



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	6
1 ГРАФЫ. ОСНОВНЫЕ ОПРЕДЕЛЕНИЯ. РЕАЛИЗАЦИЯ АЛГОРИТМОВ. ПОДСЧЕТ ХАРАКТЕРИСТИК ГРАФА.	8
1.1 Основные определения	8
1.2 Реализация графа.	8
1.3 Реализация алгоритмов на графах.	9
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
1.5 Поиск максимальной клики графа	15
2 ВИЗУАЛИЗАЦИЯ ГРАФОВ	19
3 ГЕНЕРАЦИЯ СЛУЧАЙНЫХ ГРАФОВ	21
3.1 Модель Эрдеша-Ренье	21
3.2 Модель Барабаши-Альберт	23
3.3 Модель Боллобаша-Риордана	24
3.4 Модель Боллобаша-Риодана для создания ориентированного графа	26
3.5 Модель Бакли-Остхус	28
3.6 Модель LCD	29
3.7 Модель копирования	30
3.8 Модель Чунг-Ли	32
3.9 Модель Янсона-Лучака	33
3.10 Генерация случайного геометрического графа	35
3.10.1 Эффективный алгоритм генерации геометрических графов.	35
3.10.2 Реализация алгоритма генерации геометрических графов . .	36

4	ГРАФИЧЕСКИЙ ИНТЕРФЕЙС ПОЛЬЗОВАТЕЛЯ	39
5	ВЫДЕЛЕНИЕ СООБЩЕСТВ В ГРАФЕ	41
5.1	Выделение непересекающихся сообществ	41
5.1.1	Label Propagation	41
5.1.2	FastGreedy	41
5.1.3	Сравнение алгоритмов разбиения на непересекающиеся сообщества	42
5.2	Выделение пересекающихся сообществ	42
5.2.1	DEMON	42
	ЗАКЛЮЧЕНИЕ	43
	БИБЛИОГРАФИЧЕСКИЙ СПИСОК	44

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;
- 5) изучение и сравнение алгоритмов разбиения графа на сообщества;
- 6) изучение и сравнение алгоритмов разбиения графа на сообщества;

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.

```
def bfs(self, start=1):  
    if len(self._verticies) > 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 4: Реализация обхода в ширину на языке программирования Python.

На рисунке, 4 приведена реализация алгоритма 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 5: 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 6: Поиск компонент связности.

На рисунках 5, 6 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 7: Метод для вычисления плотности графа.

На рисунке 7 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 8: Реализация вычисления диаметра графа.

На рисунке 8 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 9: 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 10: Вычисление количества треугольников, содержащих вершину v .

На рисунках 9 , 10 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 11: 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 12: Вычисление числа вилок в графе

На рисунках 11, 12 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 13: 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 14: 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 15: Вычисление среднего коэффициента кластеризации

На рисунках 13,14,15 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.vertices() if self.deg(v) == d])
        result.append(t(d, k_d/self.k_verticies()))
    return result
```

Рисунок 16: Вычисление распределения степеней.

На рисунке 16 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 17: Вычисление степени близости вершины v .

На рисунке 17 приведен метод для вычисления степени близости вершины v .

1.5 Поиск максимальной клики графа

Кликой в графе $G = (V, E)$ называется такое подмножество его вершин, $C \subset V$, что $\forall u, v, u \neq v \in C : (u, v) \in E$. Максимальной кликой называется такая клика, которая не содержится в клике большего размера.

Задача поиска максимальных клик, является NP-полной. Поэтому будет рассмотрен следующий приблизительный алгоритм поиска максимальных клик [1]

- 1) иницируется начальная клика, состоящая из одной вершины;
- 2) если есть вершины, которые можно добавить в клику, то добавляется лучшая;
- 3) если таких нет, то удаляется лучшая;

Данный алгоритм выполняется требуемое число шагов, максимальной кликой, является полученная клика наибольшего размера. Определим понятие лучшей вершины. Назовем множеством кандидатов, такое подмножество вершин графа, что вершины не входят в клику, но имеют ребро с каждой вершиной из клики. Лучшим кандидатом для добавления клику является та вершина из множества кандидатов, которая имеет наибольшее число ребер с другими вершинами из множества кандидатов. Лучшей вершиной для удаления является та вершина, у которой наименьшее число соседей из множества таких вершин, которые не входят в клику и имеют на одно ребро меньше чем вершин в клике.

Теперь рассмотрим реализацию данного алгоритма на языке программирования Python. Был создан класс «MaxClique» который при инициализации, получается граф и начальную клику. Были разработаны следующие методы

- 1) «candidates» – создает список кандидатов для добавления в текущую клику;
- 2) «best_candidate» – возвращает лучшую вершину для добавления;
- 3) «add_best» – добавляет в клику лучшую вершину;
- 4) «one_missing» – возвращает список для определения лучшей вершины для удаления;
- 5) «best_for_delete» – возвращает лучшую вершину для удаления;
- 6) «delete_best» – удаляет лучшую вершину;
- 7) «step» – выполняет шаг алгоритма;
- 8) «find_clique» – возвращает найденную максимальную клику;

Рассмотрим каждый метод более подробно.

```
def __init__(self, g: Graph, start):  
    self.graph = g  
    self.clique = start
```

Рисунок 18: Инициализация объекта типа «MaxClique».

На рисунке 18 приведена инициализация объекта для поиска максимальной клики.

```

def candidates(self):
    candidates = set()
    for v in self.graph.verticies():
        f = True
        if v not in candidates:
            for u in self.clique:
                if not self.graph.has_edge(u, v):
                    f = False
                    break
            if f:
                candidates.add(v)
    return candidates

def best_candidate(self):
    candidates = self.candidates()
    d = dict()
    for v in candidates:
        d[v] = sum(1 for u in candidates if self.graph.has_edge(u, v))
    return max(d)

def add_best(self):
    self.clique.add(self.best_candidate())

```

Рисунок 19: Реализация методов «candidates», «best_candidate», «add_best»

На рисунке 19 приведены методы для добавления лучшей вершины в клику.

```

def one_missing(self):
    result = set()
    for v in self.graph.verticies():
        if v not in self.clique:
            k = sum(1 for u in self.clique if self.graph.has_edge(u, v))
            if len(self.clique) - 1 == k:
                result.add(v)
    return result

def best_for_delete(self):
    missing = self.one_missing()
    d = dict()
    for v in self.clique:
        d[v] = sum(1 for u in missing if self.graph.has_edge(v, u))
    return min(d)

def delete_best(self):
    self.clique.remove(self.best_for_delete())

```

Рисунок 20: Методы «one_missing», «best_for_delete», «delete_best»

На рисунке 20 приведены методы для удаления лучшей вершины из клики.


```

def step(self):
    cand = self.candidates()
    if len(cand) != 0:
        self.add_best()
    else:
        self.delete_best()

def find_clique(self, k_ticks):
    result = self.clique.copy()
    for _ in range(k_ticks):
        self.step()
        if len(self.clique) > len(result):
            result = self.clique.copy()
    return result

```

Рисунок 21: Реализация методов «step» и «find_clique»

На рисунке 21 приведены методы для поиска максимальной клики графа.

Данный алгоритм имеет ряд преимуществ, такие как простота и эффективность. К недостаткам можно отнести, то что находит лишь одну максимальную клику.

2. ВИЗУАЛИЗАЦИЯ ГРАФОВ

Визуализация была разработана с помощью сторонней библиотеки «matplotlib» [2].

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 22: 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 23: Генерация словаря «coords»

На рисунке 23 представлен метод для создания координат вершин.

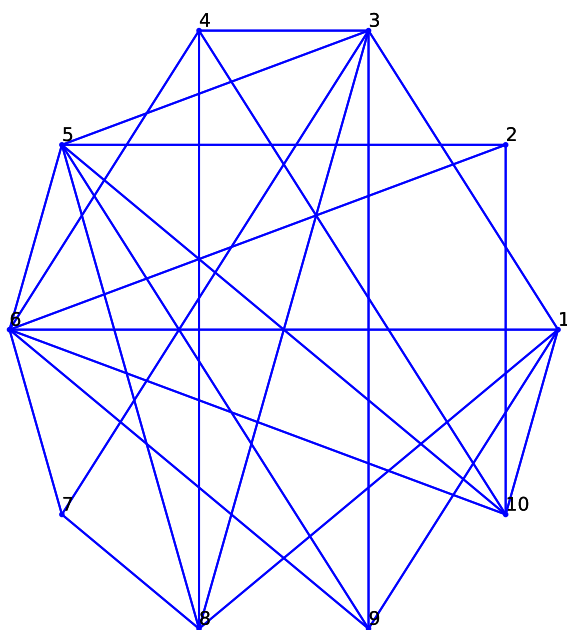


Рисунок 24: Пример диаграммы графа.

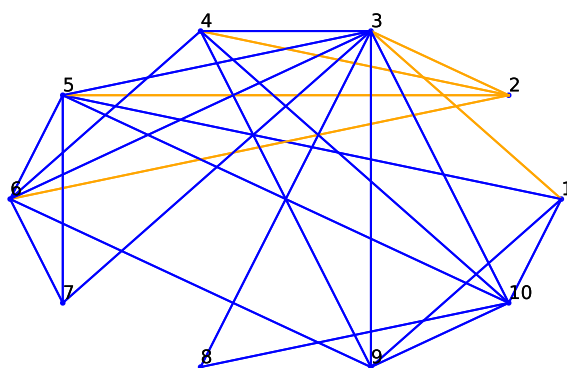


Рисунок 25: Пример диаграммы графа с оранжевыми ребрами.

На рисунках 24, 25 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 [3]. 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 26: 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 27: Генерация случайного графа по модели Эрдеша-Ренье

На рисунках 26,27 приведен код для генерация случайного графа.

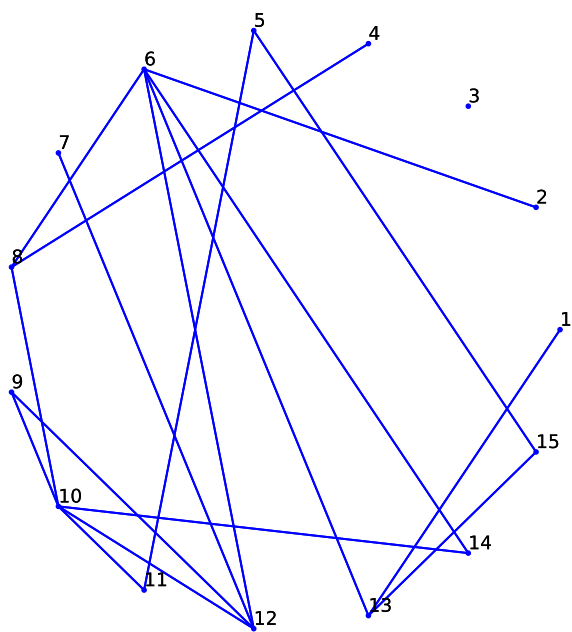


Рисунок 28: Случайный граф созданный с помощью модели Эрдеша-Ренье с параметрами $n = 15, p = 0.1$

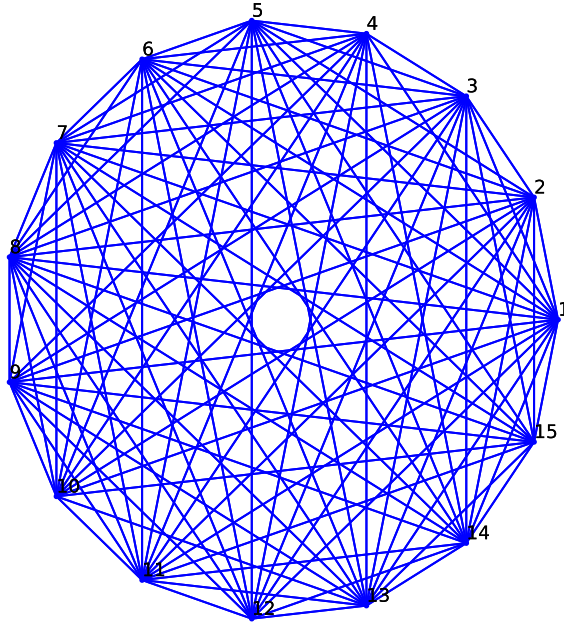


Рисунок 29: Случайный граф созданный с помощью модели Эрдеша-Ренье с параметрами $n = 15, p = 0.7$

На рисунках 28, 29 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 30: Реализация добавления вершины согласно модели Барабаши-Альберт

На рисунке 30 приведен код, который реализует добавление новой вершины в граф, согласно данной модели.

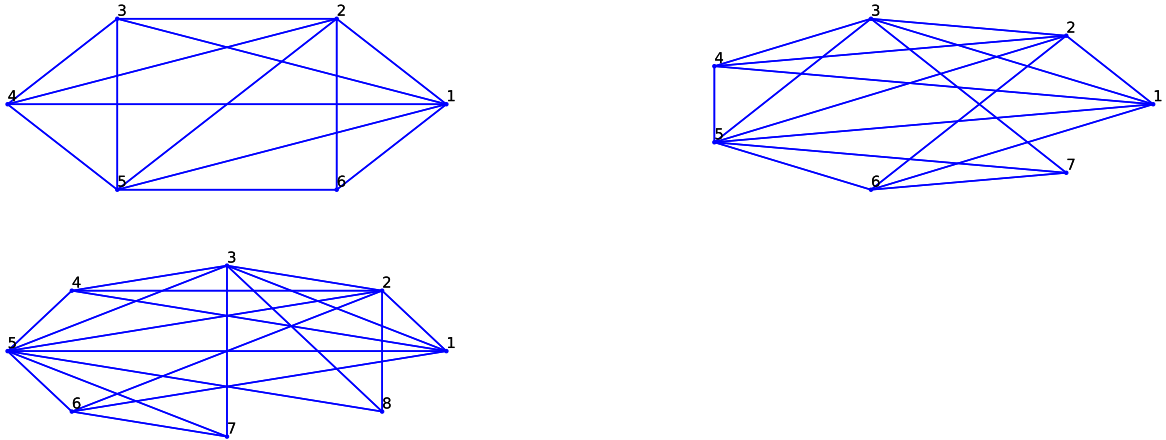


Рисунок 31: Визуализация построения случайного графа с помощью модели Барабаши-Альберт с параметрами $n = 5, m = 3$

На рисунке 31 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 32: Построение графа с n вершинами, согласно модели Боллобаша-Риодана.

На рисунке 32 приведен код для генерации случайного графа с помощью модели Боллобаша-Риодана.

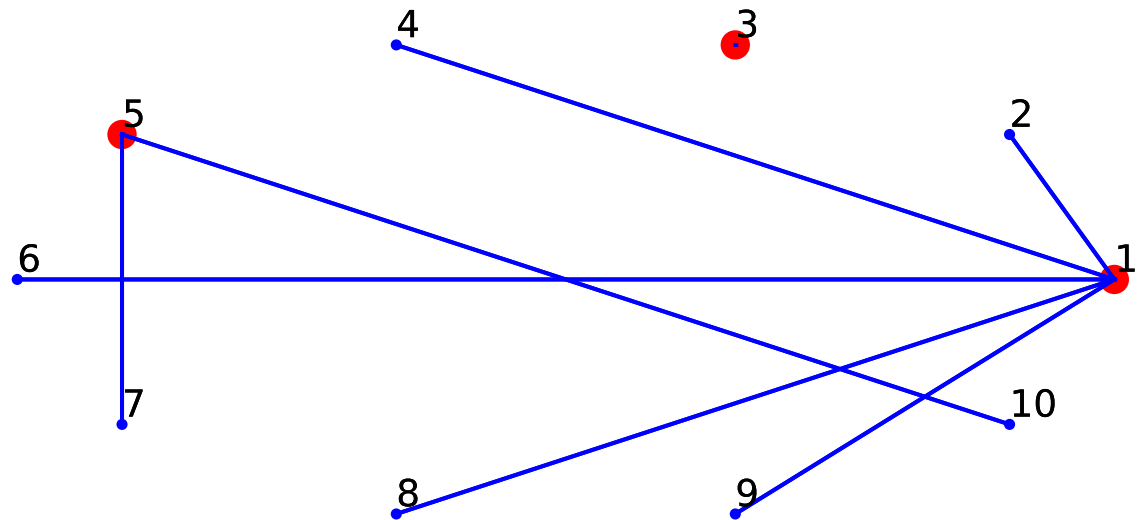


Рисунок 33: Граф с 10 вершинами, построенный согласно модели Боллобаша-Риодана

На рисунке 33 изображен граф построенный согласно данной модели.

3.4 Модель Боллобаша-Риодана для создания ориентированного графа

Рассмотрим следующую модель генерации случайного ориентированного графа [4]. Пусть даны неотрицательные вещественные числа $\alpha, \beta, \gamma, \delta_{in}, \delta_{out}$, где $\alpha + \beta + \gamma = 1$ и ориентированный граф $G(1)$. Будет создаваться последовательность $G(1), G(2), \dots, G(t), G(t+1) \dots G(n)$, где следующий граф строится по предыдущему. Пусть вершина $v \in V(t)$ выбирается согласно $d_{in} + \delta_{in}$, если вероятность выбора этой вершин из $V(t)$ равна $\frac{d_{in}(v) + \delta_{in}}{t + \delta_{in} \cdot |V(t)|}$, выбирается согласно $d_{out} + \delta_{out}$ если вероятность выбора данной вершины из $V(t)$ равна $\frac{d_{out}(v) + \delta_{out}}{t + \delta_{out} \cdot |V(t)|}$, где $d_{in}(v)$ – полустепень захода вершины v , $d_{out}(v)$ – полустепень исхода вершины v .

Граф $G(t+1)$ строится согласно следующему правилу:

- 1) с вероятностью α добавляется ребро (v, w) , где v новая вершина, w вершина выбранная согласно $d_{in} + \delta_{out}$;
- 2) с вероятностью β добавляется ребро (v, w) , где вершина v добавляется согласно $d_{out} + \delta_{out}$, вершина w выбирается согласно $d_{in} + \delta_{in}$;
- 3) с вероятностью γ добавляется ребро (v, w) , где v новая вершина, w выбирается согласно $d_{out} + \delta_{out}$;

Рассмотрим реализацию данного алгоритма.

```
def d_in_p(self, v):
    return (self.graph.in_deg(v) + self.d_in) / (self.t + self.d_in * self.graph.k_verticies())

def d_out_p(self, v):
    return (self.graph.out_deg(v) + self.d_in) / (self.t + self.d_out * self.graph.k_verticies())

def rnd_in(self):
    probs_in = [self.d_in_p(v) for v in self.graph.verticies()]
    return choices(list(self.graph.verticies()),
                  weights=probs_in, k=1)[0]

def rnd_out(self):
    probs_out = [self.d_out_p(v) for v in self.graph.verticies()]
    return choices(list(self.graph.verticies()),
                  weights=probs_out, k=1)[0]
```

Рисунок 34: Получение случайных вершин графа

На рисунке 34 приведен код для получения вершин графа согласно $d_{in} + \delta_{in}$ и $d_{out} + \delta_{out}$

```
def add_vertex(self):
    p = random()
    if p < self.alpha:
        v = self.nxt
        self.nxt += 1
        w = self.rnd_in()
        self.graph.add_edge(v, w)
    if p < self.beta:
        v = self.rnd_in()
        w = self.rnd_out()
        self.graph.add_edge(v, w)
    if p < self.gamma:
        w = self.nxt
        self.nxt += 1
        v = self.rnd_out()
        self.graph.add_edge(v, w)
    self.t += 1
```

Рисунок 35: Добавление вершины в граф согласно данной модели

На рисунке 35 приведена реализация добавления вершины в ориентированный граф.

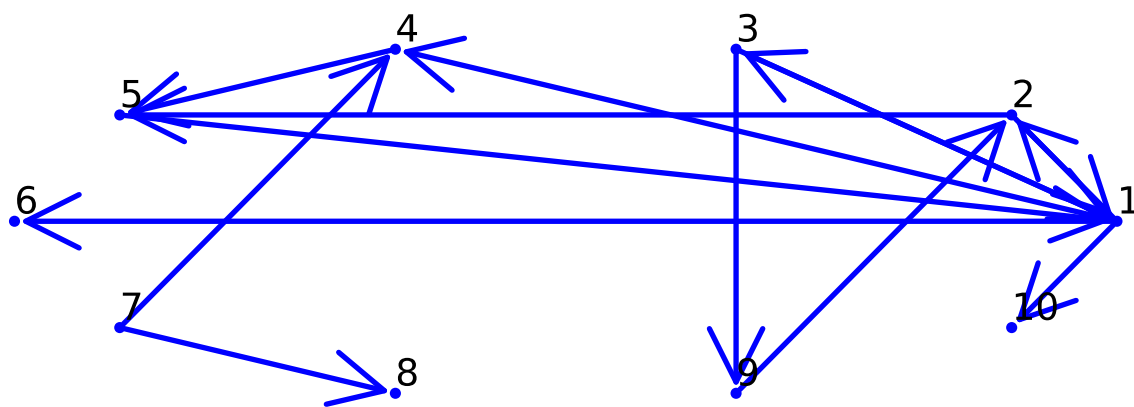


Рисунок 36: Пример графа созданного с помощью данной модели

На рисунке 36 приведена диаграмма графа созданного с помощью данной модели, где $\alpha = 0.4$, $\beta = 0.4$, $\gamma = 0.2$, $\delta_{in} = 2$, $\delta_{out} = 2$.

3.5 Модель Бакли-Остхус

Рассмотрим следующую [5] модификацию модели Боллобаша-Риодана. Дан граф с одной вершиной и петлей и параметр a . На $i + 1$ -том шаге вершина $i + 1$ добавляется в граф, из нее проводится ребро в случайную вершину, которая выбирается со следующей вероятностью $P_{k,i+1} = \frac{\deg k + a}{(a+1) \cdot (i+1) - 1}$, $P_{i+1,i+1} = \frac{a}{(a+1)(i+1) - 1}$.

Рассмотрим реализацию данной модели.

```
def add_edge(self):
    p = [(self.graph.deg(v) + self.a) / ((self.a+1) *
                                           (self.graph.k_verticies() + 1) - 1) for v in self.graph.
          verticies()]
    p.append(self.a / ((self.a+1) * (self.graph.k_verticies() + 1) - 1))
    self.graph.add_vertex(self.nxt)
    v = choices(list(self.graph.verticies()), weights=p, k=1)[0]
    self.graph.add_edge(self.nxt, v)
    self.nxt += 1
```

Рисунок 37: Добавление вершины в граф

На рисунке 37 представлен метод, реализующий добавление вершины в граф, согласно модели Бакли-Остхус.

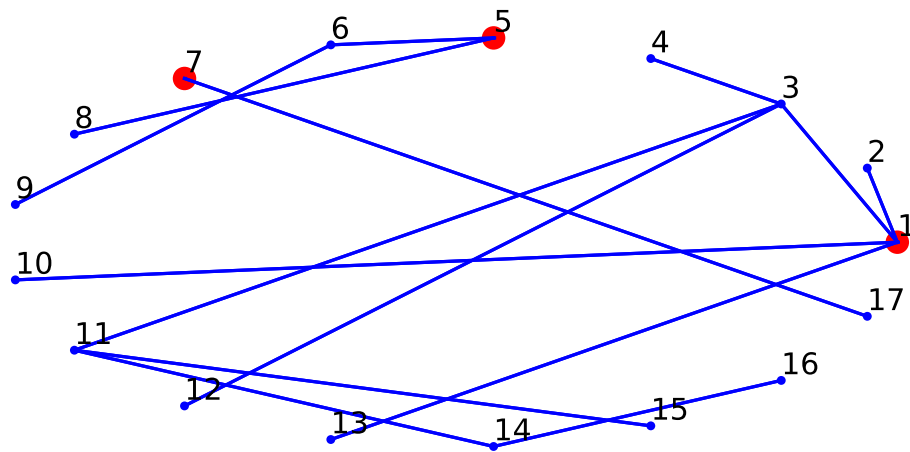


Рисунок 38: Пример графа, созданного с помощью модели Бакли-Остхус с параметром $a = 2$

На рисунке 38 представлен пример результата работы данной модели.

3.6 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 39: Реализация LCD алгоритма.

На рисунке 39 приведена реализация LCD алгоритма

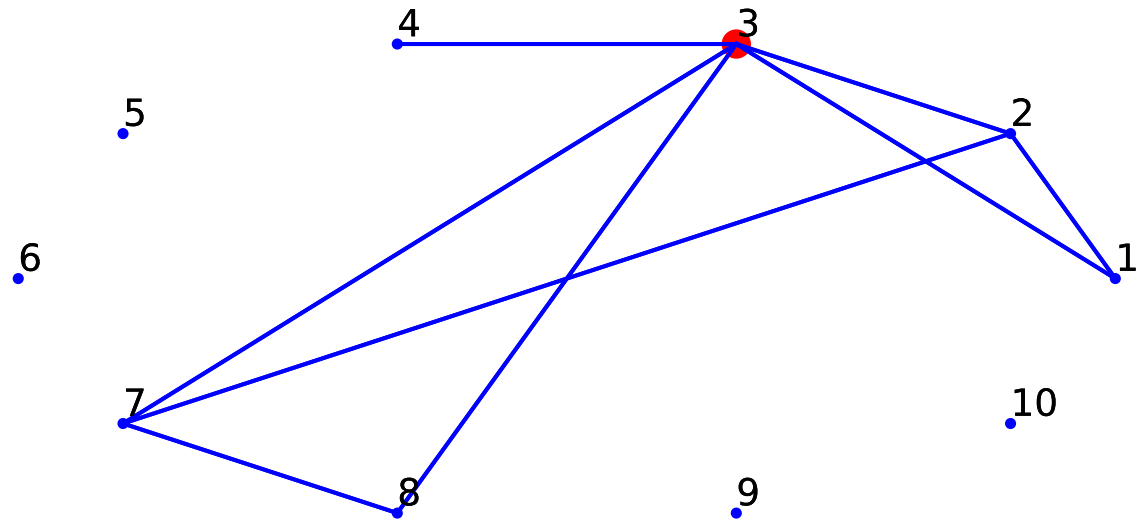


Рисунок 40: Граф, созданный с помощью LCD алгоритма

На рисунке 40 приведен пример графа, созданного с помощью данного алгоритма.

3.7 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_vertices)
    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 41: Метод, реализующий алгоритм добавления вершин, согласно модели копирования.

На рисунке 41 приведен метод добавления вершин в граф, согласно модели копирования.

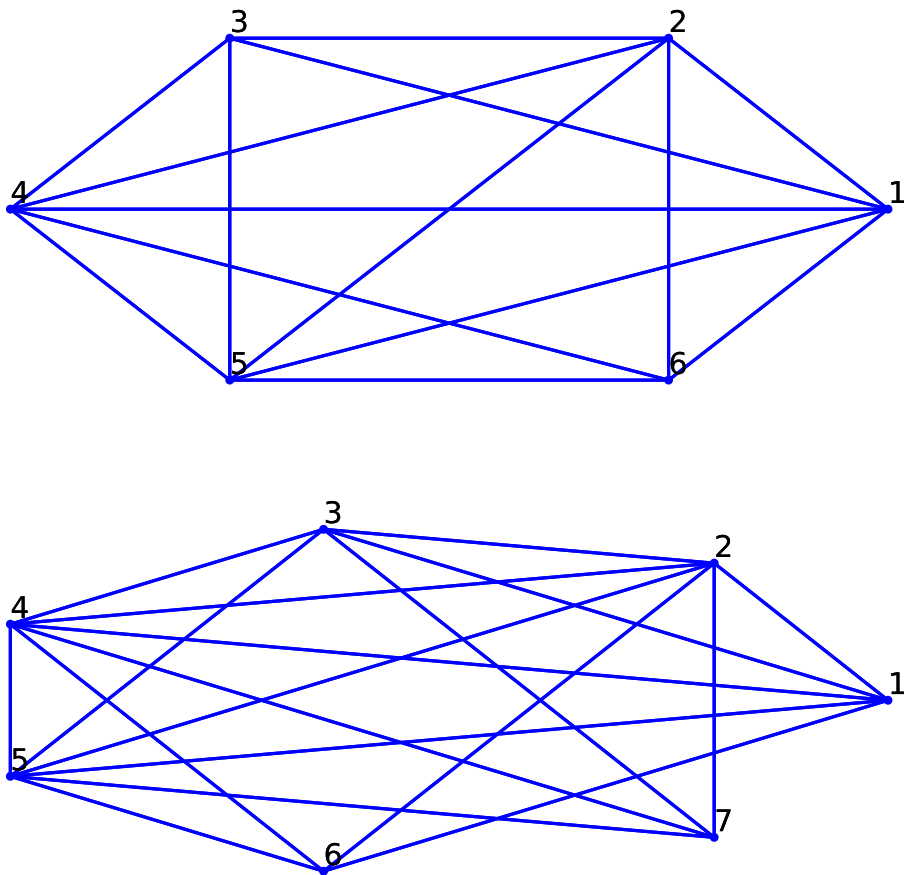


Рисунок 42: Результат добавления двух вершин в граф с параметрами $d = 4$, $\alpha = 0.1$ согласно модели копирования.

На рисунке 42 приведен пример генерации случайного графа согласно модели копирования.

3.8 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 43: Генерация случайного мультиграфа согласно модели ЧунгЛи.

На рисунке 43 приведен код для создания случайного мультиграфа согласно данной модели.

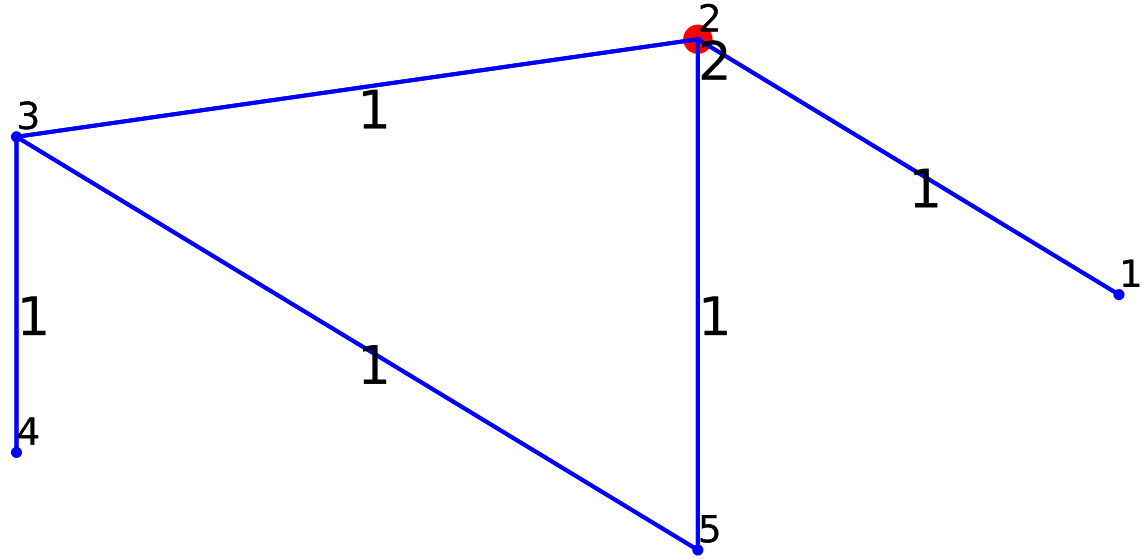


Рисунок 44: Мультиграф созданный с помощью модели Чунг-Ли, где $d = [1, 5, 2, 1, 2]$

Пример случайного мультиграфа, полученного согласно данно модели приведен на рисунке 44.

3.9 Модель Янсона-Лучака

Рассмотрим набор вершин $v = \{v_1 \dots v_n\}$ и набор весов $W = \{W_1 \dots W_n\}$. Пусть $\lambda_{ij} = \frac{W_i W_j}{n}$ – математическое ожидание случайной величины E_{ij} , имеющей Пуассоновское распределение. Для вершин i, j строится E_{ij} кратных ребер. Получившийся мультиграф можно преобразовать в граф, путем стягивания кратных ребер.


```
def gen_graph(self):
    rng = np.random.default_rng()
    graph = MultiGraph()
    for (u, v) in combinations(range(1, len(self.weights)+1), 2):
        lamb = (self.weights[u-1]*self.weights[v-1]) / len(self.weights)
        for _ in range(int(lamb)):
            graph.add_edge(u,v)
        graph.add_verticies((u,v))
    self.graph = graph
```

Рисунок 45: Генерация случайного графа согласно модели Янсона-Лучака.

На рисунке 45 приведена реализация алгоритма создания случайного графа, согласно модели Янсона-Лучака. Для генерации случайной величины используется библиотека «numpy» [6].

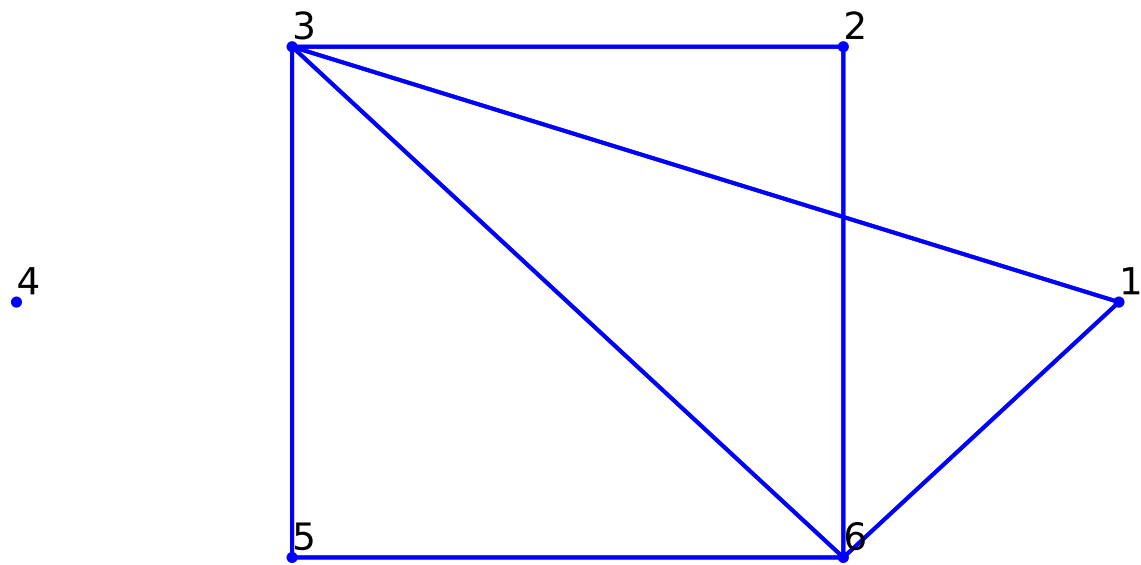


Рисунок 46: Пример графа, созданного согласно модели Янсона-Лучака

На рисунке 46 изображен граф, который был создан с помощью модели Янсона-Лучака, $w = \{2, 2, 3, 1, 3, 2\}$.

3.10 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 .

3.10.1 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 горизонтали 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 [7] :

- 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 негенерированных 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.

3.10.2 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 47: Определение класса «Node»

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

Рисунок 48: Метод для вычисления евклидова расстояния между двумя вершинами.

На рисунках 47, 48 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 49: Инициализация модели генерации случайного геометрического графа.

На рисунке 49 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 50: Реализация алгоритма генерации случайного геометрического графа

На рисунке 50 приведен метод, реализующий генерацию случайного геометрического графа.

Рассмотрим примеры графов, созданных с помощью данного алгоритма.

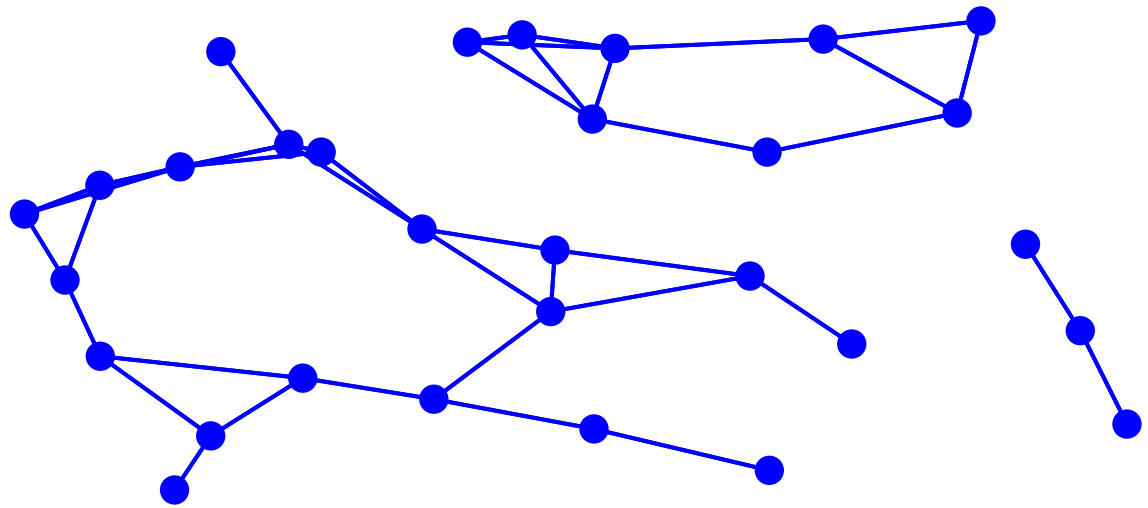


Рисунок 51: Случайный геометрический граф, $n = 30$, $R = 1$

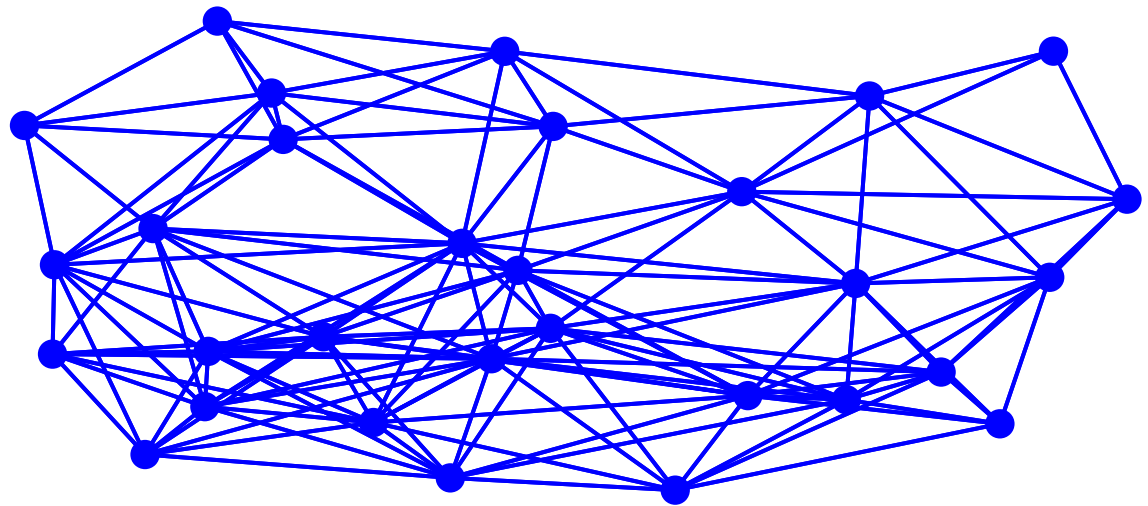


Рисунок 52: Случайный геометрический граф, $n = 30$, $R = 2$

На рисунках 51,52 приведены примеры графов, созданных с помощью алгоритма генерации случайных геометрических графов.

4. ГРАФИЧЕСКИЙ ИНТЕРФЕЙС ПОЛЬЗОВАТЕЛЯ

Графический интерфейс был разработан с помощью библиотеки «PySide6» [8]. Были созданы следующие виджеты:

- 1) виджет для визуализации графа;
- 2) виджет для создания информации о графе;
- 3) виджеты для создания графа, согласно каждой из моделей генерации случайного графа;
- 4) виджеты для визуализации поиска в глубину и ширину;
- 5) виджет для создания графа и подсветки в нем мостов;

Рассмотрим примеры работы данного приложения

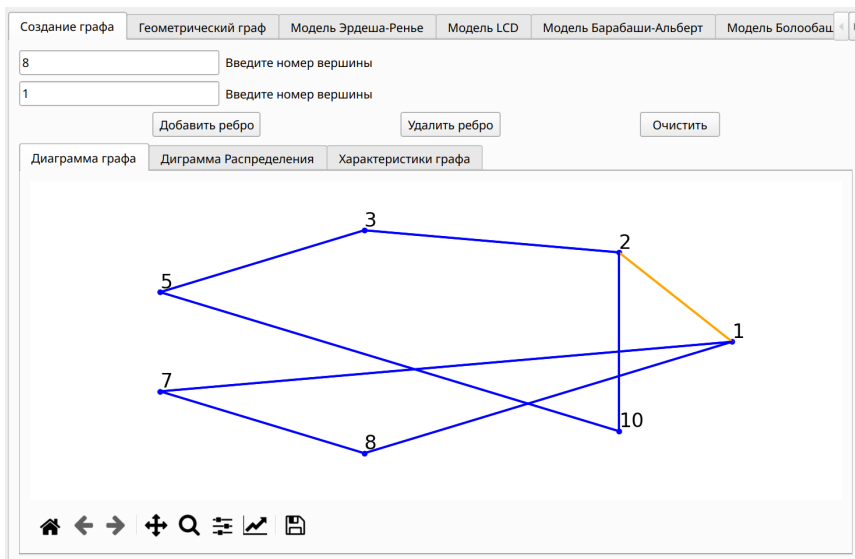


Рисунок 53: Граф с подсвеченным мостом

На рисунке 53 приведено основное окно графического приложения.

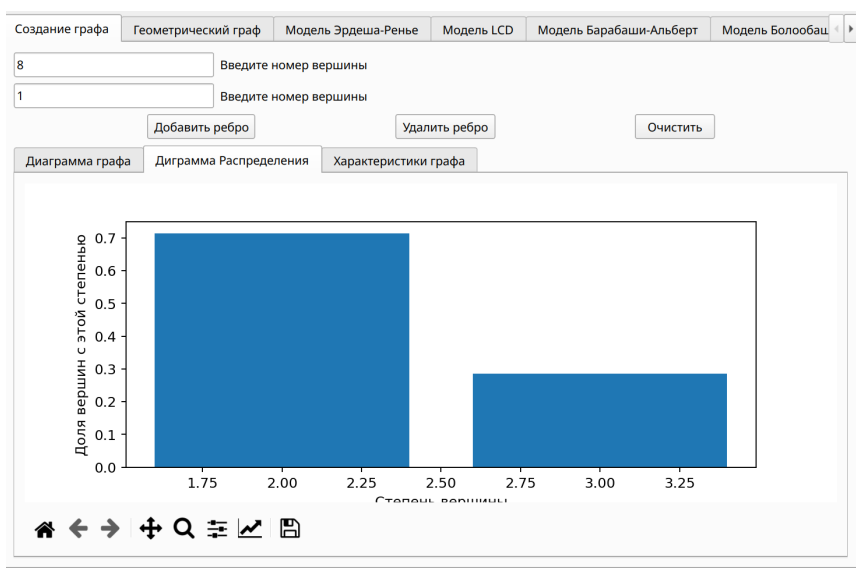


Рисунок 54: Диаграмма распределения степеней вершин графа.

На рисунке 54 изображена диаграмма распределения степеней графа, приведенного на рисунке 53.

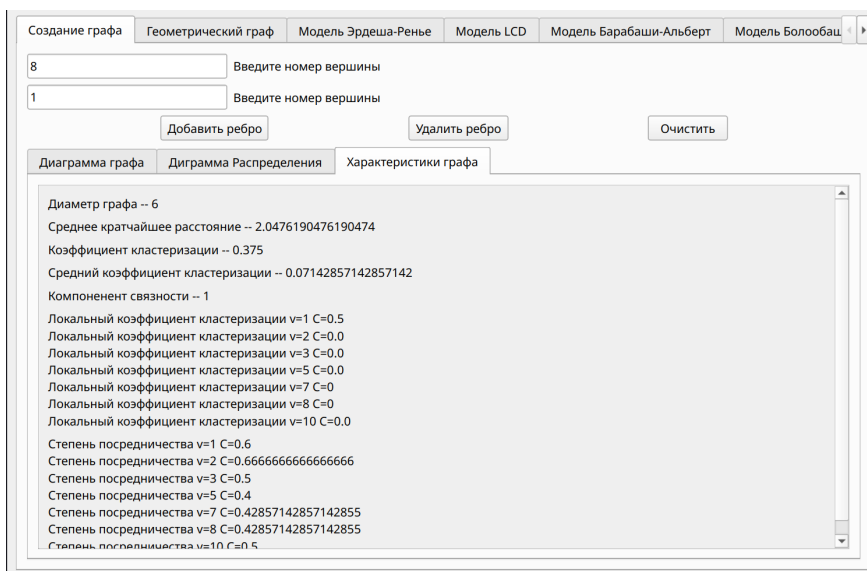


Рисунок 55: Характеристики графа.

На рисунке 55 приведены характеристики графа, приведенного на рисунке 53.

5. ВЫДЕЛЕНИЕ СООБЩЕСТВ В ГРАФЕ

Назовем сообществом или кластером некое разбиение множества вершин, так же введем меру выраженности структуры сообщества, называемую модулярностью [9]

$$Q = \frac{1}{2m} \sum_{i,j} (A_{ij} - \frac{d_i d_j}{2m}) \delta(C_i, C_j) \quad (4)$$

На рисунке 4 приведена формула модулярности, где m – количество ребер графа, A_{ij} элемент матрицы смежности, d_i, d_j – степени вершин графа, $\delta(C_j, C_i)$ – функция, которая равна 0 если сообщества не равны, иначе равна 1.

5.1 Выделение непересекающихся сообществ

Будет рассмотрен ряд алгоритмов выделения непересекающихся сообществ, основанный на максимализации модулярности.

5.1.1 Label Propagation

Данный метод основан на предположении, что соседние вершины находятся в одном сообществе. Алгоритм перемещает вершины графа в то сообщество, где находится большинство ее соседей до тех пор, когда вершин для перемещения не останется.

5.1.2 FastGreedy

Данный алгоритм жадно максимализует модулярность. В начале работы алгоритма каждая вершина находится в своем сообществе. На каждом шаге объединяются пары сообществ, чье объединение максимально увеличивает модулярность.

5.1.3 Сравнение алгоритмов разбиения на непересекающиеся сообщества

Для сравнения эффективность алгоритмов были созданы 10 графов согласно модели Эрдеша-Реньи на 30 вершин с параметром $p = 0.5$ и 10 графов согласно модели Болобаша-Риодана на 30 вершин. Была вычислена средняя модулярность.

Таблица 1: Результат сравнения алгоритмов

	Label Propagation	FastGreedy
Эрдеша-Реньи	0.188	0.186
Болобаша-Риодана	0.554	0.642

На рисунке 1 приведены результаты сравнения. Действительно модель Болобаша-Риодана создает графы, которые лучше описывают реальные сетевые графы.

5.2 Выделение пересекающихся сообществ

5.2.1 DEMON

Данный алгоритм, основан на алгоритме Label Propagation. Назовем подграф эго-сетью вершины v , если он содержит всех соседей вершины v и все ребра смежные данной вершине. Опишем алгоритм:

- 1) $\mathcal{C} = \emptyset$ – исходное разбиение
- 2) для каждой вершины v строим эго-сеть, ее разбиваем методом Label Propagation;
- 3) объединяем с сообществами из \mathcal{C} , если отношение размера меньшего к большему сообществу меньше ϵ

ЗАКЛЮЧЕНИЕ

Все задачи были выполнены, но данную работу можно расширить следующим образом:

- 1) реализовать другие алгоритмы укладки графа;
- 2) создать web-интерфейс;
- 3) реализовать другие алгоритмы кластеризации;

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. [Электронный ресурс]: URL: <https://learn.microsoft.com/ru-ru/archive/msdn-magazine/2011/november/test-run-greedy-algorithms-and-maximum-clique> (дата обращения 10.05.2025).
2. [Электронный ресурс]: URL: <https://matplotlib.org/> (дата обращения 05.06.2025).
3. Н.Н. Кузюрин М.М. Берновский. “Случайные графы, модели и генераторы безмасштабных графов”. В: *Труды Института системного программирования РАН* (2012).
4. B.Bollobas и др. Directed Scale-Free Graphs:
5. Deryk Osthus Pierce G. Buckley. “Popularity based random graph models leading to a scale-free degree sequence”. В: *Discrete Mathematics* (2004).
6. [Электронный ресурс]: URL: <https://numpy.org/> (дата обращения 05.06.2025).
7. О. Д. Соколова В. В. Шахов А. Н. Юргенсон. “ЭФФЕКТИВНЫЙ МЕТОД ГЕНЕРАЦИИ СЛУЧАЙНЫХ ГЕОМЕТРИЧЕСКИХ ГРАФОВ ДЛЯ МОДЕЛИРОВАНИЯ БЕСПРОВОДНЫХ СЕТЕЙ”. В: *ПРИКЛАДНАЯ ДИСКРЕТНАЯ МАТЕМАТИКА* (2016).
8. [Электронный ресурс]: URL: <https://pypi.org/project/PySide6/> (дата обращения 05.06.2025).
9. Никишин Евгений Сергеевич. Методы выделения сообществ в социальных графах: