GRAPH MINING ASSIGNMENT

PART - 1

By: Aayush Keval Shah
2019A7PS0137H

About the assignment:

Language used: C++
The folder contains Five folders, one for each question and also the input graph files containing the edge lists for each graph.

Contents:

Q1) Create a graph reader that can read graphs from an edge list file and can store the graph in the adjacency list and CSR representations.

sol)
The Part_1_Q1 folder contains the source code for this question. The 1_Graph_reader.cpp file contains the code for generating the graph representations asked.

The following functions are used:

1. **readEdges()** : Reads the edges present in input file and stores them in a 2D vector.
2. **construct_adjList(vector<vector<int>> &edges)** : Constructs the adjacency list representation from the edge list stored.
3. **construct_CSR(map<int,vector<int>> &adjList, vector<int> &offsets, vector<int> &neighbours) :** Constructs the CSR representation from the adjacency list representation constructed above.
4. **display_adjList(map<int,vector<int>> adjList)** : Displays the Adjacency List.
5. **display_CSR(vector<int> &offsets, vector<int> &neighbours):** Displays the CSR representation**.**
6. All the above functions are called in the main function chronologically and thus the Graph representations are generated.

Thus the Adjacency List and the Compressed Sparse Row representations are generated through the given edge list file.

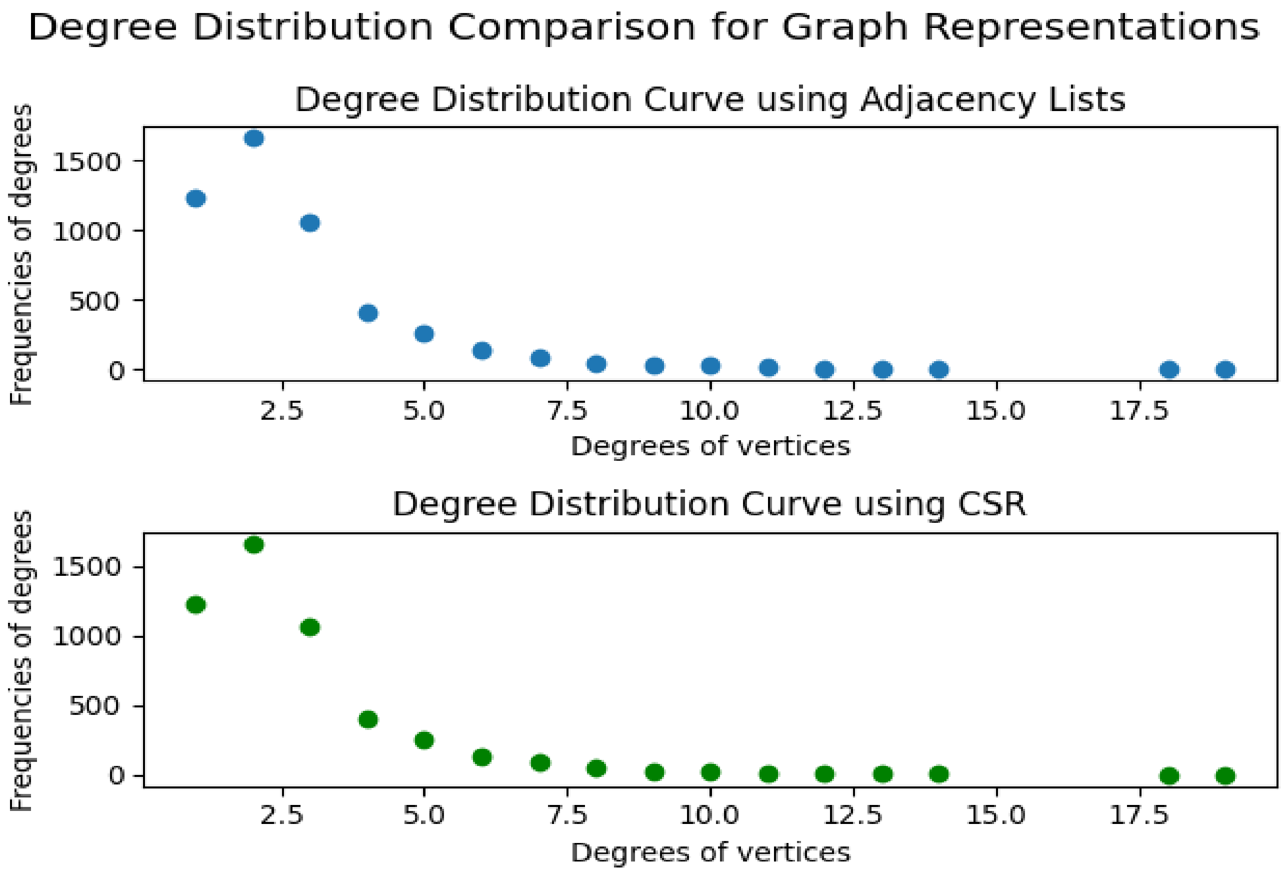
Q2) Generate a degree distribution of the given input graph using both the adjacency list and CSR representation. Check that you are getting the same degree distribution for a single graph using two different formats. Plot the degree distribution and generate pdf image of the degree distribution.

sol)
The Part_1_Q2 folder contains the source code for this question.

1. The 2_Degree_checker.cpp file contains the code for generating the adjacency list and CSR representations asked. The following functions are used:
    a. **readEdges()** : Reads the edges present in input file and stores them in a 2D vector.
    b. **construct_adjList(vector<vector<int>> &edges)** : Constructs the adjacency list representation from the edge list stored.
    c. **display_adjList(map<int,vector<int>> adjList)** : Displays the Adjacency List.
    d. **countDegree_adjList(map<int,vector<int>> adjList):** Generates the degree frequency array from the adjacency list.
    e. **storeDegree_adjList(vector<int> &degreeFreq_adjList):** Stores the degree frequency generated from the adjacency list in adjList_degreeFreq.txt file.
    f. **construct_CSR(map<int,vector<int>> &adjList, vector<int> &offsets, vector<int> &neighbours) :** Constructs the CSR representation from the adjacency list representation constructed above.
    g. **display_CSR(vector<int> &offsets, vector<int> &neighbours):** Displays the CSR representation**.**

h. **countDegree_CSR(vector<int> &offsets, vector<int> &neighbours) :** Generates the degree frequency array from the CSR.
i. **storeDegree_CSR(vector<int> &degreeFreq_CSR):** Stores the degree frequency generated from the CSR in CSR_degreeFreq.txt file.
j. All the above functions are called in the main function chronologically.

2. The plotter.py file uses matplotlib and numpy to plot the degree distribution curves after reading them from the files in which the c++ program stored them. The comparison plot is saved as a png file.
For eg:



Q3) Count the following (1) 3-cycles or triangles (2) 4 cycles (non-induced) in a given graph. Report the number of triangles and 4-cycles as well as the running time for each graph.

sol) The Part_1_Q3 folder contains the source code for this question.

1. The 3_N_Cycles.cpp file contains the code for calculating the diameter of the given graph. The following functions are used:
    a. **readEdges()** : Reads the edges present in input file and stores them in a 2D vector.
    b. **construct_adjList(vector<vector<int>> &edges)** : Constructs the adjacency list representation from the edge list stored.
    c. **display_adjList(map<int,vector<int>> adjList)** : Displays the Adjacency List.
    d. DFS(map<int,vector<int>> &adjList, vector<bool> &visited, int n, int vert, int start, int &count) : With DFS find every possible path of length n-1 (n-1 because the last edge will be closing edge for the cycle) for a particular source. Then we check if this path ends with the vertex it started with, if yes then we count this as the cycle of length n.
    e. countCycles(map<int,vector<int>> &adjList, int n) : Every possible path of length (n-1) can be searched using only V – (n – 1) vertices instead of all V vertices (where V is the total number of vertices) as these vertices cover the cycles of remaining vertices as well and thus maxmizies efficiency. We call DFS for each of these nodes. We consider all the paths of length n containing the node u where u belongs to V and then once done mark it as visited.
    f. All the above functions are called in the main function chronologically.

Overall we find the cycles of given length from each node and then count the total number of cycles. The overall time taken by this program for a graph with V vertices and E edges is O(V*(V+E)) which can be written as O(V*E) as we are calling a simple DFS function for all nodes and then traversing all the edges from it. When the graph is dense the edges E will be of the order of O(V^2) and thus the total runtime of the algorithm would be O(V^3).

Q4) Compute the diameter of each graph.

sol)
The Part_1_Q4 folder contains the source code for this question.

2. The 4_Graph_Diameter.cpp file contains the code for calculating the diameter of the given graph. The following functions are used:
    a. **readEdges()** : Reads the edges present in input file and stores them in a 2D vector.
    b. **construct_adjList(vector<vector<int>> &edges)** : Constructs the adjacency list representation from the edge list stored.
    c. **display_adjList(map<int,vector<int>> adjList)** : Displays the Adjacency List.
    d. **BFS(map<int,vector<int>> &adjList, vector<int> &distance, int source):** This function traverses the nodes in a graph breadthwise. It implements the breadth-first search starting from the source node and thus maintains the distance of all other nodes from itself. The rule for updating the distance is :
        i. **if(!visited(neighbour)) distance(neighbour)=distance(source)+1**

        ii. For a given start node it traverses all the nodes in that connected component

    e. All the above functions are called in the main function chronologically.

Overall for an undirected and unweighted graph we can call BFS starting from each node in the graph and thus calculate the distance between it and all other nodes. Then we take the maximum distance from given node and do the same for every node in the graph. Finally we will get the Longest shortest-path of the graph which is the diameter of the graph.

The overall time taken by thins program for a graph with V vertices and E edges is O(V*(V+E)) which can be written as O(V*E) as we are calling a simple BFS function for all nodes and then traversing all the edges from it. When the graph is dense the edges E will be of the order of O(V^2) and thus the total runtime of algorithm would be O(V^3).

There are still more efficient algorithms like the iFUB algorithm to find the diameter of undirected large graphs.

Q5) Compute the size of the largest connected component for each graph.

sol) The Part_1_Q5 folder contains the source code for this question.

1. The 5_Largest_Component.cpp file contains the code for finding the size of the largest connected component of the given graph using the adjacency list generated from the edge list input file. The following functions are used:

a. **readEdges()** : Reads the edges present in input file and stores them in a 2D vector.
b. **construct_adjList(vector<vector<int>> &edges)** : Constructs the adjacency list representation from the edge list stored.
c. **display_adjList(map<int,vector<int>> adjList)** : Displays the Adjacency List.
d. iterativeDFS(map<int,vector<int>> &adjList, vector<bool> &visited, int source) : This function is called for every node present in the graph. This function visits all the nodes depth wise starting from the source node. This also maintains a visited boolean array to make sure we visit a node only once. Overall this iterative Depth First Search algorithm counts the size of a single connected component.
e. All the above functions are called in the main function chronologically and thus the Graph representations are generated.

Overall, the adjacency list is used to traverse through all the connected components and thus we find the size of the largest component after each DFS call in the main function. The overall time taken by thins program for a graph with V vertices and E edges is O(V+E) as we are calling a simple DFS function for all nodes and then traversing all the edges.