Connectivity algorithm with depth first search (DFS) on simple graphs

Connectivity algorithm with depth first search (DFS) on simple graphs

O Riansanti[1,2], M Ihsan[1,2] and D Suhaimi[1,2]

[1] Department of Mathematics, Syiah Kuala University, Banda Aceh, Indonesia
[2] Multimedia Research Group, Department of Mathematics, Syiah Kuala University, Banda Aceh, Indonesia

E-mail: mahyus@unsyiah.ac.id

Abstract. This paper discusses an algorithm to detect connectivity of a simple graph using Depth First Search (DFS). The DFS implementation in this paper differs than other research, that is, on counting the number of visited vertices. The algorithm obtains *s* from the number of vertices and visits source vertex, following by its adjacent vertices until the last vertex adjacent to the previous source vertex. Any simple graph is connected if *s* equals 0 and disconnected if *s* is greater than 0. The complexity of the algorithm is $O(n^2)$.

1. Introduction

A graph is connected if any two vertices are joined by a sequence of edges [1]. A connected graph can also be defined as a graph when there is a walk between every pair of vertices [1]. A graph is connected if there exists a path between any pair of vertices [2]. A work of literature also defines a connected graph as a graph that is formed by one component [2].

The connectivity of a graph is often used in the various real-life implementation and problem-solving (algorithm) of a graph, such as a bipartite graph, Hamiltonian graph, Eulerian graph, tree, minimum spanning tree (MST), some shortest path problems (Dijkstra's algorithm), graph coloring, and planar graph. In the real world, the implementation of a connected graph solves problems in telecommunication [3], blood flow [4], computer network [5], and transportation [6]. Determining the connectivity of a graph is so important because of its various benefits.

The connectivity of a graph can be determined using methods or connectivity algorithms. One of the connectivity algorithms is the positive non-diagonal entry of matrix [7]. Other methods are Modified Prüfer's Algorithm Labelling Trees [8], the similarity of virtual vertex set and vertex set [9].

In this research, the algorithm uses the concept of Depth First Search (DFS), which is a well-known concept in computer science. The difference of this algorithm from other algorithms is the way of counting vertices that have been visited [10, 11].

2. Methodology

The connectivity algorithm presented in this article takes input any simple graf G = (V, E). Afterward, the status of each vertex is set to false and is contained in variable Pi, where $1 \leq i \leq |V|$, which means every vertex is an unvisited vertex. Variables are then initialized to |V|, which means no vertex is visited yet.

The process of checking connectivity is presented in a recursive function called Visit (source vertex). Let vertex r be source vertex, then the status of vertex r is set to true, which means that vertex r is visited. The algorithm then subtracts 1 from s. Afterward, the algorithm will search for the vertex with the smallest label that is adjacent to r, if there exists such vertex, the recursive function will run on that vertex and then subtract 1 from s. This mechanism follows DFS method. The process will be repeated in the same way as calling the recursive function, but on different source vertex, that is an adjacent vertex to r with next label. The recursive process will stop if there is no adjacent vertex with status false. The last phase is checking the variable s, if s = 0 then the graph is connected and if s > 0 then the graph is disconnected.

3. Results and Discussion

The result of this research shows that any simple graph can be checked its connectivity using connectivity algorithm with DFS. The algorithm shows that if $s = 0$, then the graph is connected, and if $s > 0$, then the graph is disconnected. In general, the algorithm to determine the connectivity of any simple graph is shown as follows.

```
1.  Make_Simple_Graph()
2.  for(i = 1; i <= |V|; i++)
3.  P[i] = 0
4.  s = |V|
5.  Initial VertexStart
6.  Visit(VertexStart)
7.  if(s == 0)
8.  Output "Connected graph"
9.  Else
10. Output "Disconnected graph"
```

The algorithm consists of three major parts. The first part is inputting any simple graph, initializing the status of each vertex and initializing variable s, and selecting the source vertex. The second part is the process of DFS, which is represented by a recursive function. In general, the second part works as follows.

```
1.  void Visit(int r)
2.  int j
3.  P[r] = 1
4.  s--
5.  for(j = 1; j <=|V|; j++)
6.  if(A[r][j] == 1)
7.  v = j
8.  if(P[v] != 1)
9.  Visit (v)
```

After selecting the source vertex, the algorithm will search for an adjacent vertex of source vertex. This algorithm corresponds to Depth First Search (DFS). After executing the recursive function, the value of variable s will be changed. The third part of the algorithm is checking the value of variable s. If s = 0, then the graph is connected and if s > 0 then the graph is disconnected. The algorithm running on simple graph G = (V, E), where V = {1, 2, 3, 4, 5, 6, 7} and E = {12, 15, 17, 23, 34, 35, 67} is shown in Figure 1 to Figure 4.

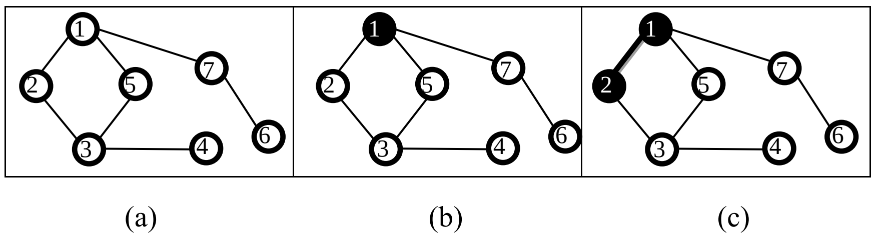

Figure 1. (a) Inputting simple graph (b) Find source vertex and (c) visited vertex adjacency.

The first step is inputting a simple graph. The graph with $|V| = 7$ is shown in Figure 1 (a). The algorithm then initializes $s = |V|$ where s is a variable to determine the connectivity and for each visited vertex, the algorithm subtracts 1 from s. The algorithm then initializes $P_i = 0$ for each unvisited vertex where $0 \leq i < n$ and $P_i = 1$ for each visited vertex where $0 \leq i < n$. So, $|v| = p_1, p_2, p_3, p_4, p_5, p_6, p_7$ and give a status for each vertices.

As shown in Figure 1 (b), vertex 1 is selected to be the source vertex. Afterward, $s = s - 1$ for each visited vertex so s = 6. Since vertex 1 is a visited vertex, the algorithm sets $P_1 = 1$. The candidates for the adjacent vertex of source vertex are vertex 2, vertex 5, and vertex 7. The algorithm chooses the adjacent vertex with the smallest label. Vertex 2 is therefore chosen to visit. Thus, $s = 5$ and $P_2 = 1$.

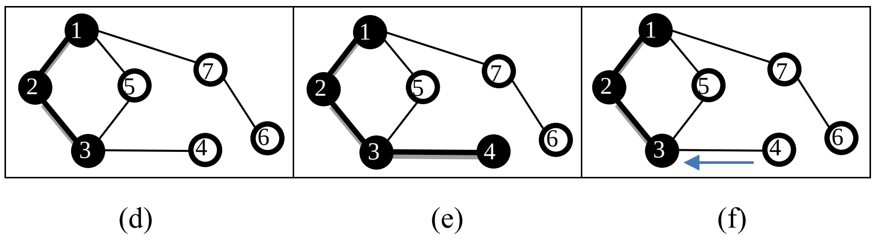

Figure 2. (d) Visiting adjacent vertex (e) Visiting adjacent vertex (f) Selecting previous source vertex.

Figure 2 (d) shows that vertex 3 is the only vertex adjacent to vertex 2, which is the current source vertex. Therefore, the vertex visited next is vertex 3. Thus, $s = 4$ and $P_3 = 1$. Figure 2 (e) shows that both vertex 4 and vertex 5 are adjacent to vertex 3.The algorithm chooses the one with the smallest label. Vertex 4 is therefore chosen as source vertex. The algorithm then sets s = 3 and $P_4 = 1$. In Figure 2 (f), no vertex is adjacent to vertex 4. The algorithm goes back to previous source vertex, which is vertex 3

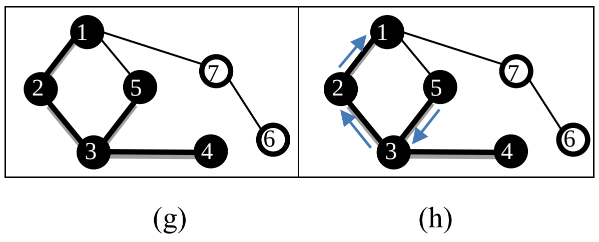

Figure 3. (g) Visiting adjacent vertex (h) Selecting previous source vertex.

Figure 3 (g) shows that vertex 5 is adjacent to vertex 3, so s = 2 and $P_5 = 1$. In Figure 3 (h), no vertex is adjacent to vertex 5. The algorithm goes back to previous source vertex, which is vertex 3. The process is repeated until vertex 1 is visited as shown in figure 2(h).

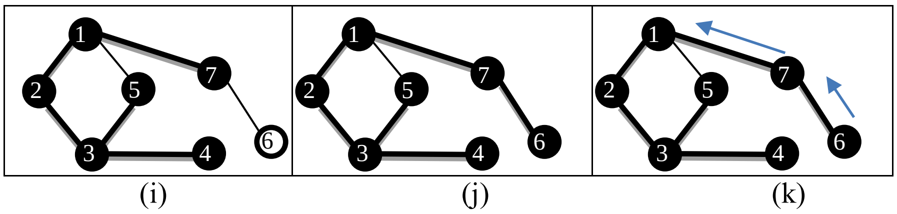

Figure 4. (i) Visiting adjacent vertex (j) Visiting adjacent vertex and (k) Selecting previous source vertex.

Figure 4 (i) shows that vertex 1 is the source vertex. The only vertex adjacent to source vertex is vertex 7. Thus, vertex 7 is visited so $s = 1$ and $P_7 = 1$. Figure 4 (j) shows that the only vertex adjacent to vertex 7 is vertex 6. Thus, vertex 6 is visited so $s = 0$ and $P_6 = 1$. Figure 4 (k) shows no vertex is adjacent to vertex 6, so the algorithm selects the previous vertex and goes back to the first source vertex, which is vertex 1. Vertex 1 has no remaining adjacent vertex. Therefore, the algorithm checks the value of variable s. If $s = 0$, then the graph is connected. However, if $s > 0$, then the graph is disconnected. The graph shown above indicates that $s = 0$, so graph G is a connected graph.

The total time for this algorithm is $O(n^2)$. That is, for each n visited vertices, the algorithm checks its adjacency n times, both adjacent vertices and non-adjacent vertices.

4. Summary

The algorithm can be used to validate an input graph in solving graph problems that need a connected graph, such as verifying a Hamiltonian graph. The proposed algorithm, unfortunately, runs only on a simple graph. Based on this research, it is shown that any simple graph is connected if $s = 0$ and disconnected if $s > 0$ where s is a variable to determine the connectivity of a graph

Acknowledgments
The authors express gratitude to Directorate of Research and Community Services for grant of Applied Products Research.

References

[1] Goodaire E G and Parmenter M M 2002 *Discrete Mathematics with Graph Theory Second Edition* (New Jersey: Prentice-Hall)
[2] Wilson R J 1996 *Introduction to Graph Theory Fourth edition* (London: Longman Group Ltd)
[3] Yang Q and Desmedt Y 2011 Secure communication in multicast graphs. *Advances in Cryptology–ASIACRYPT 2011* pp 538-555
[4] Sci & Educ 2014 Exploring Secondary Students' Epistemological Features Depending on the Evaluation Levels of the Group Model on Blood Circulation, Springer **14** 23 1075–1099
[5] Vegni A M, Campolo C and Molinaro A 2013 *Little TDC Modeling of Intermittent Connectivity in Opportunistic Networks: The Case of Vehicular Ad hoc Networks* (New York: Springer)
[6] Baruah A K 2014 *International J. Com. App.* **86** 11
[7] Aldous J M and Wilson R J 2000 *Graphs and Applications: An Introductory Approach,* (London: Springer-Verlag)
[8] Al A, Mahyus I and Rahma Z 2013 *J. Natural* **13** 19-22
[9] Reza W, Mahyus I and Rahma Z 2014 *J. Natural* **14** 30-33
[10] Hochbaum S D 1993 *The Pseudoflow Algorithm Graph Algorithms and Network Flows*, *Network* (New Jersey: Prentice-Hall)
[11] Bondy J A and Murty U S R 2008 *Graph Theory* (London: Springer-Verlag)