Project 3 Report (Phase 2)

Name: Seung Yup Yum

Student ID Number: 76777057 (Odd)

UCI Net ID: syyum

Write Up

Basic Explanation

This project is divided into 2 part: Graph and Graph Algorithm. The graph part is basic implementation of the graph itself. Graph algorithm part is the essence of this project, which calculates the characteristic of the graph. The detailed explanation for each method for each part is explained below.

Graph

Initialization & Structure

There are several ways to implement graph, for example, adjacency list or edge list, but for many parts in this project, adjacency set can provide O(1). Therefore, the graph is constructed as an adjacency set. The adjacency set is basically hash map where the key is node itself, and the associated value to key is the set of the neighbor. Therefore, in the initialization part in class graph, it goes through the edge list, add vertex1 as key and add vertex2 to vertex1's neighbor, and vice versa.

To achieve O(1) for other methods, it also contains num_edge and num_node. The variable num_node is provided with the graph, and num_edge is appropriately increased for each edge in the provided edge list.

Below is other methods in the class graph

get_num_node returns just number of nodes.

get_num_edges returns just number of edges.

get_neighbors returns the node's neighbor. Since the graph structure is adjacency set, it is O(1). The node's neighbor can be gained by doing *graph_srucure[node]*.

select_single_vertex returns one single vertex which has at least one neighbor in the graph. The single vertex is selected randomly.

Get_vertex_list returns the list of the nodes in the graph. The order is not sorted.

Graph Algorithms

This is the essential part to calculate the characteristic of the graph. It has 3 main methods to calculate degree distribution, graph diameter and graph clustering coefficient, and it has 3 additional methods to help 3 main methods.

get_degree_distributuion is one of the main methods that returns the dictionary where the key is degree, and the value is the frequency of the node with that degree. Simply, it goes through the graph structure and fill up the dictionary that will be returned. Each degree can be calculated by doing *len(graph.get_neighbors(vertex))* with O(1).

bfs_max_distance is helper function for calculate the graph's diameter. It returns the farthest node and the max distance. The method does typical breadth first search with using deque, while it also updated the dictionary where the key is node, and the value is the distance from the start vertex to particular node. After the method finish to do breadth first search, It searches dictionary to find max distance and the node that is far from the max distance.

get_diameter is another main methods that returns the diameter of the graph. The approach to get the diameter of the graph is do breadth first search from randomly selected one vertex and do breadth first search again from the farthest node from previous breadth first search. Repeat this steps until the previous breadth first search distance result is equal to current breadth first search distance result. To follow this approach, it first gets one randomly selected vertex by using *select_single_vertex()*. This is now the start vertex. With this start vertex, it calls *bfs_max_distance()* until the *current_max_distance* is equal to previous *max_distance*. Whenever *bfs_max_distance()* is done, the result node is now new start vertex.

number_of_triangle is one of the helper methods to calculate graph's clustering coefficients. In order to achieve O(n + (degeneracy * degeneracy)), the process required to produce degeneracy ordering. The actual required structure is *later_neighbor*, not degeneracy ordered list, but the *later_neighbor* structure can only be made during the process of generating degeneracy list, the exact step of degeneracy ordered list is required.

Since the actual degeneracy ordered list is not useful, this method uses hash set to store all visited node that is technically popped out from the graph and append to the degeneracy ordered list. The main reason to use hash set is *difference()* function that automatically get the difference between two sets.

With the same reason, it has mainly two hash map, called *degree_vertex_map* and *vertex_degree_map*. In the *degree_vertex_map*, the key is degree, and value is set that contains all vertices with its degree is equal to the key. In *vertex_degree_map* is another data structure to keep tracking each vertex's degree, so the key is vertex, and the value is its current degree. Another main reason to use hash map is, it is easy to get the minimum key in the current hash map. It helps a lot to get minimum key easily without having another data structure.

Another data structure, and the only data structure that is used to calculate the number of triangles is *later_neighbor*. It is hash map where the key is vertex, and the value is set of neighbor that will appear in the degeneracy ordered list. Again, the reason for using set as value is that the difference between visited set and neighbor set is easily obtained by using *difference()*.

In this method, firstly, it goes through the graph structure, initialize *degree_vertex_map* and *vertex_degree_map*. After it initializes both hash maps, it enters the loop to generate degeneracy ordered list. The loop iterates until the length of visited set is equal to number of node in the graph, which means that all the nodes it popped out from the graph and inserted into degeneracy ordered list. In the loop, for each iteration, it gets the minimum key of *degree_vertex_map* and pop out the vertex that is in the *degree_vertex_map [minimum degree]*, called *selected_vetex*. After vertex is selected, it gets its neighbor in graph. Since the already popped out nodes, which are in the visited set, are not useful to get the *later_neighbor*, it only considers elements that are in the neighbor but not in the visited. Store them in the set called later_neighbor_set. For each vertex in the *later_neighbor_set*, it updates the current vetex in the *vertex_degree* and updates *degree_vertex* accordingly. After it updates two hash maps, it updates *later_neighbor* hash map, simply *later_neighbor[selected_vertex] = later_neighbor_set*, and the selected vertex is added to visited set.

After this loop, it generates *later_neighbor* hash map, where the key is vertex, and value is all neighbors that will be appeared in later in degeneracy ordered list. Now, the number of triangle can be calculated. It goes all vertex in graph, get the *later_neighbor_set* by doing *later_neighbor[v]*. For each vertex1 and vertex 2 in *later_neighbor_set*, count if vertex1 and vertex2 are neighbor. After finishing counting, divided by 2.

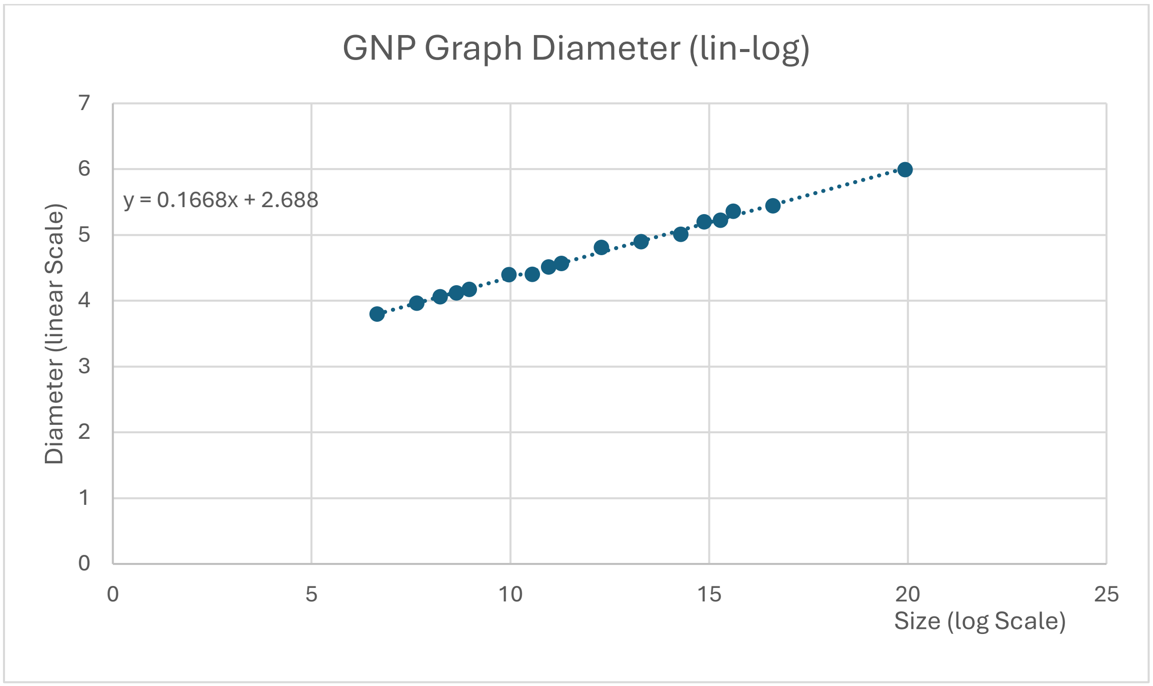
two_edge_path is another helper function to obtain the clustering coefficient of the graph. It simply gets the sum of ((degree * (degree – 1)) / 2 for each vertex in the graph.

get_clustering_coefficient is the last main method. It calls *number_of_triangle* and *two_edge_path*, and returns 3 * *number_of_triangle / two_edge_path*.

Constraints (Diameter & Clustering Coefficient)

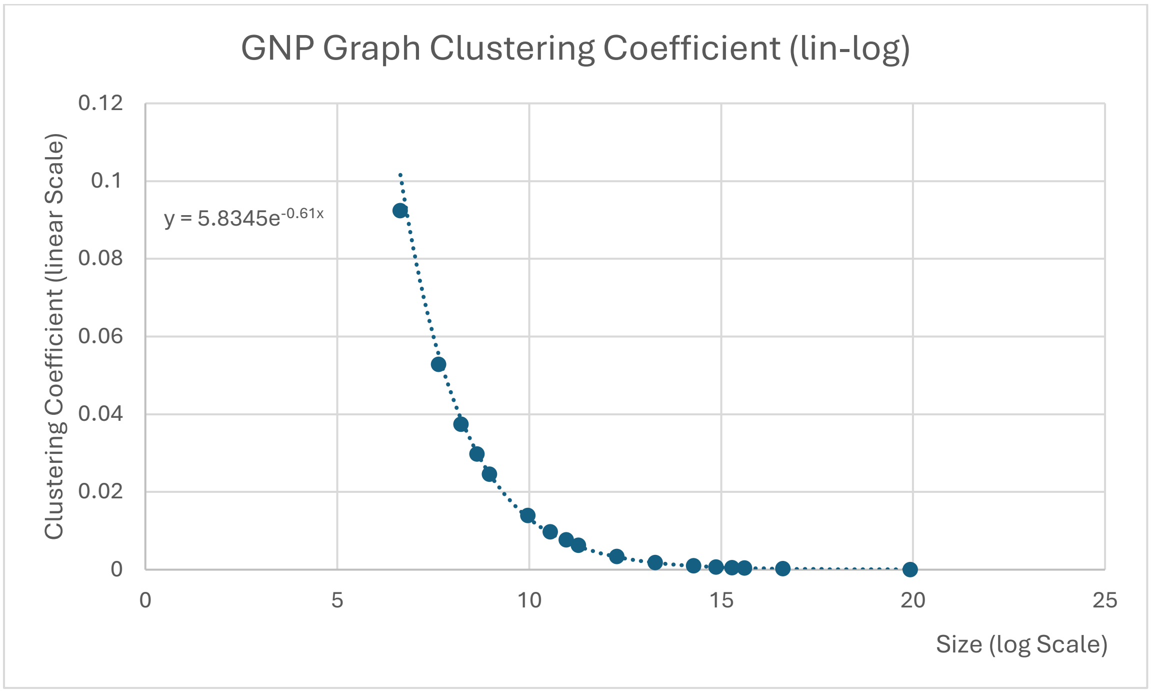
- Based on the student ID number, the graph is made by Eros Renyi Algorithm, called GNP.
- The size of input (number of nodes) for diameter and clustering coefficient experiment is [100, 200, 300, 400, 500, 1000, 1500, 2000, 2500, 5000, 10000, 20000, 30000, 40000, 50000, 100000, 1000000].
- The total iteration is 1200.
- For each iteration, the input is newly generated.
- After iteration, the average diameter and clustering coefficient is calculated, and plot it.
- The size is in the log scale with base 2, and the diameter is linear scale.
- Making plot and the trendline is done in Excel.

Plotting Result (Diameter)



By iterating 1200 times and averaging them, the result of the graph is linear with a slope of 0.1668. It can be seen that the diameter grows at a fairly constant rate as the size of the graph increases in the lin-log scale. Therefore, it grows proportional to the function $\log n$.

Plotting Result (Clustering Coefficient)



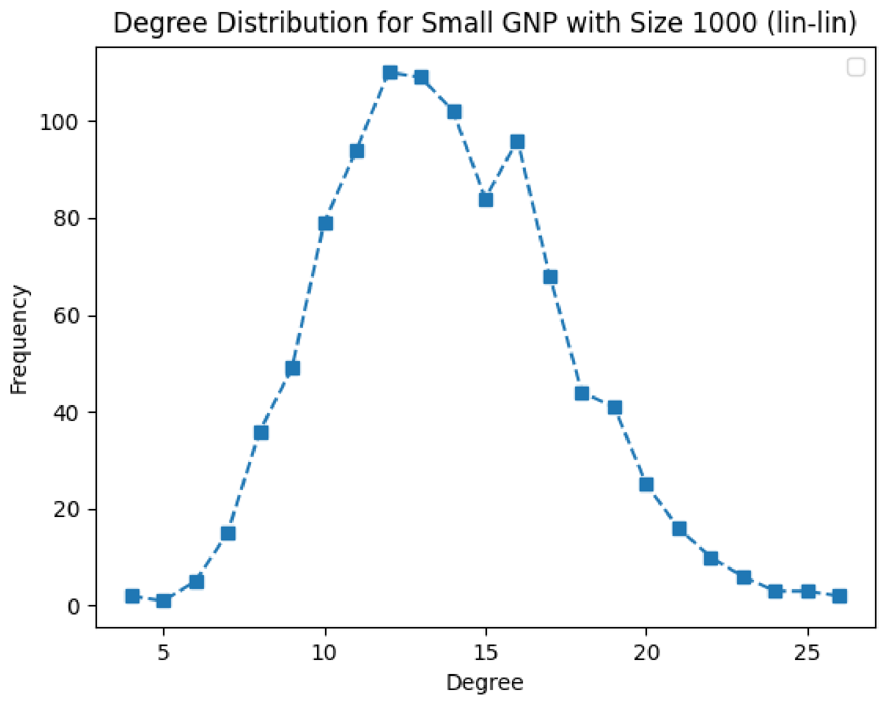
On the other hand, the clustering coefficient is not linear in line-log, which means that it does not grow proportional to the function $\log n$, and the clustering coefficient values decrease a function of n in lin-log scale.

Constraints (Degree Distribution)

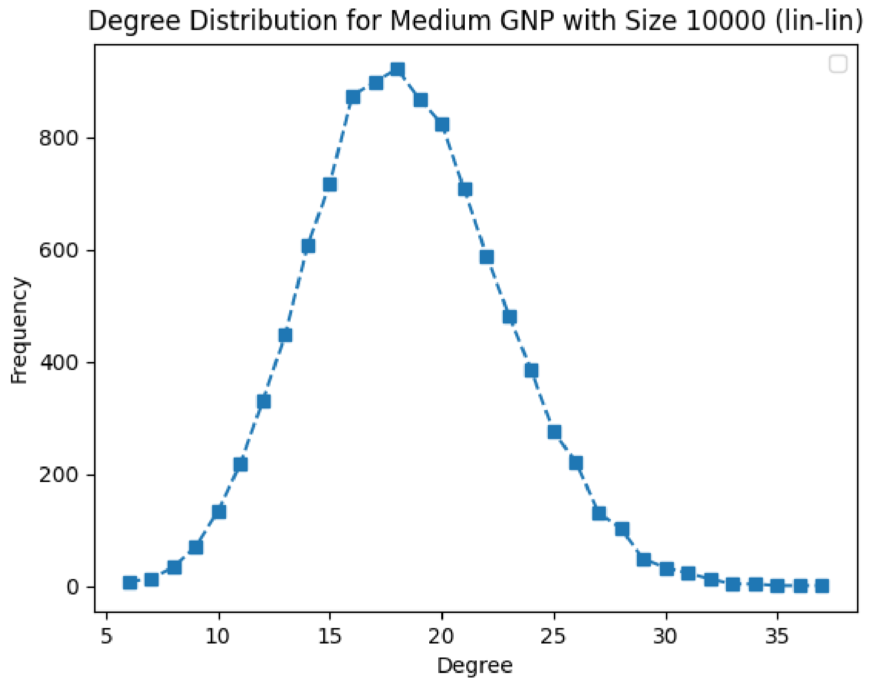
- Based on the student ID number, the graph is made by Eros Renyi Algorithm, called GNP.
- Size is 1000, 10000, 100000 for small, medium, large graph respectively.
- The graph making is done through matplotlib.pyplot and numpy.

Plotting Degree Distribution (lin-lin Scale)

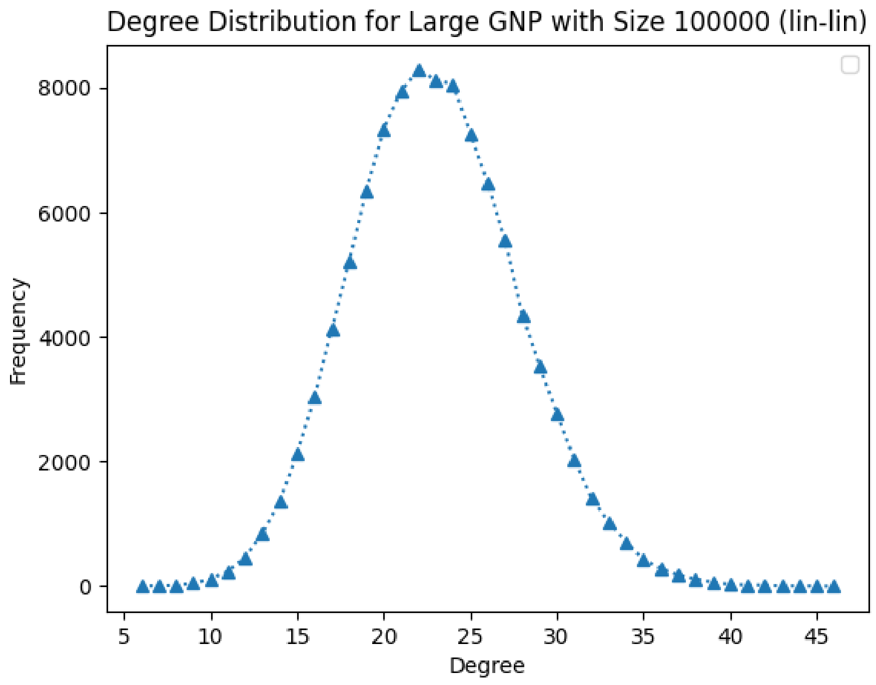
Size 1000 (lin -lin)



Size 10000 (lin-lin)

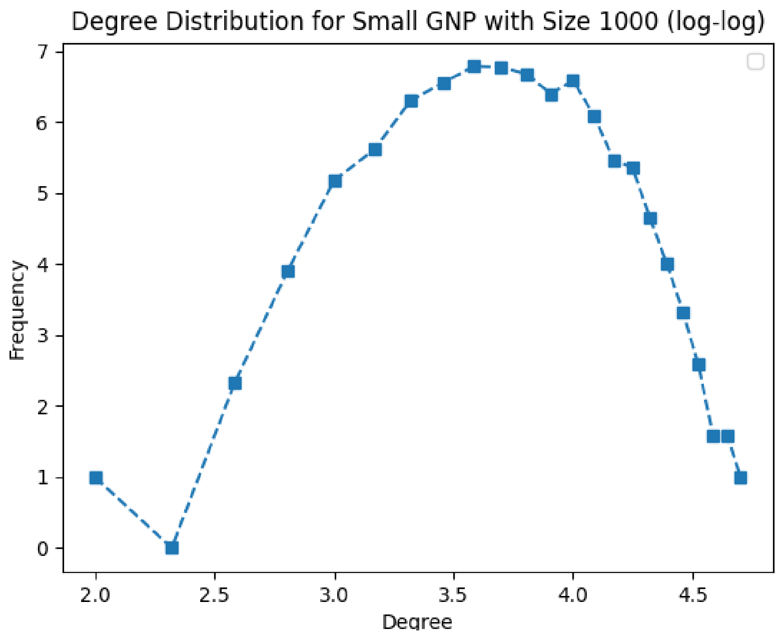


Size 100000 (lin-lin)

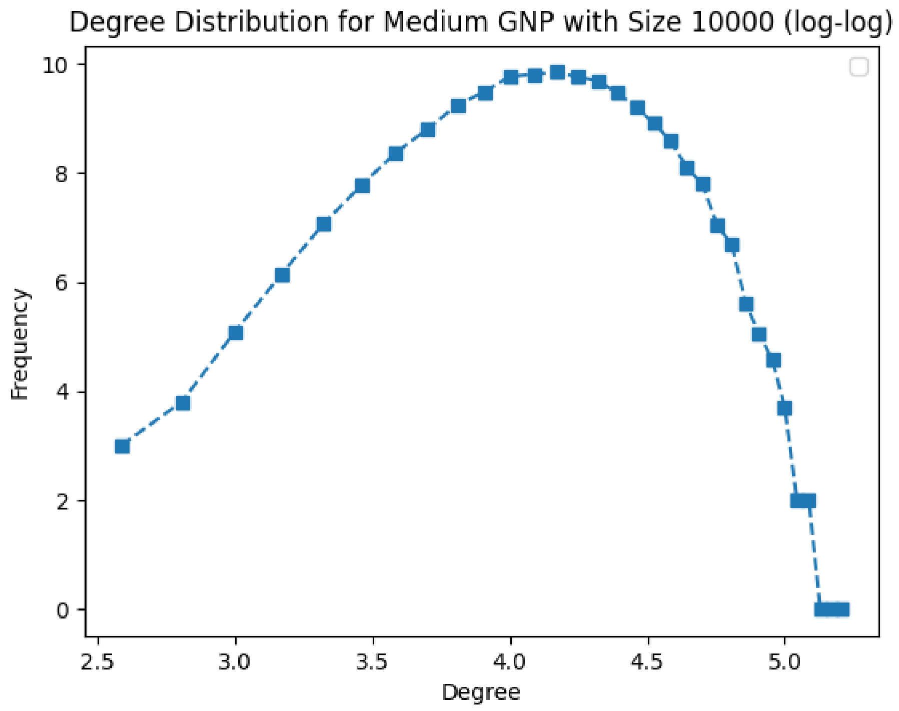


Plotting Degree Distribution (log-log Scale)

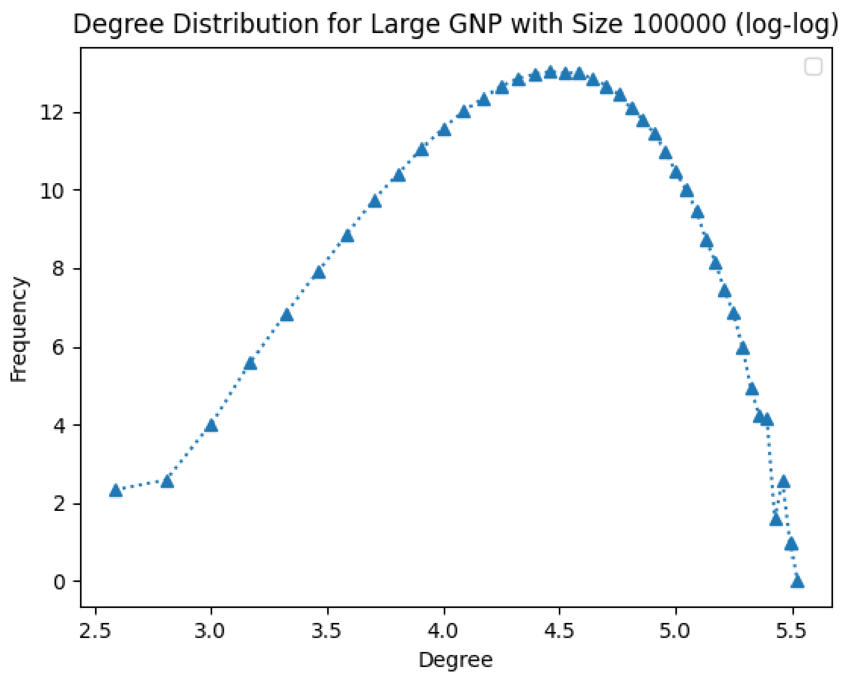
Size 1000 (log-log)



Size 10000 (log-log)



Size 100000 (log-log)



Conclusion: Based on the result graph, the degree distribution of Eros Renyi Graph does not have power law.

Code Explanation for implementing Eros Renyi Graph

Input: number of nodes in the graph

Output: A list of edges representing the graph

Pseudo code in English:

```
Initialization:

  • Initialize probability. $p = (2 * \ln n) / n$
  • Initialize output list, called edge.
  • Set $v = 1$, and $w = -1$.

Main Loop Body (while $v <$ number of nodes):

  • Draw a random number r uniformly between 0 and 1.
  • Compute the "skip step" for target node by doing:
      o $w = w + 1 + \text{floor}( \log (1 - r) / \log (1 - p) )$
  • Adjust w if it exceeds v by doing:
      o while $v \leq w$ and $v <$ number of nodes:
          ▪ $w = w - v$ (calculate next offset)
          ▪ Increment v by 1 (move to the next source node)
  • If $v <$ number of nodes, add edge (v, w) to edges.

Return the edge list.
```