ASHESI

Identification of Disconnected Components in a Graph

Richard Kweenu Quayson

Thomas Kojo Quarshie

Department of Computer Science and Information Systems, Ashesi University

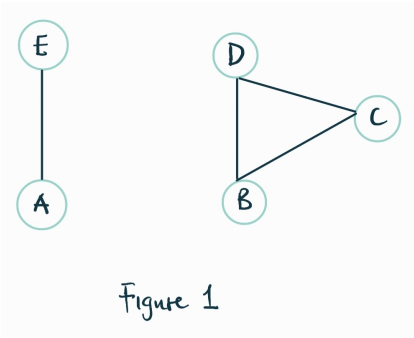
CS456_B: Final Paper

November 23, 2023

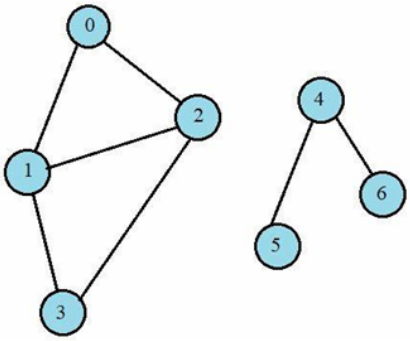
Introduction

Graph theory originated in 1735 when Lenhard Euler tackled and addressed the Königsberg Bridge problem, a mathematical puzzle involving seven bridges in the city of Königsberg [1]. Euler's innovative abstraction, representing the problem through vertices and edges, laid the foundation for the diverse applications of graph theory in solving real-world complex problems. Graphs are non-linear data structures composed of points or nodes called vertices, connected by line segments called edges. Formally, a graph G consists of set of nodes V and a set of edges E, represented mathematically as ***G=(V, E)***. Graphs are used to model relations between objects and solve real world complex problems such as social network analysis, path optimization and logistics (transportation), and network analysis. A graph can either be directed known as a digraph or undirected.

A graph is said to be directed if a graph ***G=(V, E)*** has every edge mapping unto some ordered pair of vertices (V_i, V_j) I.e. an edge between V_i and V_j has an arrow directed from V_i to V_j. A graph is undirected if these pairs of vertices are unordered i.e., a pair of vertices (V_i, V_j) is the same as the pair (V_i, V_j) connected by an edge that has no direct arrow between them [2]. This distinction lays the groundwork for exploring connected and disconnected graphs, revealing insight into the structures that underpin various networked systems. A graph G is said to be connected if its pair of vertices (V_i, V_j) is reachable from one another i.e., there exists a path between every pair of vertices in the graph [3]. A digraph is termed to be a strongly connected graph if there exists a directed path from every vertex to every other vertex in a graph.



(a) Disconnected Components in a graph.



(b) Disconnected Components in a graph.

Figure 1: The figures above shows examples of disconnected graphs.

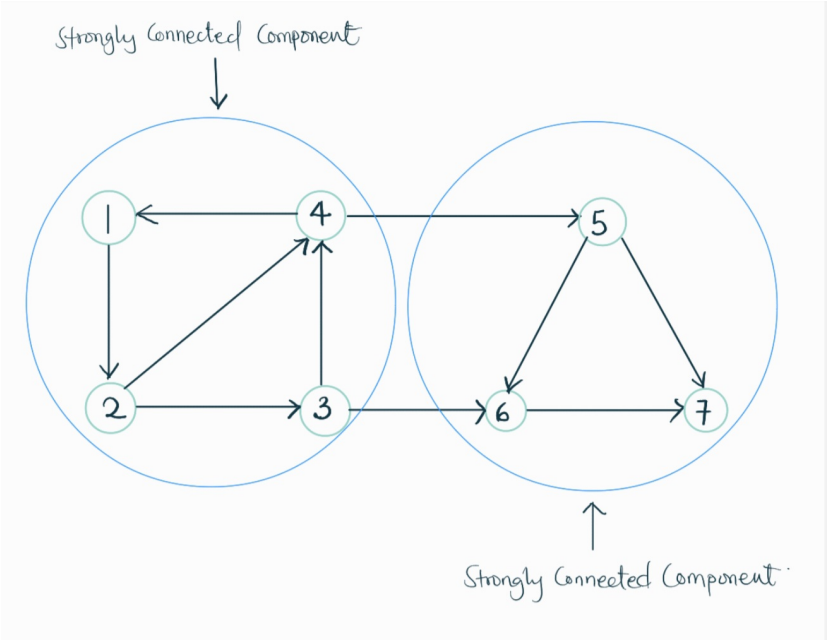


Figure 2: Strongly Connected Components in a graph.

In social network analysis and communication networks, connected graphs signify relationships or interactions, ensuring accessibility among entities and facilitating the smooth flow of information from one location to another [4].

Nevertheless, this paper shifts its focus to the examination of disconnected components within a graph, emphasising their significance in detecting anomalies such as network attacks and botnets in a social network.

Brief Description of the Problem

A graph **G** is said to be disconnected if two vertices exist in G such that no path or edge has those vertices in **G** as endpoints, i.e., the graph **G** contains at least two vertices not connected by an edge. In graph theory, a graph component is a connected subgraph (i.e., a subset of vertices and edges where there is an edge between every pair of vertices) within a larger graph [5]. As such, disconnected components (DC) in a graph are subgraphs that are not connected to any other components in a graph or network of graphs. The growth of disconnected components in a graph is in tandem with the sparsity of a graph. Therefore, enhancing the sparsity of a graph to simplify network complexity and increase the storage efficiency of a network leads to the segmentation of the graph or networks into disconnected components [6]. Disconnected components are less prominent in fully connected graphs, where all pairs of vertices are connected by an edge. However, in real-world networks, graphs are not fully connected, and the presence of disconnected components provides valuable information about the structure, connectivity, and plausible anomalies in a network [6].

Literature Review

In several pieces of literature, different authors propose different approaches to finding the disconnected components in a graph. A spectral Graph Partitioning (SGP) is proposed by [7] to identify the disconnected components in a network. This technique represents a graph in a Laplacian matrix and uses the eigenvalues and eigenvectors of the matrix to find the disconnected components in

a graph. The eigenvalues specify the topological structure of the graph and the count of zero (0) eigenvalues is used to dictate the number of disconnected components in the graph. The eigenvectors constructed with a step function, however, help in identifying the connected nodes within the graph's connected components [7]. Although the Spectral Graph Partitioning technique provides a simpler approach to identifying disconnected components in a graph by utilizing the eigenvalues and eigenvectors of a Laplacian matrix, it is only accurate in some cases. The performance of Spectral Graph Partitioning is opposed by [8], asserting that the limitations of this technique become more pronounced in highly sparse graphs. The Spectral Graph Portioning technique tends to provide inaccurate results in identifying disconnected components in scenarios where a graph or network of graphs has a less conforming graph structure [8]. Although the SGP technique has a good agreement between estimates and numerical results when eigenvector localization is absent, [8] points out the sensitivity of this technique to rare events, such as the emergence of nodes or vertices with irregular degrees in a graph. In such cases, the estimates used in the Spectral Graph Partitioning Technique (SGP) tend to deviate from its numerical results, affecting the overall precision of the method. Although [8] provides a deep understanding of the limitations of the SGP technique, particularly in sparse graphs, the author does not provide an alternative approach or a suitable algorithm that is best suited to identify the disconnected components in a graph.

Nonetheless, other algorithms that are well-known in identifying the disconnected components in a graph are Depth-First Search(DFS) and Breadth-First Search(BFS). These two algorithms help in identifying he disconnected components in a graph, but they have differences in their approaches with limitations. Unlike the BFS, the Depth-First Search algorithm uses less memory as it does

not have to store all the neighbors of a node or vertex. Moreover, the Depth-First Search algorithm is more suitable than BFS in identifying disconnected components in large sparse graphs [3]. However, thee downside of DFS is that it may get stuck in deep levels of recursion for very deep or sparse graphs. Since we are concerned about memory usage and simplicity in identifying disconnected components in a graph, this paper uses the Depth-First Search algorithm.

Description of Algorithm

This paper employs the Depth-First Search (DFS) algorithm to determine the disconnected components in a graph. The algorithm utilises the list or array data structure to store disconnected components and a set data structure to keep track of visited nodes while iterating through the vertex of the nodes in the graph. For each iteration, the algorithm chooses an unvisited vertex as the start of a subgraph and uses the DFS algorithm recursively to determine all nodes that are connected to that vertex, storing the visited nodes in the visited set. The Depth-First Search Algorithm employed in this paper functions by marking the current vertex to begin the search as visited and adding it to the disconnected component. The DFS retrieves the neighbors of the current vertex and iterates through them using an iterator. For each unvisited neighbour, the algorithm recursively explores the neighbours of that neighbour and its connected vertices to determine the disconnected components or subgraphs.

Pseudocode of Algorithm

Algorithm *FindDisconnectedComponents(graphs)*

//Finding the disconnected components in a graph

//Input: A graph

```
//Output: Number of disconnected components and their pairs
visited ← empty set
disconnected_components ← [ ] (empty list)
for each vertex in graph.vertices do
  if vertex not in visited do
  // Start a disconnected component
    disconnected_components ← empty set
    dfs(graph, vertex, disconnected_component)
    disconnected_components ← disconnected_components.add(disconnected_component)
  returndisconnected_components

function dfs(graph, vertex, visited, disconnected_components)
  visited ← visited.add(vertex)
  disconnected_components ← disconnected_component.add(vertex)
  for each neighbour in graph.edges[vertex] do
   if neighbour not in visited
      dfs(graph, neighbour, visited, disconnected_component)
```

Implementation of Algorithm

Refer to the attached zipped file for the implementation of the algorithm.

Analysis of Algorithm

In the `FindDisconnectedComponents` function, the initialization of the `visited` and `disconnected_components` variables is in constant time, $O(1)$. The outer loop iterating over the vertices in the graph has a constant time complexity in terms of the number of vertices, $O(n)$. The `is_visited` condition, initialization

of `disconnected_component`, and the update of `disconnected_components` are also constant time, $O(1)$. However, for each iteration, the `dfs` function is called. Assuming the time complexity of the DFS function is $T_{\text{DFS}}(m, n)$, where m is the number of edges and n is the number of vertices, the time complexity of the `FindDisconnectedComponents` function will be:

$$T(n, m) = O(1) \cdot O(n) \cdot O(1) \cdot T_{\text{DFS}}(m, n)$$

$$T(n, m) = O(n) \cdot T_{\text{DFS}}(m, n)$$

The time complexity of the DFS algorithm is $O(m + n)$, and as such, $T_{\text{DFS}}(m, n) = O(m + n)$. Therefore,

$$T(n, m) = O(n) \cdot O(m + n)$$

$$T(n, m) = O(n \cdot (m + n))$$

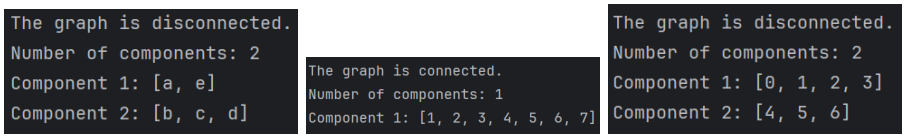
$$T(n, m) = O(mn + n^2)$$

The space complexity of the algorithm is heavily impacted by the `visited` and `disconnected_components` variables. In the worst case, when all vertices are in a single disconnected component, the space complexity of the `disconnected_components` set is $O(n)$. Likewise, the space complexity of the `visited` set is $O(n)$ since all vertices will be visited in the DFS traversal algorithm. Thus, the space complexity of the algorithm is $O(n)$.

Discussion of Results

In testing the designed algorithm, the paper utilizes the graphs cited in Figures 1a, 2, and 1b. The graphs were instantiated in Java with the help of a custom

`Graph` class that had vertices and edges as instance variables and `addEdge` as a method. The graphs cited above were created and passed as arguments to the `FindDisconnectedComponents` method. The images below represent the results from the test cases.



(a) Graph representation of Figure 4. (b) Graph representation of Figure 5. (c) Graph representation of Figure 6.

Figure 3: The images shows the result from the algorithm.

From the images above, the result from the algorithm is consistent with the analysis. The algorithm classified the graphs in Figures 3a and 3c as being disconnected and consisting of two components. Moreover, the graph in Figure 3b was classified as connected as it consisted of a single component.

Conclusion

In conclusion, this paper delves into the realm of graph theory, tracing its origins back to Euler's groundbreaking work on the Königsberg Bridge problem. Graphs, as non-linear data structures, have found widespread applications in diverse fields, modeling relationships and providing solutions to complex real-world problems. The distinction between directed and undirected graphs sets the stage for exploring connected and disconnected components, offering valuable insights into network structures.

The paper narrows its focus to disconnected components within a graph, emphasizing their significance in detecting anomalies such as network attacks and botnets in social networks. The algorithm of choice, Depth-First Search (DFS), has been thoroughly described and illustrated, showcasing its effectiveness in

uncovering disconnected components.

The literature review touches upon various approaches, including Spectral Graph Partitioning, and highlights the limitations of certain techniques in sparse graphs. While other algorithms like DFS and Breadth-First Search are acknowledged, the paper opts for DFS due to its simplicity and efficiency in finding disconnected components.

The pseudocode and implementation details provide a clear guide for applying the DFS algorithm to identify disconnected components. The analysis indicates a time complexity of $O(mn + n^2)$ and a space complexity of $O(n)$, where m is the number of edges and n is the number of vertices.

Overall, this paper contributes to the understanding of disconnected components in graphs, offering a recursive DFS-based algorithmic solution in identifying disconnected components in a graph.