CS F426 Graph Mining Lab-1 Documentation

Q1.

We need to create the graph reader, which reads the graph from the edge list file and stores graphs.
It is asked to represent the graph in two different formats:

a. Adjacency list
b. CSR representation

Step1: Reading the graph from the edge list file and creating an edge list in CPP
Declare an edge list and pass by reference to the **readGraph(**) function, where the graph is read from the file as a stream of strings
The data from the file is stored in the edge list.
The helper functions like **stringtoint()** which takes in argument a string and returns its integer value are also used.

1. To represent in the Adjacency List.
    - To represent the graph which is stored as an edge list in the form of an adjacency list, we are using a map data structure whose key is an integer and value is a vector of integers.
    - Each key of the map represents a node in the graph and its value, a vector, represents to which of all the nodes the key node is connected.
2. To represent in CSR format
    - Since CSR of a graph contains 3 vectors namely, weights, IA, JA, three vectors are declared.
    - The **CSRAdjList()** function in the code converts the graph which is in the form of an adjacency list to CSR.

The complexity of the algorithm is **O(V + E)** where E is the number of edges in the graph and V is the number of vertices of the graph.

Run time Graph1_a: 0.056s
Run time Graph2_a: 3.591s

Run time Graph1_b: 0.045s
Run time Graph2_b: 3.792s

Q2.

In the given question, 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. To show the degree distribution, in the x-axis you will be showing the degree and in the y-axis, you will be showing the frequency.

The important function that is to be noted helps in generating the plot of the graph in the png format is **plot_graph()**
The plot is generated with file name example-1.png.
The Time complexity of the algorithm is **O(V + E)** where E is the number of edges in the graph and V is the number of vertices of the graph.

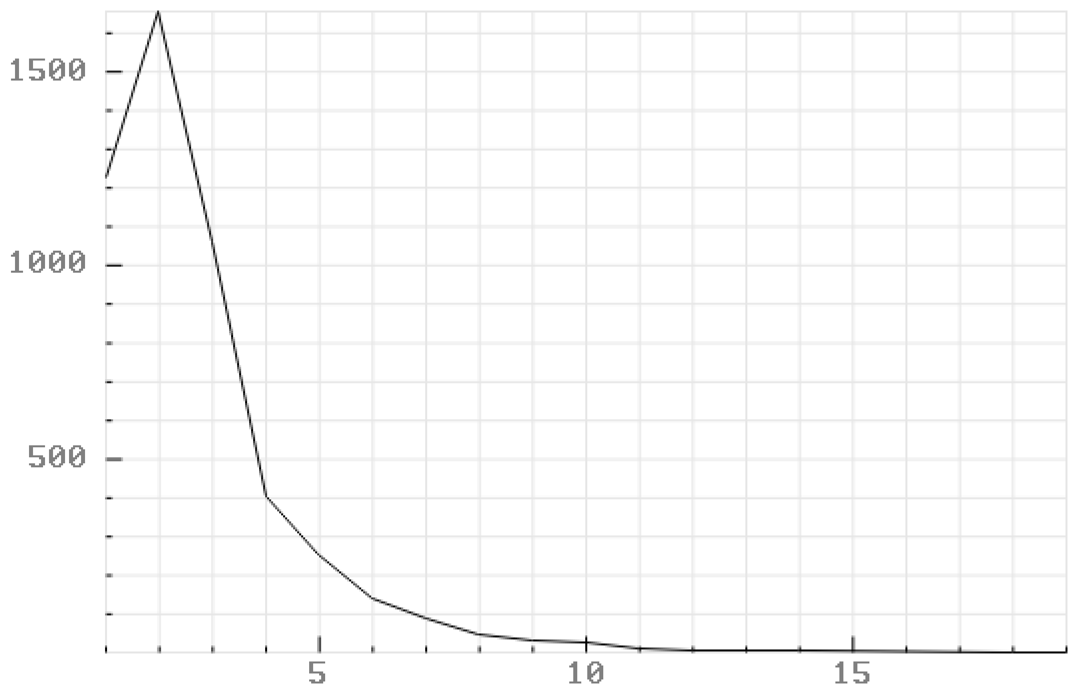

Fig 2.1 Adjacency list plot for graph 1

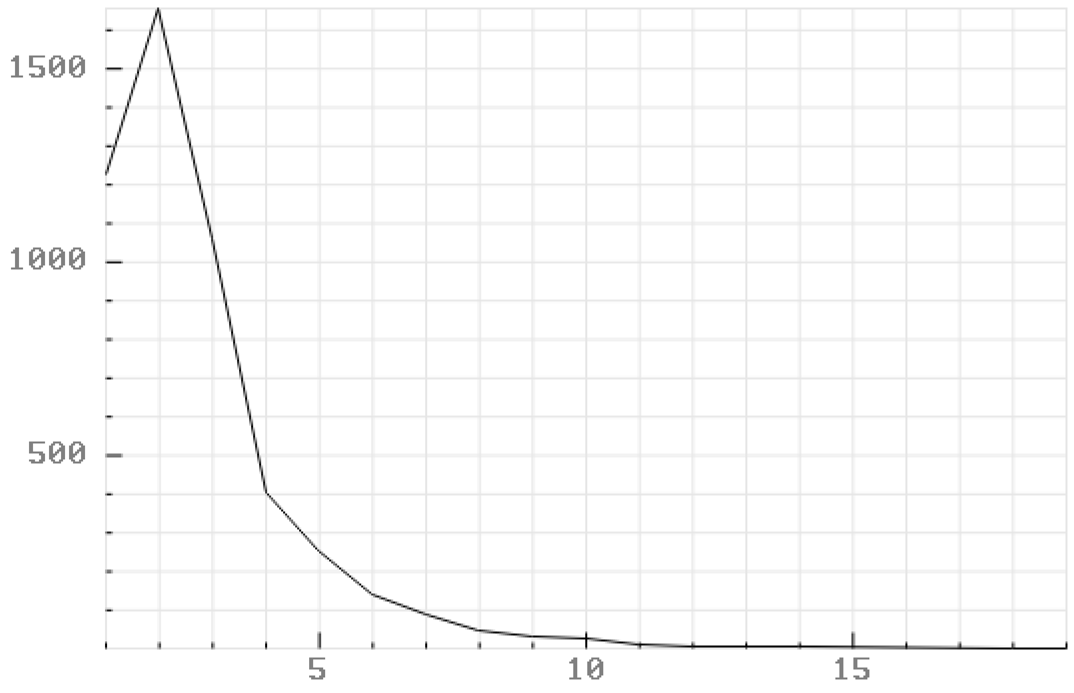

Fig 2.2 CSR Plot for graph 1

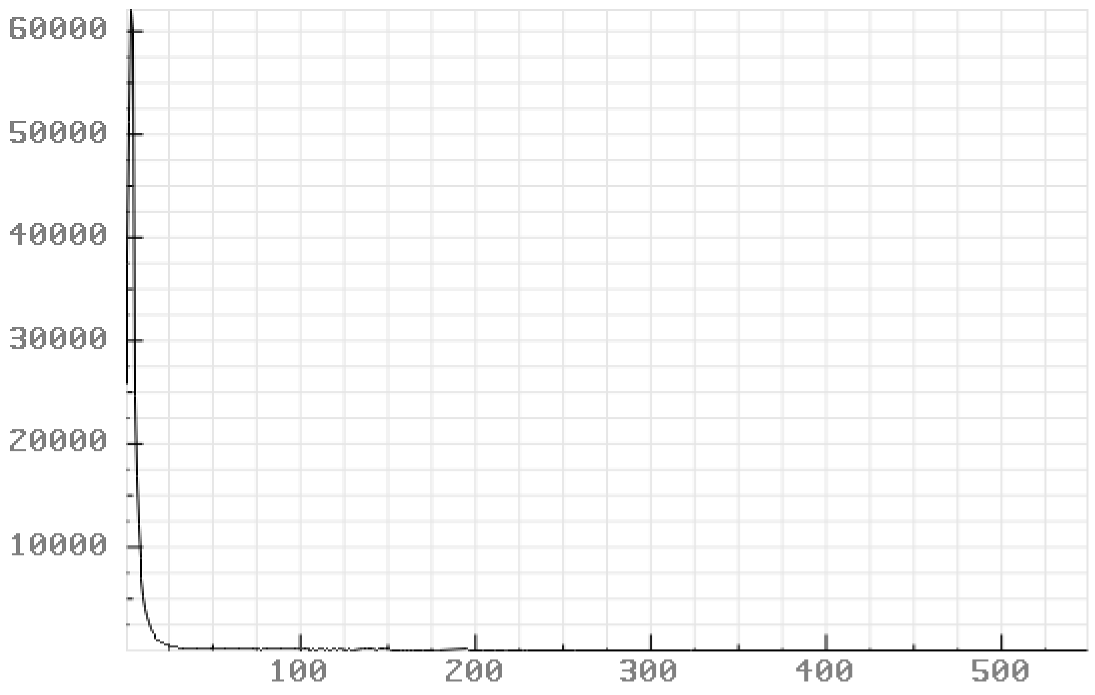

Fig 2.3 Adjacency list Plot for graph 2

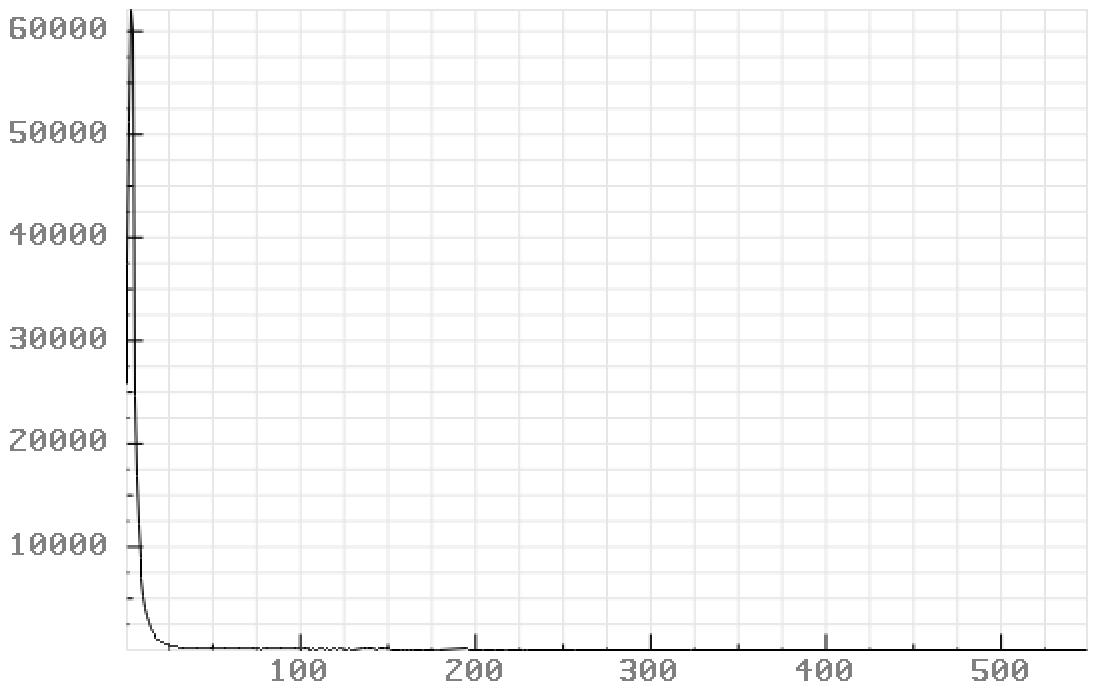

Fig 2.4 CSR Plot for graph 2

Run time Graph1: 0.463s
Run time Graph2: 4.092s

Q3.
In the given question, we need to find the number of 3-cycles and 4-cycles in the graph.
Here the function **countCycles()** in the code plays an important role.
Using DFS we find every possible path of length (n-1) for a particular source (or starting point). 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. Notice that we looked for a path of length (n-1) because the nth edge will be the closing edge of the cycle.
We have to give the argument the number of cycles that we need to count. We have used the **edge_list** and the DFS approach, to check if an edge exists between any two edges.

Run time Graph1: 0.003s
Run time Graph2: 6.800s

Q4.
In the question, it is asked to compute the diameter of the graph.
The Naive algorithm for finding the diameter of an undirected unweighted graph takes O(EV) time where n is the number of vertices in the given graph and m is the number of edges in the given graph. The Naive algorithm finds out the eccentricity of all vertices and diameter is the maximum of all eccentricities.
This algorithm finds out the diameter of the graph by initially starting with an upper bound and lower bound for it. And in each step, this bound is brought closer to find out the exact diameter. Starting from one vertex, at each step the lower bound is updated by computing the eccentricities of vertices at a distance i from u, and the upper bound is decreased by 2. Even though the worst-case Time Complexity is **O(VE),** in practice the algorithm runs in **O(E)** time.

Run time Graph1: 0.119s
Run time Graph2: 10.772s

Q5.
In the question, we need to compute the maximum connected component in the graph. I have used the Depth First Search (DFS) approach, to find the maximum connected component. The function
largestComponent(map<int,vector<int>>& adjList, vector<bool>& visited, int source, int &count)
is the important function to be noted for the question. It also functions as a helper which computes dfs from a node. A size variable is used to increment it whenever the dfs function is called for each of the connected components. The answer is the maximum of all such sizes of all the connected components.
The time complexity of the algorithm is **O(V+E)**.

Run time Graph1: 0.072s
Run time Graph2: 8.113s