

**CE2001 Algorithm**

**Example Class 4 Lab Report**

Application of Graph Algorithms

Prepared By:

Xiao Hui Yong

Xu Dan Dan

Yeong Hui Ting

Yuen Kai Him

(Group 6)

Contents

[1. Introduction 2](#_Toc370752679)

[1.1. Undirected Graph 2](#_Toc370752680)

[1.2. Depth-First Search (DFS) 2](#_Toc370752681)

[1.3. Application of DFS Algorithm 2](#_Toc370752682)

[2. DFS Algorithm Source Code 3](#_Toc370752683)

[3. Part 1 – Generate a toy graph 4](#_Toc370752684)

[4. Part 2 – Generate 5 Graphs of Various Size 8](#_Toc370752685)

[5. Conclusion 8](#_Toc370752686)

# Introduction

## Undirected Graph

An undirected graph is a set of objects that are connected together, where all the edges are bidirectional. An undirected graph is also called an undirected network.

## Depth-First Search (DFS)

DFS is a classic recursive method for systematically examining each of the vertices and edges in a graph. The DFS algorithm search through each node exactly once and every edge is traversed once in forward direction (exploring) and once in backward direction (backtracking). An example was showed below:

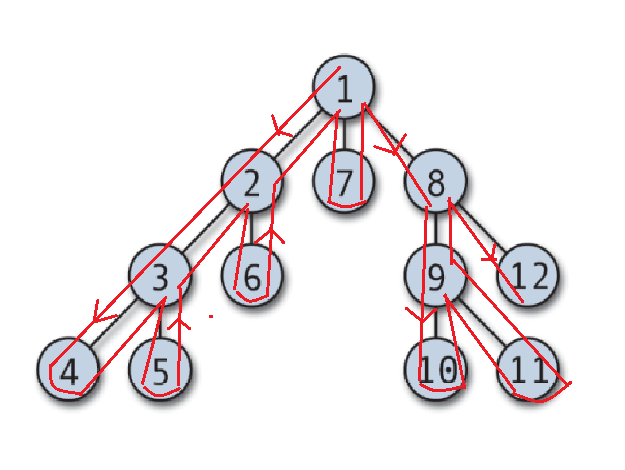


Figure 1: Example of DFS

Output: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12

## Application of DFS Algorithm

DFS algorithm can be uses to solve problems such as cycle detection, bridge and planarity. Cycle detection is to check whether a graph is cyclic if it contains one or more cycles, otherwise it is acyclic. Bridge is an edge whose deletion increases the number of connected components. An edge is a bridge if and only if it is not contained in any cycle. A graph is planar if it can be drawn in the plane such that no edges cross one another by using DFS to determines whether a graph is planar in linear time or not.

# DFS Algorithm Source Code

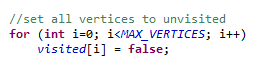


Figure 2: Default Vertices Status

First, we will set the visited status of all vertices to false at the start to indicate the vertices for not been visited. Figure 2 showed the coding of setting all vertices to false.

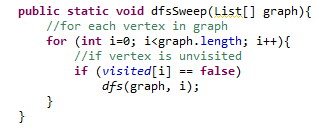


Figure 3: dfsSweep ()

Next, we create a method, dfsSweep(), for checking the status of each vertex. If the vertex’s status is false then it will go to method, dfs(), which is shown in Figure 3. dfsSweep() is needed because we need to check how many connected components is in the graph and thus dfs() needs to be repeated.

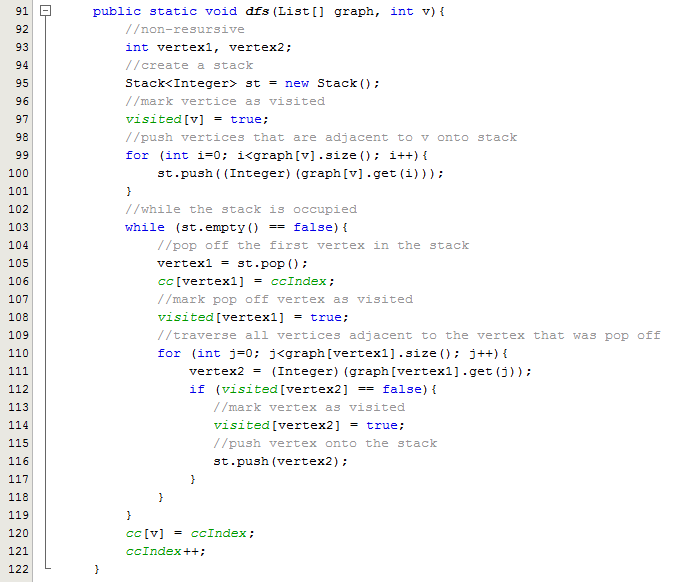


Figure 4: dfs ()

We are using stack and pop for iterative DFS algorithm in the method, dfs(), which is shown in Figure 4. When dfs() is first called, it will push all the adjacent neighbours of Vertex 0 into the stack and Vertex 0 will be marked. As long as the stack is not empty, the program will always pop off the first element in the stack and store it in vertex 1. Vertex1 will be marked as visited and the index of the connected component it is in will be stored in cc[]. The program will enter a for-loop to check the adjacent neighbours of vertex1 and if they have not been visited, it will be pushed onto the stack and marked as visited. The program will repeat this process until it reaches a dead end. For example, it reaches Vertex 4 and it does not have any adjacent neighbours. DFS will then back up from the dead end and continue to check the other adjacent neighbours of Vertex 1. DFS will only end when the stack is empty.

If there are more than 1 connected components, the vertex will enter dfs() again and repeat the same process. An iterative DFS algorithm was used to prevent any overflow of the stack from occurring. Since part 2 requires us to use large values, if recursive method was used to implement DFS algorithm, it may result in an overflow.

# Part 1 – Generate a toy graph

For part 1 of the example class, we are required to generate a toy graph of about 10 vertices represented as an adjacency list in a text file. The program must be able to load the graph from the text file.

First, we initialize the graph array to be an array of linked list which showed in Figure 5.

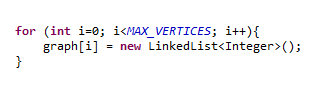
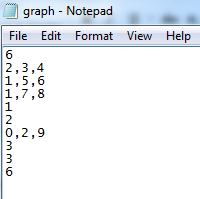


Figure 5: Array of Linked List

Next, we generate a text file with 10 vertices showed in Figure 6 and Figure 7 showed in graph view on how the graph will be look like after tracing through the vertices.



Vertex 0 🡪 6

Vertex 1 🡪 2 🡪 3 🡪 4

Vertex 2 🡪 1 🡪 5 🡪 6

Vertex 3 🡪 1🡪 7 🡪 8

Vertex 4 🡪 1

Vertex 5 🡪 2

Vertex 6 🡪 0 🡪 2 🡪 9

Vertex 7 🡪 3

Vertex 8 🡪 3

Vertex 9 🡪 6

Figure 6: graph.txt

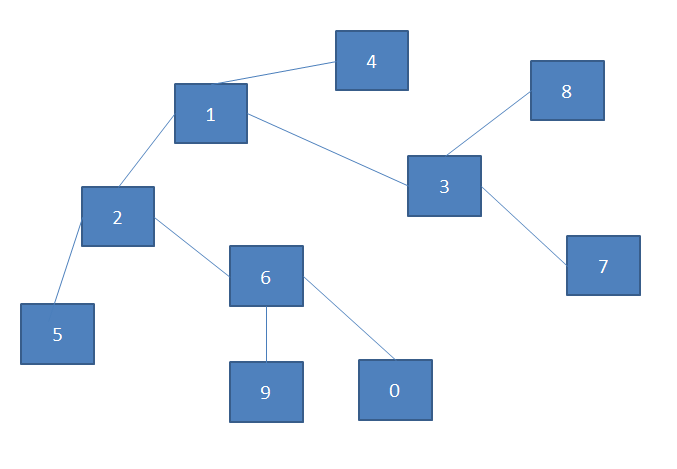


Figure 7: Graph View

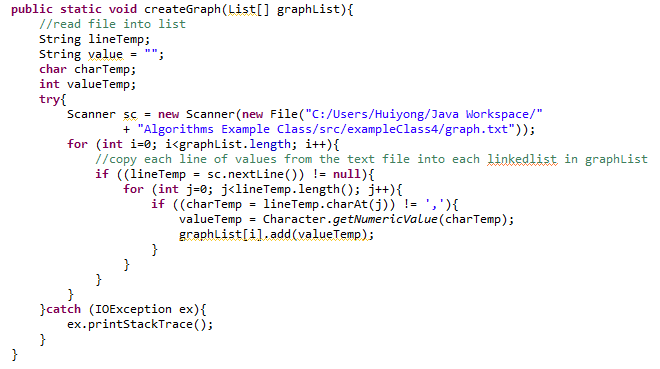


Figure 8: createGraph()

In the method of createGraph() which showed in Figure 8, we will read in the text file and use a for-loop to read through the entire text file. An if-else condition is used to check current line is not empty. If the current line is not empty, the line will be stored as a String in lineTemp and another for-loop will be used to loop through lineTemp and extract only the numbers for each vertex to store into graphList[]. This is done until the line after last number in the last line.

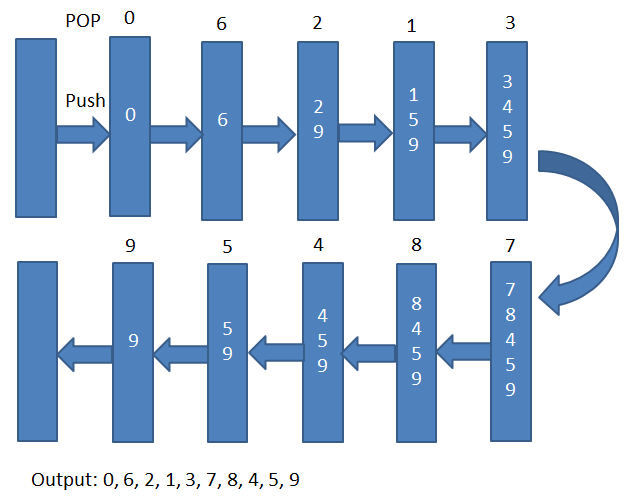


Figure 9: Stack and Pop View

Figure 9 shows the elements at the stage at each stage and which elements are pop off and push onto the stack.

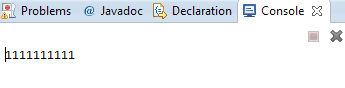


Figure 10: Check Tree

The result for part 1 is illustrated in Figure 10. Since the graph in the text file only had 1 connected component, the output was 1 from Vertex 0 to Vertex 9. This means that all vertices are in connected component 1.

Assuming the graph was changed and designed to have 3 connected components.



Figure 11: Text File for a Graph with 3 connect components

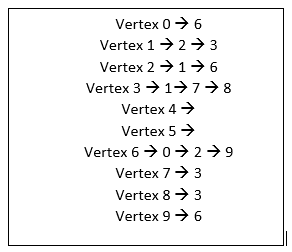


Figure 12: Interpreting textfile in figure 11

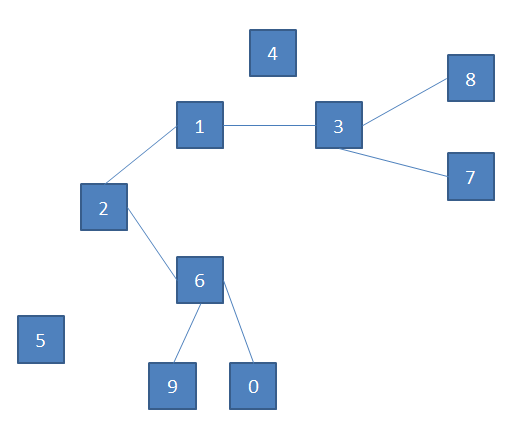


Figure 13: Graphical representation of the graph in text file



Figure 14: Output for a graph with 3 connected components

Since Vertex 4 and 5 are isolated, they both represent a new connected components. This explains the output because all Vertex 0-9 except Vertex 4 and 5 belong to connected component 1, Vertex 4 belongs to connected components 2 and Vertex 5 belongs to connected component 3.

# Part 2 – Generate 5 Graphs of Various Size

### 2.1 Implementation

For part 2 of the example class, we are required to generate 5 graphs of various sizes with number of vertices ranging from 10,000 to 50,000. Measure the CPU times of the program and analyse how the running time of your program depends on m and n.

As of Part 1, we initialize the graph array to be an array of linked list shown in Figure 5 above.

Then we set the initial number of edge to be 8n. After which, we have to run three overlapping for loops, the most inner for loop is used to reset the graph array for every new graph. The next for loop that is overlapped will loop a total of 5times increasing the vertex by 10,000 from 10,000 to 50,000 and the main functions such as creating the graph and running the depth-first search through the graph. Then it’s the most outer for loop that increases the number of edge by 1 for 5times in total. Thus, generating 5 different set of data from 8n to 12n consisting 10,000 to 50,000 vertices in each set.

In Part 2, we will have to create our own graph using the codes below:

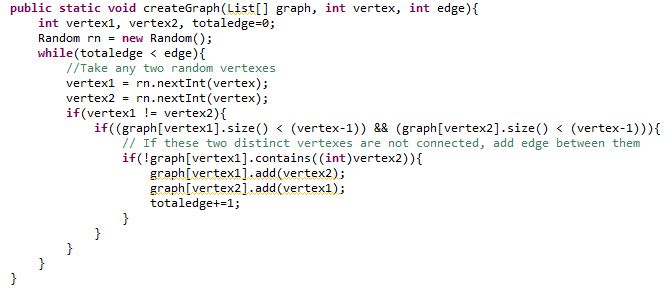


Figure 15: createGraph()

In the method of createGraph() which showed in Figure 15, it will randomly allocate 2 numbers to vertex1 and vertex2 within the range of vertices required for the graph then create an edge between them after checking the following conditions:

1. They are not equal.
2. Vertex1 and vertex2 are not already connected.

Finally, it will increase the total number of edges already present and repeat until the total number of edges present matches the stipulated number of edges.

## Statistics

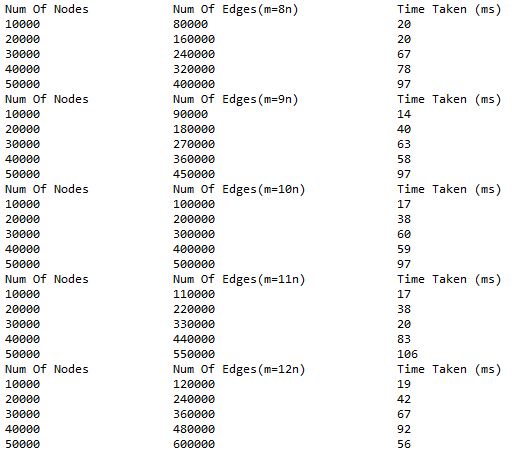


Figure 16: Output for random Generated Graph

By plotting the CPU time taken against number of vertices into a graph

Figure 17: Graphical View of the Generated Data

Through the graph, it can be observed that there is a linear increment of number of nodes and CPU time taken and from the time function at 10n, t = 0.0018n – 0.1, which belongs to time complexity O(n). In calculation, the time complexity of DFS is O(n+m). For example, when m = 8n, the time complexity of DFS will be O(n+8n) = O(9n) = O(n). Therefore the time complexity derived from experimental result is similar to the theoretical results.

# Conclusion

We were able to use a DFS graph algorithm to find connected components inside a graph. In both parts of the example class, our implementation was verified by the generated results derived in the graph. In conclusion, the implementation of the DFS graph algorithm works and the time complexity is O(|V|+|E|) for a graph G = (V,E).