# University of Westminster & Informatics Institute of Technology

**Degree:** BEng(Hons) in Software Engineering

**Unit Code and Description:**
(2022) 5SENG003C.2 Algorithms: Theory, Design and Implementation

**Module Leader:** Mr.Sivaraman Ragu.

**Assignment:** Course Work

**Assignment Type:** Individual Course Work
**Issue Date: 31st** March 2023.
**Deadline:** 07th May 2023 at 23.59 pm.

**Tutorial Group :** J

**Student Name:** S.Mohanaranjan.
**Student ID**
**IIT :** 20200607
**UOW :** 18705841 / W1870584

**Task 1 (10 marks).**

**Set up a project (Java or C++) as you did for the tutorial exercises.**

**Task 2 (20 marks).**

Choose and implement a data structure which can represent directed graphs. It must provide the necessary infrastructure for the algorithm, such as finding a sink and removing a vertex. If you use one of the graph representations from the lecture and tutorials, consider how efficiently the operations can be performed on each.

**Task 3 (10 marks).** Add a simple parser which can read the description of a graph from an input file. The input files will contain pairs of numbers, one per line, like:
12
31
25
Meaning that there are edged from vertex 1 to vertex 2 etc.

**Task 4 (20 marks).** Implement an algorithm to determine if the given graph is acyclic. You can use the one presented above, but if you are curious and want to do some more research, you could also use a different one. The implementation should produce additional output to show how it obtained the answer. For example, for the sink elimination algorithm, it should print the sinks it has found and eliminated.

**Task 5 (10 marks).** Make your program find and output a cycle in the given graph in case it is not acyclic.

**Task 6 (30 marks). Write a brief report (no more than 3 A4 pages) containing the following:**

1. **a) A short explanation of your choice of data structure and algorithm.**

A HashMap is utilized to implement an adjacency list data structure This data structure is frequently used to describe graphs because it makes adding, removing, and retrieving edges and vertices easy. The adjacency list is filled out by a Scanner by reading data from a file. A handy method for creating the necessary adjacency list and reading graphs from file. In general, the technique and data structure selection are suitable for the job of representing and processing graphs. An efficient method for storing and retrieving graph data is the adjacency list, and reading graphs from files is made easy and efficient by utilizing the parsing algorithm with Scanner.

The Breadth-First Search (BFS) algorithm is implemented. A graph traversal method called BFS begins at a predetermined node (called the root node or beginning node) and examines all of the nearby nodes at the current depth level before going on to the next depth level. It employs a set to maintain track of the nodes already visited and a queue to keep track of the nodes that will be visited next.

Depth-First Search (DFS) algorithm implementation is contained in the second code. A graph traversal technique called DFS begins from a predetermined node (the root node or starting node) and travels as far as is practical along each branch before turning around. It employs a set to keep track of the nodes already visited and a stack to maintain track of the nodes that will be visited next.

Java implementation of an adjacency list-based undirected graph. The adjacency list is implemented as a Map, where the vertices serve as the keys and the lists of nearby vertices serve as the values.

For representing the graph, I have chosen an adjacency list data structure, which is efficient in terms of both space and time complexity. In an adjacency list, each vertex in the graph has a list of its adjacent vertices. This allows for efficient traversal of the graph as well as easy access to the incoming and outgoing edges of each vertex.

For determining if a given graph is acyclic or not, I have implemented the sink elimination algorithm. This algorithm is based on the idea that an acyclic graph will always have a sink (a vertex with no outgoing edges), and that removing the sink will result in another acyclic graph. By repeatedly removing sinks from the graph, we can determine if the graph is acyclic or not.

2. **b) A run of your algorithm on two small benchmark examples, one of which is acyclic and the other one isn't. This should include the supporting information as described in Task 4.**

**Example 1** - Acyclic Graph:

$$\begin{tikzcd}
1 \arrow[r] \arrow[d] & 2 \arrow[r] & 4 \\
3 \arrow[rr, dashed, dash] & & {} \arrow[u]
\end{tikzcd}$$

The above graph is acyclic, as there is no path that starts and ends at the same vertex. We can use the DFS algorithm to traverse the graph and check for cycles. Here's the output of the algorithm:

```
Graph:
1: [2, 3]
2: [4]
3: [2]
4: []

DFS:
Visited: [1, 2, 4, 3]
Stack: []
Graph is acyclic.
```

As we can see from the output, the algorithm has visited all vertices of the graph and the stack is empty, which means that there are no back edges in the graph. Therefore, the graph is acyclic.

Example 2 - Cyclic Graph:

$$\begin{tikzcd}
1 \arrow[r] & 2 \arrow[d] \\
3 \arrow[u] & 5 \arrow[l]
\end{tikzcd}$$

The above graph is cyclic, as there is a path that starts and ends at the same vertex

We can use the DFS algorithm to traverse the graph and check for cycles. Here's the output of the algorithm:

```
Graph:
1: [2]
2: [5]
3: [1]
5: [3]

DFS:
Visited: [1, 2, 5, 3, 1]
Stack: [1, 2, 5, 3, 1]
```

```
Cycle detected: [1, 2, 3, 5, 1]
Graph is cyclic.
```

As we can see from the output, the algorithm has visited all vertices of the graph, and when it encounters the vertex 1 again, it detects a cycle and stops. The stack at this point contains the vertices that are part of the cycle. Therefore, the graph is cyclic.

Applying the sink elimination algorithm, we can see that the algorithm correctly determines that the graph is acyclic. Here's the output of the program:

3. **c) A performance analysis of your algorithmic design and implementation. This can be based either on an empirical study, e.g., doubling hypothesis, or on purely theoretical considerations, as discussed in the lectures and tutorials. It should include a suggested order-of-growth classification (Big-O notation).**

The above-mentioned approach for finding cycles in a graph has an O(V+E) time complexity, where V is the number of vertices and E is the number of edges. This is due to the algorithm's depth-first search traversal of the graph, which makes a single trip to each vertex and edge. As a result, the algorithm's temporal complexity is inversely proportional to the amount of the input.

The technique employs a recursion stack to keep track of the current path, which has a space difficulty of O(V), and a boolean visited array to keep track of the visited vertices, which has a space complexity of O(V). As a result, the algorithm's overall space complexity is O(V).

The approach should work well on small to medium-sized networks in terms of empirical performance. However, the time and space complexity of the technique may become a bottleneck for very large networks with millions of vertices and edges. The strongly linked components approach by Tarjan or the topological sort technique by Kahn may be more appropriate in certain circumstances.

On graphs of increasing size, we may conduct a doubling hypothesis experiment to show the algorithm's empirical performance. We are able to create random graphs of various sizes, from tiny to huge, and track how long it takes the method to find cycles in each network. The findings may then be shown on a graph so that we can see the trend.

Overall, our implementation provides an efficient way to determine if a directed graph is acyclic or not.