National Institute of Technology Calicut
Department of Computer Science and Engineering
Third Semester B. Tech.(CSE)-Monsoon 2024
CS2091E Data Structures Laboratory
Assignment 5

Submission deadline (on or before): 11:59 PM, 09/10/2024

Policies for Submission and Evaluation:

- You must submit all the solutions of this assignment following the below-mentioned guidelines in the Eduserver course page, on or before the submission deadline.
- Ensure that your programs will compile and execute using GCC compiler without errors. The programs should be compiled and executed in the SSL/NSL.
- During the evaluation, failure to execute programs without compilation errors may lead to zero marks for that evaluation.
- Your submission will also be tested for plagiarism, by automated tools. In case your code fails to pass the test, you will be straightaway awarded zero marks for this assignment and considered by the examiner for awarding an F grade in the course. Detection of ANY malpractice related to the lab course can lead to awarding an F grade in the course.

Naming Conventions for Submission

- Submit a single ZIP (.zip) file (do not submit in any other archived formats like .rar, .tar, .gz). The name of this file must be

  `ASSG<NUMBER>_<ROLLNO>_<FIRST-NAME>.zip`

  (Example: *ASSG1_BxxyyyyCS_LAXMAN.zip*). DO NOT add any other files (like temporary files, input files, etc.) except your source code, into the zip archive.
- The source codes must be named as

  `ASSG<NUMBER>_<ROLLNO>_<FIRST-NAME>_<PROGRAM-NUMBER>.c`

  (For example *ASSG1_BxxyyyyCS_LAXMAN_1.c*). If you do not conform to the above naming conventions, your submission might not be recognized by our automated tools and hence will lead to a score of 0 marks for the submission. So, make sure that you follow the naming conventions.

Standard of Conduct

- Violation of academic integrity will be severely penalized. Each student is expected to adhere to high standards of ethical conduct, especially those related to cheating and plagiarism. Any submitted work MUST BE an individual effort. Any academic dishonesty will result in zero marks in the corresponding exam or evaluation and will be reported to the department council for record keeping and for permission to assign an F grade in the course. The department policy on academic integrity can be found at: https://minerva.nitc.ac.in/?q=node/650.

Questions

1. You are given an adjacency matrix representation of an undirected simple graph G with n vertices and m edges, where $n \geq 1$ and $m \geq 0$. An undirected simple graph does not contain self-loops or parallel edges between any two vertices.
   Implement a menu-driven program that performs the following operations using Depth-First Search (DFS) as a subroutine:

   Operations

   (a) *noOfConnectedComponents(G)*: Calculates and print the total number of connected components in the graph G. A connected component is a maximal subgraph where any two vertices are connected by a simple path.

   (b) *sizeOfComponents(G)*: Prints the sizes of all connected components in the graph G, sorted in non-decreasing order of their sizes. The size of a component is the number of vertices it contains.

   (c) *noOfBridges(G)*: Prints the total number of bridges in the graph G. A bridge (or cut-edge) is an edge whose removal increases the number of connected components in the graph. If there are no bridges in the graph, the function should print 0.

   (d) *noOfArticulationPoints(G)*: Prints the total number of articulation points in the graph G. An articulation point (or cut-vertex) is a vertex whose removal increases the number of connected components. If there are no articulation points, print 0.

   Notes

   i. A graph with one vertex will always have one connected component and no bridges or articulation points.

   ii. An empty graph with n vertices and 0 edges will result in n connected components.

   iii. A fully connected graph has 1 component and no bridges or articulation points.

   Input Format

   - The first line contains an integer n (number of vertices, $n \geq 1$).
   - The next n lines each contain a space separated n integers representing the adjacency matrix of the graph. The vertices are numbered from 1 to n. A value of 1 at position *matrix[i][j]* means there is an edge between vertex i and vertex j, and a value of 0 means there is no edge. Note that the adjacency matrix is symmetric for undirected simple graph.
   - Each subsequent line contains a character from {'a', 'b', 'c', 'd', 'x'}.
   - Character 'a' calls the function *noOfConnectedComponents(G)*.
   - Character 'b' calls the function *sizeOfComponents(G)*.
   - Character 'c' calls the function *noOfBridges(G)*.
   - Character 'd' calls the function *noOfArticulationPoints(G)*.
   - Character 'x' terminates the program.

   Output Format

   - The output of each command should be printed on a separate line. However, no output is printed for 'x'.
   - For option 'a': Print the total number of connected components in the graph.
   - For option 'b': Print the sizes of all connected components in non decreasing order separated by a space.
   - For option 'c': Print the total number of bridges in the graph. Print 0 if no bridges are present.

- For option 'd': Print the total number of articulation points in the graph. Print 0 if no articulation points are present.

Sample test case 1

Input

```
5
0 1 0 0 0
1 0 0 0 0
0 0 0 1 0
0 0 1 0 1
0 0 0 1 0
a
b
c
d
x
```

Output

```
2
2 3
3
1
```

2. You are given an adjacency matrix representation of an undirected simple graph G with n vertices and m edges, where $n \geq 1$ and $m \geq 0$. An undirected simple graph does not contain self-loops or parallel edges between any two vertices.
Implement a menu-driven program that performs the following operations using Breadth-First Search (BFS) as a subroutine:

Operations

(a) *isBipartite(G)*: Checks whether the given graph G is bipartite or not. A bipartite graph is a graph whose vertices can be partitioned into two disjoint sets so that every edge connects a vertex in one set to a vertex in the other. The function should print 1 if the graph is bipartite, and -1 if it is not.

(b) *hasCycle(G)*: Checks if the graph G contains a cycle. A cycle is a path that starts and ends at the same vertex, with no repeated edges. The function should print 1 if a cycle is present, and -1 if no cycle is present in G.

(c) *isTree(G)*: Checks if the given graph G is a valid tree. A valid tree is a *connected acyclic* graph. The function should print 1 if the graph is a tree, and -1 if it is not. Note that a valid tree must have exactly $n - 1$ edges.

Notes

i. A single vertex with no edges is a bipartite graph and a tree, but it contains no cycles.
ii. A disconnected graph cannot be a tree, but it may still be bipartite or contain cycles.

Input Format

- The first line contains an integer n (number of vertices).
- The next n lines each contain a space separated n integers representing the adjacency matrix of the graph. A value of 1 at position *matrix[i][j]* means there is an edge between vertex i and vertex j, and a value of 0 means there is no edge. The matrix is symmetric for undirected graph.

- Each subsequent line contains a character from {'a', 'b', 'c', 'x'}.
- Character 'a' calls the function *isBipartite(G)*.
- Character 'b' calls the function *hasCycle(G)*.
- Character 'c' calls the function *isTree(G)*.
- Character 'x' terminates the program.

Output Format

- The output of each command should be printed on a separate line. However, no output is printed for 'x'.
- For option 'a': Print 1 if the graph is bipartite, and -1 if it is not.
- For option 'b': Print 1 if the graph contains a cycle, and -1 if G does not contain a cycle.
- For option 'c': Print 1 if the graph is a tree, and -1 if it is not.

Sample test case 1

Input

```
4
0 1 1 0
1 0 1 0
1 1 0 1
0 0 1 0
a
b
c
x
```

Output

```
-1
1
-1
```

3. You are given a directed simple graph G with n vertices and m edges. A graph is represented by an adjacency list, where $n \geq 1$ and $m \geq 0$.
Implement a menu-driven program that performs the following operations using Depth-First Search (DFS) as a sub-routine:

Operations

(a) *isDAG(G)*: Checks whether a topological sort is possible for the given directed acyclic graph G. A topological sort is a linear ordering of the vertices in a Directed Acyclic Graph (DAG) such that for every directed edge $e_1 \rightarrow e_2$, vertex e_1 appears before e_2 in the ordering. Topological sorting is possible only if the graph is a Directed Acyclic Graph (DAG). The function should print 1 if topological sort is possible, and -1 if it is not.

(b) *countStronglyConnectedComponents(G)*: Counts the number of strongly connected components in the graph. A strongly connected component is a maximal subgraph in which every vertex is reachable from every other vertex. The function should print the number of strongly connected components in the graph.

Notes

i. An empty graph with n vertices and 0 edges will results n number of strongly connected components.

ii. For a single vertex with no edges, topological order is possible and the graph has one strongly connected component.

Input Format

- The first line contains two integers n (number of vertices, $n \geq 1$) and m (number of edges, $m \geq 0$).
- The subsequent n lines contain the label of the respective node, followed by a space-separated nodes adjacent to it in ascending order of their labels.
- Each subsequent line contains a character from {'a', 'b', 'x'}.
- Character 'a' calls the function *isDAG(G)*.
- Character 'b' calls the function *countStronglyConnectedComponents(G)*.
- Character 'x' terminates the program.

Output Format

- The output of each command should be printed on a separate line. However, no output is printed for 'x'.
- For option 'a': Print 1 if topological sort is possible, and -1 if if it is not.
- For option 'b': Print the total number of strongly connected components in the graph.

Sample test case 1

Input

```
6 7
1 2 3
2 4
3 4 5
4 6
5 6
6
a
b
x
```

Output

```
1
6
```