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

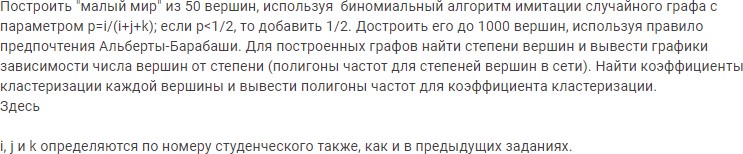
Лабораторная работа №8

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



i = 1, j = 0, k = 0

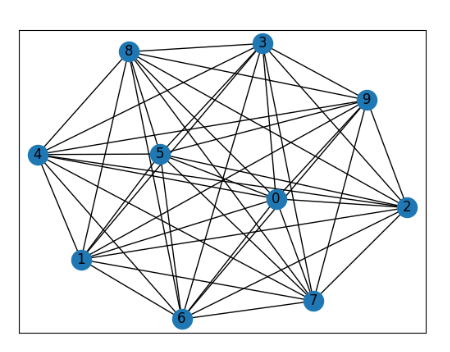
# Выполнение работы

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:

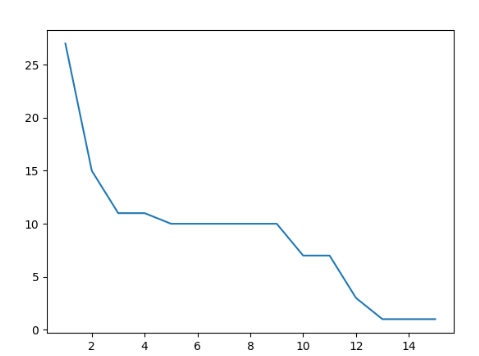
# При n = 10

1. Граф:



# При n = 1000

Зависимость числа вершин от степени:



Зависимость числа вершин от коэффициента кластеризации:

