



МИНОБРНАУКИ РОССИИ
федеральное государственное бюджетное образовательное
учреждение высшего образования
«САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭКОНОМИЧЕСКИЙ УНИВЕРСИТЕТ»
(СПбГЭУ)

Факультет информатики и прикладной математики
Кафедра прикладной математики и экономико-математических методов

ОТЧЕТ
по производственной практике

Наименование организации прохождения практической подготовки:
СПбГЭУ

Направление: 01.03.02 Прикладная математика и информатика

Направленность:
Прикладная математика и информатика в экономике и управлении

Обучающийся: Титилин Александр Михайлович

Группа: ПМ-2201

Подпись _____

Руководитель по практической подготовке от СПбГЭУ:
Салина Татьяна Константиновна, кандидат экономических наук, доцент
кафедры прикладной математики и экономико-математических методов

(подпись руководителя)

Оценка по итогам защиты отчета

Санкт-Петербург
2025 г.



МИНОБРНАУКИ РОССИИ
федеральное государственное бюджетное образовательное
учреждение высшего образования
«САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭКОНОМИЧЕСКИЙ УНИВЕРСИТЕТ»
(СПбГЭУ)

СОГЛАСОВАНО:

Руководитель по практической подготовке от
профильной организации

УТВЕРЖДАЮ:

Заведующий кафедрой прикладной
математики и экономико-математических
методов

Фридман Григорий Морицович

(подпись)

М.П.

(подпись)

« 19 » мая 2025 г.

Индивидуальное задание
на производственную практику
(научно-исследовательскую работу)

Обучающегося: 3 курса Титилина Александра Михайловича _____

Направление: 01.03.02 Прикладная математика и информатика _____

Направленность:

Прикладная математика и информатика в экономике и управлении _____

Наименование организации прохождения практической подготовки:
СПбГЭУ _____

Сроки практической подготовки с 21.05.2025 по 18.06.2025 г. _____

Руководитель по практической подготовке от СПбГЭУ

Салина Татьяна Константиновна, кандидат экономических наук, доцент
кафедры прикладной математики и экономико-математических методов

**Совместный рабочий график
с указанием видов работ,
связанных с будущей профессиональной деятельностью**

№ п/п	Перечень заданий, подлежащих разработке	Календарные сроки (даты выполнения)
1.	Ознакомление с правилами внутреннего распорядка на предприятии, ЛНА, прохождение инструктажа по технике безопасности и охране труда	21.05.2025
2.	Ознакомление с аналитическими задачами, решаемыми подразделением. Согласование с руководителем практики от предприятия индивидуального задания на практику.	21.05.2025–22.05.2025
3.	Изучение научных источников, сбор и обобщение информации по теме исследования	22.05.2025–27.05.2025
4.	Сбор данных по теме исследования. Реализация основных алгоритмов.	27.05.2025–01.06.2025
5	Реализация и тестирование алгоритмов генерации случайных графов.	01.06.2025–07.06.2025
6	Проектирование и разработка графического интерфейса пользователя.	07.06.2025–16.06.2025
7	Обобщение материалов и подготовка отчёта по результатам практики	16.06.2025–18.06.2025

С заданием ознакомлен _____

(подпись обучающегося)

Руководитель по практической подготовке от СПбГЭУ

_____ Салина Т.К.

(подпись)

Руководитель по практической подготовке от профильной организации

_____ (подпись)

Обучающийся прошел инструктаж по ознакомлению с требованиями охраны труда, техники безопасности, пожарной безопасности, а также с правилами внутреннего распорядка. Вводный инструктаж и инструктаж на рабочем месте пройдены с оформлением установленной документации.

Руководитель по практической подготовке от организации/профильной организации назначен приказом № _____ дата _____ и соответствует требованиям трудового законодательства Российской Федерации о допуске к педагогической деятельности.

_____ (подпись)

Содержание

ВВЕДЕНИЕ	5
1 ГРАФЫ. ОСНОВНЫЕ ОПРЕДЕЛЕНИЯ. РЕАЛИЗАЦИЯ АЛГОРИТМОВ. ПОДСЧЕТ ХАРАКТЕРИСТИК ГРАФА.	7
1.1 Основные определения	7
1.2 Реализация графа.	7
1.3 Реализация алгоритмов на графах.	8
1.4 Вычисление характеристик графа	10
1.4.1 Плотность сети	11
1.4.2 Диаметр графа	11
1.4.3 Коэффициенты кластеризации	12
1.4.4 Распределение степеней.	14
1.4.5 Степень близости.	14
2 ВИЗУАЛИЗАЦИЯ ГРАФОВ	16
3 ГЕНЕРАЦИЯ СЛУЧАЙНЫХ ГРАФОВ	18
3.1 Модель Эрдеша-Ренье	18
3.2 Модель Барабаши-Альберт	20
3.3 Модель Боллобаша-Риордана	21
3.4 Модель LCD	22
3.5 Модель копирования	24
3.6 Модель Чунг-Ли	26
3.7 Генерация случайного геометрического графа	28
3.7.1 Эффективный алгоритм генерации геометрических графов.	28
3.7.2 Реализация алгоритма генерации геометрических графов	29

ВВЕДЕНИЕ

Целью данной работы является разработка графического приложения пользователя для визуализации и анализа графов.

Задачи работы:

- 1) разработка представления графа и базовых алгоритмов на графах;
- 2) изучение и разработка алгоритмов генерации случайных графов;
- 3) создание визуализации графов и работы алгоритмов;
- 4) разработка графического приложения для визуализации работы разработанных алгоритмов;

1. ГРАФЫ. ОСНОВНЫЕ ОПРЕДЕЛЕНИЯ. РЕАЛИЗАЦИЯ АЛГОРИТМОВ. ПОДСЧЕТ ХАРАКТЕРИСТИК ГРАФА.

1.1 Основные определения

Для того, чтобы реализовать граф, необходимо дать определение данного понятия. Пусть задано конечное множество вершин $V = \{v_1 \dots v_n\}$ и множество ребер $E = \{e_1 \dots e_m\}$, где $e_k = \{v_i, v_j\}$. Пару $G = (V, E)$ назовем графом. Если $E = \{e_1^{i_1} \dots e_m^{i_k}\}$ мультимножество ребер, то это мультиграф. Две вершины $v_1, v_2 \in V$ называются смежными, если $\{e_1, e_2\} \in E$.

1.2 Реализация графа.

Рассмотрим реализацию графов с помощью языка программирования Python. Был создан пакет «graph_lib», который содержит реализованные классы для представления графа, визуализации и генерации случайных графов.

Класс «Graph» для реализации графа содержится в модуле «graph.py». Рассмотрим поля данного класса:

- 1) `_adjacency_list` – список смежности графа. Представлен с помощью словаря, в котором ключи – это вершины, а значения – это множества вершин, которым данная вершина смежна;
- 2) `_verticies` – множество вершин графа;
- 3) `_k_edges` – количество ребер в графе;

```
def __init__(self):
    self._adjacency_list = defaultdict(set)
    self._verticies = set()
    self._k_edges = 0
```

Рисунок 1: Инициализация графа.

На рисунке 1 представлена функции инициализации пустого графа. Добавление ребра определяется следующим образом:

$$(V, G) \rightarrow (V \cup \{v, u\}, E \cup \{\{v, u\}\}) \quad (1)$$

Формула 1 легко переносится на язык программирования Python

```
def add_vertex(self, vertex):  
    self._verticies.add(vertex)
```

Рисунок 2: Добавления вершины в граф.

```
def add_edge(self, a, b):  
    self._adjacency_list[a].add(b)  
    self._adjacency_list[b].add(a)  
    self.add_vertex(a)  
    self.add_vertex(b)  
    self._k_edges += 1
```

Рисунок 3: Добавление ребра в граф.

На рисунках 2, 3 приведены методы, которые добавляют в граф вершину и ребро. Данные методы обобщаются на список вершин или ребер.

1.3 Реализация алгоритмов на графах.

Для успешного анализа графов необходимо реализовать некоторые базовые алгоритмы. Рассмотрим алгоритмы обхода графа. Начнем с поиска в ширину.

```

 $Q \leftarrow \{s\}$ 
visited =  $\emptyset$ 
while  $Q \neq \emptyset$ 
     $v = \text{pop}(Q)$ 
    yield  $v$ 
    for  $\{u, v\}$  in  $E$ 
        if  $u \notin \text{visited}$ 
             $\text{push}(Q, u)$ 
             $\text{visited} = \text{visited} \cup \{v_1\}$ 
        end if
    end for
end while

```

Рисунок 4: Псевдокод алгоритма обхода в ширину.

```

def bfs(self, start=1):
    if len(self._vertices) > 0:
        visited = {start}
        queue = deque([start])
        while queue:
            a = queue.popleft()
            yield a
            for b in self._adjacency_list[a]:
                if b not in visited:
                    visited.add(b)
                    queue.append(b)

```

Рисунок 5: Реализация обхода в ширину на языке программирования Python.

На рисунках 4 , 5 приведен алгоритм обхода графа в ширину

Данный алгоритм будет использоваться для поиска кратчайших путей в графе и поиске компонент связности.


```
def dist(self, v, g):
    d = defaultdict(lambda: None)
    d[v] = 0
    for u1 in self.bfs(v):
        for u2 in self.neib(u1):
            if d[u2] is None:
                d[u2] = d[u1] + 1
    return d[g]
```

Рисунок 6: Поиск кратчайшего расстояния между v и g

```
def connected_components(self):
    visited = set()
    components = []
    for v in self.vertices():
        if v not in visited:
            component = []
            for u in self.bfs(v):
                visited.add(u)
                component.append(u)
            components.append(component)
    return components
```

Рисунок 7: Поиск компонент связности.

На рисунках 6, 7 представлены методы, которые используют поиск в ширину.

Так же аналогично реализован поиск в глубину и алгоритм поиска мостов, основанный на нем.

1.4 Вычисление характеристик графа

Были разработаны методы для вычисления следующих характеристик графа:

- 1) плотность сети;
- 2) диаметр графа;
- 3) среднее кратчайшее расстояние;
- 4) коэффициент кластеризации;

- 5) локальный коэффициент кластеризации;
- 6) средний коэффициент кластеризации;
- 7) распределение степеней вершин;
- 8) степень близости вершины;

Рассмотрим каждую характеристику более подробно

1.4.1 Плотность сети

Плотность сети – отношение количества ребер к максимально возможному, $\frac{n(n-1)}{2}$, где n – количество вершин графа.

```
def density(self):  
    max_edges = self.k_verticies() * (self.k_verticies() - 1) // 2  
    return self._k_edges / max_edges
```

Рисунок 8: Метод для вычисления плотности графа.

На рисунке 8 представлен метод для вычисления плотности сети.

1.4.2 Диаметр графа

Диаметр графа – максимальное расстояние между парами вершин. Вычисляется следующим образом, необходимо пройти из каждой вершины поиском в ширину, выбрать вершину, обход из которой пометил больше всего вершин.

```
def diameter(self):  
    diam_lst = [len(tuple(self.bfs(v))) - 1 for v in self._verticies]  
    return max(diam_lst)
```

Рисунок 9: Реализация вычисления диаметра графа.

На рисунке 9 представлен метод для вычисления диаметра графа.

1.4.3 Коэффициенты кластеризации

Введем следующие понятия:

- 1) треугольник – граф состоящий из 3 вершин, степень каждой 2;
- 2) вилка – граф состоящий из 3 вершин, степень двух вершин 1, другой 2. Вершина степени 2 называется центром вилки;

Рассмотрим коэффициенты кластеризации:

- 1) коэффициент кластеризации $\frac{3 \cdot \text{количество треугольников в графе}}{\text{количество вилок в графе}}$;
- 2) локальный коэффициент кластеризации для вершины v – $\frac{\text{число треугольников с вершиной } v}{\text{число вилок с центром } v}$;
- 3) средний коэффициент кластеризации – среднее арифметическое локальных коэффициентов кластеризации;

Для вычисления данных коэффициентов нужно реализовать методы поиска числа вилок и треугольников

```
def k_triangles(self):
    triplets = set()
    k = 0
    for v in self._vertices:
        if self.deg(v) >= 2:
            for v1 in self.neib(v):
                for v2 in self.neib(v):
                    triplet = frozenset((v, v1, v2))
                    if self.has_edge(v1, v2) and triplet not in triplets:
                        k += 1
                        triplets.add(triplet)
    return k
```

Рисунок 10: Вычисление количества треугольников в графе.

```
def k_local_triangles(self, v):
    k = 0
    if self.deg(v) >= 2:
        for v1 in self.neib(v):
            for v2 in self.neib(v):
                if v1 < v2 and self.has_edge(v1, v2):
                    k += 1
    return k
```

Рисунок 11: Вычисление количества треугольников, содержащих вершину v .

На рисунках 10 , 11 представлены методы для вычисления числа треугольников в графе.

```
def k_local_forks(self, v):
    k = 0
    if self.deg(v) >= 2:
        for v1 in self.neib(v):
            for v2 in self.neib(v):
                if v1 < v2 and not self.has_edge(v1, v2):
                    k += 1
    return k
```

Рисунок 12: Вычисление вилок с центром в вершине v

```
def k_forks(self):
    k = 0
    for v in self._verticies:
        k += self.k_local_forks(v)
    return k
```

Рисунок 13: Вычисление числа вилок в графе

На рисунках 12, 13 представлены методы для вычисления числа вилок.

```
def cluster_k(self):
    if self.k_forks() == 0:
        return 0
    return 3 * self.k_triangles() / self.k_forks()
```

Рисунок 14: Вычисление коэффициента кластеризации.

```
def local_cluster_k(self, v):
    if self.k_local_forks(v) > 0:
        return self.k_local_triangles(v) / self.k_local_forks(v)
    return 0
```

Рисунок 15: Вычисление локального коэффициента кластеризации.

```
def mean_cluster_k(self):
    return sum(self.local_cluster_k(v) for v in self._verticies) / self.k_verticies()
```

Рисунок 16: Вычисление среднего коэффициента кластеризации

На рисунках 14,15,16 представлены методы для вычисления коэффициентов кластеризации.

1.4.4 Распределение степеней.

Необходимо для всех степеней вершин найти долю вершин, которые имеют данную степень.

```
def deg_distribution(self):
    result = []
    t = namedtuple("DegDistrNode", ("k", "p"))
    for d in set(self.deg_list()):
        k_d = len([1 for v in self.verticies() if self.deg(v) == d])
        result.append(t(d, k_d/self.k_verticies()))
    return result
```

Рисунок 17: Вычисление распределения степеней.

На рисунке 17 приведен метод для вычисления распределения степеней.

1.4.5 Степень близости.

Степень близости вершины v — $C(v) = \frac{n-1}{\sum_u d(u,v)}$ где n — количество вершин в графе, $d(u, v)$ — кратчайшее расстояние от u до v .

```
def closeness(self, v):  
    d = [self.dist(v, u)  
          for u in self.verticies() if self.dist(v, u) is not None]  
    if sum(d) == 0:  
        return 0  
    return (self.k_verticies() - 1) / sum(d)
```

Рисунок 18: Вычисление степени близости вершины v .

На рисунке 18 приведен метод для вычисления степени близости вершины v .

2. ВИЗУАЛИЗАЦИЯ ГРАФОВ

Визуализация была разработана с помощью сторонней библиотеки «matplotlib».

Для создания диаграмм графов был разработан класс «GraphPlotter». Данный класс содержит следующие поля:

- 1) «graph» – граф, визуализация которого создается;
- 2) «coords» – словарь, где ключ – это вершина, а значение координата;
- 3) «orange_edges» – список ребер, которые должны быть оранжевыми;

```
def __init__(self, g: Graph | GeoGraph | MultiGraph, orange_edges=[]):
    self.graph: Graph | GeoGraph | MultiGraph = g
    self.coords = {}
    self.gen_coords(g)
    self.orange_edges = orange_edges
```

Рисунок 19: Инициализация объекта типа «GraphPlotter».

Рассмотрим генерацию координат для вершины v_i

$$\begin{cases} r_i = \frac{2 \cdot i \cdot \pi}{n} \\ x_i = \cos r_i \\ y_i = \sin r_i \end{cases} \quad (2)$$

Формула 2 задает укладку вершин графа по окружности

```
def __gen_coords(self, g: Graph | MultiGraph):
    for i, v in enumerate(g.vertices()):
        r = 2 * i * pi / self.graph.k_verticies()
        self.coords[v] = self.point(cos(r), sin(r))
```

Рисунок 20: Генерация словаря «coords»

На рисунке 20 представлен метод для создания координат вершин.

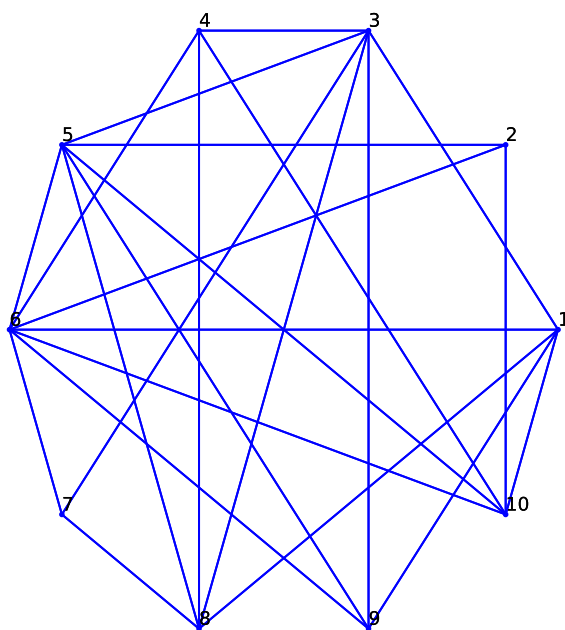


Рисунок 21: Пример диаграммы графа.

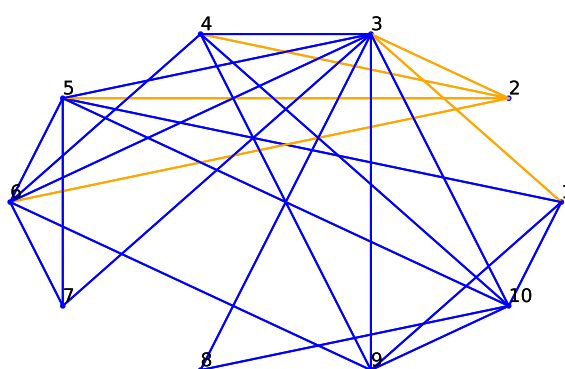


Рисунок 22: Пример диаграммы графа с оранжевыми ребрами.

На рисунках 21, 22 представлены диаграммы графов, созданные с помощью «GraphPlotter».

3. ГЕНЕРАЦИЯ СЛУЧАЙНЫХ ГРАФОВ

Для многих задач требуется генерация случайного графа. Будут рассмотрены разные модели генерации случайного графа. Каждая модель представлена отдельным модулем.

3.1 Модель Эрдеша-Ренье

Рассмотрим простейшую модель генерации случайного графа. Даны $n \in \mathbb{N}$, $p \in (0, 1)$. Создается полный граф с n вершинами, каждое ребро берется с вероятностью p .

```
def complete_graph_edges(k):  
    edges = []  
    for a in range(1, k+1):  
        for b in range(a+1, k+1):  
            edges.append((a, b))  
            edges.append((b, a))  
    return edges
```

Рисунок 23: Генерация списка ребер полного графа с k вершинами

```
def erdos_renyi_random_graph(n, p):  
    edges = [e for e in ErdosRenyiGraph.complete_graph_edges(  
        n) if random() < p]  
    g = Graph()  
    g.add_edges(edges)  
    g.add_vertices(range(1, n+1))  
    return g
```

Рисунок 24: Генерация случайного графа по модели Эрдеша-Ренье

На рисунках 23,24 приведен код для генерация случайного графа.

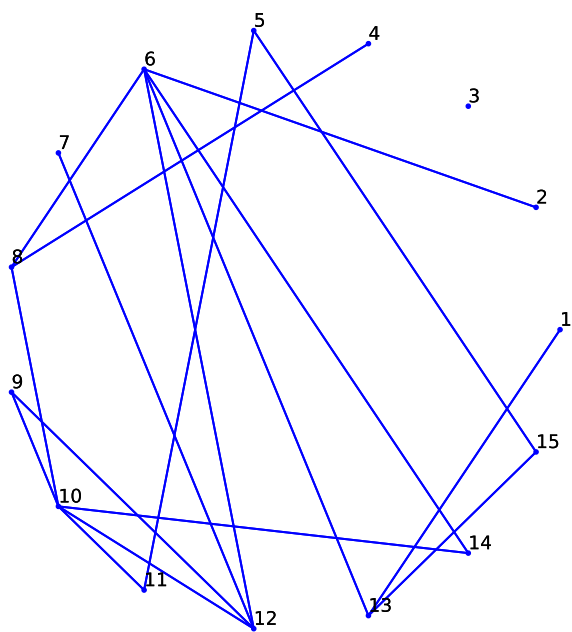


Рисунок 25: Случайный граф созданный с помощью модели Эрдеша-Ренье с параметрами $n = 15, p = 0.1$

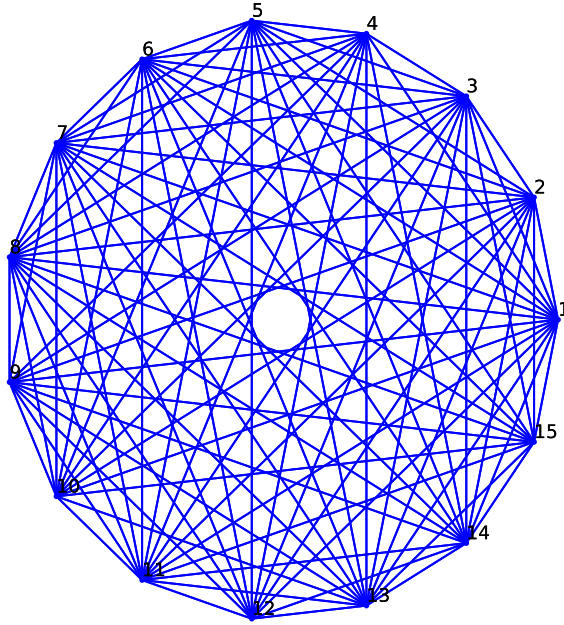


Рисунок 26: Случайный граф созданный с помощью модели Эрдеша-Ренье с параметрами $n = 15, p = 0.7$

На рисунках 25, 26 приведены примеры графов, созданных с помощью данной модели.

Графы созданные с помощью модели Эрдеша-Ренье плохо описывают реальные сетевые структуры такие, как социальные сети.

3.2 Модель Барабаши-Альберт

Пусть дан граф с n вершинами и $m \leq n \in \mathbb{N}$. Будем добавлять вершины пошагово, каждая добавленная вершина должна иметь m ребер.

$$P_{in} = \frac{\deg i}{\sum_j \deg j} \quad (3)$$

Формула 3 обозначает вероятность добавления ребра $\{i, n\}$

```

def add_vertex(self, vertex):
    if vertex not in self.graph.vertices():
        total_deg = sum(self.graph.deg(v) for v in self.graph.vertices())
        p = [self.graph.deg(v) / total_deg for v in self.graph.vertices()]
        end = choice(list(self.graph.vertices()),
                     p=p, size=self.m, replace=False)
        for v1 in end:
            self.graph.add_edge(vertex, v1)

```

Рисунок 27: Реализация добавления вершины согласно модели Барабаши-Альберт

На рисунке 27 приведен код, который реализует добавление новой вершины в граф, согласно данной модели.

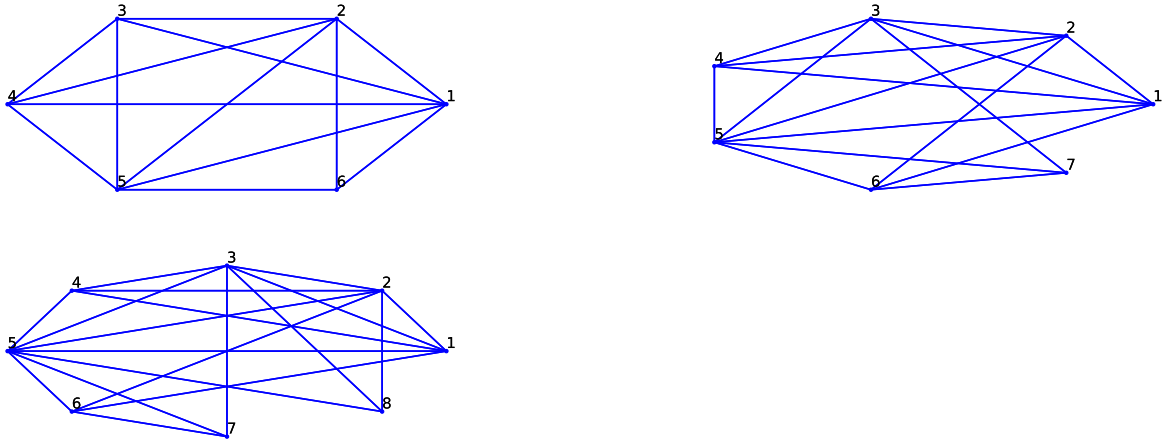


Рисунок 28: Визуализация построения случайного графа с помощью модели Барабаши-Альберт с параметрами $n = 5, m = 3$

На рисунке 28 приведены три шага итерации построения случайного графа, согласно модели Барабаши-Альберт.

3.3 Модель Боллобаша-Риордана

Пусть дан граф с одной вершиной и петлей. Добавляем пошагово вершины. Пусть граф с $n - 1$ вершиной построен, тогда $P_{in} = \frac{\deg i}{2n-1}$, $P_{nn} = \frac{1}{2n-1}$

```
def __init__(self, n):
    g = Graph()
    g.add_edge(1, 1)
    for v in range(2, n+1):
        p = [g.deg(vertex) / (2*n - 1) for vertex in g.vertices()]
        p.append(1 / (2*n - 1))
        g.add_vertex(v)
        end = choices(list(g.vertices()), weights=p, k=1)
        g.add_edge(v, end[0])
    self.graph = g
```

Рисунок 29: Построение графа с n вершинами, согласно модели Боллобаша-Риодана.

На рисунке 29 приведен код для генерации случайного графа с помощью модели Боллобаша-Риодана.

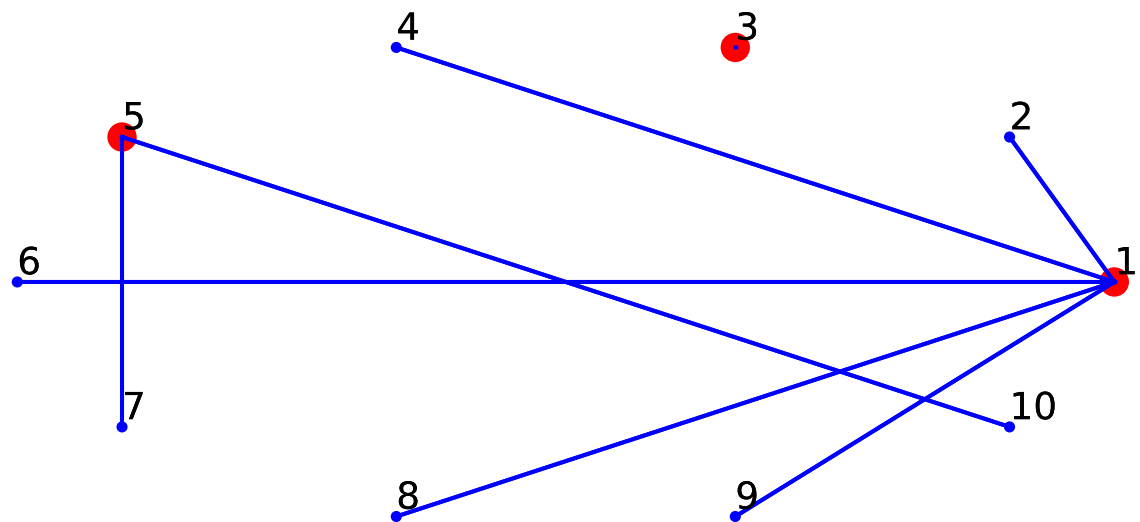


Рисунок 30: Граф с 10 вершинами, построенный согласно модели Боллобаша-Риодана

На рисунке 30 изображен граф построенный согласно данной модели.

3.4 Модель LCD

Рассмотрим алгоритм генерации графа с помощью линейной хордовой диаграммы (LCD):

- 1) идем слева направо;
- 2) добавляем в набор вершины, пока не встретим конец дуги;
- 3) собранный набор становится вершиной графа, дуги становятся дугами графа;

```
def __init__(self, n):
    k = 2*n
    nums = list(range(1, k+1))
    self.left = []
    self.right = []
    for _ in range(n):
        l = choice(nums)
        nums.remove(l)
        r = choice(nums)
        nums.remove(r)
        self.left.append(l)
        self.right.append(r)
    v = []
    curr = []
    nums = list(range(1, k+1))
    for i in nums:
        curr.append(i)
        if i in self.right:
            v.append(curr.copy())
            curr = []
    g = Graph()
    for i, lst in enumerate(v, 1):
        g.add_vertex(i)
        for j, v2 in enumerate(lst):
            if v2 in self.left:
                r = self.right[j]
                for k in range(len(v)):
                    if r in v[k]:
                        g.add_edge(i, k+1)
    self.graph = g
```

Рисунок 31: Реализация LCD алгоритма.

На рисунке 31 приведена реализация LCD алгоритма

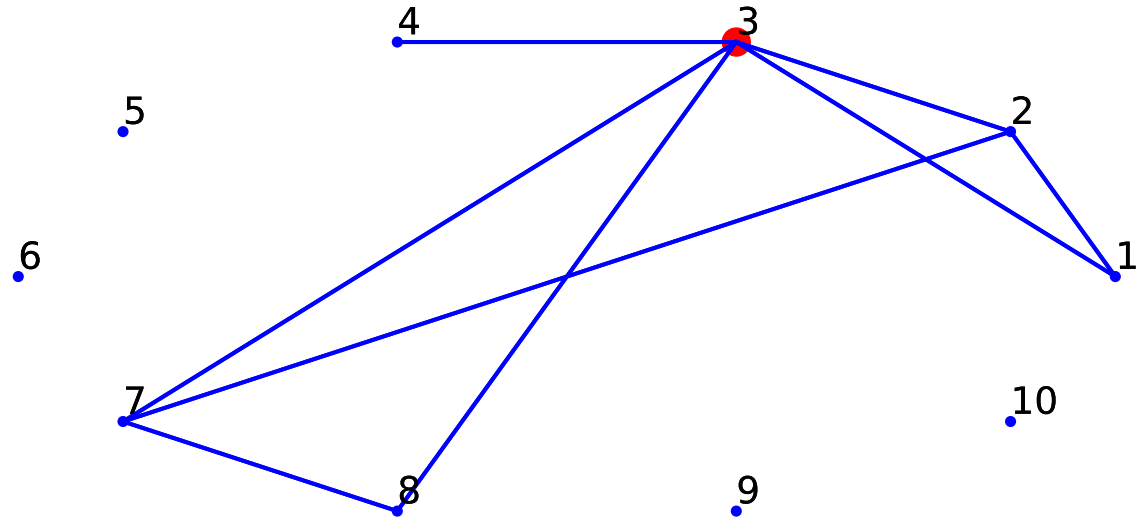


Рисунок 32: Граф, созданный с помощью LCD алгоритма

На рисунке 32 приведен пример графа, созданного с помощью данного алгоритма.

3.5 Модель копирования

Пусть даны $\alpha \in (0, 1)$ и d -регулярный граф $d \geq 1$, V – множество вершин этого графа. Рассмотрим алгоритм добавления вершины в граф:

- 1) выберем случайную вершину $p \in V$;
- 2) добавляем d вершин по следующему правилу: с вероятностью α строим ребро из новой вершины в p , с вероятностью $1 - \alpha$ строим ребро из новой вершины в i -го соседа вершины p ;

```
def add_vertex(self):
    v = self.j + 1
    self.j+=1
    p = choice(self.start_verticies)
    for i in range(self.d):
        if random() < self.alpha:
            self.graph.add_edge(p, v)
        else:
            self.graph.add_edge(list(self.graph.neib(p))[i], v)
```

Рисунок 33: Метод, реализующий алгоритм добавления вершин, согласно модели копирования.

На рисунке 33 приведен метод добавления вершин в граф, согласно модели копирования.

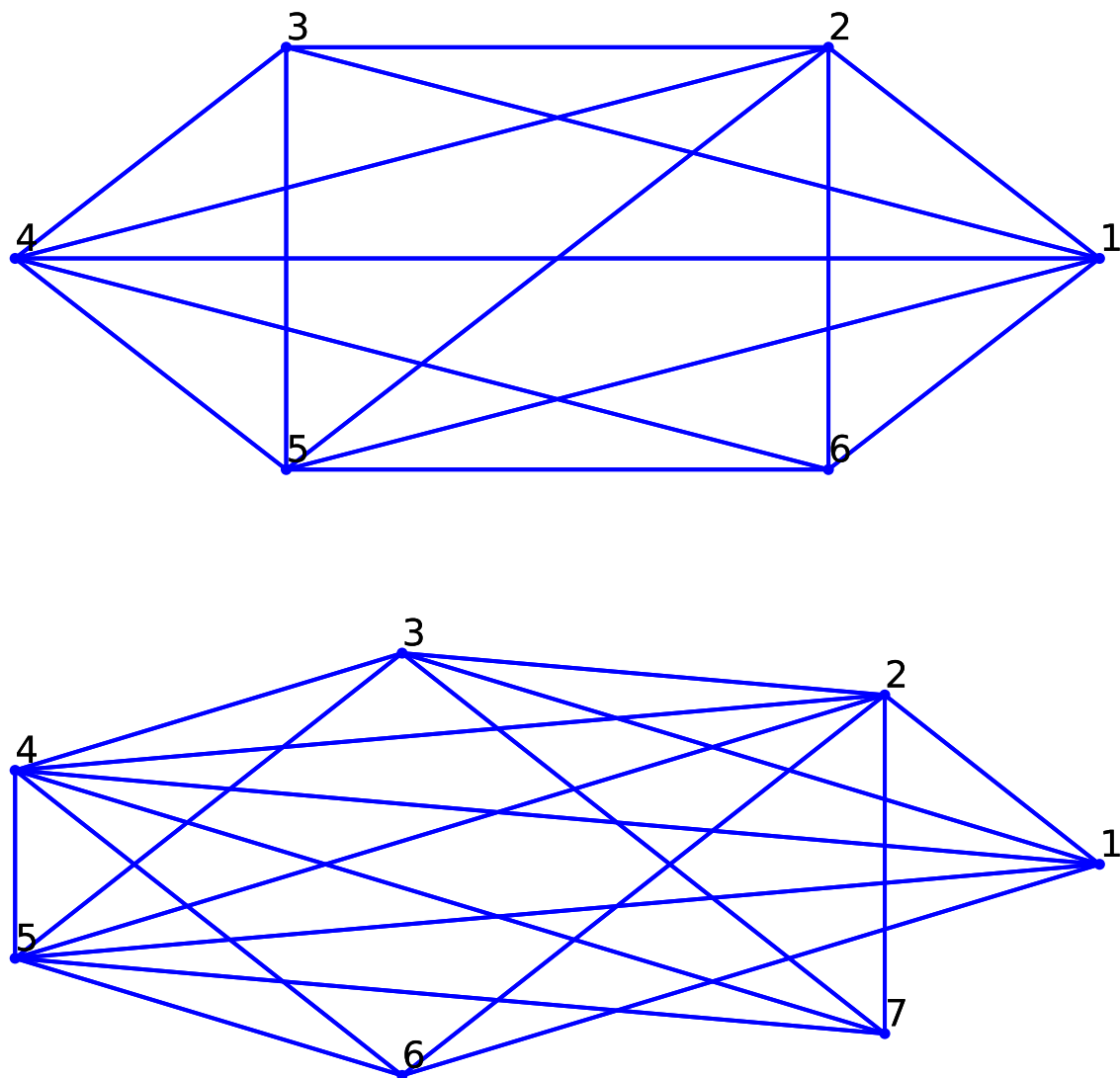


Рисунок 34: Результат добавления двух вершин в граф с параметрами $d = 4$, $\alpha = 0.1$ согласно модели копирования.

На рисунке 34 приведен пример генерации случайного графа согласно модели копирования.

3.6 Модель Чунг-Ли

Рассмотрим модель генерации случайного мультиграфа. Пусть нам дана степень каждой вершины, степень i -ой вершины обозначим как d_i . Граф генерируется

следующим образом:

- 1) строится множество L , которое состоит из d_i копий вершины i ;
- 2) задаются случайные паросочетания на L ;
- 3) число парасочетаний между копиями u и v — число ребер между u и v ;

```
def __init__(self, d: [int]):
    self.d = d
    self.l_set = []
    for i, elem in enumerate(d, 1):
        for _ in range(elem):
            self.l_set.append(i)

    self.edges = []
    for _ in range(sum(d)//2):
        a = choice(self.l_set)
        self.l_set.remove(a)
        b = choice(self.l_set)
        self.l_set.remove(b)
        self.edges.append((a, b))
    self.graph = MultiGraph()
```

Рисунок 35: Генерация случайного мультиграфа согласно модели ЧунгЛи.

На рисунке 35 приведен код для создания случайного мультиграфа согласно данной модели.

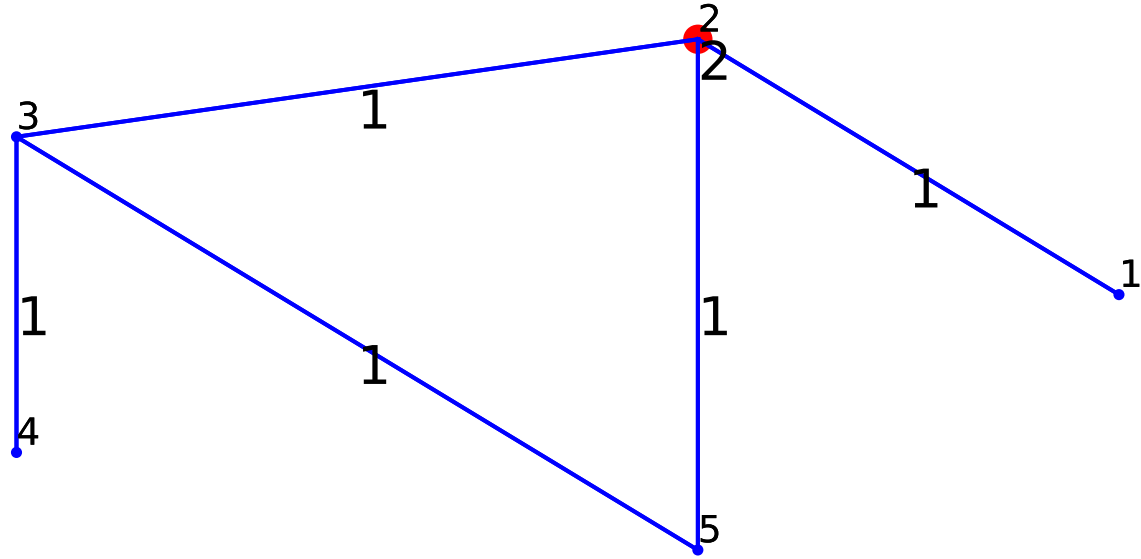


Рисунок 36: Мультиграф созданный с помощью модели Чунг-Ли, где $d = [1, 5, 2, 1, 2]$

Пример случайного мультиграфа, полученного согласно данно модели приведен на рисунке 36.

3.7 Генерация случайного геометрического графа

Назовем граф $G(n, R)$ случайным геометрическим графом, если он получен путем размещения на плоскости n вершин и две вершины инцидентны, если евклидово расстояние между ними не превышает R .

3.7.1 Эффективный алгоритм генерации геометрических графов.

Пусть на плоскость наложена сетка, состоящая из квадратных ячеек с шагом $\frac{R}{\sqrt{2}}$. Обозначим ячейку i -тую по вертикали и j -тую по вертикали как L_{ij} , всего ячеек L_{size}^2 . Назовем множество $\Omega_{ij} = \{(L_{kl} \mid k \in -2 \dots 2, n \in -2 \dots 2, 0 \leq$

$|k| + |m| < 3$ псевдоокрестностью L_{ij} Рассмотрим следующий алгоритм генерации случайных геометрических графов :

- 1) в каждой ячейке создается случайная вершина;
- 2) для i, j вершины создаются ребра смежные вершинам из Ω_{ij} , если расстояние между ними не превышает R ;
- 3) $n_{\text{curr}} = n - L_{\text{size}}^2$ – число несгенерированных вершин;
- 4) $N = n - L_{\text{size}}^2$;
- 5) $p = \frac{1}{L_{\text{size}}^2}$;
- 6) для L_{ij} вычисляется $s_{ij} = \min(n_{\text{curr}}, B(N, p))$, где $B(N, p)$ случайная величина распределенная по биномиальному закону;
- 7) в L_{ij} создается s_{ij} вершин;
- 8) каждая созданная вершина соединяется с вершинами из Ω_{ij} , если расстояние не превышает R ;
- 9) n_{curr} уменьшается на s_{ij} ;
- 10) если $n_{\text{curr}} = 0$ то граф построен;

Данный алгоритм является достаточно эффективным и создает графы похожие на реальные структуры, такие как беспроводные сети.

3.7.2 Реализация алгоритма генерации геометрических графов

Геометрический граф будет реализован с помощью класса «GeoGraph», который является потомком класса «Graph». Для реализации вершин был реализован класс «Node», имеющий следующие поля и методы:

- 1) поле «x» – координата по оси X ;

- 2) поле «y» – координата по оси Y ;
- 3) метод «dist» – вычисляет евклидово расстояние между двумя вершинами;

Рассмотрим реализацию на языке программирования Python.

```
@dataclass(frozen=True)
class Node:
    x: float
    y: float
```

Рисунок 37: Определение класса «Node»

```
def dist(self, other):
    return sqrt((self.x - other.x) ** 2 + (self.y - other.y) ** 2)
```

Рисунок 38: Метод для вычисления евклидова расстояния между двумя вершинами.

На рисунках 37, 38 приведена реализация вершины геометрического графа.

Модель генерации случайного геометрического графа была определена в классе «GeoGraphRndModel». В данном классе был определен метод инициализации и метод создания графа.

```
def __init__(self, r, n):
    self.n = n
    self.r = r
    self.l_step = r/(sqrt(2))
    self.l_size = int(sqrt(n))
    self.grid = Grid(self.l_size)
    self.gen_graph()
```

Рисунок 39: Инициализация модели генерации случайного геометрического графа.

На рисунке 39 приведен метод создания модели случайного геометрического графа. Рассмотрим реализацию алгоритма генерации случайного графа.

```

def gen_graph(self):
    self.graph = GeoGraph()
    for lst in self.grid.grid.values():
        self.graph.add_verticies(lst)
    for i in range(self.l_size):
        for j in range(self.l_size):
            for n1 in self.grid.grid[i, j]:
                omega = self.grid.omega(i, j)
                for n2 in omega:
                    if n1.dist(n2) <= self.r and n2 != n1:
                        self.graph.add_edge(n1, n2)
    N = self.n - self.l_size**2
    p = (1/self.l_size)**2
    n_curr = self.n - self.l_size**2
    for i in range(self.l_size):
        for j in range(self.l_size):
            s = binomial(N, p)
            s = min(s, n_curr)
            for _ in range(s):
                n = Node(i+random(), j+random())
                for n2 in self.grid.grid[i, j]:
                    self.graph.add_edge(n, n2)
                omega = self.grid.omega(i, j)
                for n2 in omega:
                    if n.dist(n2) <= self.r:
                        self.graph.add_edge(n, n2)
                self.grid.grid[i, j].append(n)
                self.graph.add_vertex(n)
            n_curr -= s
        if n_curr <= 0:
            break

```

Рисунок 40: Реализация алгоритма генерации случайного геометрического графа

На рисунке 40 приведен метод, реализующий генерацию случайного геометрического графа.

Рассмотрим примеры графов, созданных с помощью данного алгоритма.

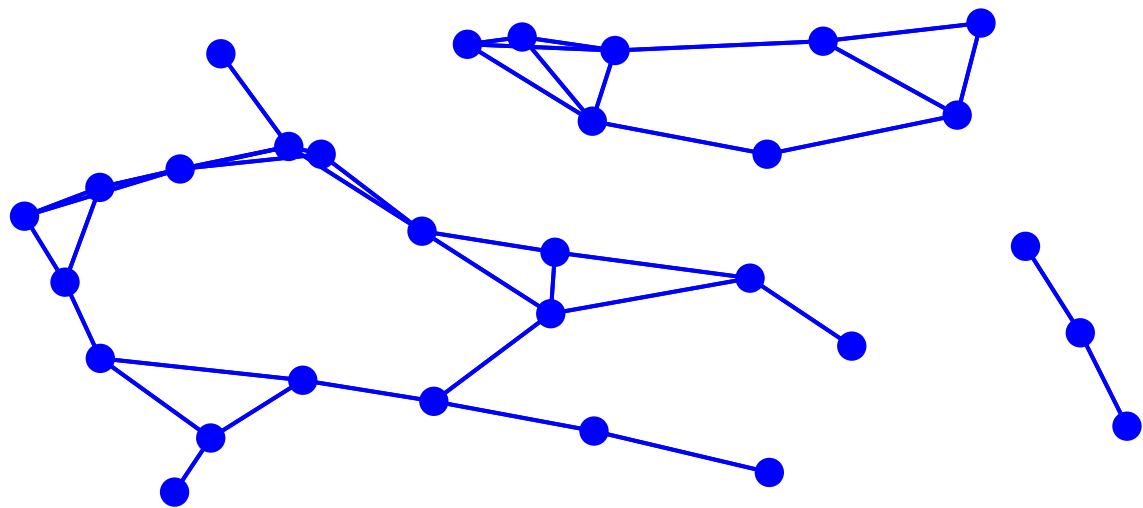


Рисунок 41: Случайный геометрический граф, $n = 30$, $R = 1$

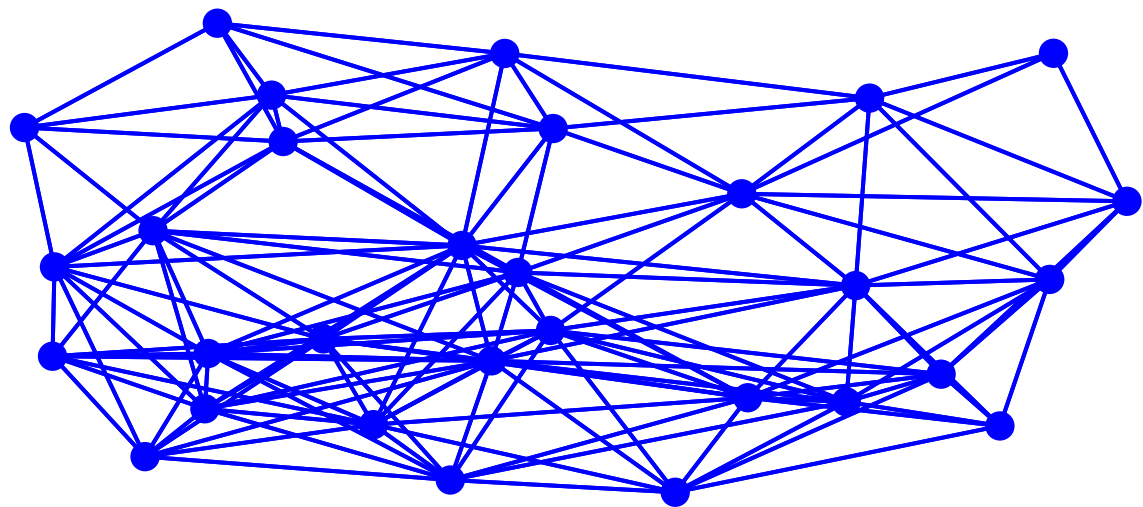


Рисунок 42: Случайный геометрический граф, $n = 30$, $R = 2$

На рисунках 41,42 приведены примеры графов, созданных с помощью алгоритма генерации случайных геометрических графов.