CCDSALG MCO2

Submitted by: MP Group 35

BANIQUED, Lourenz Jhay G.

GUTIERREZ, Allen Andrei D.

REYES, Ma. Julianna Re-an Dg.

UNISAN, Ryu Kisen P.

JAO, Jazzie R.

July 31,2023

I. Introduction

With the rise of technology and social media platforms nowadays, it has now become a trend to have an account that you could use on a social media platform (i.e. Facebook) to further connect with your family and friends. Being able to communicate with your loved ones and also having a platform to share stuff with every now and then. Every account has a number of friends that you personally know in real life but as we could see social media can also be a way to make friends through just having a connection with another person. For example, Person A is friends with Person B, Person B is friends with Person C, and through the use of "mutual friends" on Facebook, users can easily identify if they have a friend who is connected to that person.

The project is a Java program designed to display the friends associated with that account and also find connections between one another even if it is through a number of people, it can still find a way to find a connection between the two users. Given a social network/database, this is an excellent tool to easily identify the connections that link the two users together and also individually display the friend list of any account in the database.

The report is composed of Explaining the Data Structure used in the program, What algorithms were used to display friends and find the connections, Algorithm analysis in terms of time complexity, and lastly reflections/summary of findings.

II. Data Structure

Adjacency Lists

The Data Structure that was implemented in the program is Adjacency Lists. In simple definitions, Adjacency List is an array of linked lists used to represent a graph where each node in the graph stores a list of its neighboring vertices (zaidkhan, 2023).
To further explain the nature of this Data Structure we have an example below:

Given an undirected graph,

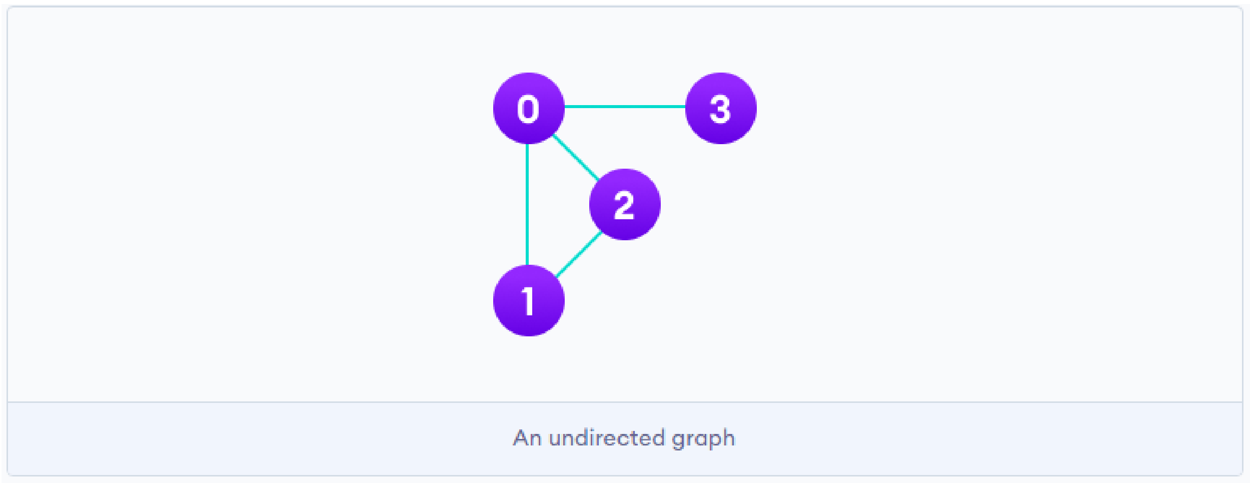


Image 2.a: undirected graph (programiz, n.d.)

We can represent this graph in the form:

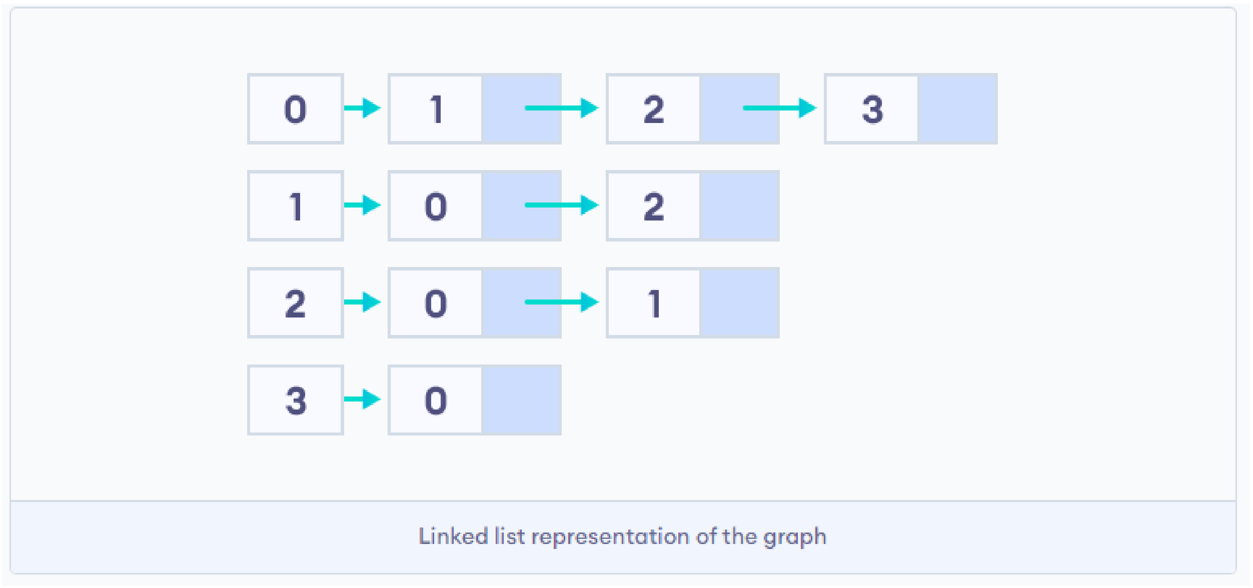


Image 2.b: representation graph (programiz, n.d.)

The representation above shows that 0,1,2,3 are the vertices and each of them forms a linked list with all of its adjacent vertices or connected nodes. For instance, vertex 2 has two adjacent vertices which are 0 and 1. Therefore, 2 is linked with 0 and 1.

The **Rationale** behind the use of Adjacency List:

1. Memory Efficient

   The data set to be utilized in performing the project is a social network structure that is collected from Facebook. This social network would be composed of nodes that represent the account of the user followed by edges to denote if two of the accounts are friends. Accordingly, we could define this type of social graph as a sparse graph since an account would only have a limited number of friends meaning the edges would only be a few nodes for designating the friend connection between the accounts. And that's why storing the graph as an adjacency list avoids the need to allocate memory to determine all the possible connections between each account. For example, suppose we have 1000 accounts in the social network and each account has an average of 10 friends. If we use a different data structure to represent the graph we would need to allocate memory for all possible connections even if most of them are empty compared to an adjacency list where it would only allocate memory if there really is a connection between the two accounts.
2. Dynamic structure

   Considering that an adjacency list is also a collection of linked lists this means that it is much easier to edit the connection between two accounts since we won't be needing to resize the structure.
3. Easy to traverse

   Adjacency list structure can utilize Depth-First Search (DFS) because of its capability to have fast access to its neighboring nodes. When performing a Depth-First Search (DFS) for graph traversal, having direct access to the neighbors of each vertex allows for efficient exploration throughout the graph.

III. Algorithm

Depth-First Search

The algorithm that is used to display the connection between two people in the network is DFS or Depth First Search Algorithm. Depth-First Search (DFS) is a graph traversal algorithm that explores as far as possible along each branch before backtracking. It starts from a source vertex and explores as far as possible along each branch before backtracking. The algorithm uses a stack data structure to keep track of vertices in the traversal process. The DFS algorithm is recursive in nature because when it reaches a vertex with no unvisited adjacent vertices, it backtracks to the previous vertex and continues the exploration from there. The process continues until all vertices are visited.

The Steps in DFS

1. Initialization

   Choose a starting vertex as the source and mark it as visited. Push this vertex onto the stack.

2. Explore

   While the stack is not empty:

   Pop a vertex from the stack. This will be the current vertex.

   Process the current vertex.

   Visit all the adjacent vertices of the current vertex that have not been visited yet. For each unvisited adjacent vertex, mark it as visited, and push it onto the stack.

3. Backtrack

   If there are no unvisited adjacent vertices for the current vertex, backtrack by popping the previous vertex from the stack. This will take you back to the last vertex that has unvisited adjacent vertices.

4. Termination

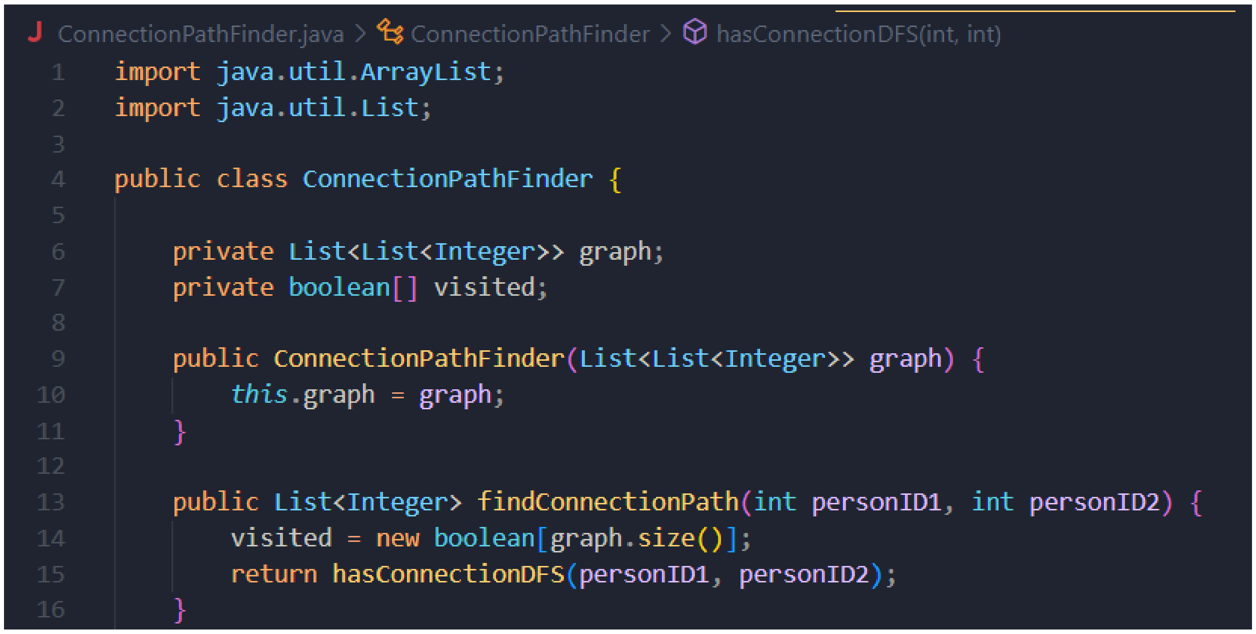
   Repeat steps 2 and 3 until the stack becomes empty. This means you have visited all the reachable vertices from the source vertex.

IV. Algorithm Analysis

Analysis of Depth-First Search Algorithm

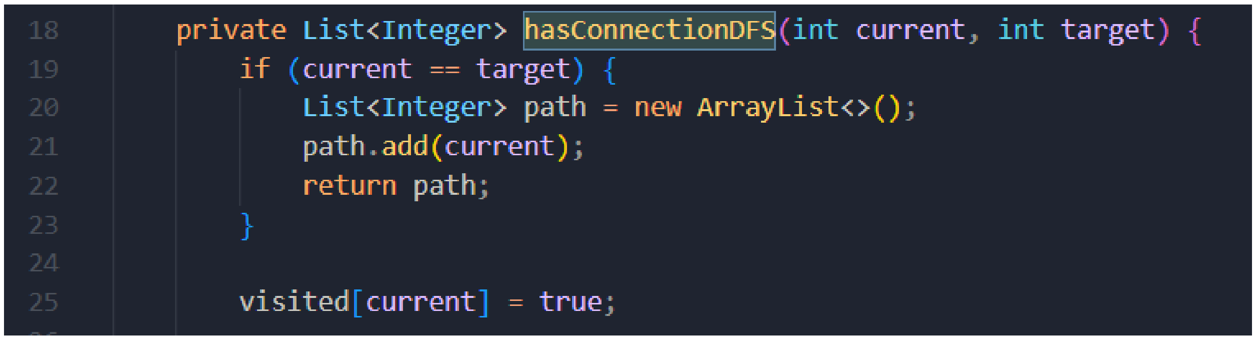
The Depth-First Search algorithm is utilized to determine whether two given user inputs of a person's ID have a connection with each other. The time complexity of the DFS method can be divided into parts:

1. Initialization (Current Vertices / Parent)
   Starting at a vertex, marking it as its source we consider as visited will give us a **O(V) time complexity**. The V stands for the number of vertices or stored ID's of the people that will be visited by the code snippet "*findConnectionPath method*" (refer to image below).



```
ConnectionPathFinder.java > ConnectionPathFinder > hasConnectionDFS(int, int)
1   import java.util.ArrayList;
2   import java.util.List;
3
4   public class ConnectionPathFinder {
5
6       private List<List<Integer>> graph;
7       private boolean[] visited;
8
9       public ConnectionPathFinder(List<List<Integer>> graph) {
10          this.graph = graph;
11      }
12
13      public List<Integer> findConnectionPath(int personID1, int personID2) {
14          visited = new boolean[graph.size()];
15          return hasConnectionDFS(personID1, personID2);
16      }
```

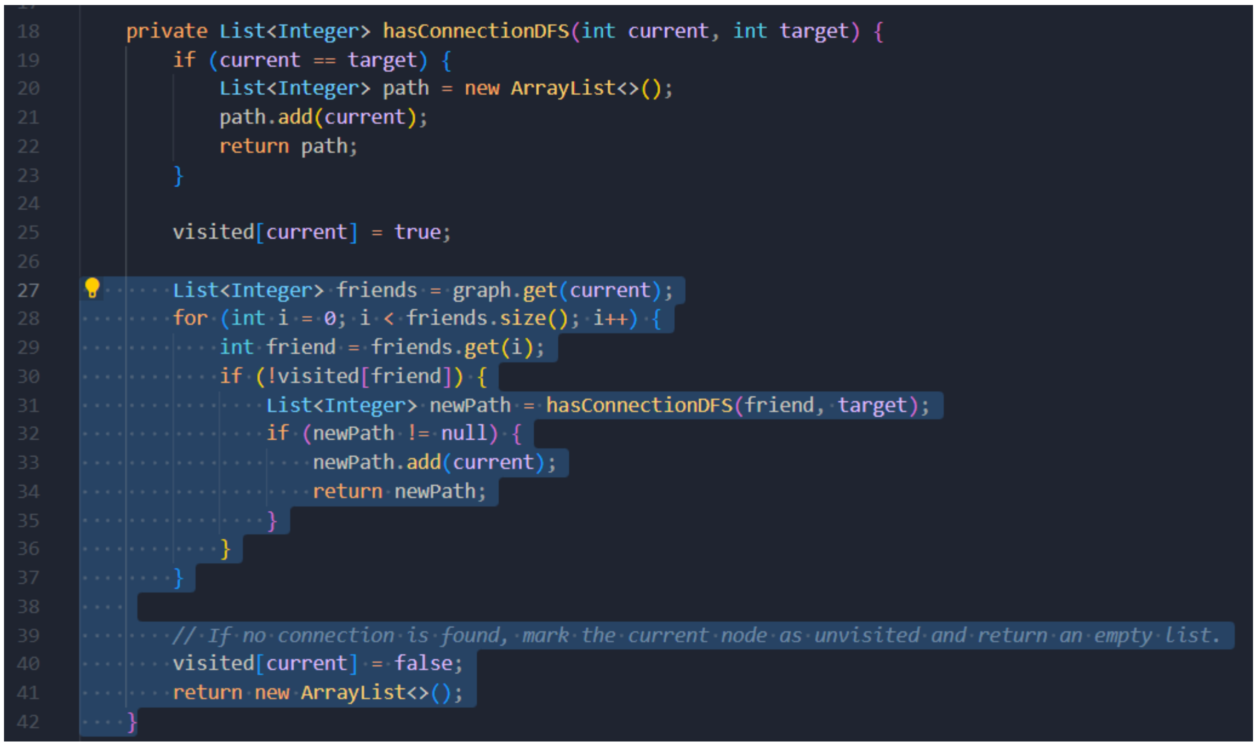
2. Explore
   After choosing a specific vertex, the code in line 15 (refer to image above) transfers the parameter values of both IDs to the "*hasConnectionDFS method*" (refer to image below) which is the algorithm that explores the depth of the given vertex to analyze whether there is a valid connection.



```
18      private List<Integer> hasConnectionDFS(int current, int target) {
19          if (current == target) {
20              List<Integer> path = new ArrayList<>();
21              path.add(current);
22              return path;
23          }
24
25          visited[current] = true;
```

Shown above, when the current and target IDs are the same, the program will add the path as its current and will automatically return the path back. In short, this will be the base case.

When for instance the base case is not met, the program will continue to the recursive part of the method and explore the depth of the current vertices until it reaches a connection that will find its target (refer to image below).



```
18      private List<Integer> hasConnectionDFS(int current, int target) {
19          if (current == target) {
20              List<Integer> path = new ArrayList<>();
21              path.add(current);
22              return path;
23          }
24
25          visited[current] = true;
26
27          List<Integer> friends = graph.get(current);
28          for (int i = 0; i < friends.size(); i++) {
29              int friend = friends.get(i);
30              if (!visited[friend]) {
31                  List<Integer> newPath = hasConnectionDFS(friend, target);
32                  if (newPath != null) {
33                      newPath.add(current);
34                      return newPath;
35                  }
36              }
37          }
38
39          // If no connection is found, mark the current node as unvisited and return an empty list.
40          visited[current] = false;
41          return new ArrayList<>();
42      }
```

The highlighted code snippet shows the DFS traversal code which will be utilized recursively until it reaches a visited current node (current ID) accessing its friends to find adjacent friends (hence, we utilized an adjacency list) that will lead to the target node (target ID). To avoid revisiting past nodes, line 25 will ensure that visited nodes == true will be marked. If the traversal code of DFS has a connection, the adjacent friends will be saved and added to the "*newPath integer List*". After adding all adjacent nodes to the newPath integer list, the method will end by returning the newPath or list of connections.

Lines 39-41 will only happen when there is no connection between the current and target node. This will mark the current node as unvisited returning an empty ArrayList.

Since the graph implements an adjacency list or an array list of linked lists, each node traversal will have a list of adjacent edges. However, each node and its adjacent edges will only be traversed once the overall time complexity of the DFS traversal will be a sum of vertices and the number of each vertices' edges **O(V + E) time complexity**.

V. Summary of findings

The group gained multiple lessons, insights and realizations with this project. We gained first hand experience with the use of Data Structures and which specific Data structure to use for certain situations. The usage of graphs was helpful because the team dealt with large amounts of unsorted data. It would be difficult to see and establish connections between people without the use of graphs. The usage of graphs helped the team visualize the data and its connections to one another more clearly. The group realized first hand the importance of choosing the right data structure for the problem given to us. Initially, the team thought of using an Array because of its accessibility and that the group was familiar with the data structure already. What we hadn't realized was that Lists would've made a better data structure for the problem. Lists was the better option for our program because Lists offer more advantages than arrays can in this situation because we're working to find connections between thousands of data points. Additionally the group used a specific type of List called Adjacent List. As stated before the simplest explanation of an Adjacent List is an array of linked lists used to represent a graph. How the List was ordered was crucial to the creation of the graph and Lists made sure that the order of elements being inserted was always uniform, it always made sure the sequence of data was based on insertion of elements to the list.

VI. Contributions

MEMBER NAME	CONTRIBUTIONS
Baniqued, Lourenz Jhay G.	- Introduction and Data Structure part of the Report
Gutierrez, Allen Andrei D.	- Algorithm used and Summary of Findings part of the Report
Reyes, Ma. Julianna Re-an Dg.	- Summary of Findings part of the Report
Unisan, Ryu Kisen Unisan	- Algorithm analysis and Summary of Findings part of the Report

VII. References

Adjacency List- Programiz. (n.d). from https://www.programiz.com/dsa/graph-adjacency-list#:~:text=An%20adjacency%20list%20represents%20a,an%20edge%20with%20the%20vertex.

Depth First Search or DFS for a Graph. (n.d). from https://www.geeksforgeeks.org/depth-first-search-or-dfs-for-a-graph/

Zaidkhan (2023). *Adjacency List meaning & definition in DSA – GeeksforGeeks.* from https://www.geeksforgeeks.org/adjacency-list-meaning-definition-in-dsa/

VIII. Images

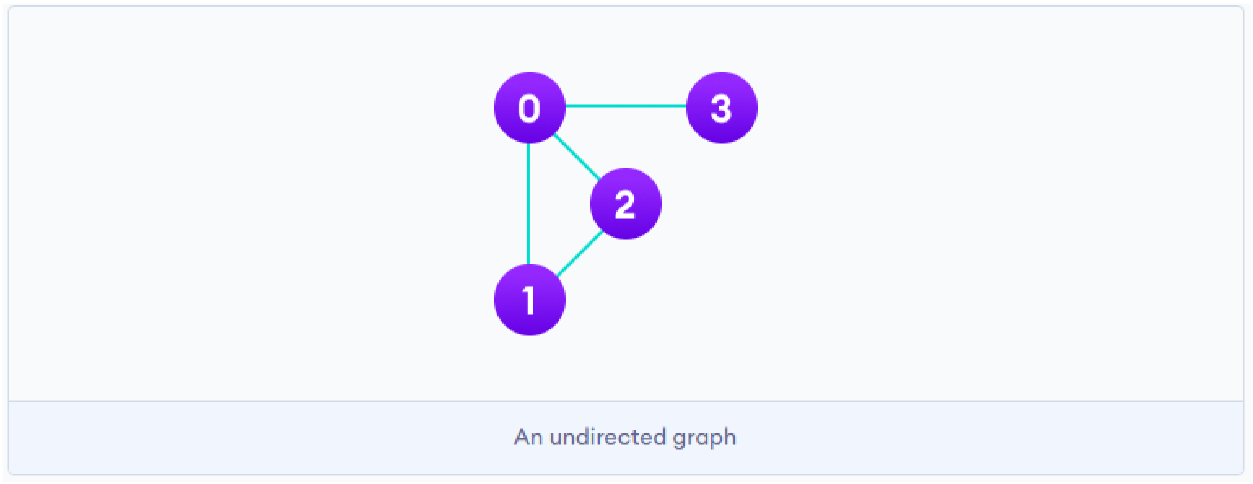


Image 2.a: undirected graph (programiz, n.d.)

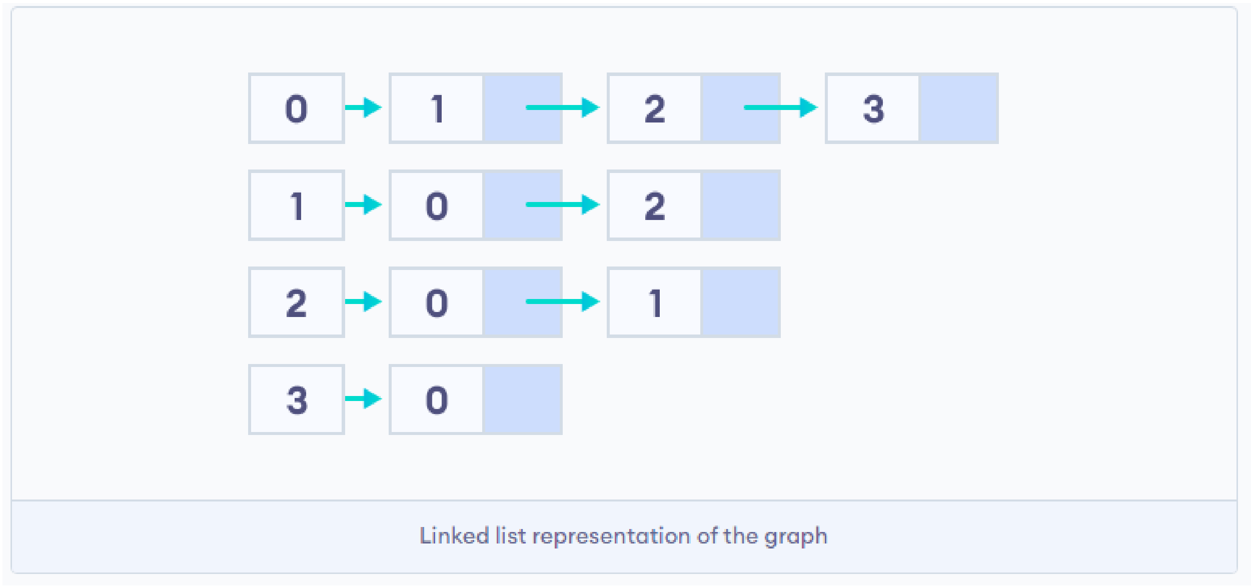


Image 2.b: representation graph (programiz, n.d.)