CSCI 665 Foundations of Algorithms

Mohan Kumar

SPRING 2014

Lab Assignment II

Due at 4:30 PM April 14, 2014

Assignment Problem

Write programs to implement the following graph algorithms:

1. Kruskal's Minimum Spanning Tree
2. Dijkstra's Single Source Shortest Paths
3. Floyd Warshall's All Pairs Shortest Paths
4. Transitive Closure

Execute your sorting programs for the following sets of graphs:

a. Set_1: Node connectivity less than 10
b. Set_2: Node connectivity more than $n/2$
c. Set_3: Random Graphs

Presentation of Results: Measure CPU time and present the final result (in the form of matrices) for graph sizes 40, 60 and 100. Present your time results using tables.

Required Submissions: Code with comments, sample results in the form of matrices and tables to show the CPU times. Include instructions for executing your code.
*Results: Each data point should be an average of 10 or more reruns of the execution.

Programming Language and Data Structure: **Your choice**. Please mention your choices and provide a justification in the summary report.

Assignment submissions are due by 4:30 PM April 14, 2014. Submission instructions will be posted on myCourses.

PLEASE NOTE THAT your PROGRAMS WILL BE TESTED WITH DIFFERENT DATA SETS.

Please refer to the following files for input data and general instructions – GeneralLabInstructions.txt, SampleOutput.txt and GraphInputGenerator.java.

General programming instructions

- Write your programs neatly. COMMENT your programs reasonably (include brief comments describing the main purpose of a specific block of code).
- All the programs should take input from STANDARD INPUT, not from a file.
- All programs should write output to STANDARD OUTPUT, not a file.
- Submit only the necessary files in the dropbox by compressing them into ONE zip/rar/tar/tar.gz file.
- Your programs will be tested using the provided input samples along with other inputs that are not being provided.
- Look out for specific instructions regarding each problem.

```
Non-weighted undirected graph:
4 6
0 1
0 2
0 3
1 2
1 3
2 3
Weighted undirected graph:
4 6
0 1 10
0 2 1
0 3 3
1 2 12
1 3 5
2 3 18
Non-weighted directed graph:
4 8
0 2
1 0
1 2
1 3
2 0
2 1
2 3
3 2
Weighted directed graph:
4 8
0 2 12
1 0 16
1 2 27
1 3 19
2 1 28
2 3 22
3 0 22
3 2 32
```

```
import java.util.Random;

/**
 * Class to generate graph input data.
 *
 * @author Chinmay
 *
 */
public class GraphInputGenerator {

	private static Random rand = new Random();
	// Adjacency matrix
	private int[][] adj;
	// Number of vertices & edges
	private int vertices, edges;
	private boolean isWeighted, isDirected;

	public GraphInputGenerator(int v, int e, boolean isWeighted,
			boolean isDirected) {
		adj = new int[v][v];
		this.vertices = v;
		this.isWeighted = isWeighted;
		this.isDirected = isDirected;
		// Maximum possible edges check
		this.edges = isDirected ? Math.min(e, v * (v - 1)) :
Math.min(e, v
				* (v - 1) / 2);
		fillAdj();
	}

	/**
	 * Fills adjacency matrix randomly.
	 */
	private void fillAdj() {
		int count = 0;
		while (count < edges) {
			int v1 = rand.nextInt(vertices);
			int v2 = v1;
			while (v2 == v1) {
				v2 = rand.nextInt(vertices);
			}
			if (addEdge(v1, v2))
				count += 1;
		}
	}

	/**
	 * Adds a edge between vertices v1 and v2 and makes entry into
the adjacency
	 * matrix accordingly. For non weighted graphs, the edge cost
```

```
is 1. For non
         * directed graphs, the adjacency matrix is updated for both
the vertices.
         *
         */
        private boolean addEdge(int v1, int v2) {
                if (adj[v1][v2] == 0) {
                        int e = 1;
                        if (isWeighted) {
                                e += rand.nextInt(edges * edges / 2);
                        }
                        adj[v1][v2] = e;
                        if (!isDirected)
                                adj[v2][v1] = e;
                        return true;
                }
                return false;
        }

        /**
         * Displays adjacency matrix.
         */
        public void showAdj() {
                for (int[] row : adj) {
                        for (int col : row) {
                                System.out.print(col + " ");
                        }
                        System.out.println();
                }
        }

        /**
         * Prints the graph data. First line contains number of
vertices and number
         * of edges. Following lines show edges in the graph. If the
graph is
         * weighted, the cost is also shown.
         */
        public void genGraphInput() {
                System.out.println(vertices + " " + edges);
                for (int i = 0; i < adj.length; i++) {
                        for (int j = isDirected ? 0 : i; j <
adj.length; j++) {
                                if (adj[i][j] != 0) {
                                        System.out.println(i + " " + j
                                                        +
((isWeighted) ? " " + adj[i][j] : ""));
                                }
                        }
                }
```

```
        }

        public static void main(String[] args) {
                int v = Integer.parseInt(args[0]);
                int e = Integer.parseInt(args[1]);
                System.out.println("Non-weighted undirected graph:");
                GraphInputGenerator g = new GraphInputGenerator(v, e,
false, false);
                g.genGraphInput();

                System.out.println("Weighted undirected graph:");
                g = new GraphInputGenerator(v, e, true, false);
                g.genGraphInput();

                System.out.println("Non-weighted directed graph:");
                g = new GraphInputGenerator(v, e, false, true);
                g.genGraphInput();

                System.out.println("Weighted directed graph:");
                g = new GraphInputGenerator(v, e, true, true);
                g.genGraphInput();

        }
}
```