**РОССИЙСКОЙ ФЕДЕРАЦИИ**

**ПЕНЗЕНСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ**

**Кафедра «Вычислительная техника»**

Курсовая работа

по дисциплине «Логика и основы алгоритмизации в инженерных задачах»

на тему «Реализация алгоритма поиска максимальных паросочетаний»

ПГУ 09.03.01 -

Направление подготовки – 09.03.01 Информатика и вычислительная техника

Профиль подготовки – Программная инженерия

|  |  |  |
| --- | --- | --- |
| Выполнил студент |  | Усов А.С. |
| Группа |  | 23ВВВ1 |
|  |  |  |
| Руководитель |  |  |
| д.т.н., профессор |  | Митрохин М.А. |

|  |  |
| --- | --- |
| Работа защищена с оценкой |  |
| Преподаватели |  |
|  |
| Дата защиты |  |

2025

**Содержание**

Введение………………………………………………………………….3

Постановка задачи…………………………………………………….....4

Теоретическая часть задания……………………………………………4

Описание алгоритма поставленной задачи…………………………….7

Алгоритмы ручного расчета задачи и вычислений……………...........14

Описание самой программы……………………………………………16

Тестирование…………………………………………………………….18

Заключение……………………………………………………………....22

Список литературы……………………………………………………...23

Приложение А. Листинг программы…………………………………...24

# **Введение**

Поиск максимальных паросочетаний в графах – важная задача теории графов, находящая применение в таких областях, как распределение ресурсов, составление расписаний, логистика и анализ сетей. В основе этой задачи лежит поиск множества рёбер в графе, не имеющих общих вершин, что позволяет решить ряд практических задач оптимизации. В данной курсовой работе исследуются и реализуются два алгоритма решения задачи поиска максимальных паросочетаний: алгоритм Куна и алгоритм Форда-Фалкерсона. Первый метод основывается на концепции поиска увеличивающих цепей в двудольном графе с использованием обхода в глубину, тогда как второй применяет подход максимального потока, позволяющий решать более общие задачи.

1. **Постановка задачи**
   1. **Описание задачи**

Задача состоит в нахождении максимального паросочетания в двудольном графе. Двудольный граф содержит две группы вершин U и V, между которыми могут быть рёбра, но внутри каждой группы рёбра отсутствуют. Паросочетание — это набор рёбер, таких что никакие два из них не имеют общих вершин. Максимальное паросочетание представляет собой паросочетание с максимальным числом рёбер.

Для разработки программы в рамках курсовой работы необходимо выполнить следующие задачи:

1. Проанализировать поставленные задачи и требования;
2. Разработать интерфейс программы;
3. Реализовать поиск максимальных паросочетаний в двудольных графах;
4. Протестировать разработанную программу.
5. Сделать выводы и отобразить их в графическом формате.

Для разработки программы были выбраны язык программирования Python и среда разработки PyCharm Community Edition 2024.

# **Теоретическая часть задания**

**Паросочетание** M в двудольном графе — произвольное множество рёбер двудольного графа такое, что никакие два ребра не имеют общей вершины.

Вершины двудольного графа, инцидентные рёбрам паросочетания M, называются **покрытыми**, а неинцидентные — **свободными**.

**Числом рёберного покрыти** называется размер минимального рёберного покрытии графа G и обозначается через ρ(G).

Число рёбер в наибольшем паросочетании графа G называется **числом паросочетания**.

**Максимальное паросочетание** (по включению) — это такое паросочетание M в графе G, которое не содержится ни в каком другом паросочетании этого графа, то есть к нему невозможно добавить ни одно ребро, которое бы являлось несмежным ко всем рёбрам паросочетания.

**Максимальное паросочетание** (по мощности) — это паросочетание M в графе G максимальное по мощности.

Паросочетание M графа G называется **совершенным** (или **полным**), если оно покрывает все вершины графа.

**Чередующаяся цепь** — путь в двудольном графе, для любых двух соседних рёбер которого верно, что одно из них принадлежит паросочетанию M, а другое нет.

**Дополняющая цепь** (или **увеличивающая цепь**) — чередующаяся цепь, у которой оба конца свободны.

**Уменьшающая цепь** — чередующаяся цепь, у которой оба конца покрыты.

**Сбалансированная цепь** - чередующаяся цепь, у которой один конец свободен, а другой покрыт.

**Свойство:** В любом графе без изолированных вершин, число паросочетания и число рёберного покрытия в сумме дают число вершин. Если существует совершенное паросочетание, то оба числа равны |V|/2.

Пример максимального и полного паросочетания, чередующейся цепи:

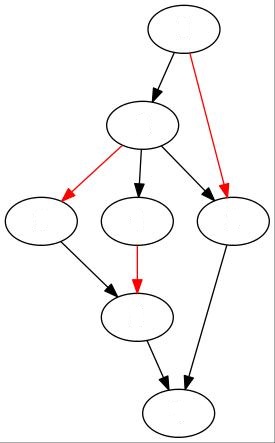


Рисунок 1 - красные рёбра являются рёбрами максимального паросочетания

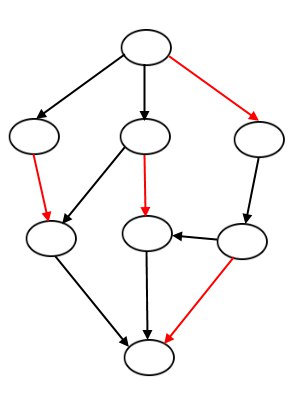


Рисунок 2 - красные рёбра являются рёбрами полного паросочетания.

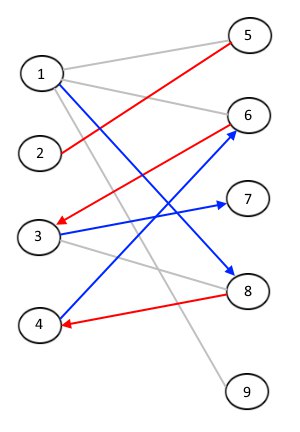


Рисунок 3 - чередующаяся цепь: 1−8−4−6−3−7

Теорема о максимальном паросочетании и дополняющих цепях: Паросочетание M в двудольном графе Gявляется максимальным тогда и только тогда, когда в G нет дополняющей цепи.

1. **Описание алгоритма поставленной задачи**

Курсовая работа направлена на разработку интерактивного приложения для решения задачи поиска максимального паросочетания в двудольных графах. В проекте реализованы методы нахождения паросочетания с использованием **алгоритма Куна** и **метода Форда-Фалкерсона**, а также средства визуализации графов и взаимодействия с пользователем. Ниже представлен пошаговый разбор алгоритма решения задачи:

**Алгоритм Куна:**

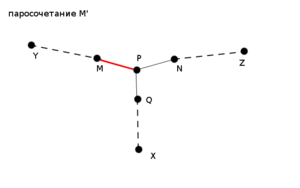


Рисунок 4 - Ребро красного цвета лежит в паросочетании, а черного - нет.

Задан граф G⟨V,E⟩, про который известно, что он двудольный, но разбиение не задано явно. Требуется найти наибольшее паросочетание в нём

Алгоритм можно описать так: сначала возьмём пустое паросочетание, а потом — пока в графе удаётся найти увеличивающую цепь, — будем выполнять чередование паросочетания вдоль этой цепи, и повторять процесс поиска увеличивающей цепи. Как только такую цепь найти не удалось — процесс останавливаем, — текущее паросочетание и есть максимальное.

В массиве matching хранятся паросочетания (v,matching[v]) (Если паросочетания с вершиной v не существует, то matching[v]=−1). А used — обычный массив "посещённостей" вершин в обходе в глубину (он нужен, чтобы обход в глубину не заходил в одну вершину дважды). Функция dfs возвращает true, если ей удалось найти увеличивающую цепь из вершины v, при этом считается, что эта функция уже произвела чередование паросочетания вдоль найденной цепи.

Внутри функции просматриваются все рёбра, исходящие из вершины v, и затем проверяется: если это ребро ведёт в ненасыщенную вершину to, либо если эта вершина to насыщена, но удаётся найти увеличивающую цепь рекурсивным запуском из matching[to], то мы говорим, что мы нашли увеличивающую цепь, и перед возвратом из функции с результатом true производим чередование в текущем ребре: перенаправляем ребро, смежное с to, в вершину v.В основной программе сначала указывается, что текущее паросочетание —пустое (массив matching заполняется числами −1). Затем перебирается вершина v, и из неё запускается обход в глубину dfs, предварительно обнулив массив used

**Время работы алгоритма Куна:**

Алгоритм Куна можно представить как серию из n запусков обхода в глубину на всём графе.

Следовательно, всего этот алгоритм исполняется за время O(nm), где m — количество рёбер, что в худшем случае есть O(n3)

Если явно задано разбиение графа на две доли размером n1 и n2, то можно запускать dfs только из вершин первой доли, поэтому весь алгоритм исполняется за время O(n1m). В худшем случае это составляет O(n21n2).

**Разбор алгоритма Куна в проекте:**

def max\_matching\_kun\_algorithm(n, m, edges):

n: Количество вершин в первой доле графа.

Первая доля содержит вершины с номерами от 0 до n−1.

m: Количество вершин во второй доле графа.

Вторая доля содержит вершины с номерами от n до n+m−1.

edges: Список рёбер графа.

Каждое ребро представлено в виде кортежа (u,v), где u принадлежит первой доле, а v — второй доле.

global g, mt, used

g = defaultdict(list)

Создаётся граф g, представленный в виде списка смежности.

g[u] будет содержать список всех вершин v−n, соединённых с вершиной u.

for u, v in edges:

if u < 0 or u >= n or v < n or v >= n + m:

raise ValueError(

f"Некорректное ребро: ({u}, {v}). Убедитесь, что u в [0, {n - 1}], а v в [{n}, {n + m - 1}].")

g[u].append(v - n)

Перебираются все рёбра (u,v):

Проверяется, что u принадлежит диапазону [0,n−1], а v — диапазону [n,n+m−1]. Если это не так, выбрасывается ошибка.

Если проверка пройдена, v−n добавляется в список соседей u в графе g. Это приводит индексацию правой доли в диапазон [0,m−1].

mt = [-1] \* m

Создаётся массив mt размера mt[to] будет содержать вершину v, соединённую с вершиной to. Если вершина to не покрыта паросочетанием, mt[to]=−1.

for v in range(n):

used = [False] \* n

try\_kuhn(v)

Массив used сбрасывается в [False,…,False]. Это гарантирует, что вершина v будет рассматриваться в текущем вызове try\_kuhn.

Вызывается try\_kuhn(v) для поиска увеличивающего пути.

matching = [(u, v + n) for v, u in enumerate(mt) if u != -1]

Если mt[to]=u=−1, добавляется пара (u,v+n) в результат.

return matching

**Алгоритм Форда-Фалкерсона:**

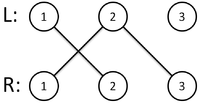


Рисунок 5 - Пример графа G

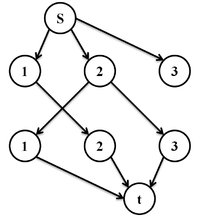


Рисунок 6 - Соответствующий граф G′

Обозначим как p′ путь p из s в t без первого и последнего ребра. Пусть он является дополняющей цепью для исходного графа G, и пусть также существование дополняющей цепи в графе G приводит к существованию пути p′. Тогда из теоремы: если мы на каком-то шаге можем найти новый путь, т.е дополняющую цепь, то мы увеличиваем текущее паросочетание. Если путь найти мы уже не можем, значит дополняющих цепей в графе нет и текущее паросочетание — искомое. Осталось доказать что сделанное предположение действительно верно.

Т. к. p′ — путь в двудольном графе, начинающийся в L и заканчивающийся в R, то он нечетной длины. Вершины в нем не повторяются (т.к. это путь в дереве поиска в глубину). Рассмотрим текущее паросочетание. Согласно поддерживаемому инварианту (R,L)-ребра в паросочетании, а (L,R)-ребра — нет. В таком случае ребра пути p′ можно пронумеровать так, чтобы нечетные ребра были свободными, а четные — покрытыми ребрами текущего паросочетания. Заметим, что путь может начинаться и заканчиваться только в свободной вершине, т. к. из s ведут ребра только в свободные вершины и только из свободных вершин ведут ребра в t. Итак, теперь ясно, что p′ — дополняющая цепь для графа G.

Обратно, пусть существует дополняющая цепь в графе G. В одной из ориентаций она начинается в какой-то свободной вершине и заканчивается в свободной вершине , далее будем рассматривать именно эту ориентацию. Ребра поочередно то не лежат, то лежат в паросочетании, значит в нашей ориентации эти ребра поочередно ориентированы то (L,R), то (R,L). Заметим что эта ориентация совпадает с ориентацией ребер на пути, а значит в нашем ориентированом графе существует путь из свободной вершины u∈L

в свободную вершину v∈R. Нo каждая свободная вершина из L связана ребром с s в графе G′, аналогично каждая свободная вершина из R связана ребром с t. Не сложно заметить, что, в таком случае, t достижим из s, а значит в процессе поиска в глубину будет найден некий s→t путь p и соответствующий ему p′.

**Время работы алгоритма Форда-Фалкерсона:**

Поиск в глубину запускается от вершины s не более чем L раз, т.к. из s ведет ровно L ребер, и при каждом запуске одно из них инвертируется. Сам поиск работает за O(E), каждая инвертация и перезапись паросочетания так жезанимает O(E) времени. Тогда все время алгоритма ограничено O(VE).

**Разбор алгоритма Форда-Фалкерсона в проекте:**

def max\_matching\_ford\_fulkerson(n, m, edges):

n: Количество вершин в первой доле графа.

Первая доля содержит вершины с номерами от 0 до n−1.

m: Количество вершин во второй доле графа.

Вторая доля содержит вершины с номерами от n до n+m−1.

edges: Список рёбер графа.

Каждое ребро представлено в виде кортежа (u,v), где u принадлежит первой доле, а v — второй доле.

Инициализация списка смежности:

adj = {x: [] for x in range(n)}

for u, v in edges:

adj[u].append(v)

Создается словарь adj, где для каждой вершины первой доли хранится список смежных ей вершин второй доли.

Перебирая список рёбер edges, мы заполняем этот словарь.

Инициализация массивов соответствия:

px = {x: -1 for x in range(n)}

py = {y: -1 for y in range(n, n + m)

px: Словарь соответствия для вершин первой доли. Изначально все вершины не связаны (−1).

py: Словарь соответствия для вершин второй доли. Аналогично, все вершины из второй доли не связаны

Основной цикл поиска пути увеличения:

is\_path = True

while is\_path:

is\_path = False

vis = {x: False for x in range(n)}

for x in range(n):

if px[x] == -1:

if dfs(x, vis, px, py, adj):

is\_path = True

Инициализация флага is\_path:

* Этот флаг указывает, удалось ли найти хотя бы один путь увеличения на текущей итерации.

vis:

* Словарь для отслеживания посещенных вершин первой доли. Это необходимо, чтобы избежать повторных посещений одной и той же вершины в рамках одного поиска.

Внутренний цикл:

* Для каждой свободной вершины x первой доли (px[x]=−1) запускается поиск в глубину (dfs).
* Если поиск успешен (возвращает True), значит найден путь увеличения, и флаг is\_path устанавливается в True.

Формирование результата:

matching = [(x, px[x]) for x in range(n) if px[x] != -1]

return matching

1. **Алгоритмы ручного расчета задачи и вычислений**

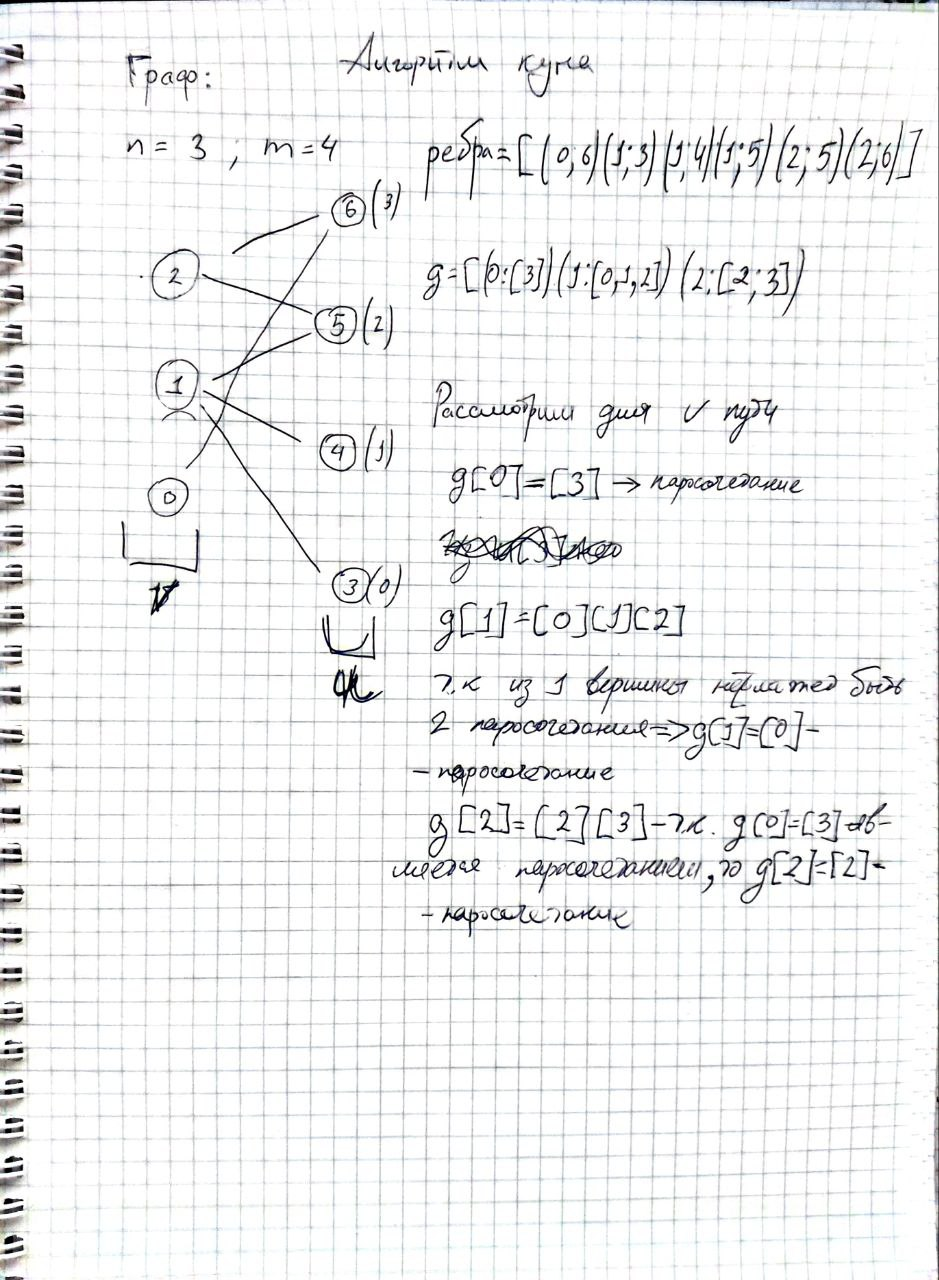


Рисунок 7 - ручная проверка

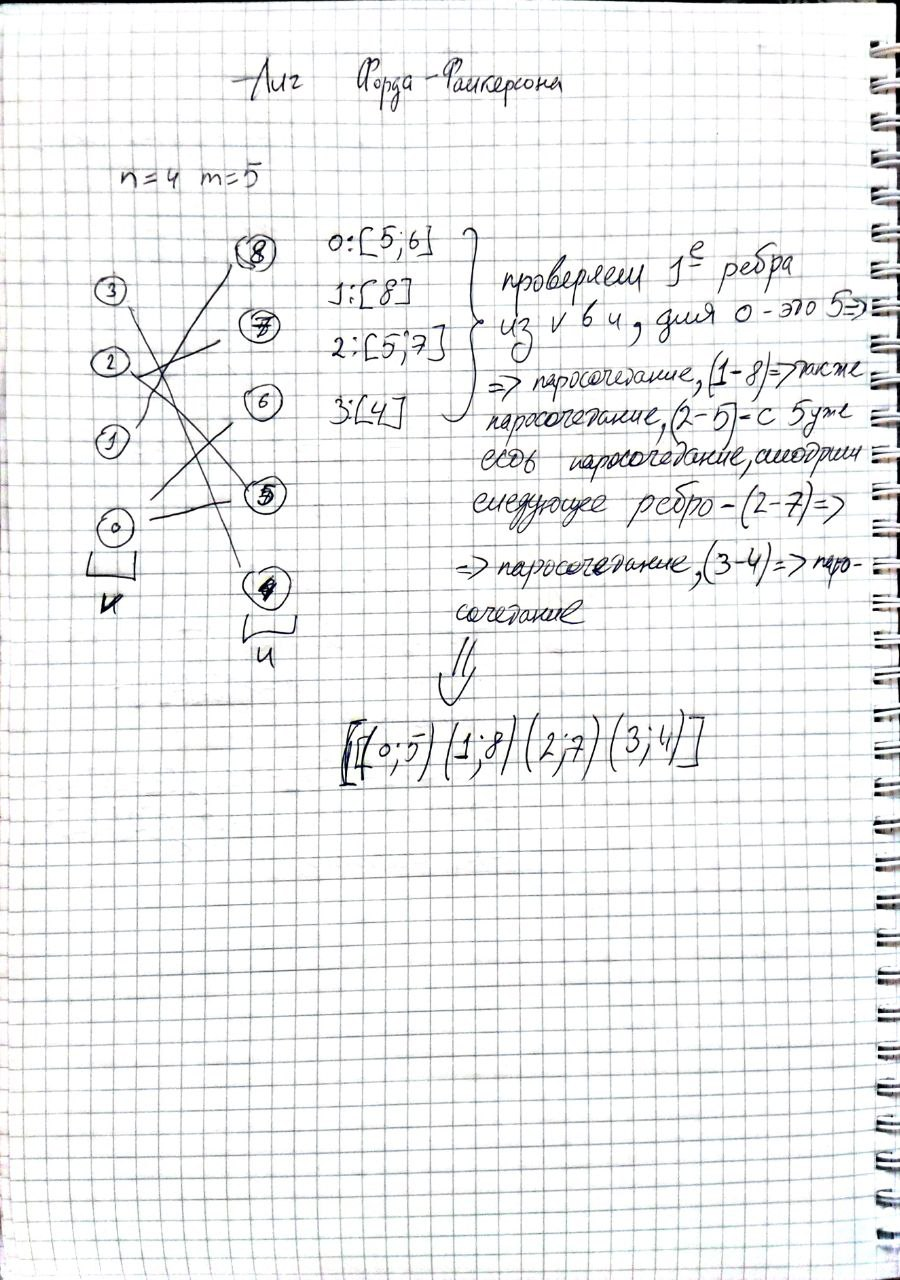


Рисунок 8 - ручная проверка

1. **Описание программы**

Для написания данной программы использован язык программирования Python. Python — это высокоуровневый интерпретируемый язык программирования, известный своей простотой, читабельностью кода и широким спектром библиотек, что делает его удобным для решения задач в области визуализации, алгоритмов и разработки пользовательских интерфейсов.

Проект был создан в виде графического приложения с использованием библиотеки Tkinter для интерфейса, Matplotlib для построения графиков, а также NetworkX для работы с графовыми структурами.

Данная программа является многомодульной, поскольку состоит из следующих основных функций:

**graphics** — основная функция, отвечающая за создание графического интерфейса, включая элементы ввода и вывода данных, выбор алгоритмов и визуализацию графов.

**draw\_graph** — функция построения и отображения графа с подсветкой рёбер, входящих в максимальное паросочетание.

**handle\_manual\_input** — обработка графа, заданного пользователем вручную через текстовый ввод.

**handle\_random\_generation** — генерация случайного двудольного графа на основе заданного количества вершин и вероятности появления рёбер.

**max\_matching\_kun\_algorithm** — реализация алгоритма Куна для поиска максимального паросочетания.

**max\_matching\_ford\_fulkerson** — реализация алгоритма Форда-Фалкерсона для поиска максимального паросочетания.

**generate\_random\_bipartite\_edges** — функция для случайного создания рёбер двудольного графа.

Работа программы начинается с запуска графического интерфейса. Пользователю предлагается:

1.Выбрать алгоритм поиска паросочетания: алгоритм Куна или алгоритм Форда-Фалкерсона.

2.Указать параметры графа:

Количество вершин в первой и второй долях графа.

Вероятность существования рёбер между вершинами двух долей.

3.Задать граф вручную через ввод рёбер или автоматически сгенерировать случайный граф.

4.Если пользователь выбирает генерацию графа, то на экран выводится запрос на количество вершин в каждой доле, а также вероятность появления рёбер между ними. В случае самостоятельного ввода графа программа ожидает матрицу смежности или список рёбер, после чего выполняет обработку данных.

5.После выполнения выбранного алгоритма программа выводит на экран:

Максимальное паросочетание в виде списка рёбер.

Визуализацию графа с выделением рёбер паросочетания.

1. **Тестирование**

Для разработки и тестирования программы использовалась среда выполнения Python с интегрированной средой разработки, поддерживающей библиотеки Tkinter, Matplotlib и NetworkX. Данная среда предоставляет все необходимые инструменты для написания, отладки и тестирования многомодульной программы.

Тестирование проводилось в рабочем порядке на всех этапах разработки, а также после завершения написания кода. В процессе тестирования были выявлены и устранены проблемы, связанные с вводом данных пользователем, корректностью алгоритмов, визуализацией графов и взаимодействием модулей программы. Особое внимание уделялось проверке алгоритмов Куна и Форда-Фалкерсона на корректное выполнение задачи нахождения максимального паросочетания.

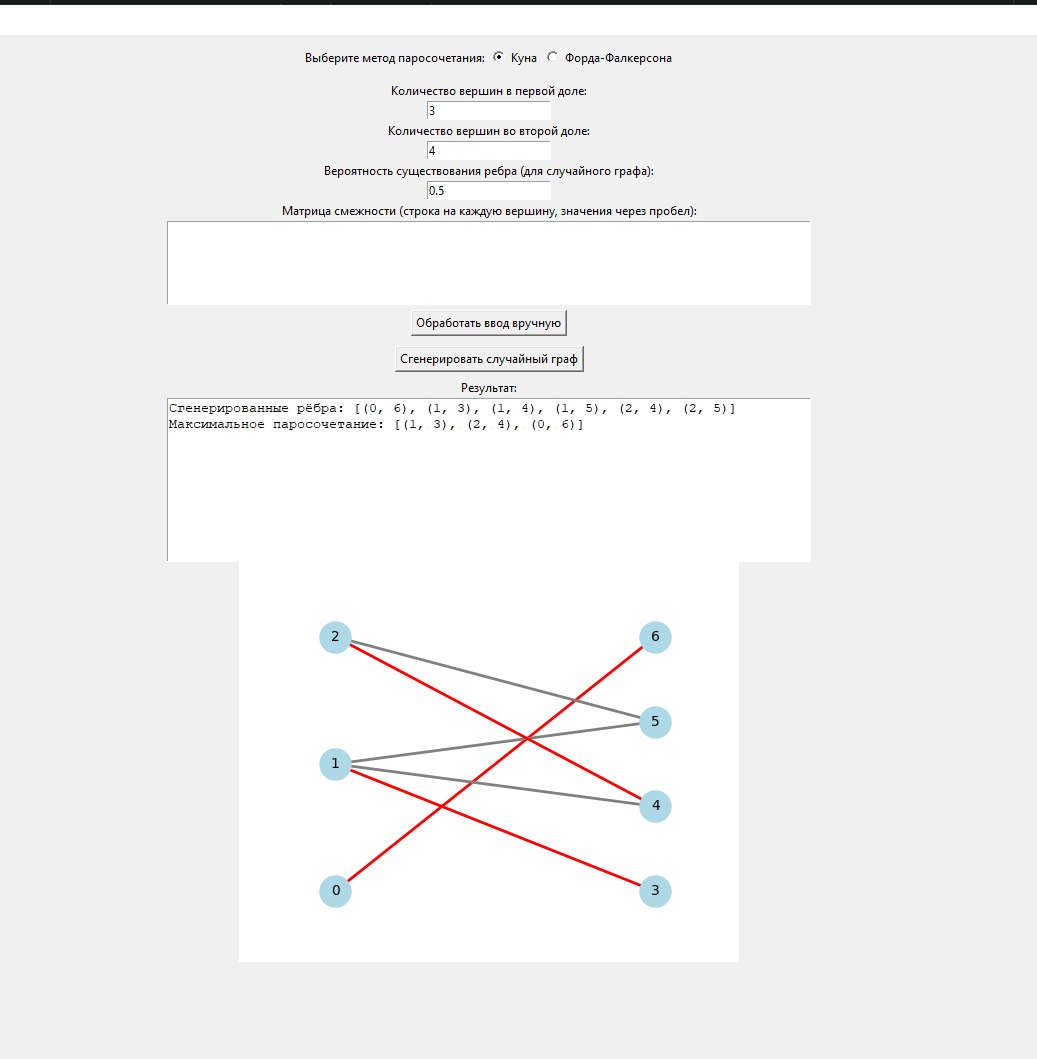
****

Рисунок 9 - n = 3 m = 4 вероятность сущ-ния ребра = 0.5 алгоритм Куна

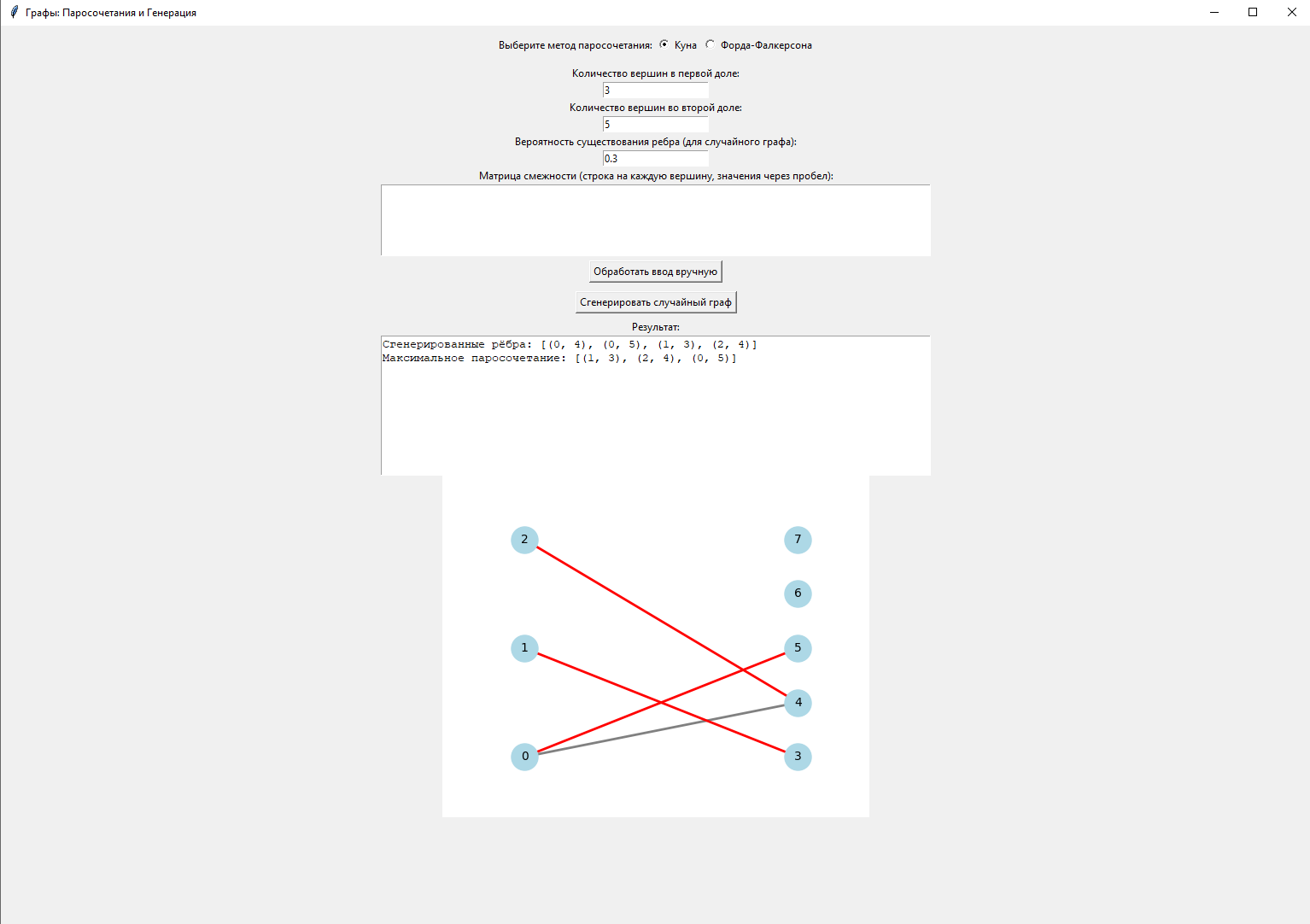
****

Рисунок 10 - n = 3 m = 5 вероятность сущ-ния ребра = 0.5 алгоритм Куна

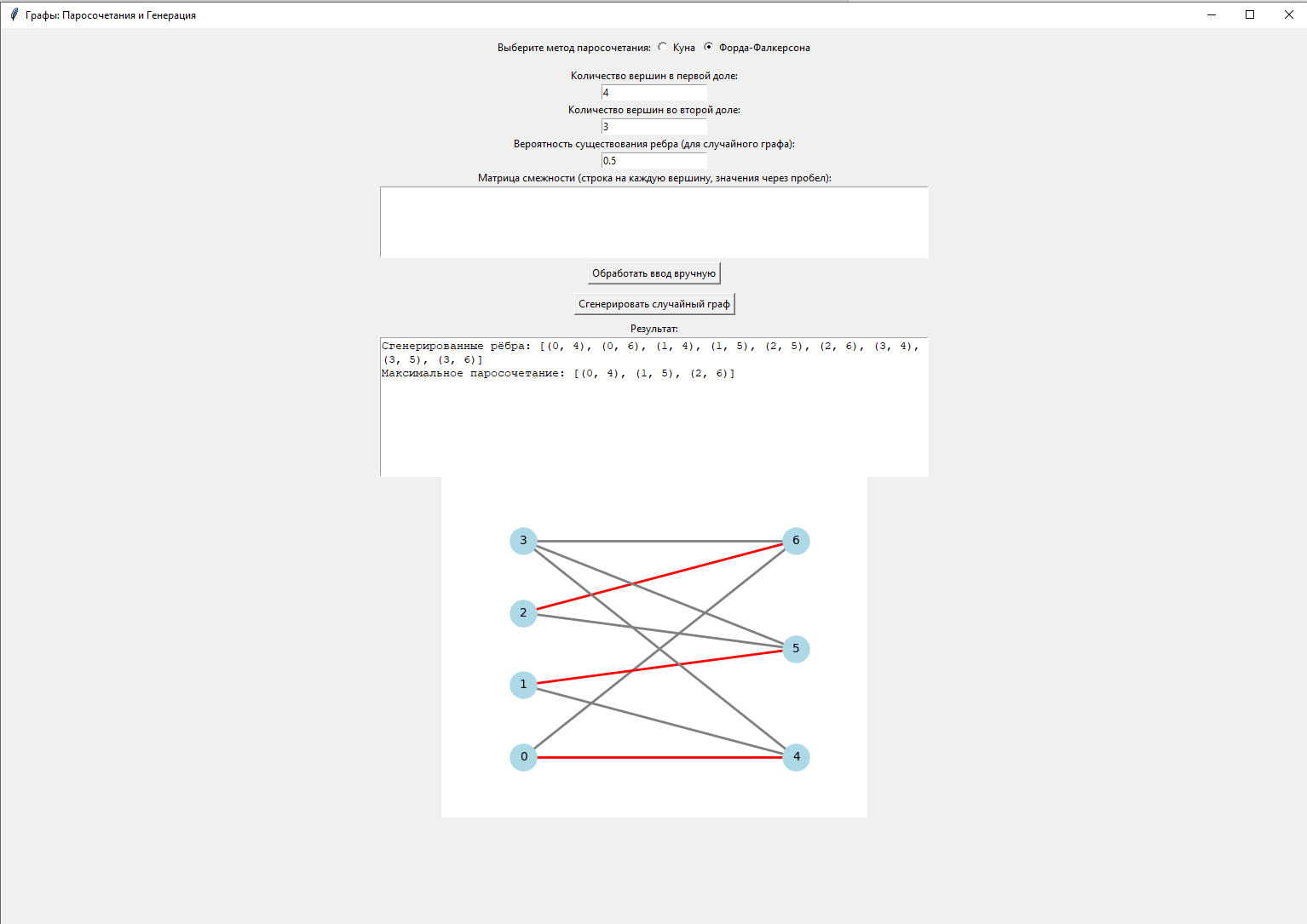
****

Рисунок 11 - n = 4 m = 4 вероятность сущ-ния ребра = 0.5 алгоритм Форда-фалкерсона

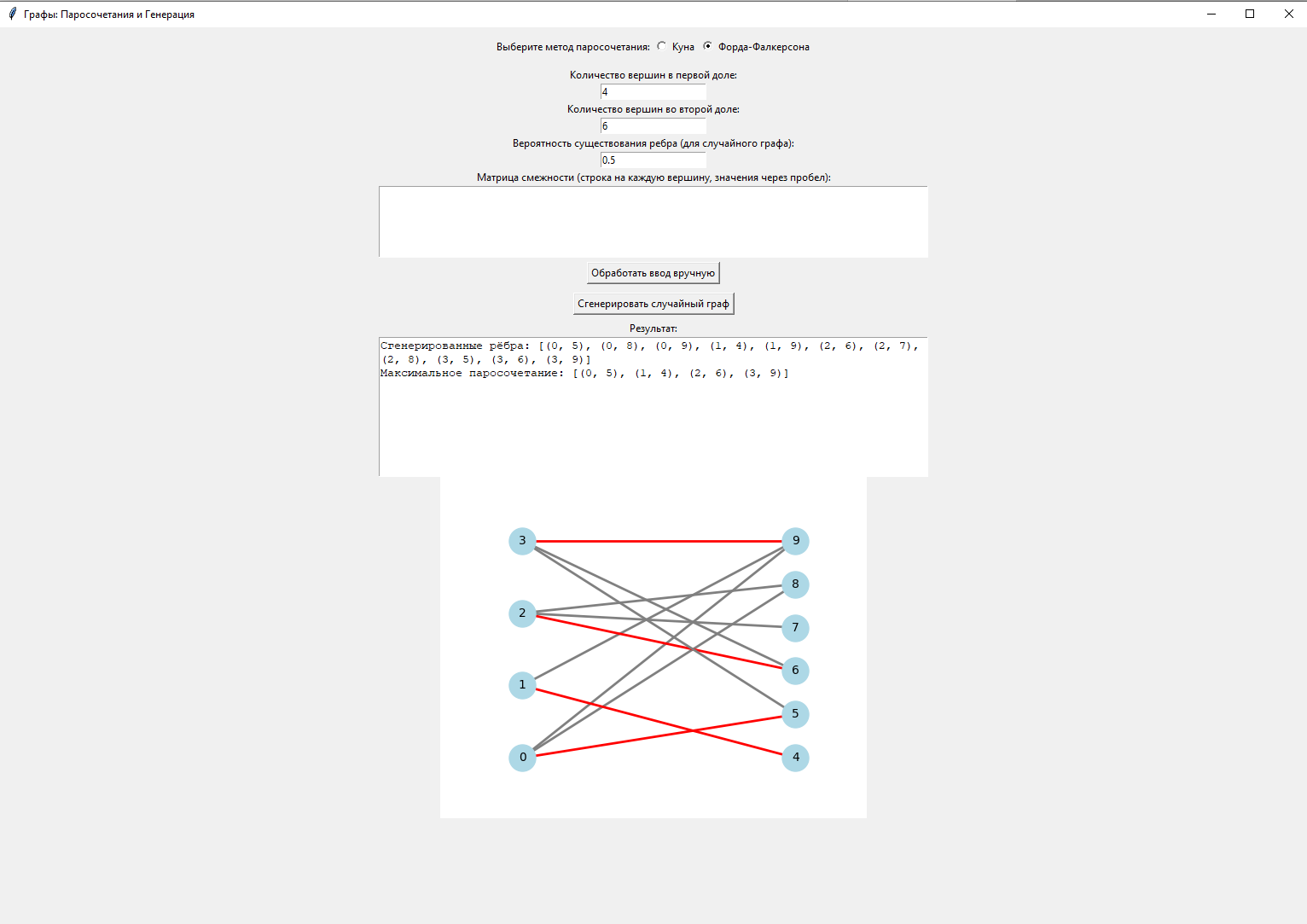
****

Рисунок 12 - n = 4 m = 6 вероятность сущ-ния ребра = 0.5 алгоритм Форда-фалкерсона

Таблица 1 – Описание поведения программы при тестировании

|  |  |  |
| --- | --- | --- |
| Описание теста | Ожидаемый результат | Полученный результат |
| Запуск программы | Вывод графического интерфейса программы | Верно |
| Выбор алгоритма (Куна/Форда-Фалкерсона) | Визуальные подсказки какой из алгоритмов сейчас выбран | Верно |
| Ручной ввод двумерного графа | Видимый ввод символов для графа | Верно |
| Генерирование случайного графа | Вывод самого графа для визуального представления | Верно |
| Вывод графической схемы графа | Вывод корректен и ребра корректны ко всем вершинам | Верно |

**Заключение**

Таким образом, в процессе создания данного проекта была разработана программа, реализующая алгоритмы нахождения максимального паросочетания в двудольных графах с использованием методов Куна и Форда-Фалкерсона. Программа создана с применением языка программирования Python и библиотек Tkinter, Matplotlib и NetworkX.

При выполнении данной курсовой работы были получены навыки проектирования и разработки многомодульных программ, освоены методы работы с графами, включая создание и обработку матриц смежности и списков рёбер, а также визуализацию графов. Также были углублены знания в области алгоритмов теории графов, включая алгоритмы поиска максимального паросочетания.

Основным недостатком программы является ограничение пользовательского интерфейса рамками графического окна Tkinter, что может быть недостаточно удобно для масштабной обработки данных. Тем не менее, функционал программы достаточен для решения поставленной задачи, а возможности визуализации графов и подсветки рёбер паросочетания делают её удобным инструментом.

Программа может быть расширена в будущем, например, за счёт добавления новых алгоритмов работы с графами или улучшения интерфейса с использованием более современных библиотек для создания приложений.

**Список литературы**

1. Кристофидес Н. «Теория графов. Алгоритмический подход» Мир,  1978

2. Оре О. Графы и их применение: Пер. с англ. 1965. 176 с.

1. [Алгоритм Куна для поиска максимального паросочетания — Викиконспекты](https://neerc.ifmo.ru/wiki/index.php?title=%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC_%D0%9A%D1%83%D0%BD%D0%B0_%D0%B4%D0%BB%D1%8F_%D0%BF%D0%BE%D0%B8%D1%81%D0%BA%D0%B0_%D0%BC%D0%B0%D0%BA%D1%81%D0%B8%D0%BC%D0%B0%D0%BB%D1%8C%D0%BD%D0%BE%D0%B3%D0%BE_%D0%BF%D0%B0%D1%80%D0%BE%D1%81%D0%BE%D1%87%D0%B5%D1%82%D0%B0%D0%BD%D0%B8%D1%8F)
2. [Алгоритм Форда-Фалкерсона для поиска максимального паросочетания — Викиконспекты](https://neerc.ifmo.ru/wiki/index.php?title=%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC_%D0%A4%D0%BE%D1%80%D0%B4%D0%B0-%D0%A4%D0%B0%D0%BB%D0%BA%D0%B5%D1%80%D1%81%D0%BE%D0%BD%D0%B0_%D0%B4%D0%BB%D1%8F_%D0%BF%D0%BE%D0%B8%D1%81%D0%BA%D0%B0_%D0%BC%D0%B0%D0%BA%D1%81%D0%B8%D0%BC%D0%B0%D0%BB%D1%8C%D0%BD%D0%BE%D0%B3%D0%BE_%D0%BF%D0%B0%D1%80%D0%BE%D1%81%D0%BE%D1%87%D0%B5%D1%82%D0%B0%D0%BD%D0%B8%D1%8F)
3. [Алгоритм Форда-Фалкерсона - Библиотека алгоритмов на графах](https://urban-sanjoo.narod.ru/ford-fulkerson.html)

**Приложение А.**

**Листинг программы.**

Файл main.py:

import tkinter as tk

import networkx as nx

import matplotlib.pyplot as plt

from matplotlib.backends.backend\_tkagg import FigureCanvasTkAgg

from alg\_kun import max\_matching\_kun\_algorithm, generate\_random\_bipartite\_edges

from alg\_ford\_fulkerson import max\_matching\_ford\_fulkerson

def graphics():

def clear\_output():

output\_text.delete(1.0, tk.END)

def append\_output(text):

output\_text.insert(tk.END, text + "\n")

def draw\_graph(edges, bipartite=False, n=0, m=0, matching=None):

fig, ax = plt.subplots(figsize=(5, 4))

G = nx.Graph()

if bipartite:

G.add\_nodes\_from(range(n), bipartite=0)

G.add\_nodes\_from(range(n, n + m), bipartite=1)

pos = nx.bipartite\_layout(G, nodes=range(n))

else:

pos = nx.spring\_layout(G)

G.add\_edges\_from(edges)

# Подсветка рёбер паросочетания

edge\_colors = []

for edge in edges:

if matching and edge in matching:

edge\_colors.append('red') # Цвет для рёбер паросочетания

else:

edge\_colors.append('gray') # Обычный цвет для остальных рёбер

nx.draw(G, pos, with\_labels=True, ax=ax, node\_color='lightblue', edge\_color=edge\_colors,

node\_size=500, font\_size=10, width=2)

return fig

def handle\_manual\_input():

clear\_output()

try:

nonlocal canvas\_widget

if canvas\_widget:

canvas\_widget.destroy()

n = int(n\_entry.get())

m = int(m\_entry.get())

edges = []

# Парсинг списка рёбер

edges\_input = input\_text.get(1.0, tk.END).strip().split("\n")

for edge\_str in edges\_input:

edge = tuple(map(int, edge\_str.split()))

if len(edge) != 2:

raise ValueError("Каждое ребро должно содержать ровно два числа.")

edges.append(edge)

method = selected\_method.get()

if method == "Куна":

matching = max\_matching\_kun\_algorithm(n, m, edges)

elif method == "Форда-Фалкерсона":

matching = max\_matching\_ford\_fulkerson(n, m, edges)

else:

raise ValueError("Неизвестный метод паросочетания")

append\_output(f"Максимальное паросочетание: {matching}")

fig = draw\_graph(edges, bipartite=True, n=n, m=m, matching=matching)

canvas = FigureCanvasTkAgg(fig, master=canvas\_frame)

canvas\_widget = canvas.get\_tk\_widget()

canvas\_widget.pack()

except Exception as e:

append\_output(f"Ошибка: {str(e)}")

def handle\_random\_generation():

clear\_output()

try:

nonlocal canvas\_widget

if canvas\_widget:

canvas\_widget.destroy()

n = int(n\_entry.get())

m = int(m\_entry.get())

edge\_probability = float(probability\_entry.get())

edges = generate\_random\_bipartite\_edges(n, m, edge\_probability)

method = selected\_method.get()

if method == "Куна":

matching = max\_matching\_kun\_algorithm(n, m, edges)

elif method == "Форда-Фалкерсона":

matching = max\_matching\_ford\_fulkerson(n, m, edges)

else:

raise ValueError("Неизвестный метод паросочетания")

append\_output(f"Сгенерированные рёбра: {edges}")

append\_output(f"Максимальное паросочетание: {matching}")

fig = draw\_graph(edges, bipartite=True, n=n, m=m, matching=matching)

canvas = FigureCanvasTkAgg(fig, master=canvas\_frame)

canvas\_widget = canvas.get\_tk\_widget()

canvas\_widget.pack()

except Exception as e:

append\_output(f"Ошибка: {str(e)}")

root = tk.Tk()

root.title("Графы: Паросочетания и Генерация")

root.geometry("1000x700")

selected\_method = tk.StringVar(value="Куна")

method\_frame = tk.Frame(root)

method\_frame.pack(pady=10)

tk.Label(method\_frame, text="Выберите метод паросочетания:").pack(side=tk.LEFT)

tk.Radiobutton(method\_frame, text="Куна", variable=selected\_method, value="Куна").pack(side=tk.LEFT)

tk.Radiobutton(method\_frame, text="Форда-Фалкерсона", variable=selected\_method, value="Форда-Фалкерсона").pack(

side=tk.LEFT)

# Ввод данных

tk.Label(root, text="Количество вершин в первой доле:").pack()

n\_entry = tk.Entry(root)

n\_entry.pack()

tk.Label(root, text="Количество вершин во второй доле:").pack()

m\_entry = tk.Entry(root)

m\_entry.pack()

tk.Label(root, text="Вероятность существования ребра (для случайного графа):").pack()

probability\_entry = tk.Entry(root)

probability\_entry.pack()

tk.Label(root, text="Матрица смежности (строка на каждую вершину, значения через пробел):").pack()

input\_text = tk.Text(root, height=5)

input\_text.pack()

# Кнопки

manual\_button = tk.Button(root, text="Обработать ввод вручную", command=handle\_manual\_input)

manual\_button.pack(pady=5)

random\_button = tk.Button(root, text="Сгенерировать случайный граф", command=handle\_random\_generation)

random\_button.pack(pady=5)

# Вывод данных

tk.Label(root, text="Результат:").pack()

output\_text = tk.Text(root, height=10)

output\_text.pack()

# Холст для графиков

canvas\_frame = tk.Frame(root)

canvas\_frame.pack(fill=tk.BOTH, expand=True)

canvas\_widget = None

root.mainloop()

graphics()

Файл alg\_ford\_fulkerson.py

def dfs(x, vis, px, py, adj):

if vis[x]:

return False

vis[x] = True

for y in adj[x]:

if py[y] == -1:

py[y] = x

px[x] = y

return True

elif dfs(py[y], vis, px, py, adj):

py[y] = x

px[x] = y

return True

return False

def max\_matching\_ford\_fulkerson(n, m, edges):

adj = {x: [] for x in range(n)}

for u, v in edges:

adj[u].append(v)

px = {x: -1 for x in range(n)}

py = {y: -1 for y in range(n, n + m)}

is\_path = True

while is\_path:

is\_path = False

vis = {x: False for x in range(n)}

for x in range(n):

if px[x] == -1:

if dfs(x, vis, px, py, adj):

is\_path = True

matching = [(x, px[x]) for x in range(n) if px[x] != -1]

return matching

Файл alg\_kun.py

import random

from collections import defaultdict

g = defaultdict(list)

mt = []

used = []

def generate\_random\_bipartite\_edges(n, m, edge\_probability):

edges = []

for i in range(n):

for j in range(m):

if random.random() < edge\_probability:

edges.append((i, n + j))

return edges

def try\_kuhn(v):

if used[v]:

return False

used[v] = True

for to in g[v]:

if mt[to] == -1 or try\_kuhn(mt[to]):

mt[to] = v

return True

return False

def max\_matching\_kun\_algorithm(n, m, edges):

global g, mt, used

g = defaultdict(list)

for u, v in edges:

if u < 0 or u >= n or v < n or v >= n + m:

raise ValueError(

f"Некорректное ребро: ({u}, {v}). Убедитесь, что u в [0, {n - 1}], а v в [{n}, {n + m - 1}].")

g[u].append(v - n)

mt = [-1] \* m

for v in range(n):

used = [False] \* n

try\_kuhn(v)

matching = [(u, v + n) for v, u in enumerate(mt) if u != -1]

return matching