UNIVERSITY OF WESTMINSTER

INFORMATICS INSTITUTE OF TECHNOLOGY

BEng (Hons) Software Engineering

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

COURSEWORK REPORT

Name: - Srinayaka Palliyaguru Ge Kalindu Lokith

UOW number: - w1867088

IIT number: - 20210332

Level: - L5

Choice of Algorithm:-

I. Among the many possible algorithms we apply problem described in the coursework, I select the Depth-First Search (DFS) algorithm as the type of algorithm for the problem's solution.

There are several algorithms for navigating graphs. Depth-First Search (DFS), a well-known and often-used graph traversal method that is excellent at spotting cycles in directed graphs, is used to tackle this problem. DFS has a variety of noteworthy advantages and disadvantages when compared to other graph traversal algorithms including Breadth-First Search (BFS), Dijkstra's Algorithm, and A* Algorithm.

There are several algorithms for navigating graphsDFS.it has ease of implementation and comprehension is its main advantage. The algorithm considers all possible routes from a given starting vertex, and in the case of directed graphs, it retraced its steps if it came across a previously visited vertex that was still on the stack of recursive calls. Keeping two sets of visited vertices and vertices in the current stack allows DFS to detect cycles in directed graphs effectively.

There are several algorithms for navigating graphsDFS.it has ease the biggest benefit of DFS is how simple it is to use and understand. In directed graphs, the algorithm considers all potential paths from a given beginning vertex and reverses direction if it comes across a previously visited vertex that is now on the stack of recursive calls. DFS is able to identify cycles in directed graphs by keeping two sets of visited vertices and vertices in the current stack.

As a result, DFS is an excellent approach for detecting cycles in directed graphs.

Choice of Data Structure:-

The efficiency and adaptability of the data structures utilized in this implementation were chosen for their suitability for expressing and interacting with a directed graph.

The graph's adjacency list is kept in HashMap because it enables constant-time (O (1)) access to vertices and their neighbors. This is essential for algorithms like depth-first search that require speedy access to the adjacency list.

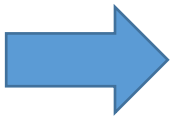
- It is also easy to add and remove vertices and edges when using a HashMap.
- In addition to maintaining track of the vertices that are now in the stack of recursive calls (HashSetInteger> currentStack), HashSet is also used to keep track of the vertices that have been traversed throughout the DFS traversal. It is used because it provides constant-time complexity while adding and checking if an element is present in the set. During the DFS traversal, the visited and currentStack sets frequently need to perform these actions, therefore this is beneficial to them.

The ArrayList provides constant-time (0(1)) access to every neighbor by its index, it is utilized to store the neighbors of each vertex in the adjacency list as well as any cycles that are found during the DFS traversal. It is also useful for adding and removing things from lists.

Acyclic Benchmark Examples :-

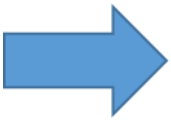
The Time Complexity of Depth-First Search

6	7
7	8
8	9
9	10



```
"C:\Users\Lokith Srinayaka\.jdks\corretto-18.0.2\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA 2
--------------------------------------------------------------------------------------------
Retrieved Graph Data :-
6 ----> 7
7 ----> 8
8 ----> 9
9 ----> 10

Amount of Vertices : 5
Graph Edges are : [[6, 7], [7, 8], [8, 9], [9, 10]]

Adjacency List :
6: 7
7: 8
8: 9
9: 10
10:

Visiting vertex 6 (adding it to the visited set and current stack)
  > Vertex 7 is not visited yet, recursively visiting it
Visiting vertex 7 (adding it to the visited set and current stack)
  > Vertex 8 is not visited yet, recursively visiting it
Visiting vertex 8 (adding it to the visited set and current stack)
  > Vertex 9 is not visited yet, recursively visiting it
Visiting vertex 9 (adding it to the visited set and current stack)
  > Vertex 10 is not visited yet, recursively visiting it
Visiting vertex 10 (adding it to the visited set and current stack)
Removing vertex 10 from the current stack
Removing vertex 9 from the current stack
Removing vertex 8 from the current stack
Removing vertex 7 from the current stack
Removing vertex 6 from the current stack

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

Process finished with exit code 0
```

6	7
7	8
8	9
9	1
1	6



```
"C:\Users\Lokith Srinayaka\.jdks\corretto-18.0.2\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\IntelliJ IDEA
--------------------------------------------------------------------------------------------
Retrieved Graph Data :-
6 ----> 7
7 ----> 8
8 ----> 9
9 ----> 1
1 ----> 6

Amount of Vertices : 5
Graph Edges are : [[1, 6], [6, 7], [7, 8], [8, 9], [9, 1]]

Adjacency List :
1: 6
6: 7
7: 8
8: 9
9: 1

Visiting vertex 1 (adding it to the visited set and current stack)
  > Vertex 6 is not visited yet, recursively visiting it
Visiting vertex 6 (adding it to the visited set and current stack)
  > Vertex 7 is not visited yet, recursively visiting it
Visiting vertex 7 (adding it to the visited set and current stack)
  > Vertex 8 is not visited yet, recursively visiting it
Visiting vertex 8 (adding it to the visited set and current stack)
  > Vertex 9 is not visited yet, recursively visiting it
Visiting vertex 9 (adding it to the visited set and current stack)
  > Detected cycle involving vertex 1 (adding it to the cycle list)
  > Detected cycle involving vertex 9 (adding it to the cycle list)
  > Detected cycle involving vertex 8 (adding it to the cycle list)
  > Detected cycle involving vertex 7 (adding it to the cycle list)
  > Detected cycle involving vertex 6 (adding it to the cycle list)

Cycle detected: 1 -> 6 -> 7 -> 8 -> 9 -> 1

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

Process finished with exit code 0
```

The Depth-First Search (DFS) algorithm's temporal complexity varies depending on the graph's size and how it's implemented. In the worst scenario, the approach might traverse each vertex and edge in the graph, with an O(|V|+|E|) time complexity for each vertex and edge visited.

In other words, the number of graph vertices and edges directly correlates with the temporal complexity of DFS. This suggests that the algorithm's execution time increases linearly as the size of the input network increases. This indicates that the time requirements of the algorithm scale directly with the size of the input network.

Either a recursive or an iterative method can be used to implement depth-first search. Each vertex is visited only once in the recursive implementation we used in this case, and the time required to visit each vertex is O(1).

The DFS algorithm visits all neighboring vertices for each vertex. If there are any, the algorithm will once visit each of the k adjacent vertices that make up a vertex. As a result, the time required to visit each neighboring vertex is proportional to the number of vertex neighbors, which is O(k).

The total number of nearby vertices for all graph nodes affects how long the DFS method takes. Since each edge is counted twice, the sum of neighboring vertices is equal to twice the number of edges (|E|). As a result, DFS has an O(|V| + |E|) time complexity.

In conclusion, the number of vertices and edges in the graph has an inverse relationship with the DFS method's temporal complexity.

The Space Complexity of Depth-First Search

The amount of memory used by the algorithm is gauged by the DFS space complexity. The call stack's maximum size affects how space-complex the DFS algorithm is.

In the worst scenario, the call stack might include all of the vertices of the graph at once. Since |V| is the total number of graph vertices, the DFS algorithm has a space complexity of O(|IV|).

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