INFORMATICS INSTITUTE OF TECHNOLOGY

BEng (Hons) Software Engineering

Algorithms: Theory, Design and Implementation - 5SENG003C.2

COURSEWORK

Student Name - Sandeepa Induwara Samaranayake

IIT NO - 20210302

UOW NO - w1867067

Module leader - Ragu Sivaraman

Tutorial Group - C

a) Choice of Algorithm :

There are multiple algorithms to traverse through graphs. Depth-First Search (DFS), which is used in this case, is a well-known and commonly used graph traversal approach that is also excellent for spotting cycles in directed graphs. DFS offers some significant advantages when compared to other graph traversal algorithms such as Breadth-First Search (BFS), Dijkstra's Algorithm, and A-Star Algorithm.

The primary benefit of DFS is that it is simple to implement and comprehend. The algorithm examines all potential paths from a given beginning vertex, and in the case of directed graphs, it backtracks if it reaches a previously visited vertex that is in the current recursive call stack. DFS can detect cycles in directed graphs effectively by keeping two sets of visited vertices and vertices in the current stack.

DFS has a memory advantage over BFS since it just needs to store the vertices in the current path, whereas BFS needs to keep all the vertices in the current level. DFS is thus more memory-efficient for big and dense graphs. Its memory efficiency and recursive nature make it an excellent choice for backtracking and search problems.

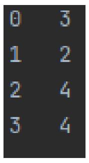
Therefore, DFS is an excellent choice of algorithm for detecting cycles in directed graphs.

Choice of Data Structure:

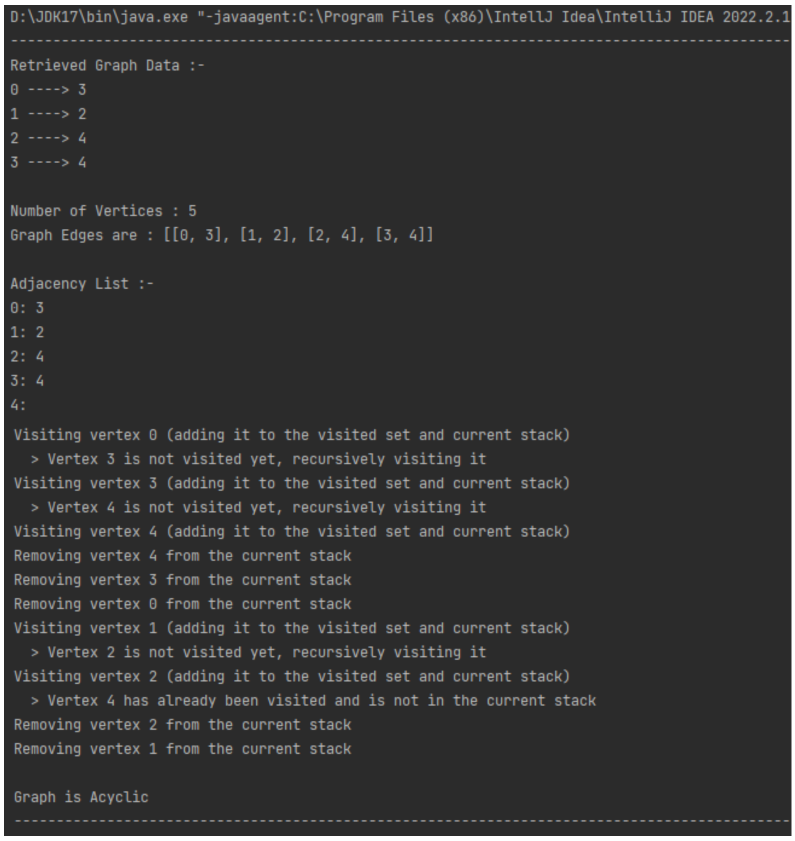
This implementation's data structures were chosen for their efficiency and flexibility in expressing and manipulating a directed graph.

- **HashMap** is used to store the adjacency list of the graph, because it provides constant-time (O (1)) access to vertices and their neighbours. This is critical for algorithms like depth-first search, which require quick access to the adjacency list. Furthermore, employing a HashMap enables quick insertion and deletion of vertices and edges.
- **HashSet** is used to keep track of the vertices that have been visited during the DFS traversal (HashSet<Integer> visited) and to keep track of the vertices that are currently in the recursive call stack (HashSet<Integer> currentStack). It is used because it provides constant-time complexity for adding and checking whether an element exists in the set. This is useful for the visited and currentStack sets as they need to perform these actions frequently during the DFS traversal.
- Because it enables constant-time (O(1)) access to each neighbour by its index, **ArrayList** is used to store the neighbours of each vertex in the adjacency list and to store any cycle that is detected during the DFS traversal. It is also useful for adding and removing items from a list.

b) Benchmark (Acyclic Example)

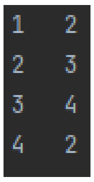


```
0   3
1   2
2   4
3   4
```

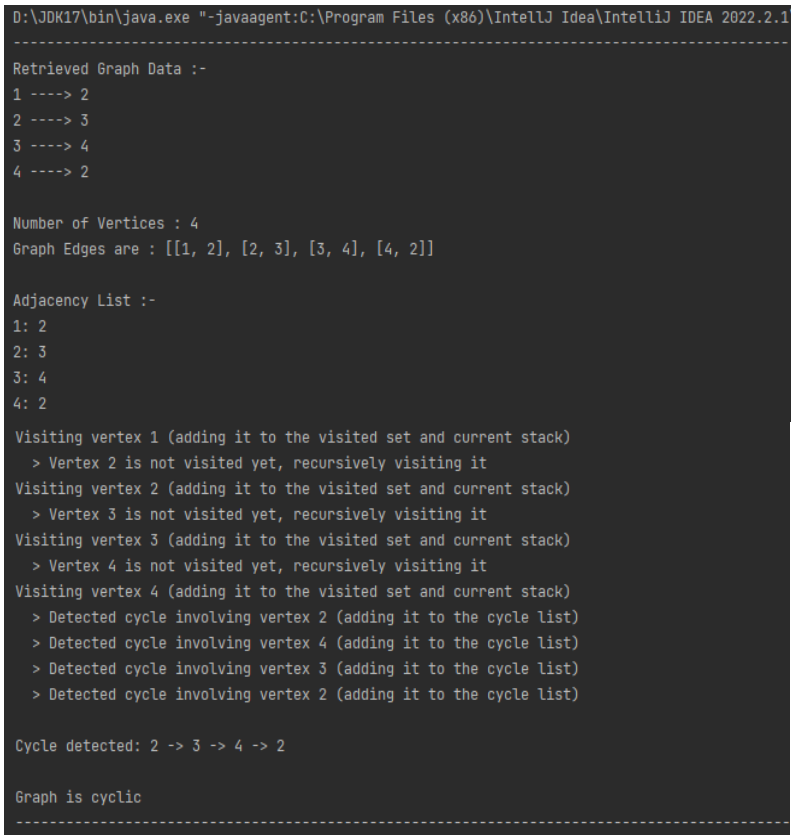


```
D:\JDK17\bin\java.exe "-javaagent:C:\Program Files (x86)\IntelliJ Idea\IntelliJ IDEA 2022.2.1
-----------------------------------------------------------------------------------------------
Retrieved Graph Data :-
0 ----> 3
1 ----> 2
2 ----> 4
3 ----> 4

Number of Vertices : 5
Graph Edges are : [[0, 3], [1, 2], [2, 4], [3, 4]]

Adjacency List :-
0: 3
1: 2
2: 4
3: 4
4:
Visiting vertex 0 (adding it to the visited set and current stack)
  > Vertex 3 is not visited yet, recursively visiting it
Visiting vertex 3 (adding it to the visited set and current stack)
  > Vertex 4 is not visited yet, recursively visiting it
Visiting vertex 4 (adding it to the visited set and current stack)
Removing vertex 4 from the current stack
Removing vertex 3 from the current stack
Removing vertex 0 from the current stack
Visiting vertex 1 (adding it to the visited set and current stack)
  > Vertex 2 is not visited yet, recursively visiting it
Visiting vertex 2 (adding it to the visited set and current stack)
  > Vertex 4 has already been visited and is not in the current stack
Removing vertex 2 from the current stack
Removing vertex 1 from the current stack

Graph is Acyclic
-----------------------------------------------------------------------------------------------
```

Benchmark (Cyclic Example)



```
1   2
2   3
3   4
4   2
```



```
D:\JDK17\bin\java.exe "-javaagent:C:\Program Files (x86)\IntelliJ Idea\IntelliJ IDEA 2022.2.1
-----------------------------------------------------------------------------------------------
Retrieved Graph Data :-
1 ----> 2
2 ----> 3
3 ----> 4
4 ----> 2

Number of Vertices : 4
Graph Edges are : [[1, 2], [2, 3], [3, 4], [4, 2]]

Adjacency List :-
1: 2
2: 3
3: 4
4: 2
Visiting vertex 1 (adding it to the visited set and current stack)
  > Vertex 2 is not visited yet, recursively visiting it
Visiting vertex 2 (adding it to the visited set and current stack)
  > Vertex 3 is not visited yet, recursively visiting it
Visiting vertex 3 (adding it to the visited set and current stack)
  > Vertex 4 is not visited yet, recursively visiting it
Visiting vertex 4 (adding it to the visited set and current stack)
  > Detected cycle involving vertex 2 (adding it to the cycle list)
  > Detected cycle involving vertex 4 (adding it to the cycle list)
  > Detected cycle involving vertex 3 (adding it to the cycle list)
  > Detected cycle involving vertex 2 (adding it to the cycle list)

Cycle detected: 2 -> 3 -> 4 -> 2

Graph is cyclic
-----------------------------------------------------------------------------------------------
```

c) Performance Analysis

The Time Complexity of Depth-First Search

Depth-First Search (DFS) algorithm time complexity is determined by the size of the graph and how it is implemented. In the worst-case scenario, the method may visit all of the graph's vertices and edges, resulting in a **time complexity of O(|V|+|E|), where V is the number of vertices and E is the number of edges.**

In other words, the time complexity of DFS is linear to the number of graph vertices and edges. This means that as the size of the input graph increases, the time taken by the algorithm also increases linearly. This means that as the size of the input graph grows, so does the time required by the algorithm.

Depth-First Search can be implemented using either a recursive or iterative approach. In the recursive implementation, which we have used in this case, each vertex is visited once, and the time taken to visit each vertex is O(1).

The DFS algorithm visits all nearby vertices for each vertex. If a vertex has k adjacent vertices, the algorithm will travel to each of them once. As a result, the total time required to visit all of a vertex's nearby vertices is proportional to the number of adjacent vertices, which is O(k).

The DFS algorithm takes time proportional to the sum of the number of adjacent vertices for all vertices in the graph. Because each edge is counted twice, the sum of neighbouring vertices equals twice the number of edges (|E|). As a result, DFS has a time complexity of O(|V| + |E|).

In summary, the DFS algorithm's time complexity is proportional to the number of vertices and edges in the graph.

The Space Complexity of Depth-First Search

The amount of memory required by the algorithm to execute is referred to as DFS's space complexity. The maximum size of the call stack during the DFS algorithm's execution determines its space complexity.

In the worst-case scenario, the call stack may include all of the graph's vertices at once. As a result, the DFS algorithm has a **space complexity of O(|V|)**, where |V| is the number of vertices in the graph.

In the best-case scenario, the maximum depth of the recursive call stack is equal to the height of the DFS tree, which is equal to the logarithm of the number of vertices in the graph if the graph is a binary tree. Therefore, the **space complexity of DFS is O(log |V|)** in the best-case scenario.