20-14.24.
         */
        public static Graph<String, Integer> figure14_15() {
                String[][] edges = { { "SFO", "LAX", "337" }, { "SFO", "BOS", "2704" },
{ "SFO", "ORD", "1846" },
                                { "SFO", "DFW", "1464" }, { "LAX", "DFW", "1235" },
{ "LAX", "MIA", "2342" }, { "DFW", "ORD", "802" },
                                { "DFW", "JFK", "1391" }, { "DFW", "MIA", "1121" },
{ "ORD", "BOS", "867" }, { "ORD", "PVD", "849" },
                                { "ORD", "JFK", "740" }, { "ORD", "BWI", "621" },
{ "MIA", "BWI", "946" }, { "MIA", "JFK", "1090" },
                                { "MIA", "BOS", "1258" }, { "BWI", "JFK", "184" },
{ "JFK", "PVD", "144" }, { "JFK", "BOS", "187" }, };
                return graphFromEdgelist(edges, false);
        }


        public static void main(String[] args) {
                System.out.println("Figure 14.3");
                System.out.println(figure14_3());


                System.out.println("Figure 14.8");
                System.out.println(figure14_8());


                System.out.println("Figure 14.9");
                System.out.println(figure14_9());
```

```
        System.out.println("Figure 14.11");
        System.out.println(figure14_11());


        System.out.println("Figure 14.12");
        System.out.println(figure14_12());


        System.out.println("Figure 14.14");
        System.out.println(figure14_14());


        System.out.println("Figure 14.15");
        System.out.println(figure14_15());
    }

}
```

Listing 3

```
import java.util.Set;
import java.util.HashSet;

/**
 * A collection of graph algorithms.
 */
public class GraphAlgorithms {
```

```
	/**
	 * Performs depth-first search of the unknown portion of Graph g starting at
	 * Vertex u.
	 *
	 * @param g
	 *          Graph instance
	 * @param u
	 *          Vertex of graph g that will be the source of the search
	 * @param known
	 *          is a set of previously discovered vertices
	 * @param forest
	 *          is a map from nonroot vertex to its discovery edge in DFS
	 *          forest
	 *
	 *          As an outcome, this method adds newly discovered vertices
	 *          (including u) to the known set, and adds discovery graph edges
	 *          to the forest.
	 */
	public static <V, E> void DFS(Graph<V, E> g, Vertex<V> u, Set<Vertex<V>> known, Map<Vertex<V>, Edge<E>> forest) {
//write your code here
	}


	/**
	 * Performs DFS for the entire graph and returns the DFS forest as a map.
```

```
     *
     * @return map such that each nonroot vertex v is mapped to its discovery
     *        edge (vertices that are roots of a DFS trees in the forest are
     *        not included in the map).
     */
    public static <V, E> Map<Vertex<V>, Edge<E>> DFSComplete(Graph<V, E> g)
{
        Set<Vertex<V>> known = new HashSet<>();
        Map<Vertex<V>, Edge<E>> forest = new ProbeHashMap<>();
        for (Vertex<V> u : g.vertices())
            if (!known.contains(u))
                DFS(g, u, known, forest); // (re)start the DFS process at u
        return forest;
    }

    /**
     * Performs breadth-first search of the undiscovered portion of Graph g
     * starting at Vertex s.
     *
     * @param g
     *          Graph instance
     * @param s
     *          Vertex of graph g that will be the source of the search
     * @param known
     *          is a set of previously discovered vertices
```

```
         * @param forest
         *           is a map from nonroot vertex to its discovery edge in DFS
         *           forest
         *
         *           As an outcome, this method adds newly discovered vertices
         *           (including s) to the known set, and adds discovery graph edges
         *           to the forest.
         */
        public static <V, E> void BFS(Graph<V, E> g, Vertex<V> s, Set<Vertex<V>>
known, Map<Vertex<V>, Edge<E>> forest) {
                //write your code here
        }


        /**
         * Performs BFS for the entire graph and returns the BFS forest as a map.
         *
         * @return map such that each nonroot vertex v is mapped to its discovery
         *         edge (vertices that are roots of a BFS trees in the forest are
         *         not included in the map).
         */
        public static <V, E> Map<Vertex<V>, Edge<E>> BFSComplete(Graph<V, E> g) {
                Map<Vertex<V>, Edge<E>> forest = new ProbeHashMap<>();
                Set<Vertex<V>> known = new HashSet<>();
                for (Vertex<V> u : g.vertices())
                        if (!known.contains(u))
```

```
                    BFS(g, u, known, forest);
        return forest;
    }
}
```