GRAPHS

1. OBJECTIVES

(a) To review the concept of graphs.
(b) To review different ways of traversing a graph.

2. LABORATORY

This lab will be conducted in the Computing Lab 1 (N4-B1-8) in SCE. This is an individual experiment. You cannot use the classes that implement the **Collection** interface (please refer to http://java.sun.com/docs/books/tutorial/collections/), except for Array class. We are going to implement these classes (like Vector, LinkedList, Stack, TreeSet, etc.) in CPE/CSC105, instead of using them directly from Java.

No make-up is allowed for students who have missed their stipulated lab classes without any acceptable excuse like having a valid leave of absence from the school or on medical leave. The students will be deemed to have failed the particular lab work. In case you have valid reasons to do makeup, you must inform the lecturer in charge. The makeup should be done within the same week, during the lab sessions attended by other groups.

3. EQUIPMENT

Hardware: The PCs running under the LINUX environment in Computing Lab 1.
Software: NetBeans, SUN JAVA compiler (**javac**) and interpreter (**java**). Your programming will only be tested by the lab markers using javac and java on the lab PCs.

4. EXPERIMENT

In this lab, we will implement a system that manages a graph using Java. The system will:

- Read the graph from a text file.
- Traverse the graph in a depth-first manner, using a given node as a starting point.
- Traverse the graph by breadth-first, starting from a given node.
- Compute the number of connected components in a graph.

Specifically, we need to do the following:

(a) Write a method **readFile** that read the graph from a text file **graph.txt** that contains the adjacency lists. The graph file is in the following format:

num_of_nodes
node1_id num_of_adj_nodes adj_node1_id adj_node2_id…
…
noden_id num_of_adj_nodes adj_node1_id adj_node2_id…

The first line of the file contains an integer representing the number of graph nodes. Each of the remaining lines corresponds to a graph node and contains the id of the node, number of its adjacent nodes and a list of adjacent node ids (all are integers). It is assumed that the node id starts with 0 and increases by 1 for the next node in the file.

An example text file and its corresponding graph are shown in Figure 8.1 (notice that not all the graph nodes are connected and there are two connected components in the example). We assume that the input file is always correct and no validation is required. After the graph is read from the file, it should be converted into **adjacency matrix** representation and kept in main memory.

```
6
0 2 1 2
1 2 0 2
2 3 0 1 3
3 1 2
4 1 5
5 1 4
```

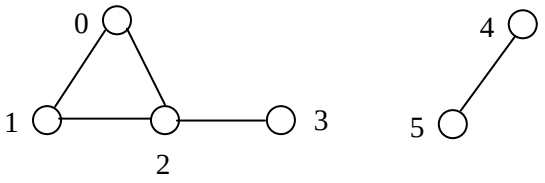

Figure 8.1: A graph and the corresponding text file

(b) Write a method **depthFirst** that traverses the graph in a depth-first manner. depthFirst first reads from the standard input the id of the starting node (an integer). Then it traverses the graph by depth-first and prints the node ids in their traversal order. It is obvious that the id of the starting node will appear at the beginning of the output.

(c) Write a method **breadthFirst** that does similar function to depthFirst, yet traverses the graph in a breadth-first manner.

(d) Write a method **connectedComponents** that finds all the connection components in the graph. connectedComponents outputs the node ids of each connected component in one line. The final line of the output contains an integer, indicating the number of connected components in the graph. The output of connectedComponents for our example graph can be:

```
0 1 2 3
4 5
2
```

(**Hint**: you may consider using depthFirst or breadthFirst method in connectedComponents)

(e) Write a function **menu** that displays all the above methods (as shown below) for test run.

```
(1)   Read data to graph
(2)   Depth first
(3)   Breadth first
(4)   Connected components
(5)   Exit
```

When command 1 is chosen, the program should prompt for 1 file name.

Although parts (b) to (d) are to be implemented as a single method, it may be necessary to create additional sub-method(s) to handle portions of the method. This is especially true if the original method is too long or contains more than one functionally related group of statements. When you write a method, remember that this method is to work for all possible inputs. Not on just your test inputs. You must test for all conditions that might possibly arise; print out error messages as needed.

A text file **graph.txt** has been created in the directory **/home/staff1/csc105/lab8/** to assist the development and testing of your program. Its content will be changed (while still adhering to the same data format) during grading in order to test your programs' robustness. The objective is to ensure that you do not hardcode your program to work exclusively for the given sample customer records. Check also the **readme** file for any last minute hints or changes.

Create a sub-directory called **cpe105/lab8** or **csc105/lab8** (depending on whether you are taking CPE105 or CSC105) in your home directory. Please note that the grader will only look into this sub-directory to find, compile and test run your program. You should provide a readme file that gives instructions to compile and run your program. You should not have irrelevant source files in the directory.

5. ERROR HANDLING

Your can assume that the input is always correct; thus no input validation is required.

6. REPORT

(a) Please check the posted final due date on Edventure for this lab.

(b) For the report, submit hard copies of the readme file, program listing and **script** file(s) recording results of the testing of your program.

(c) Not every lab report will be graded. You will be notified only after all the hard copies mentioned in part (b) have been received.

(d) The grading of your work will be based on the criteria listed in lab1 (c) and (d).

7. ACADEMIC HONESTY AND COLLABORATION

Cooperation is recommended in understanding various concepts and system features. But the actual solution of the assignments, the programming and debugging must be your individual work, except for what you specifically credit to other sources. (Your grade will be based on your own contribution.) For example, copying without attribution any part of someone else's program is plagiarism, even if you modify it and even if the source is a textbook. You can document the credit to other sources at the start of your program code listing. The University takes acts of cheating and plagiarism very seriously: first time violators may fail the coursework component of CPE105 / CSC105. Any wholly (or partly) copied (or being copied) programs will receive zero mark.

8. REFERENCES

[1] Reference book for CSC105.

[2] http://java.sun.com/docs/index.html; http://java.sun.com/j2se/