МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ
ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ
**«БЕЛГОРОДСКИЙ ГОСУДАРСТВЕННЫЙ
ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ им. В. Г. ШУХОВА»
(БГТУ им. В.Г. Шухова)**

Кафедра программного обеспечения вычислительной техники и автоматизированных систем

Лабораторная работа №8
по дисциплине «Теория надежности»
тема: «Анализ живучести сетей»

Выполнил: ст. группы ВТ-32
Черных Воскобойников И.
Проверил: Кабалянц П.С.

Белгород 2021 г.

Данные варианта

Построить "малый мир" из 50 вершин, используя биномиальный алгоритм имитации случайного графа с параметром p=i/(i+j+k); если p<1/2, то добавить 1/2. Достроить его до 1000 вершин, используя правило предпочтения Альберты-Барабаши. Для построенных графов найти степени вершин и вывести графики зависимости числа вершин от степени (полигоны частот для степеней вершин в сети). Найти коэффициенты кластеризации каждой вершины и вывести полигоны частот для коэффициента кластеризации.
Здесь

i, j и k определяются по номеру студенческого также, как и в предыдущих заданиях.

$i = 1, j = 0, k = 0$

Выполнение работы

```
import networkx as nx
import matplotlib.pyplot as plt
import random as rand
class Graph:

    def __init__(self, nodes_count: int):
        self.edges = []
        self.powers = {node: 0 for node in range(nodes_count)}
        self.claster_coefs = {node: 0 for node in range(nodes_count)}

    def add_node(self):
        self.powers[self.get_nodes_count()] = 0

    def add_edge(self, a, b):
        if a < b:
            self.edges.append([a, b])
        else:
            self.edges.append([b, a])
        self.powers[a] += 1
        self.powers[b] += 1

    def random_add_edge(self, p: float, a, b):
        if rand.random() < p:
            self.add_edge(a, b)

    def get_nodes_count(self):
        return len(self.powers)

    def get_power(self, node) -> int:
        return self.powers[node]

    def get_sum_of_powers(self) -> int:
        return sum(self.powers)


    def get_claster_coef(self, node) -> None:

            node neighbours = list()

            for edge in self.edges:
              if edge[0] == node:
                 node_neighbours.append(edge[1])
              elif edge[1] == node:
                 node_neighbours.append(edge[0])

            neighbours_count = len(node_neighbours)
            max_count = neighbours_count*(neighbours_count - 1)/2
            if max_count == 0:
```

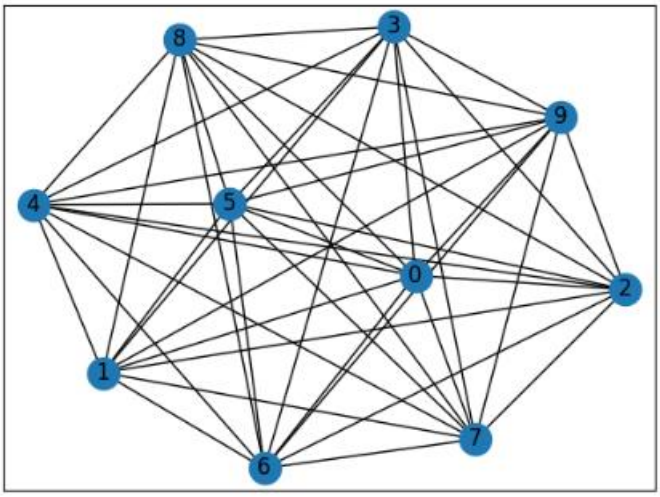
```
                return None

            current_count = 0
            for edge in self.edges:
                if edge[0] == node or edge[1] == node:
                  continue
                elif edge[0] in node_neighbours and edge[1] in node_neighbours:
                 current_count += 1
            claster_coef = current_count/max_count

            self.claster_coefs[node] = claster_coef
            return claster_coef

    def calculate_claster_coefs(self):
        nodes = self.get_nodes_count()
        for node in range(nodes):
            self.get_claster_coef(node)


# Выводит изображения графа.
    def visualize(self):
        graph = nx.Graph()
        graph.add_edges_from(self.edges)
        nx.draw_networkx(graph)
        plt.show()

    @staticmethod
    def random_graph(nodes_count: int, p: float):
        graph = Graph(nodes_count)

        for i in range(nodes_count - 1):

            for j in range(i + 1, nodes_count):
                graph.random_add_edge(p, i, j)
        return graph

    def do_build(self, n_end: int):
        n = len(self.powers)
        for new_node in range(n, n_end):
            sum_of_powers = self.get_sum_of_powers()
            self.add_node()
            for node in range(new_node):
                p add = self.get power(node) / sum of powers
                self.random_add_edge(p_add, new_node, node)
        self.edges.sort(key=lambda edge: edge[0])

def out_powers_func(graph: Graph, n_end: int):

    y_dict = { power: 0 for power in range(n_end) }
    for (node, power) in graph.powers.items():
        y_dict[power] += 1
    for power in range(len(y_dict) - 2, -1, -1):
        y_dict[power] += y_dict[power + 1]
    y_dict.pop(0)

    flag = True
    while flag:
        to_new = False
        for power in y_dict.keys():
            if y_dict[power] == 0:
                y_dict.pop(power)
                to_new = True
                break
        if not to_new:
```

```
                flag = False
        plt.plot(y_dict.keys(), y_dict.values())
        plt.show()

def out_clasters_func(graph: Graph, n_end: int):

        y_dict = dict()
        for (node, coef) in graph.claster_coefs.items():
            y_dict[coef] = 1
        keys_list = sorted(list(y_dict.keys()))
        for coef in range(len(y_dict) - 2, -1, -1):
            y_dict[keys_list[coef]] += y_dict[keys_list[coef + 1]]

        plt.plot(keys_list, [y_dict[key] for key in keys_list])
        plt.show()
if __name__ == '__main__':
        i = 1
        j = 0
        k = 0

        n = 10

        n_end = 1000

        p = i/(i + j + k)
        if p < 1/2:
            p += 1/2

        graph = Graph.random_graph(n, p)
        graph.calculate_claster_coefs()
        out_powers_func(graph, n)
        out_clasters_func(graph, n)
        graph.visualize()

        graph.do_build(n_end)
        graph.calculate_claster_coefs()
        out_powers_func(graph, n_end)
        out_clasters_func(graph, n_end)
        graph.visualize()
```

Результаты работы программы:

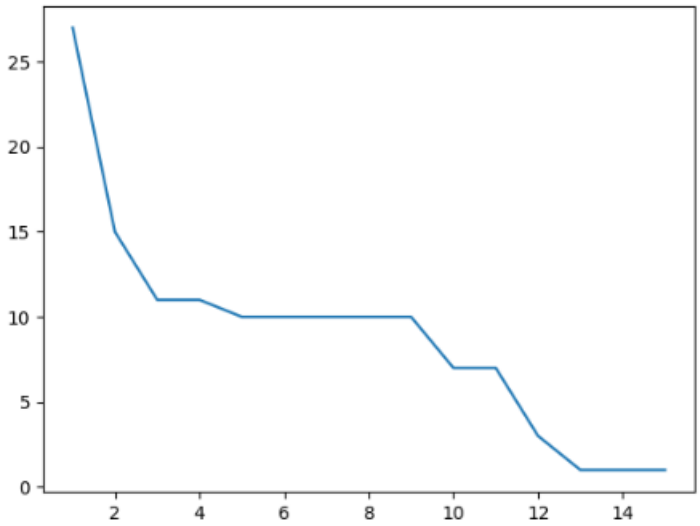
При n = 10

1. Граф:



При n = 1000

Зависимость числа вершин от степени:



Зависимость числа вершин от коэффициента кластеризации: