Assignment Solutions : Graph - 4 Problems on graph Coloring

Q1 An image is represented by an m x n integer grid image where image[i][j] represents the pixel value of the image.

You are also given three integers sr, sc, and color. You should perform a flood fill on the image starting from the pixel image[sr][sc].

To perform a flood fill, consider the starting pixel, plus any pixels connected 4-directionally to the starting pixel of the same color as the starting pixel, plus any pixels connected 4-directionally to those pixels (also with the same color), and so on. Replace the color of all of the aforementioned pixels with color.

Return the modified image after performing the flood fill.

Code link: https://pastebin.com/3HyGKGtV

Explanation:

- **The dfs function performs a Depth-First Search (DFS) on the image. It takes the starting row (sr), starting column (sc), new color, and old color as input.**
- **If the starting row or column is out of bounds or the color of the current pixel is not the same as the old color, return from the function.**
- **Set the color of the current pixel to the new color.**
- **Recursively call the dfs function for the four adjacent pixels (up, down, left, right) to perform the flood fill.**
- **The floodFill function first checks if the new color is the same as the old color. If it is, there is no need to perform the flood fill, so it returns the original image.**
- **Otherwise, it calls the dfs function to perform the flood fill starting from the given starting row and column.**
- **Finally, it returns the modified image.**

Q2 You are given a positive integer n representing the number of nodes in an undirected graph. The nodes are labeled from 1 to n.

You are also given a 2D integer array edges, where edges[i] = [ai, bi] indicates that there is a bidirectional edge between nodes ai and bi. Notice that the given graph may be disconnected.

Divide the nodes of the graph into m groups (1-indexed) such that:

- **Each node in the graph belongs to exactly one group.**
- **For every pair of nodes in the graph that are connected by an edge [ai, bi], if ai belongs to the group with index x, and bi belongs to the group with index y, then |y - x| = 1.**

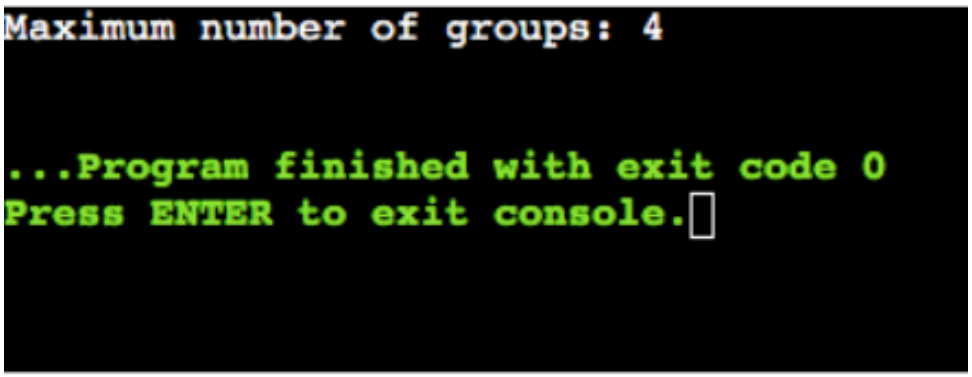
Return the maximum number of groups (i.e., maximum m) into which you can divide the nodes. Return -1 if it is impossible to group the nodes with the given conditions.

Code link: https://pastebin.com/nGDSZLqC

Explanation:

- **The normalBfs function performs a Breadth-First Search (BFS) starting from the given start node to calculate the level (depth) of the BFS traversal. It takes the start node, a hash table representing the edges (hashEdges), and a vector of boolean values indicating the visited nodes as input. It returns the level of the BFS traversal.**
- **Inside the normalBfs function:**
  - **Initialize a queue (q) and push the start node into the queue.**
  - **Mark the start node as visited by setting the corresponding element in the visited vector to true.**
  - **Initialize the level variable to 0 to keep track of the BFS level.**
  - **Perform the BFS traversal:**
    - **For each level of the BFS, iterate through the nodes in the queue.**
    - **For each node, iterate through its neighboring nodes (obtained from the hashEdges hash table).**
    - **If a neighboring node is not visited, enqueue it and mark it as visited.**
  - **Increment the level variable after processing each level.**
  - **Finally, return the level.**

- **The checkConnectedBfs function performs a modified BFS to check if all the nodes at the same level have any connections among themselves. It also calculates the level of the modified BFS. It takes the start node, the hashEdges hash table, and a vector of boolean values indicating the visited nodes as input. It returns the level of the modified BFS traversal.**
- **Inside the checkConnectedBfs function:**
  - **Initialize a queue (q), a vector (members) to store the nodes at the same level, and a vector (visitedCopy) to create a copy of the visited vector.**
  - **Push the start node into the queue, add it to the members vector, and mark it as visited.**
  - **Initialize the level variable to 0 to keep track of the modified BFS level.**
  - **Perform the modified BFS traversal:**
    - **For each level of the BFS, iterate through the nodes in the queue.**
    - **For each node, iterate through its neighboring nodes (obtained from the hashEdges hash table).**
    - **If a neighboring node is not visited, enqueue it, mark it as visited, and add it to the collections vector.**
    - **After processing each level, check if all nodes at the same level have any connections among themselves. If any connections are found, return -1.**
    - **Enqueue the nodes in the collections vector, add them to the members vector, and increment the level variable.**
  - **After the modified BFS traversal, perform a normal BFS for each member in the members vector using the normalBfs function to obtain the maximum level.**
  - **Finally, return the maximum level.**
- **The magnificentSets function takes the number of nodes n and the edges of the graph (edges) as input. It constructs a hash table (hashEdges) to represent the graph and performs the grouping and checking using the checkConnectedBfs and normalBfs functions. It returns the maximum number of groups into which the nodes can be divided according to the given conditions. If it is not possible to group the nodes, it returns -1.**
- **In the main function, the code calculates the maximum number of groups into which the nodes can be divided based on the provided number of nodes (n) and edges (edges) of the graph. The output will be "Maximum number of groups: 4".**



Q3 There is a bi-directional graph with n vertices, where each vertex is labeled from 0 to n - 1. The edges in the graph are represented by a given 2D integer array edges, where edges[i] = [ui, vi] denotes an edge between vertex ui and vertex vi. Every vertex pair is connected by at most one edge, and no vertex has an edge to itself.
Return the length of the shortest cycle in the graph. If no cycle exists, return -1.
A cycle is a path that starts and ends at the same node, and each edge in the path is used only once.

Code link: https://pastebin.com/P9DzT2ba

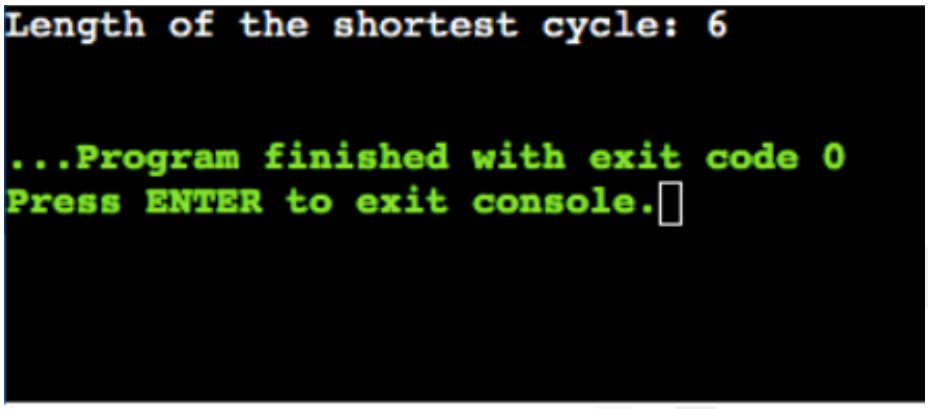
Explanation:

- **The findShortestCycle function takes the number of vertices n and the edges of the graph as input. It returns the length of the shortest cycle in the graph. If no cycle exists, it returns -1.**
- **Inside the findShortestCycle function:**
  - **Construct an unordered map graph to represent the graph using the given edges. Each vertex is mapped to a vector of its adjacent vertices.**
- **Iterate through each vertex i from 0 to n-1:**
  - **Initialize two vectors, dist and parent, of size n to keep track of the distances and parent vertices during the BFS traversal. Initialize all elements of dist to a large value (1e9) and all elements of parent to -1.**
  - **Initialize a queue q and enqueue the current vertex i.**
  - **Initialize the distance of the current vertex i to 0 (dist[i] = 0).**
- **Perform a modified BFS traversal from each vertex:**
  - **While the queue is not empty, dequeue a vertex curr from the queue.**
  - **Iterate through the adjacent vertices next of curr.**
  - **If the distance of next is 1e9, it means it has not been visited:**
    - **Update the distance of next to dist[curr] + 1.**
    - **Set the parent of next as curr.**
    - **Enqueue next into the queue.**
  - **If the parent of next is not equal to curr and the parent of curr is not equal to next, it means an edge forms a cycle:**
    - **Calculate the cycle length as 1 + dist[next] + dist[curr].**
    - **Update minCycle with the minimum cycle length found.**
- **If minCycle is still 1e9, it means no cycle was found. Return -1.**
- **Otherwise, return minCycle, which represents the length of the shortest cycle found in the graph.**

The code performs a modified BFS traversal starting from each vertex in the graph. During the traversal, it keeps track of the distances and parent vertices to detect cycles. If an edge is found that forms a cycle, the code calculates the length of the cycle and updates the minimum cycle length (minCycle). Finally, it returns the length of the shortest cycle found in the graph or -1 if no cycle exists.

The BFS approach guarantees that the first cycle encountered will be the shortest cycle in the graph. By maintaining the distances and parent vertices during the traversal, the code can identify cycles formed by edges that are not immediate neighbors. This allows it to find the shortest cycle even in graphs with complex connectivity patterns.

Output:



Q4 There are n cities. Some of them are connected, while some are not. If city a is connected directly with city b, and city b is connected directly with city c, then city a is connected indirectly with city c.
A province is a group of directly or indirectly connected cities and no other cities outside of the group.
You are given an n x n matrix isConnected where isConnected[i][j] = 1 if the ith city and the jth city are directly connected, and isConnected[i][j] = 0 otherwise.
Return the total number of provinces.

Code link: https://pastebin.com/XCRmSVHz

Explanation:

- **The findCircleNum function takes the matrix isConnected as input and returns the total number of provinces.**
- **Inside the findCircleNum function:**
  - **Initialize variables n, i, and j.**
  - **Create an adjacency list g to represent the graph using the given matrix isConnected.**
  - **Iterate through each pair of cities i and j (with i from 0 to n-1 and j from i+1 to n-1):**
    - **If isConnected[i][j] is 1, it means the cities i and j are connected.**
    - **Add an edge between cities i and j in the adjacency list g.**

- Initialize a variable ans to keep track of the number of provinces.
- Create a vector vis of size n to keep track of visited cities. Set all elements of vis to false.

- Iterate through each city i from 0 to n-1:
  - If the city i has not been visited (!vis[i]), it means a new province is found.
  - Increment ans by 1 (to count the current province).
  - Call the solve function to perform a depth-first search starting from city i.
    - Inside the solve function:
      - Mark the current city as visited by setting v[s] to true.
      - Iterate through each neighboring city i of the current city.
      - If the neighboring city has not been visited, recursively call the solve function on the neighboring city.
- Return the total number of provinces (ans).

The provided code uses a depth-first search (DFS) approach to find the total number of provinces in the graph. It constructs an adjacency list from the given matrix isConnected and performs DFS on each unvisited city to find connected provinces. The count of the provinces is incremented for each new starting city of DFS traversal. Finally, the code returns the total number of provinces found.

Output: