МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное автономное образовательное учреждение
высшего образования «Санкт-Петербургский политехнический университет
Петра Великого»

Институт компьютерных наук и кибербезопасности

Направление: 02.03.01 Математика и компьютерные науки

Теория графов

ОТЧЁТ О ВЫПОЛНЕНИИ ЛАБОРАТОРНЫХ РАБОТ №1 - 5

«Алгоритмы на графах»

Вариант 23

Студент,
группы 5130201/30002 _____ Михайлова А. А.

Преподаватель _____ Востров А. В.

«____»_____ 2025 г.

Санкт-Петербург, 2025

Содержание

Введение

Отчёт состоит из описания 5 лабораторных работ. Каждая работа включает в себя реализацию различных алгоритмов на случайно созданных связных ациклических графах. Генерация графов осуществляется с использованием распределения Эрланга.

ЛАБОРАТОРНАЯ РАБОТА № 1

1. Сформировать случайным образом связный ациклический граф в соответствии с заданным распределением (параметры распределения задаются как константы) с вводимым пользователем количеством вершин. Распределение брать из справочника Вадзинского (есть в списке литературы).

2. Реализовать метод Шимбелла для сгенерированной с помощью заданного распределения весовой матрицы (пользователь вводит количество ребер), в ответе представить минимальный и максмиальный путь (в виде матрицы).

3. Определить возможность построения маршрута от одной заданной точки до другой (вершины вводит пользователь) и указывать количество таковых маршрутов.

ЛАБОРАТОРНАЯ РАБОТА № 2

1. Для заданных графов (случайно сгенерированных в предыдущей работе) реализовать один из алгоритмов:

a) упорядочить граф по алгоритму Фалкерсона,

b) выполнить обход вершин графа поиском в ширину,

c) выполнить обход вершин графа поиском в глубину,

d) найти точки сочленения в графе,

e) выполнить обход ребер графа поиском в ширину,

f) выполнить обход ребер графа поиском в глубину.

2. Сгенерировать весовую матрицу (положительную или с отрицательными весами – выбирает пользователь) и найти кратчайший путь для двух выбранных пользователем вершин. В результате выводится не только расстояние, но и сам путь в виде последовательности вершин. Для алгоритмов Дейкстры и Беллмана-Форда необходимо выводить вектор расстояний, для Флойда-Уоршалла - матрицу расстояний.

g) классический алгоритм Дейкстры,

h) алгоритм Дейкстры для отрицательных весов,

i) алгоритм Беллмана-Форда,

j) алгоритм Флойда-Уоршалла.

3. Сравнить скорости работы реализованных алгоритмов (по количеству итераций).

ЛАБОРАТОРНАЯ РАБОТА № 3

1. Сформировать связный ациклический граф случайным образом в соответствии с заданным распределением. На его основе сгенерировать матрицы пропускных способностей и стоимости.

2. Для полученного графа найти максимальный поток по алгоритму Форда-Фалкерсона (или любого из перечисленных в лекции).

3. Вычислить поток минимальной стоимости (в качестве величины потока брать значение, равное [2/3*max], где max – максимальный поток). Для этого использовать ранее реализованные алгоритмы Дейкстры и/или Беллмана – Форда, Флойда-Уоршалла.

ЛАБОРАТОРНАЯ РАБОТА № 4

1. Для заданных графов (случайно сгенерированных в первой работе) найти число остовных деревьев, используя матричную теорему Кирхгофа.

2. Построить минимальный по весу остов для сгенерированного (неориентированного) взвешенного графа, используя алгоритм:

a) Краскала;

b) Борувки;

c) Прима.

Полученный остов закодировать с помощью кода Прюфера и декодировать его. Сохранять веса при кодировании обязательно.

3. Реализовать на исходном графе и полученном остове (по выбору пользователя) алгоритм:

d) нахождения максимального независимого множества вершин;

e) нахождения максимального независимого множества ребер;

f) нахождения минимального вершинного покрытия;

g) нахождения минимального реберного покрытия;

h) нахождения минимальной раскраски графа.

ЛАБОРАТОРНАЯ РАБОТА № 5

1. Для заданных графов (случайно сгенерированных в первой работе) проверить, является ли граф эйлеровым и гамильтоновым.

2. Проверить, является ли граф эйлеровым. Если нет, то модифицировать граф (показывать, что изменено). Построить эйлеров цикл.

3. Проверить, является ли граф гамильтоновым. Если нет, то модифицировать граф (показывать, что изменено). Решить задачу коммивояжера на гамильтоновом графе (все гамильтоновы циклы с суммарным весом выводим либо на экран, если их мало, либо в файл). За реализованные эвристики отдельные бонусы (по умолчанию - полный перебор).

Лабораторные работы выполнены на языке C++ в среде разработки Xcode.

1 Математическое описание

1.1 Граф

Простой граф $G(V, E)$ есть совокупность двух множеств – непустого множества V и множества E неупорядоченных пар различных элементов множества V. Множество V называется множеством вершин, множество E называется множеством рёбер.

$G(V, E) = \langle V, E \rangle, \quad V \neq \emptyset, \quad E \subseteq V \times V, \quad \{v, v\} \notin E, \quad v \in V,$

то есть множество E состоит из 2-элементных подмножеств множества V.

Сопутствующие термины:

- Вершина графа G есть элемент множества вершин $v \in V(G)$;
- Ребро графа G есть элемент множества рёбер $e \in E(G)$, или $e = \{v_1, v_2\}$, где $v_1 \in V(G)$, $v_2 \in V(G)$;
- Элементами графа G называются его вершины $v \in V(G)$ и рёбра $e \in E(G)$ графа;
- Вершина v инцидентна ребру e, если $v \in e$; тогда ещё говорят, что e есть ребро при v;
- Соседние (смежные) вершины есть такие вершины v_1 и v_2, что $\{v_1, v_2\} \in E(G)$ или, другими словами, обе вершины являются конечными для одного ребра;
- Степень вершины $d(v)$ есть количество инцидентных ей рёбер.

1.2 Распределение Эрланга

Функция распределения:

$$F(x) = 1 - e^{-\lambda x} \sum_{i=0}^{m-1} \frac{(\lambda x)^i}{i!} = e^{-\lambda x} \sum_{i=m}^{\infty} \frac{(\lambda x)^i}{i!}$$

1. Распределение Эрланга порядка m описывает распределение случайной величины $X = X_1 + X_2 + \ldots + X_m$, представляющей собой сумму m независимых случайных величин, каждая из которых распределена по показательному закону распределения с одним и тем же параметром λ.
2. При $m = 1$ распределение Эрланга совпадает с показательным распределением.

3. Распределение Эрланга порядка m является частным случаем гамма-распределения, параметром формы которого является целое положительное число m (то есть $\alpha = m$). Все характеристики распределения Эрланга порядка m определяются по тем же формулам, что и соответствующие характеристики гамма-распределения (с заменой во всех формулах α на m).

4. Распределение Эрланга тесно связано с распределением Пуассона.

5. Распределение Эрланга порядка $m + 1$ с параметром масштаба $\lambda = 1$ иногда называют показательным распределением с параметром формы m.

6. Распределение Эрланга порядка m является непрерывным аналогом отрицательного биномиального распределения с параметрами m, p, которое описывает распределение суммы m независимых случайных величин, каждая из которых имеет геометрическое распределение 1 с одним и тем же параметром p.

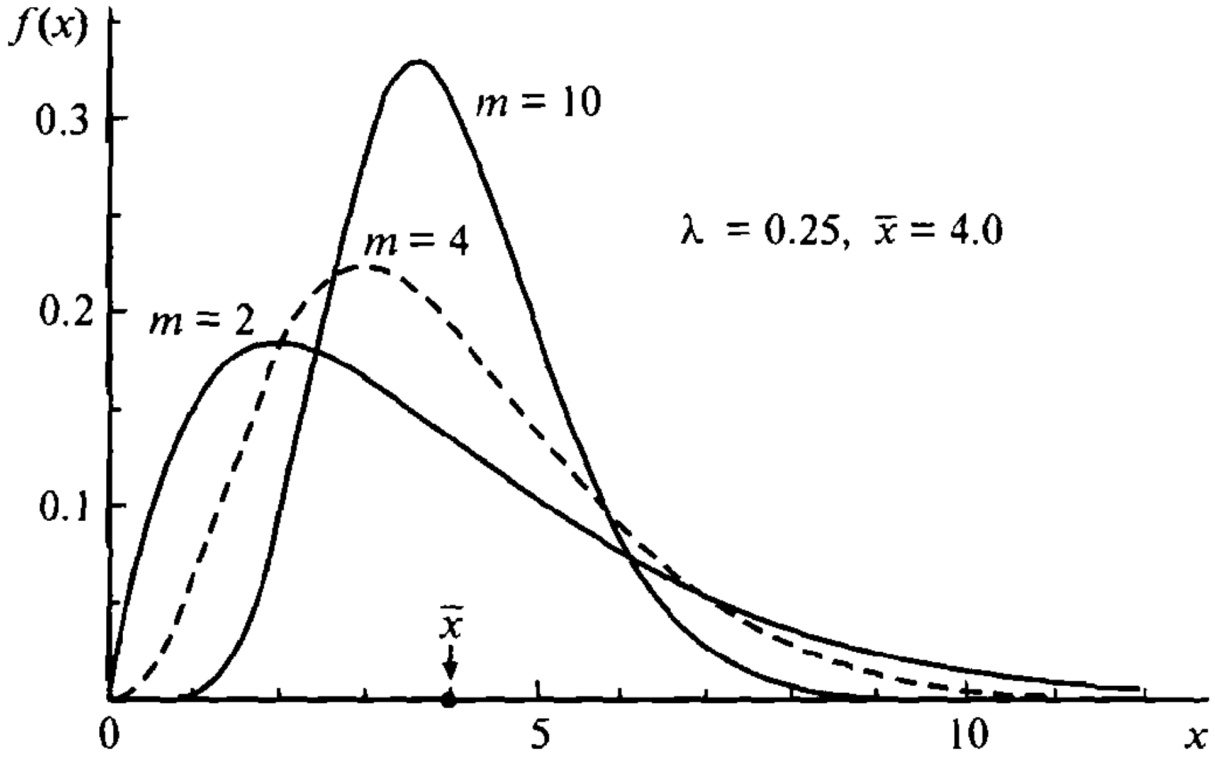

Рис. 1: Плотность вероятности распределения Эрланга

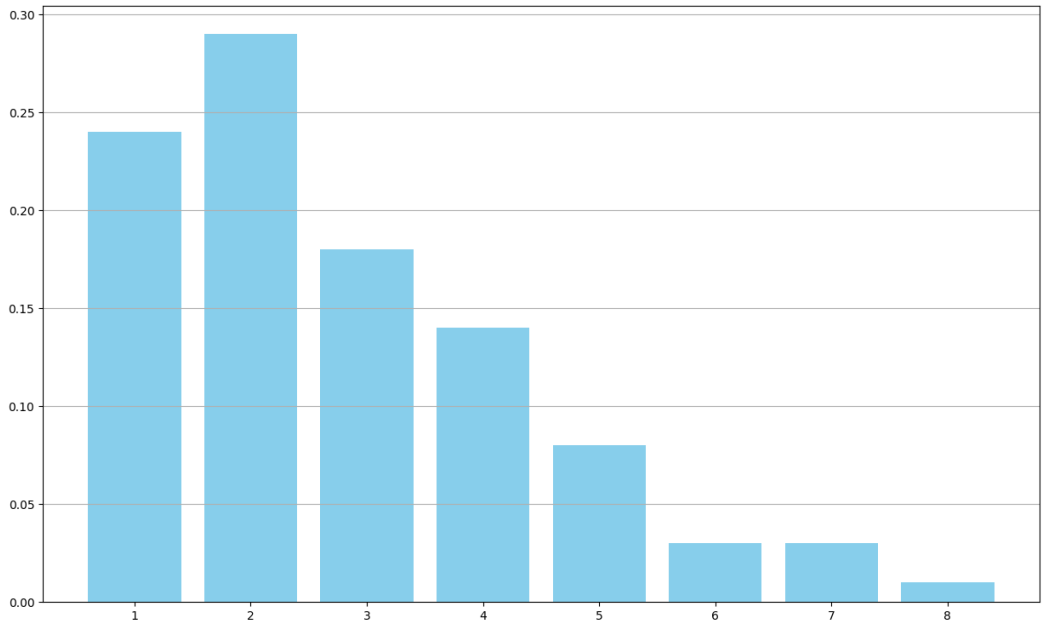

Рис. 2: Плотность вероятности распределения Эрланга

1.3 Метод Шимбелла

Метод Шимбелла позволяет находить минимальные (максимальные) пути между вершинами, состоящие из заданного количества ребер.

- Операция умножения двух величин a и b при возведении матрицы в степень соответствует их алгебраической сумме, то есть:

$$a * b = b * a \rightarrow a + b = b + a$$
$$a * 0 = 0 * a = 0 \rightarrow a + 0 = 0$$

- Операция сложения двух величин a и b заменяется выбором из этих величин максимума или минимума, то есть:

$$a + b = b + a = \min(\max)a, b$$

Нули при этом игнорируются.
$w_{ij} = \min(\max(w_{i1} + w_{1j}), \ldots, (w_{in} + w_{nj}))$

1.4 Алгоритм Дейкстры

Алгоритм Дейкстры — это алгоритм для вычисления кратчайших путей в графе с положительными весами ребер. Описание алгоритма:

- Инициализация:
  - Создается массив D, где $D[i]$ - это минимальное расстояние от вершины 0 до вершины i. Изначально все $D[i]$ равны бесконечности, кроме $D[0]$, которое равно 0.
  - Создается множество вершин M, которое содержит все вершины, кроме вершин 0. Это множество будет использоваться для выбора вершин, которые еще не были рассмотрены.
- Выбирается вершина v с наименьшим значением $D[v]$ из множества M и удаляется из M. Если M пустое, алгоритм заканчивается.
- Для каждой вершины w, которая еще не рассмотрена (т.е. не в M), обновляем $D[w]$ с учетом ребра (v, w) следующим образом: если $D[v]+$ длина ребра (v, w) меньше, чем $D[w]$, то $D[w]$ присваивается значение $D[v]+$ длина ребра (v, w).
- Возвращаемся к шагу 2 и повторяем процесс до тех пор, пока множество M не станет пустым.

Трудоемкость алгоритма – $O(p^2)$. Недостаток алгоритма: некорректно работает, если граф имеет ребра (дуги) отрицательного веса. В моем варианте нужно было использовать алгоритм Дейкстры для графа с положительными весами дуг.

1.5 Обход графа поиском в глубину

Обход в глубину – один из основных методов обхода графа, часто используемый для проверки связности, поиска цикла и компонент сильной связности и для топологической сортировки.

Общая идея алгоритма состоит в следующем: для каждой не пройденной вершины необходимо найти все не пройденные смежные вершины и повторить поиск для них.

1. Выбираем любую вершину из еще не пройденных, обозначим ее как u;
2. Запускаем процедуру dfs(u):
   - Помечаем вершину u как пройденную;

- Для каждой не пройденной смежной с u вершиной (назовем ее v) запускаем dfs(v);

3. Повторяем шаги 1 и 2, пока все вершины не окажутся пройденными.

Время работы алгоритма оценивается как O(V+E)

1.6 Алгоритм Форда-Фалкерсона

Алгоритм Форда-Фалкерсона решает задачу нахождения максимального потока в сети. Идея алгоритма заключается в следующем:

- Изначально величине потока присваивается значение 0 для всех u и v в транспортной сети.
- Затем величина потока итеративно увеличивается посредством поиска, увеличивающего пути от источника s к стоку t.
- Процесс повторяется, пока можно найти увеличивающий путь.

Алгоритм не конкретизирует, какой именно путь мы ищем на шаге 2 или как мы это делаем. По этой причине алгоритм гарантированно сходится только для целых пропускных способностей, но даже для них при больших значениях пропускных он может работать очень долго. Алгоритм Форда-Фалкерсона может иметь асимптотическую сложность O(VE).

Алгоритм поиска заданного потока минимальной стоимости может использовать любой из алгоритмов поиска кратчайшего пути из истока в сток. Для заданного суммарного потока - n последовательно n раз ищется кратчайший по стоимости путь потока 1 и далее на этом пути пропускная способность всех ребер уменьшается на 1. Далее n таких стоимостей суммируются. Целевая функция:

$$\min\left(\sum_{(u,v)\in E} c(u,v)\cdot f(u,v)\right),$$

где E — множество всех рёбер в графе, $c(u,v)$ — стоимость потока по ребру (u,v), $f(u,v)$ — поток по ребру (u,v), $\sum_{(u,v)\in E}$ — сумма стоимости потока по всем рёбрам в графе.

Асимптотика алгоритма - $O(n^*(V+E))$.

1.7 Остов

Пусть G = (V, E) - граф. Остовный подграф графа G = (V, E) – подграф, содержащий все вершины. Остовный подграф, являющийся деревом,

называется остовом. Несвязный граф не имеет остова. Связный граф может иметь много остовов.

Минимальное остовное дерево графа G - остовное дерево $T = (V, E_T)$ графа G, сумма весов ребер которого минимальна.

1.8 Матричная теорема Кирхгофа

- **Матрица Кирхгофа**
  Матрица B графа G размера $n \times n$ определяется следующим образом:

$$B[i,j] = \begin{cases} -1, & \text{если вершины с номерами } i \text{ и } j \text{ смежны;} \\ 0, & \text{если } i \neq j; \\ \deg(v_i), & \text{если } i = j. \end{cases}$$

- **Свойства матриц Кирхгофа**

  1. Суммы элементов в каждой строке и каждом столбце матрицы равны 0.
  2. Алгебраические дополнения всех элементов матрицы равны между собой.
  3. Определитель матрицы Кирхгофа равен нулю.

- **Матричная теорема Кирхгофа**
  Число остовных деревьев в связном графе G порядка $n \geq 2$ равно алгебраическому дополнению любого элемента матрицы Кирхгофа B графа G.

1.9 Алгоритм Прима

Алгоритм Прима — это алгоритм построения минимального остовного дерева взвешенного связного неориентированного графа. Он был открыт в 1930 году чешским математиком Войцехом Ярником, затем повторно открыт Робертом Примом в 1957 году и независимо от них Э. Дейкстрой в 1959 году.

Алгоритм работает следующим образом:

1. Берётся произвольная вершина, и находится ребро с наименьшей стоимостью, инцидентное этой вершине.
2. Найденное ребро и соединяемые им две вершины образуют дерево.
3. Рассматриваются рёбра графа, один конец которых — уже принадлежащая дереву вершина, а другой — нет. Из этих рёбер выбирается ребро наименьшей стоимости.

4. Присоединяется выбранное ребро к дереву. Рост дерева продолжается до тех пор, пока не будут исчерпаны все вершины исходного графа.

Результатом работы алгоритма является остовное дерево минимальной стоимости. Общая асимптотика составит $O(E + V \log V)$.

1.10 Кодирование Прюфера

Алгоритм кодирования Прюфера состоит из следующих шагов:
Пока количество вершин больше двух:

- Выбираем лист v с минимальным номером.
- В код Прюфера добавляем номер вершины, смежной с v.
- Вершина v и инцидентное ей ребро удаляются из дерева.

Полученная последовательность называется кодом Прюфера для заданного дерева.

Рядом с задачей построения кода Прюфера стоит задача восстановления закодированного дерева. Рассмотрим алгоритм декодирования подробно. Помимо кода нам нужен список всех вершин графа. Мы знаем, что код Прюфера состоит из n-2 вершин, где n – это число вершин в графе. То есть, мы можем по размеру кода определить число вершин в закодированном дереве. В результате, в начале работы алгоритма мы имеем массив из кода Прюфера размера n-2 и массив всех вершин графа: [1...n]. Далее n-2 раза повторяется такая процедура: берется первый элемент массива, содержащего код Прюфера, и в массиве с вершинами дерева производится поиск наименьшей вершины, не содержащейся в массиве с кодом. Найденная вершина и текущий элемент массива с кодом Прюфера составляют ребро дерева. Данные вершины удаляются из соответствующих массивов, и описанная выше процедура повторяется, пока в массиве с кодом не закончатся элементы. В конце работы алгоритма в массиве с вершинами графа останется две вершины, они составляют последнее ребро дерева. В результате получаем список всех ребер графа, который был закодирован. Асимптотика алгоритмов - O(n).

1.11 Минимальная раскраска графа

Правильной раскраской графа $G(V, E)$ называется такое отображение ϕ из множества вершин V в множество красок $\{c_1 \dots c_t\}$, что для любых двух смежных вершин u и v выполняется $\phi(u)! = \phi(v)$. Так же её называют t-раскраской.

Минимальная раскраска графа — это такая раскраска, которая использует наименьшее количество цветов среди всех возможных вариантов раскраски.

Пусть $\chi(G)$ обозначает минимальное количество цветов, необходимых для раскраски графа G. Тогда можно записать:

$$\chi(G) = \min\{|C| : C \text{ является раскраской графа } G\}$$

1.12 Эйлеров граф

Эйлеров граф — это граф, в котором существует эйлеров цикл, то есть простой цикл, проходящий по всем рёбрам графа.

Эйлеров граф является эйлеровым тогда и только тогда, когда он связен и степени всех его вершин чётны.

Эйлеровым путем называется простой путь, содержащий все ребра. В данной работе производится поиск Эйлерового цикла. Сначала осуществляется проверка, является ли граф Эйлеровым. Если нет, то граф достраивается так, чтобы степень каждой вершины была четной.

Для поиска эйлерова цикла в графе можно использовать алгоритм, основанный на обходе в глубину (DFS). Он состоит из следующих шагов:

- Выбрать произвольную вершину графа и поместить её в стек.
- Пока стек не пуст, повторять следующие действия:
  - извлечь вершину из стека;
  - если у этой вершины есть непосещённые соседние вершины, выбрать одну из них и поместить её в стек;
  - если у вершины нет непосещённых соседних вершин, добавить её в список эйлерова цикла.
- Вернуться к списку Эйлерова цикла в обратном порядке. Асимптотика алгоритма поиска эйлерова цикла оценивается как O(V).

Этот алгоритм гарантирует нахождение эйлерова цикла, если он существует в графе. Асимптотика работы алгоритма оценивается как O(V+E).

1.13 Гамильтонов граф

Гамильтоновым графом называется граф, содержащий гамильтонов цикл.

Гамильтоновым циклом называется простой цикл, который проходит через все вершины рассматриваемого графа.

Теорема Дирака. Если количество вершин (n) не менее 3 и степень каждой вершины не менее n/2 для любой вершины неориентированного графа G, то G — гамильтонов граф.

Проверка графа на гамильтоновость осуществляется при помощи полного перебора всех перестановок длины n. При этом если граф не Гамильтонов, он достраивается до гамильтонова по теореме Дирака.

1.14 Задача коммивояжера

Задача коммивояжера — это одна из самых известных задач комбинаторной оптимизации. Она заключается в поиске самого выгодного маршрута, проходящего через указанные города хотя бы по одному разу с последующим возвращением в исходный город. Асимптотика алгоритма поиска самого выгодного маршрута — $O(n!)$.

Можно описать задачу коммивояжера, как задачу поиска наименьшего значения целевой функции, где целевая функция это:

$$\sum_{i=1}^{n} \text{weights}[cycle_i][cycle_{i+1}]$$

Здесь weights — матрица весов графа, *cycle* — массив вершин, образующих гамильтонов цикл. То есть для решения задачи коммивояжера нужно найти минимальный гамильтонов цикл. Это можно сделать, используя перебор всех перестановок вершин графа.

2 Особенности реализации

2.1 Класс AbstractGraph

Класс AbstractGraph – абстракция произвольного графа. Является родителем для OrientedGraph (ориентированный граф).

Содержит следующие атрибуты:

- *int numVertices* - количество вершин графа;
- *int numEdges* - количество рёбер графа;
- *vector < vector < int >> adjacencyMatrix* - матрица смежности графа;
- *vector < vector < int >> weightmatrix* - матрица весов графа;
- *vector < vector < int >> cost* - матрица стоимости графа;
- *vector < vector < int >> propspos* - матрица пропускных способностей графа.

```
1  class AbstractGraph {
2          protected:
3          int numVertices;
4          int numEdges;
5          std::vector<std::vector<int>> adjacencyMatrix;
6          std::vector<std::vector<int>> weightmatrix;
7          vector<vector<int>> cost;
8          vector<vector<int>> euler_mx;
9          vector<vector<int>> gam_mx;
10         vector<vector<int>> gam_cycles;
11         vector<int> optimal_gam;
12         public:
13         vector<vector<int>> propspos;
14
15         AbstractGraph(int vertices) : numVertices(
   vertices), numEdges(0) {
16                 adjacencyMatrix.resize(vertices, std::
   vector<int>(vertices, 0));
17                 weightmatrix.resize(vertices, std::
   vector<int>(vertices, 0));
18                 propspos.resize(vertices, std::vector<
   int>(vertices, 0));
19                 cost.resize(vertices, std::vector<int>(
   vertices, 0));
```

```
20                euler_mx.resize(vertices, std::vector<
    int>(vertices, 0));
21                gam_mx.resize(vertices, std::vector<int
    >(vertices, 0));
22        }
23
24        virtual void generateRandomGraph() = 0;
25
26        void printAdjacencyMatrix() const {
27                std::cout << "матрица смежности:\n";
28                for (const auto& row : adjacencyMatrix)
     {
29                        for (int val : row) {
30                                std::cout << setw(4) <<
     val << " ";
31                        }
32                        std::cout << '\n';
33                }
34        }
35
36        void printweightmatrix() const {
37                std::cout << "матрица весов:\n";
38                for (const auto& row : weightmatrix) {
39                        for (int val : row) {
40                                std::cout << setw(4) <<
     val << " ";
41                        }
42                        std::cout << '\n';
43                }
44        }
45
46        void printpropspos() const {
47                std::cout << "матрица пропускных
    способностей:\n";
48                for (const auto& row : propspos) {
49                        for (int val : row) {
50                                std::cout << setw(4) <<
     val << " ";
51                        }
52                        std::cout << '\n';
53                }
54        }
55
```

```
56          void printcost() const{
57                  std::cout << "матрица␣стоимости:\n";
58                  for (const auto& row : cost) {
59                          for (int val : row) {
60                                  std::cout << setw(4) <<
    val << "␣";
61                          }
62                          std::cout << '\n';
63                  }
64          }
65
66          void printMatrix(const std::vector<std::vector<
    int>>& matrix) {
67                  std::cout << "граф:\n";
68                  for (const auto& row : matrix) {
69                          for (int val : row) {
70                                  std::cout << setw(4) <<
    val << "␣";
71                          }
72                          std::cout << '\n';
73                  }
74          }
75
76          bool hasNegativeWeights() const {
77                  for (const auto& row : weightmatrix) {
78                          for (int weight : row) {
79                                  if (weight < 0) {
80                                          return true;
81                                  }
82                          }
83                  }
84                  return false;
85          }
86
87
88          int getNumVertices() const { return numVertices
    ; }
89          int getNumEdges() const { return numEdges; }
90  };
```

2.2 Класс OrientedGraph

Класс OrientedGraph представляет собой абстракцию некоторого ориентированного графа и является наследником класса AbstractGraph. В этом классе происходит заполнение матриц, реализованы все алгоритмы, которые были нужны для выполнения работы.

2.2.1 Функция erlangRandomDegree()

Функция erlangRandomDegree() моделирует распределение Эрланга для генерации случайной степени в графе. Она использует параметры λ и k для расчетов и вычисляет вероятность на основе суммы членов рядов. В итоге функция возвращает случайную степень, округленную до ближайшего целого числа, пропорционально количеству вершин в графе.

Вход: параметры распределения λ, k

Выход: случайное число, соответствующее распределению

```
int erlangRandomDegree() {
        const double lambda = 3;
        const int k = 1;
        double sum = 0.0;

        for (int i = 0; i < k; ++i) {
                sum += pow(lambda, i) / factorial(i);
        }

        double result = 1.0 - exp(-lambda) * sum;
        return static_cast<int>(round(result *
  numVertices));
}
```

2.2.2 Метод Шимбелла

Функция shimbell_mul реализует умножение матриц по методу Шимбелла для нахождения максимальных путей между вершинами графа. Она принимает две матрицы m1 и m2, представляющие веса рёбер графа, и возвращает новую матрицу, в которой каждый элемент на позиции (i, j) содержит максимальную сумму путей от вершины i до j, рассматривая все возможные прямые пути через промежуточные вершины. Если путь между двумя вершинами не существует (представлен нулем), то он игнорируется.

Вход: матрицы весов ребёр графа

Выход: матрица максимальных весов

Функция shimbell_mul_min аналогична первой, но находит минимальные пути. Она возвращает матрицу, где каждый элемент на позиции (i, j)

содержит минимальную сумму весов путей от вершины i до вершины j. Для отсутствующих путей используется значение 10000 (большое по сравнению с любой реальной суммой весов), которое будет заменено на 0 при выводе.

Вход: матрица весов рёбер графа

Выход: матрица минимальных весов

Функция shimbell инициализирует результаты для матриц на основе заданной матрицы весов, обновляя их с каждым шагом итерации. Она выполняет несколько операций умножения по Шимбеллу (как для максимальных, так и для минимальных весов) в цикле, который повторяется len - 1 раз. После выполнения вычислений она выводит результирующие матрицы максимальных и минимальных путей на экран.

Вход: количество ребёр

Выход: минимальный и максимальный путь

```
vector<vector<int>> shimbell_mul(const vector<vector<int>>& m1, const vector<vector<int>>& m2) {
        size_t v = m1.size();
        vector<vector<int>> res(v, vector<int>(v, 0));

        for (size_t i = 0; i < m1.size(); i++) {
                for (size_t j = 0; j < m1.size(); j++) {
                        res[i][j] = 0;
                        for (size_t k = 0; k < m1.size(); k++) {
                                if (m1[i][k] != 0 && m2[k][j] != 0) {
                                        res[i][j] = max(res[i][j], m1[i][k] + m2[k][j]);
                                }
                        }
                }
        }

        return res;
}

vector<vector<int>> shimbell_mul_min(const vector<vector<int>>& m1, const vector<vector<int>>& m2) {
        size_t v = m1.size();
        vector<vector<int>> res(v, vector<int>(v, 0));

        for (size_t i = 0; i < m1.size(); i++) {
```

```
                for (size_t j = 0; j < m1.size(); j++)
    {
                        res[i][j] = 10000;
                        for (size_t k = 0; k < m1.size
    (); k++) {
                                if (m1[i][k] != 0 && m2
    [k][j] != 0) {
                                        res[i][j] = min
    (res[i][j], m1[i][k] + m2[k][j]);
                                }
                        }
                }
        }

        return res;
}

void shimbell(int len) {
        vector<vector<int>> res(numVertices, vector<int
    >(numVertices,0));
        vector<vector<int>> res_min(numVertices, vector
    <int>(numVertices, 0));
        for (int i = 0; i < numVertices; i++) {
                for (size_t j = 0; j < numVertices; j
    ++)
                {
                        res[i][j] = weightmatrix[i][j];
                        res_min[i][j] = weightmatrix[i
    ][j];
                }
        }
        for (int i = 0; i < len-1; i++) {
                res = shimbell_mul(res, weightmatrix);
                res_min = shimbell_mul_min(res_min,
    weightmatrix);
        }
        cout << "Maximum␣matrix␣is:\n";
        for (size_t i = 0; i < res.size(); i++)
        {
                for (size_t j = 0; j < res.size(); j++)
                {
                        cout << setw(5) << res[i][j];
                }
```

```
58                  cout << endl;
59          }
60          cout << "Minimum␣matrix␣is:\n";
61          for (size_t i = 0; i < res.size(); i++)
62          {
63                  for (size_t j = 0; j < res.size(); j++)
64                  {
65                          if (res_min[i][j] != 10000) {
66                                  cout << setw(5) <<
    res_min[i][j];
67                          }
68                          else {
69                                  cout << setw(5) << "0";
70                          }
71                  }
72                  cout << endl;
73          }
74  }
```

2.2.3 Построение маршрута от одной заданной точки до другой

Функция countPaths отвечает за подсчет всех возможных путей в графе от текущей вершины до целевой. Она принимает матрицу смежности, текущую вершину, целевую вершину, вектор посещенных вершин и счетчик путей. Если текущая вершина равна целевой, счетчик путей увеличивается. Текущая вершина помечается как посещенная, после чего для каждого соседа проверяется наличие ребра и его посещаемость. Если сосед не посещен, происходит рекурсивный вызов функции для дальнейшего обхода. После завершения обхода текущая вершина помечается как непосещенная для возможности использования в других путях.

Вход: матрица смежности, целевая вершина, текущая вершина, матрица посещённых вершин, счетчик путей

Выход: количество маршрутов

Функция findPaths служит интерфейсом для запуска подсчета путей. Она принимает начальную и конечную вершины и сначала проверяет корректность введенных данных. Если начало или конец находятся вне допустимого диапазона, возвращается 0. Далее инициализуются вектор посещенных вершин и счетчик путей, после чего вызывается функция countPaths для подсчета доступных путей от начальной до конечной вершины. В результате возвращается общее количество найденных путей.

Вход: начальная вершина, конечная вершина

Выход: возможность маршрута от одной точки до другой и количество

таких маршрутов

```
void countPaths(const vector<vector<int>>&
  adjacencyMatrix, int current, int target,
vector<bool>& visited, int& pathCount) {
        if (current == target) {
                pathCount++;
                return;
        }

        visited[current] = true;

        for (int neighbor = 0; neighbor <
  adjacencyMatrix.size(); ++neighbor) {
                if (adjacencyMatrix[current][neighbor]
  && !visited[neighbor]) {
                        countPaths(adjacencyMatrix,
  neighbor, target, visited, pathCount);
                }
        }

        visited[current] = false;
}

int findPaths(int start, int end) {
        int vertexCount = adjacencyMatrix.size();
        if (start < 0 || start >= vertexCount || end <
  0 || end >= vertexCount) {
                return 0;
        }

        vector<bool> visited(vertexCount, false);
        int pathCount = 0;

        countPaths(adjacencyMatrix, start, end, visited
  , pathCount);

        return pathCount;
}
```

2.2.4 Поиск в глубину

Функция depthFirstSearch выполняет поиск в глубину для нахождения пути между двумя вершинами в графе, представленном матрицей смежности. Она принимает начальную и конечную вершины как параметры. Внутри функции инициализируется вектор посещенных вершин и вектор родителей, которые отслеживают путь. Также используется стек для хранения вершин для дальнейшего обхода и счетчик итераций для оценки количества выполненных операций.

Процесс начинается с добавления начальной вершины в стек и отметки ее как посещенной. Внутри цикла, который продолжается, пока стек не пуст, вершина извлекается из стека, и увеличивается счетчик итераций. Формируется строка текущего пути, начиная с текущей вершины и разбирая родительские вершины, чтобы вывести полный маршрут. Если текущая вершина совпадает с конечной, выводится количество итераций, и функция завершает свою работу.

Если конечная вершина не достигнута, цикл продолжает обход соседей текущей вершины. Для каждого соседа, к которому ведет ребро, проверяется, был ли он посещен, и если нет, то он отмечается как посещенный, его родитель устанавливается на текущую вершину, и сосед добавляется в стек для дальнейшего изучения. Если по окончании обхода путь не найден, выводится сообщение о неудаче и количество итераций.

Вход: начальная вершина, конечная вершина

Выход: количество итераций, обход графа

```
1  void depthFirstSearch(int start, int end) {
2          vector<bool> visited(numVertices, false);
3          vector<int> parent(numVertices, -1);
4          int iterationCount = 0;
5          stack<int> s;
6          s.push(start);
7          visited[start] = true;
8          vector<string> paths;
9          cout << "ищем путь из " << (start + 1) << " в "
     << (end + 1) << ":\n";
10
11         while (!s.empty()) {
12                 int current = s.top();
13                 s.pop();
14                 iterationCount++;
15                 string currentPath;
16                 int temp = current;
17                 currentPath += to_string(temp + 1);
18
```

```
                for (int p = current; parent[p] != -1; p = parent[p]) {
                        currentPath = to_string(parent[p] + 1) + " " + currentPath;
                }
                paths.push_back(currentPath);
                cout << currentPath << endl;

                if (current == end) {
                        cout << "итераций: " << iterationCount << endl;
                        return;
                }
                for (int neighbor = 0; neighbor < numVertices; ++neighbor) {
                        if (adjacencyMatrix[current][neighbor] == 1 && !visited[neighbor]) {
                                visited[neighbor] = true;
                                parent[neighbor] = current;
                                s.push(neighbor);
                        }
                }
        }
        cout << "не найдено пути из " << (start + 1) << " в " << (end + 1) << endl;
        cout << "итераций: " << iterationCount << endl;
}
```

2.2.5 Алгоритм Дейкстры

Функция dijkstra реализует алгоритм Дейкстры для нахождения кратчайших путей в графе от заданной начальной вершины до конечной. Она принимает начальную и конечную вершины в качестве параметров и начинает с инициализации векторов для хранения расстояний до вершин, посещенных статусов и информации о предыдущих вершинах. Расстояния до всех вершин устанавливаются в максимальное значение, кроме начальной, которая равна нулю, а также инициализируется счетчик итераций для учета пройденных шагов.

Алгоритм выполняется в цикле, который продолжается до тех пор, пока имеются непосещенные вершины. На каждой итерации выбирается вершина

с наименьшим расстоянием и отмечается как посещенная. Затем проверяются все ее соседи. Если соседнее ребро существует и его вес меньше уже известного расстояния, то расстояние обновляется, и записывается родительская вершина.

Если конечная вершина недоступна (расстояние остается максимальным), выводится соответствующее сообщение. В противном случае выводится кратчайшее расстояние. После этого функция восстанавливает путь от конечной вершины до начальной, собирая веса рёбер и выводя все пройденные шаги. В конце отображаются кратчайшие пути от начальной вершины ко всем остальным вершинам, а также количество итераций, затраченных на выполнение алгоритма.

Вход: начальная вершина, конечная вершина

Выход: вектор кратчайших расстояний, количество итераций, путь и вес пути для заданных вершин

```
void dijkstra(int start, int end) {
        vector<int> distances(numVertices, INT_MAX);
        vector<bool> visited(numVertices, false);
        vector<int> previous(numVertices, -1);
        distances[start] = 0;
        int iterationCount = 0;

        for (int i = 0; i < numVertices; ++i) {
                int minIndex = -1;
                int minDistance = INT_MAX;

                for (int j = 0; j < numVertices; ++j) {
                        if (!visited[j] && distances[j]
   < minDistance) {
                                minIndex = j;
                                minDistance = distances
  [j];
                        }
                        iterationCount++;
                }

                if (minIndex == -1) break;

                visited[minIndex] = true;
                for (int neighbor = 0; neighbor <
  numVertices; ++neighbor) {
                        if (adjacencyMatrix[minIndex][
  neighbor] == 1 && !visited[neighbor]) {
```

```
25                                  int edgeWeight =
    weightmatrix[minIndex][neighbor];
26                                  if (distances[minIndex]
     + edgeWeight < distances[neighbor]) {
27                                      distances[
    neighbor] = distances[minIndex] + edgeWeight;
28                                      previous[
    neighbor] = minIndex;
29                                  }
30                              }
31                          }
32              }
33
34              if (distances[end] == INT_MAX) {
35                      cout << "нет пути из " << (start + 1) <<
     " в " << (end + 1) << endl;
36                      cout << "итераций: " << iterationCount
    << endl;
37                      return;
38              }
39
40              cout << "кратчайшее расстояние: " << distances[end
    ] << endl;
41              vector<int> path;
42              vector<int> weights;
43
44              for (int v = end; v != -1; v = previous[v]) {
45                      path.push_back(v + 1);
46                      if (previous[v] != -1) {
47                              int edgeWeight = weightmatrix[
    previous[v]][v];
48                              weights.push_back(edgeWeight);
49                      }
50              }
51
52              reverse(path.begin(), path.end());
53              reverse(weights.begin(), weights.end());
54
55              cout << "путь: ";
56              for (size_t i = 0; i < path.size(); ++i) {
57                      if (i > 0) cout << " ";
58                      cout << path[i];
59                      if (i < weights.size()) {
```

```
60                          cout << "␣(" << weights[i] << "
  )";
61                  }
62          }
63          cout << endl;
64          cout << "Кратчайшие␣пути␣от␣" << (start + 1) << "
  ␣к␣остальным␣вершинам:" << endl;
65          for (int v = 0; v < numVertices; ++v) {
66                  if (distances[v] == INT_MAX) {
67                          cout << (start + 1) << "␣→␣" <<
   (v + 1) << ":␣нет␣пути" << endl;
68                  } else {
69                          cout << (start + 1) << "␣→␣" <<
   (v + 1) << "␣вес(:␣" << distances[v] << "):␣";
70                          vector<int> path;
71                          for (int cur = v; cur != -1;
  cur = previous[cur]) {
72                                  path.push_back(cur + 1)
  ;
73                          }
74                          reverse(path.begin(), path.end
  ());
75                          for (size_t i = 0; i < path.
  size(); ++i) {
76                                  cout << path[i];
77                                  if (i < path.size() -
  1) {
78                                          cout << "␣→␣";
79                                  }
80                          }
81                          cout << endl;
82                  }
83          }
84
85          cout << "итераций:␣" << iterationCount << endl;
86  }
```

2.2.6 Алгоритм Форда-Фалкерсона

Алгоритм Форда-Фалкерсона решает задачу нахождения максимального потока в сети. В алгоритме используется поиск пути в глубину. В каждой итерации в остаточной сети ищется путь из истока в сток.

Вход: матрица пропускной способности, номер вершины истока, номер вершины стока

Выход: величина максимального потока

```
int fordFulkerson(vector<vector<int>>graph, int s, int
  t) {
        int u, h;
        vector<vector<int>> rGraph(numVertices, vector<
  int>(numVertices)); // остаточная сеть
        for (u = 0; u < numVertices; u++) {
                for (h = 0; h < numVertices; h++)
                rGraph[u][h] = propspos[u][h];
        }
        vector<int> parent(numVertices);
        int max_flow = 0;
        while (bfs(rGraph, s, t, parent)) {
                int path_flow = 100000000;
                for (h = t; h != s; h = parent[h]) {
                        u = parent[h];
                        path_flow = std::min(path_flow,
   rGraph[u][h]);
                }
                for (h = t; h != s; h = parent[h]) {
                        u = parent[h];
                        rGraph[u][h] -= path_flow;
                        rGraph[h][u] += path_flow;
                }
                max_flow += path_flow;
        }
        cout << max_flow << endl;
        return max_flow;
}
```

2.2.7 Поток минимальной стоимости

Функция отвечает за нахождение минимальной стоимости для заданного потока в графе, используя алгоритм для поиска кратчайших путей. Если входной поток равен нулю, выводится сообщение о том, что поток слишком мал, и функция завершает выполнение с возвращаемым значением 0.

Изначально создаются две матрицы: newcost для обновленных стоимостей и newflow для текущих остатков потока. Эти матрицы заполняются значениями из исходных матриц cost и propspos для всех пар связанных вершин.

Далее выводятся целевой поток и матрица стоимости. Затем начинается

основная часть алгоритма, где для каждого единичного потока вызывается вспомогательная функция, которая вычисляет кратчайший путь от начальной до конечной вершины. После получения пути стоимость обновляется, и остаток потока уменьшается. Если остаток потока на ребре становится равным нулю, цена этого ребра устанавливается в максимально возможное значение, чтобы его игнорировать при последующих итерациях.

Все найденные пути и их стоимости собираются в формате отображения, где ключами являются пути, а значениями — количество единиц потока и стоимость для каждого из них. В конце функции выводится общая минимальная стоимость потока, и эта стоимость возвращается в качестве результата.

Вход: величина целевого потока

Выход: величина минимальной стоимости для потока, пути для потока минимальной стоимости

```
int find_min_cost(int flow) {

        if (flow == 0) {
                cout << "поток очень мал, найти поток минимальной стоимости\n";
                return 0;
        }

        vector<vector<long long int>> newcost(numVertices, vector<long long int>(numVertices));
        vector<vector<long long int>> newflow(numVertices, vector<long long int>(numVertices));

        for (int i = 0; i < numVertices; i++) {
                for (int j = 0; j < numVertices; j++) {
                        if (adjacencyMatrix[i][j]) {
                                newcost[i][j] = cost[i][j];
                                newflow[i][j] = propspos[i][j];
                        }
                }
        }

        cout << "целевой поток: " << flow << endl;
        cout << "матрица стоимости: \n";
        for (int i = 0; i < numVertices; i++) {
                for (size_t j = 0; j < numVertices; j++) {
```

```
                cout << setw(5) << cost[i][j];
            }
            cout << endl;
        }

        cout << "пути для потока минимальной стоимости:\n";
        int s = 0;
        map<vector<int>, pair<int, long long int>> mp;

        for (size_t i = 0; i < flow; i++) {
            pair<vector<int>, int> info_floyd = floyd_util(0, numVertices - 1, newcost);
            vector<int> path = info_floyd.first;
            long long int path_cost = 0;

            for (size_t j = 0; j < path.size() - 1; j++) {
                newflow[path[j]][path[j + 1]] -= 1;
                path_cost += cost[path[j]][path[j + 1]];
                if (newflow[path[j]][path[j + 1]] == 0) {
                    newcost[path[j]][path[j + 1]] = 2147483647;
                }
            }

            mp[path].first += 1;
            mp[path].second += path_cost;
            s += path_cost;
        }

        for (const auto &it : mp) {
            for (size_t i = 0; i < it.first.size(); i++) {
                cout << setw(5) << it.first[i];
            }
            cout << " - " << it.second.first << " единиц потока, стоимость: " << it.second.second << endl;
        }

        cout << "минимальная стоимость для потока: " << s
```

```
    << endl;
59          return s;
60  }
```

2.2.8 Алгоритм Прима

Алгоритм Прима ищет минимальный по весу остов в графе. Берётся произвольная вершина, и находится ребро с наименьшей стоимостью, инцидентное этой вершине. Найденное ребро и две соединяемые им вершины образуют дерево. Рассматриваются рёбра графа, один конец которых — уже принадлежащая дереву вершина, а другой — нет. Из этих рёбер выбирается ребро наименьшей стоимости. Присоединяется выбранное ребро к дереву. Рост дерева продолжается до тех пор, пока не будут исчерпаны все вершины исходного графа.

Вход: матрица весов графа

Выход: матрица весов для минимального остова, сумма весов ребер

```
1   vector<vector<int>> prim(bool b) {
2           int it = 0;
3           vector<vector<int>> g(numVertices, vector<int>(
    numVertices, 2147483647));
4           vector<vector<int>> res(numVertices, vector<int
    >(numVertices));
5
6           for (int i = 0; i < numVertices; i++) {
7                   for (size_t j = 0; j < numVertices; j
    ++)
8                   {
9                           if (adjacencyMatrix[i][j]) {
10                                  g[i][j] = weightmatrix[
    i][j];
11                                  g[j][i] = weightmatrix[
    i][j];
12                          }
13                  }
14          }
15          vector<bool> used(numVertices);
16          vector<int> min_e(numVertices, 2147483647),
    sel_e(numVertices, -1);
17          min_e[0] = 0;
18          for (int i = 0; i < numVertices; ++i) {
19                  int u = -1;
20                  for (int j = 0; j < numVertices; ++j) {
```

```
21                          it++;
22                          if (!used[j] && (u == -1 ||
    min_e[j] < min_e[u]))
23                          u = j;
24                  }
25                  if (min_e[u] == 2147483647) {
26                          cout << "не MST!\n";
27                          exit(0);
28                  }
29
30                  used[u] = true;
31                  if (sel_e[u] != -1) {
32                          int t1, t2;
33                          t1 = max(u, sel_e[u]);
34                          t2 = min(u, sel_e[u]);
35                          res[t2][t1] = weightmatrix[t2][
    t1];
36                          res[t1][t2] = weightmatrix[t2][
    t1];
37                  }
38
39
40
41                  for (int to = 0; to < numVertices; ++to
    ) {
42                          it++;
43                          if (g[u][to] < min_e[to]) {
44                                  min_e[to] = g[u][to];
45                                  sel_e[to] = u;
46                          }
47                  }
48          }
49          int sum = 0;
50          for (int i = 0; i < numVertices; i++) {
51                  for (size_t j = i+1; j < numVertices; j
    ++)
52                  {
53                          sum += res[i][j];
54                  }
55          }
56          if (b) {
57                  cout << "суммарный вес: " << sum << endl;
58          }
```

```
59
60
61          return res;
62  }
```

2.2.9 Матричная теорема Кирхгофа

Функция kirghoff вычисляет матрицу Кирхгофа для заданного графа и на основании этой матрицы определяет количество остовных деревьев. В начале она создает матрицы: a для хранения значений по формуле Кирхгофа, kirghoff для хранения элементов матрицы Кирхгофа, и S для отслеживания связей между вершинами.

В первой части функции происходит заполнение матрицы S, где каждое ребро между вершинами помечается значением -1. Вторая часть функции считает количество соседей для каждой вершины и пополняет матрицу Кирхгофа: диагональные элементы этой матрицы получают значение количества соседей, а для двух связанных вершин устанавливаются значения -1.

Затем на основе заполненной матрицы Кирхгофа создается матрица a, которая служит входом для дальнейших вычислений. После этого выводится сама матрица Кирхгофа, и вызывается функция, которая рассчитывает определитель матрицы a, что соответствует числу остовных деревьев в графе.

Вход: матрица смежности

Выход: количество остовных деревьев

```
1  int kirghoff() {
2          vector<vector<double>> a(numVertices - 1,
   vector<double>(numVertices - 1));
3          vector<vector<int>> kirghoff(numVertices,
   vector<int>(numVertices));
4          vector<vector<int>> S(numVertices, vector<int>(
   numVertices));
5          for (size_t i = 0; i < numVertices; i++)
6          {
7                  for (size_t j = 0; j < numVertices; j
   ++)
8                  {
9                          if (adjacencyMatrix[i][j]) {
10                                 S[i][j] = -1;
11                                 S[j][i] = -1;
12                         }
13                 }
14
15         }
```

```
16
17          for (size_t i = 0; i < numVertices; i++)
18          {
19                  int sum = 0;
20                  for (size_t j = 0; j < numVertices; j
    ++)
21                  {
22                          if (S[i][j]==-1) {
23                                  sum++;
24                                  kirghoff[i][j] = S[i][j
    ];
25                          }
26                  }
27                  kirghoff[i][i] = sum;
28          }
29          for (size_t i = 1; i < numVertices; i++)
30          {
31                  for (size_t j = 1; j < numVertices; j
    ++)
32                  {
33                          a[i-1][j-1] = kirghoff[i][j];
34                  }
35          }
36          cout << "матрица Кирхгофа:\n";
37          for (size_t i = 0; i < numVertices; i++)
38          {
39                  for (int j = 0; j < numVertices; j++) {
40                          cout << setw(5) << kirghoff[i][
    j];
41                  }
42                  cout << endl;
43          }
44          cout << "число остовных деревьев: ";
45          return det(a);
46  }
```

2.2.10 Код Прюфера

Кодирование Прюфера реализовано для дерева, которое получается в результате работы алгоритма Прима. В алгоритме кодируется и декодируется дерево, причем сохраняются веса ребер.

Вход: матрица весов дерева из алгоритма Прима

Выход: закодированная последовательность и декодированная матрица весов

```
void prufer() {
        vector<pair<int, int>> prufer;
        vector<vector<int>> tree = prim(0);
        vector<int> degrees(numVertices);
        for (size_t i = 0; i < numVertices; i++)
        {
                degrees[i] = deg(tree[i]);
        }
        int num = 0;
        for (int i = 0; i < numVertices - 1; i++) {
                for (size_t j = 0; j < numVertices; j
++)
                {
                        if (degrees[j] == 1) {
                                num = j;
                                break;
                        }
                }
                for (size_t j = 0; j < numVertices; j
++)
                {
                        if (tree[num][j] != 0 &&
degrees[j] != 0) {
                                prufer.push_back({ j,
tree[num][j] });
                                degrees[j]--;
                                degrees[num]--;
                                break;
                        }

                }
        }
        cout << "код Прюфера: " << endl;
        for (size_t i = 0; i < prufer.size(); i++)
        {
                cout << prufer[i].first << " вес: " <<
prufer[i].second<<endl;
        }
        vector<pair<int, int>> prufer_ch = prufer;
        multiset<int> pruf_v;
```

```
36          for (int i = 0; i < prufer.size(); i++) {
37                  pruf_v.insert(prufer[i].first);
38          }
39          vector<vector<int>> check_tree(numVertices,
    vector<int>(numVertices));
40          vector<bool> vosst(numVertices,0);
41          int min_v = 0;
42          while (prufer_ch.size() != 0) {
43                  for (size_t i = 0; i < numVertices; i
    ++)
44                  {
45                          if (vosst[i] == 0 && find(
    pruf_v.begin(), pruf_v.end(), i) == pruf_v.end()) {
46                                  check_tree[i][prufer_ch
    [0].first] = prufer_ch[0].second;
47                                  check_tree[prufer_ch
    [0].first][i] = prufer_ch[0].second;
48                                  pruf_v.erase(find(
    pruf_v.begin(), pruf_v.end(), prufer_ch[0].first));
49                                  prufer_ch.erase(
    prufer_ch.begin());
50                                  vosst[i] = 1;
51                                  break;
52                          }
53                  }
54          }cout << "декодирование: \n";
55          for (int i = 0; i < numVertices; i++) {
56                  for (int j = 0; j < numVertices; j++) {
57                          cout << setw(5) << check_tree[i
    ][j];
58                  }cout << endl;
59          }
60
61  }
```

2.2.11 Эйлеров граф

Для поиска эйлерова цикла в графе используется функция euler_check(). Если количество вершин не более двух, то в графе невозможно найти эйлеров цикл. Иначе граф достраивается до эйлерового при помощи функции fix_euler(), если эйлеровым не является. После этого вызывается функция findEulerCycle(), выполняющая поиск эйлерового цикла.

Вход: матрица весов
Выход: эйлеров цикл по графу

```
void fix_euler(vector<vector<int>> &unoriented, vector<int> degs) {
        while (!checkdiv2(degs)) {
                int c = 0;
                while (count(degs) >= 2) {
                        int v1, v2;
                        for (int i = 0; i < numVertices; i++) {
                                if (degs[i] == 1) {
                                        v1 = i;
                                        break;
                                }
                        }
                        for (size_t i = v1+1; i < numVertices; i++)
                        {
                                if (degs[i] == 1) {
                                        v2 = i;
                                        break;
                                }
                        }
                        connect(v1, v2, unoriented, degs);
                }
                if (!checkdiv2(degs)) {
                        int v1, v2;
                        for (int i = 0; i < numVertices; i++) {
                                if (degs[i]%2== 1) {
                                        v1 = i;
                                        break;
                                }
                        }
                        for (size_t i = v1 + 1; i < numVertices; i++)
                        {
                                if (degs[i] %2== 1) {
                                        v2 = i;
                                        break;
                                }
```

```
35                          }
36                          if (unoriented[v1][v2] == 0) {
37                                  connect(v1, v2,
    unoriented, degs);
38                          }
39                          else {
40                                  if (degs[v1] > 1 &&
    degs[v2] > 1) {
41                                          disconnect(v1,
    v2, unoriented, degs);
42                                  }
43                                  else {
44                                          int little, big
    ;
45                                          if (degs[v1] <
    degs[v2]) {
46                                                  little
    = v1;
47                                                  big =
    v2;
48                                          }
49                                          else {
50                                                  big =
    v1;
51                                                  little
    = v2;
52                                          }
53                                          int neighbour;
54                                          for (int i = 0;
     i < numVertices; i++) {
55                                                  if (
    unoriented[big][i] != 0) {
56
            neighbour = i;
57
            break;
58                                                  }
59                                          }
60                                          disconnect(big,
     neighbour, unoriented, degs);
61                                          connect(little,
     neighbour, unoriented, degs);
62                                  }
```

```
63                              }
64                      }
65
66              }
67
68      }
69
70      void findEulerCycle(int n, int start)
71      {
72              cout << "\Эйлеровn цикл: \n";
73              vector<vector<int>> temp = euler_mx;
74              stack<int> s;
75              vector<int> res;
76              s.push(start);
77              while (!s.empty())
78              {
79                      int w = s.top();
80                      bool found_edge = false;
81                      for (int i = 0; i < n; i++) {
82                              if (temp[w][i] != 0) {
83                                      s.push(i);
84                                      temp[w][i] = 0;
85                                      temp[i][w] = 0;
86                                      found_edge = true;
87                                      break;
88                              }
89                      }
90
91                      if (!found_edge) {
92                              s.pop();
93                              cout << w << " ";
94                      }
95              }
96      }
97
98      void euler_check() {
99              vector<vector<int>> unoriented(numVertices,
        vector<int>(numVertices));
100             if (numVertices <= 2) {
101                     cout << "граф с двумя или менее вершинами
        не может быть эйлеровым" << endl;
102                     return;
103             }
```

```
104         for (size_t i = 0; i < numVertices; i++)
105         {
106                 for (int j = 0; j < numVertices; j++) {
107                         if (adjacencyMatrix[i][j]) {
108                                 unoriented[i][j] =
    weightmatrix[i][j];
109                                 unoriented[j][i] =
    weightmatrix[i][j];
110                         }
111                 }
112         }
113         vector<int> degs(numVertices);
114         bool b = true;
115         for (int i = 0; i < numVertices; i++) {
116                 degs[i] = deg(unoriented[i]);
117                 if (degs[i] % 2 == 1) {
118                         b = false;
119                 }
120         }
121         if (!b) {
122                 cout << "Граф␣не␣Эйлеров!\n";
123                 cout << "Граф␣был␣изменён:\n";
124                 fix_euler(unoriented, degs);
125         }
126         euler_mx = unoriented;
127         cout << "Граф␣Эйлеров!\n";
128         cout << "Эйлеров␣граф:\n";
129         for (int i = 0; i < numVertices; i++) {
130                 for (size_t j = 0; j < numVertices; j
    ++)
131                 {
132                         cout << setw(5) << euler_mx[i][
    j];
133                 }cout << endl;
134         }
135         findEulerCycle(numVertices, 0);
136
137 }
```

Функции connect и disconnect выполняют соединение и разъединение двух вершин графа.

Вход: номера вершин, матрица весов графа, вектор степеней вершин графа

Выход: обновленная матрица весов графа и вектор степеней вершин графа

```
void connect(int v1, int v2, vector<vector<int>>&
   unoriented, vector<int> &degs) {
        unoriented[v1][v2] = rand()%3+1;
        unoriented[v2][v1] = unoriented[v1][v2];
        degs[v1]++;
        degs[v2]++;
        cout << "добавлено ребро между " << v1 << " и " <<
    v2 << endl;

}
void disconnect(int v1, int v2, vector<vector<int>>&
   unoriented, vector<int>& degs) {
        unoriented[v1][v2] = 0;
        unoriented[v2][v1] = 0;
        degs[v1]--;
        degs[v2]--;
        cout << "удалено ребро между " << v1 << " и " <<
   v2 << endl;

}
```

Фунцкция count возвращает количество вершин степени 1 в графе.

Вход: вектор степеней вершин графа

Выход: количество вершин степени 1

```
int count(vector<int> degs) {
        int i = 0;
        for (int j = 0; j < numVertices; j++) {
                if (degs[j] == 1) {
                        i += 1;
                }
        }
        return i;
}
```

2.2.12 Гамильтонов граф

Поиск гамильтонова цикла осуществляется с помощью функции check_gam. В ней цикл достраивается до гамильтонова и проверяются все перестановки вершин, начинающиеся с нулевой вершины.

Вход: матрица весов графа

Выход: векторы вершин всех гамильтоновых циклов, их стоимости и минимальный гамильтонов цикл

```
void check_gam() {
        vector<vector<int>> combine = comb();
        find_gam();
        int cst = 10000;

        for (int i = 0; i < combine.size(); i++) {
                if (is_ok(combine[i])) {
                        gam_cycles.push_back(combine[i
]);
                        if (count_c(combine[i]) < cst)
{
                                optimal_gam = combine[i
];
                                cst = count_c(combine[i
]);

                        }
                }

        }
        if (gam_cycles.size() <=0) {
                cout << "Граф не Гамильтонов!\n";
                cout << "граф был изменён\n";
                fix_gam();
                for (int i = 0; i < combine.size(); i
++) {
                        if (is_ok(combine[i])) {
                                gam_cycles.push_back(
combine[i]);
                                if (count_c(combine[i])
 < cst) {
                                        optimal_gam =
combine[i];
                                        cst = count_c(
combine[i]);
                                }
                        }

                }

```

```
        }
        cout << "Граф Гамильтонов!\n";
        cout << "Гамильтонов граф:\n";

        for (size_t i = 0; i < numVertices; i++)
        {
                for (size_t j = 0; j < numVertices; j
++)
                {
                        cout << setw(5) << gam_mx[i][j
];
                }cout << endl;

        }
        cout << "Гамильтонов цикл:\n";
        for (int i = 0; i < gam_cycles.size();i++) {

                for (int j = 0; j < gam_cycles[i].size
(); j++) {
                        cout << setw(5) << gam_cycles[i
][j];
                }cout <<setw(5)<<"0"<< " цена: "<<
count_c(gam_cycles[i]) << endl;

        }
        cout << endl;
        cout << "минимальный Гамильтонов цикл: " << endl;
        for (int i = 0; i < optimal_gam.size(); i++) {
                cout << setw(5) << optimal_gam[i];
        }cout << setw(5) << "0"<< "\ценаn: " << count_c
(optimal_gam);
}
```

В функции fix_gam граф достраивается до гамильтонова по теореме Дирака.

Вход: матрица весов неориентированного графа

Выход: матрица весов гамильтонова графа

```
void fix_gam() {
        for (int i = 0; i < gam_mx.size(); i++) {
                int j = 0;
                while (deg(gam_mx[i]) < (numVertices /
2) + 1) {
                        if (gam_mx[i][j] == 0 && i!=j)
```

```
    {
                                        gam_mx[i][j] = rand() %
     3 + 1;
                                        gam_mx[j][i] = rand() %
     3 + 1;
                                        cout << "добавлено ребро
    между " << i << " и " << j << endl;

                                }
                                j++;
                        }
                }
}
```

В функции comb генерируются все перестановки вершин от 1 до n. Это происходит при помощи функций save и NextSet. Функция NextSet генерирует следующую перестановку из предыдущей. При помощи функции save перестановка сохраняется в вектор в нужном виде, то есть вершины в ней нумеруются с 0.

Вход: вектор предыдущей перестановки, количество вершин

Выход: вектор новой перестановки

```
bool NextSet(vector<int> &a, int n)
{
        int j = n - 2;
        while (j != -1 && a[j] >= a[j + 1]) j--;
        if (j == -1)
        return false;
        int k = n - 1;
        while (a[j] >= a[k]) k--;
        swap_ij(a, j, k);
        int l = j + 1, r = n - 1;
        while (l < r)
        swap_ij(a, l++, r--);
        return true;
}
```

Вход: вектор перестановки, количество вершин

Выход: вектор перестановки с нумерацией вершин от 0 до n-1

```
vector<int> save(vector<int> a, int n)
{
        vector<int> av;
        for (int i = 0; i < n; i++)
        av.push_back(a[i]-1);
```

```
        return av;
}
```

2.3 Функция main

Функция main выполняет функцию меню для всей программы. В ней происходят проверки на неверный ввод, создание графов и использование реализованных алгоритмов.

Вход: инициализация параметров программы

Выход: результаты вызванных алгоритмов

```
int main() {
        int v;
        do {
                cout << "введите количество вершин: \n";
                cin >> v;

                if (cin.fail()) {
                        cin.clear();
                        cin.ignore(std::numeric_limits<
  std::streamsize>::max(), '\n');
                }
        } while (v < 2);
        OrientedGraph graph(v);
        graph.generateRandomGraph();
        graph.printAdjacencyMatrix();
        cout << "весовая матрица с положительными весами - 
  0, с отрицательными - 1\n";
        int a;
        cin >> a;
        graph.generateweightmatrix(a);
        graph.printweightmatrix();
        cout << OUTPUTPHRASE;
        int symbol;
        while(true){
                cin >> symbol;
                switch (symbol) {
                        case 1:{
                                cout << "введите 
  количество ребер:\n";
                                int rebra;
                                cin >> rebra;
```

```
29                                      if(rebra > graph.
    getNumVertices() || rebra == 0 || rebra < 0){
30                                              cout << "
    введите количество ребер от 1 до " << graph.
    getNumVertices() << '\n';
31                                      }
32                                      else{
33                                              graph.shimbell(
    rebra);
34                                      }
35                                      cout << OUTPUTPHRASE;
36                                      break;
37                              }
38                              case 2:{
39                                      cout << "введите первую
    вершину:\n";
40                                      int one;
41                                      cin >> one;
42                                      cout << "введите вторую
    вершину:\n";
43                                      int two;
44                                      cin >> two;
45                                      int kolvo = graph.
    findPaths(one - 1, two - 1);
46                                      if(one > graph.
    getNumVertices() || two > graph.getNumVertices() ||
    one < 1 || two < 1){
47                                              cout << "
    введите номер вершины от 1 до " << graph.getNumVertices
    () << '\n';
48                                              cout <<
    OUTPUTPHRASE;
49                                              break;
50                                      }
51                                      if(kolvo == 0){
52                                              cout << "
    построение маршрута между вершинами невозможно. \n";
53                                              cout <<
    OUTPUTPHRASE;
54                                              break;
55                                      }
56                                      if(one == two){
57                                              cout << "мы
```

```
находимся в этой вершине\n";
58                                                  cout <<
OUTPUTPHRASE;
59                                                  break;
60                                          }
61                                          else{
62                                                  cout << "
количество маршрутов: "<< kolvo;
63                                          }
64                                          cout << OUTPUTPHRASE;
65                                          break;
66                                  }
67                                  case 0:{
68                                          int versh;
69                                          do {
70                                                  cout << "
введите количество вершин: \n";
71                                                  cin >> versh;
72
73                                                  if (cin.fail())
 {
74                                                          cin.
clear();
75                                                          cin.
ignore(std::numeric_limits<std::streamsize>::max(),
'\n');
76                                                  }
77                                          } while (versh < 2);
78                                          graph = OrientedGraph(
versh);
79                                          graph.
generateRandomGraph();
80                                          graph.
printAdjacencyMatrix();
81                                          cout << "весовая матрица
с положительными весами - 0, с отрицательными - 1\n";
82                                          int a;
83                                          cin >> a;
84                                          graph.
generateweightmatrix(a);
85                                          graph.printweightmatrix
();
86                                          cout << OUTPUTPHRASE;
```

```
87                                  break;
88                              }
89                              case 3:{
90                                  cout << "введите␣
    начальную␣вершину:␣\n";
91                                  int first;
92                                  cin >> first;
93                                  cout << "введите␣
    конечную␣вершину:␣\n";
94                                  int second;
95                                  cin >> second;
96                                  if(first > graph.
    getNumVertices() || second > graph.getNumVertices()
    || first < 1 || second < 1){
97                                      cout << "
    введите␣номер␣вершины␣от␣1␣до␣" << graph.getNumVertices
    () << '\n';
98                                      cout <<
    OUTPUTPHRASE;
99                                      break;
100                                 }
101                                 if(first == second){
102                                     cout << "мы␣
    находимся␣в␣этой␣вершине!\n";
103                                     cout <<
    OUTPUTPHRASE;
104                                     break;
105                                 }
106                                 else{
107                                     graph.
    depthFirstSearch(first - 1, second - 1);
108                                 }
109                                 cout << OUTPUTPHRASE;
110                                 break;
111                             }
112                             case 4:{
113                                 if (graph.
    hasNegativeWeights()) {
114                                     cout << "нельзя␣
    использовать␣алгоритм␣Дейкстры:␣отрицательные␣веса.\n";
115                                     cout <<
    OUTPUTPHRASE;
116                                     break;
```

```
117                                     }
118                                     cout << "введите␣
    начальную␣вершину:␣\n";
119                                     int first;
120                                     cin >> first;
121                                     cout << "введите␣
    конечную␣вершину:␣\n";
122                                     int second;
123                                     cin >> second;
124                                     if(first > graph.
    getNumVertices() || second > graph.getNumVertices()
    || first < 1 || second < 1){
125                                             cout << "
    введите␣номер␣вершины␣от␣1␣до␣" << graph.getNumVertices
    () << '\n';
126                                             cout <<
    OUTPUTPHRASE;
127                                             break;
128                                     }
129                                     if(first == second){
130                                             cout << "
    кратчайшее␣расстояние:␣0\n";
131                                             cout <<
    OUTPUTPHRASE;
132                                             break;
133                                     }
134                                     else{
135
136                                             graph.dijkstra(
    first - 1, second - 1);
137                                     }
138                                     cout << OUTPUTPHRASE;
139                                     break;
140                             }
141                             case 5:{
142                                     cout << "введите␣
    начальную␣вершину:␣\n";
143                                     int first;
144                                     cin >> first;
145                                     cout << "введите␣
    конечную␣вершину:␣\n";
146                                     int second;
147                                     cin >> second;
```

```
148                                     if(first > graph.
    getNumVertices() || second > graph.getNumVertices()
    || first < 1 || second < 1){
149                                         cout << "
    введите номер вершины от 1 до " << graph.getNumVertices
    () << '\n';
150                                         cout <<
    OUTPUTPHRASE;
151                                         break;
152                                     }
153                                     else{
154                                         cout << "
    алгоритм Дейкстры: ";
155                                         graph.dijkstra5
    (first - 1, second - 1);
156                                         cout << "поиск
    в глубину: ";
157                                         graph.
    depthFirstSearch5(first - 1, second - 1);
158                                     }
159                                     cout << OUTPUTPHRASE;
160                                     break;
161                             }
162                             case 6:{
163                                     graph.
    generatepropsposcost();
164                                     graph.printpropspos();
165                                     cout << "максимальный
    поток: \n";
166                                     int r = graph.
    fordFulkerson(graph.propspos, 0, graph.
    getNumVertices() - 1);
167                                     graph.find_min_cost(2 *
     r / 3);
168                                     cout << OUTPUTPHRASE;
169                                     break;
170                             }
171                             case 7:{
172                                     vector<vector<int>> pr
    = graph.prim(0);
173                                     cout << "остовное дерево:
    \n";
174                                     for (size_t i = 0; i <
```

```
graph.getNumVertices(); i++)
175                                     {
176                                             for (int j = 0;
 j < graph.getNumVertices(); j++) {
177                                                     cout <<
 setw(5) << pr[i][j];
178                                             }
179                                             cout << endl;
180                                     }
181                                     graph.prim(1);
182                                     int m;
183                                     m = graph.kirghoff();
184                                     cout << m;
185                                     cout << endl;
186                                     graph.prufer();
187                                     cout << OUTPUTPHRASE;
188                                     break;
189                             }
190                             case 8:{
191                                     graph.graphcolor();
192                                     cout << OUTPUTPHRASE;
193                                     break;
194                             }
195                             case 9:{
196                                     if (graph.
getNumVertices() <= 2) {
197                                             cout << "цикл␣
Эйлера␣невозможен␣менее,␣чем␣на␣3␣вершинах\n";
198                                             cout <<
OUTPUTPHRASE;
199                                             break;
200                                     }
201                                     graph.euler_check();
202                                     cout << OUTPUTPHRASE;
203                                     break;
204                             }
205                             case 10:{
206                                     if (graph.
getNumVertices() <= 2) {
207                                             cout << "цикл␣
Гамильтона␣невозможен␣менее,␣чем␣на␣3␣вершинах\n";
208                                             cout <<
OUTPUTPHRASE;
```

```
209                                                 break;
210                                             }
211                                             graph.check_gam();
212                                             cout << OUTPUTPHRASE;
213                                             break;
214                                 }
215                                 default:
216                                 cout << "введён символ не из
    списка. попробуйте снова. \n";
217                                 cout << OUTPUTPHRASE;
218                                 break;
219                         }
220                 }
221                 return 0;
222 }
```

3 Результаты работы программы

После запуска программы пользователя просят ввести количество вершин и выбрать знак для матрицы весов. После чего выводится меню – рис. 3.

```
введите количество вершин:
7
матрица смежности:
   0    0    1    1    0    1    1
   0    0    1    0    0    1    1
   0    0    0    1    1    1    1
   0    0    0    0    0    1    1
   0    0    0    0    0    1    0
   0    0    0    0    0    0    1
   0    0    0    0    0    0    0
весовая матрица с положительными весами – 0, с отрицательными – 1
0
матрица весов:
   0    0    9    5    0    2    9
   0    0    4    0    0    3    1
   0    0    0    4    5    2    1
   0    0    0    0    0    2    2
   0    0    0    0    0    2    0
   0    0    0    0    0    0    2
   0    0    0    0    0    0    0

выберите следующее действие:
0 – генерировать новый граф
1 – метод Шимбелла
2 – количество маршрутов между двумя вершинами
3 – поиск в глубину
4 – алгоритм Дейкстры
5 – сравнение скоростей по количеству итераций
6 – алгоритм Форда–Фалкерсона, поток минимальной стоимости
7 – число остовных деревьев, минимальный по весу остов, код Прюфера
8 – минимальная раскраска графа или остова
9 – Эйлеров цикл
10 – Гамильтонов цикл
```

Рис. 3: Запуск программы и начальное меню

При выборе «0» всё повторяется – рис 4.

```
0
введите количество вершин:
6
матрица смежности:
   0    1    1    1    1    0
   0    0    0    1    1    1
   0    0    0    1    1    1
   0    0    0    0    0    1
   0    0    0    0    0    1
   0    0    0    0    0    0
весовая матрица с положительными весами – 0, с отрицательными – 1
1
матрица весов:
   0   -9    4   -8    6    0
   0    0    0    6    4   -9
   0    0    0    4   -4    7
   0    0    0    0    0   -2
   0    0    0    0    0   -4
   0    0    0    0    0    0

выберите следующее действие:
0 – генерировать новый граф
1 – метод Шимбелла
2 – количество маршрутов между двумя вершинами
3 – поиск в глубину
4 – алгоритм Дейкстры
5 – сравнение скоростей по количеству итераций
6 – алгоритм Форда–Фалкерсона, поток минимальной стоимости
7 – число остовных деревьев, минимальный по весу остов, код Прюфера
8 – минимальная раскраска графа или остова
9 – Эйлеров цикл
10 – Гамильтонов цикл
```

Рис. 4: Генерация нового графа

При выборе «1» пользователь вводит количество ребер и выводится минимальный и максимальный пути в виде матриц – рис. 5.

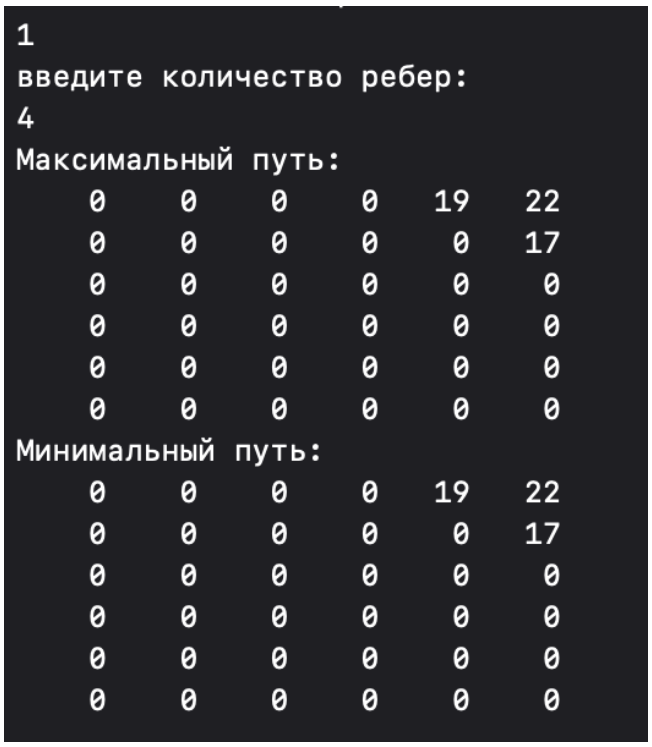

Рис. 5: метод Шимбелла

При выборе «2» пользователь вводит номера вершин и выводится количество маршрутов между ними – рис. 6.

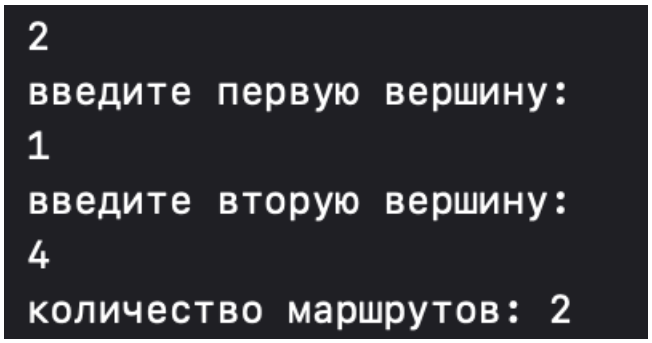

Рис. 6: Количество маршрутов между двумя вершинами

При выборе «3» пользователь вводит номера вершин и выводится обход вершин – рис. 7.

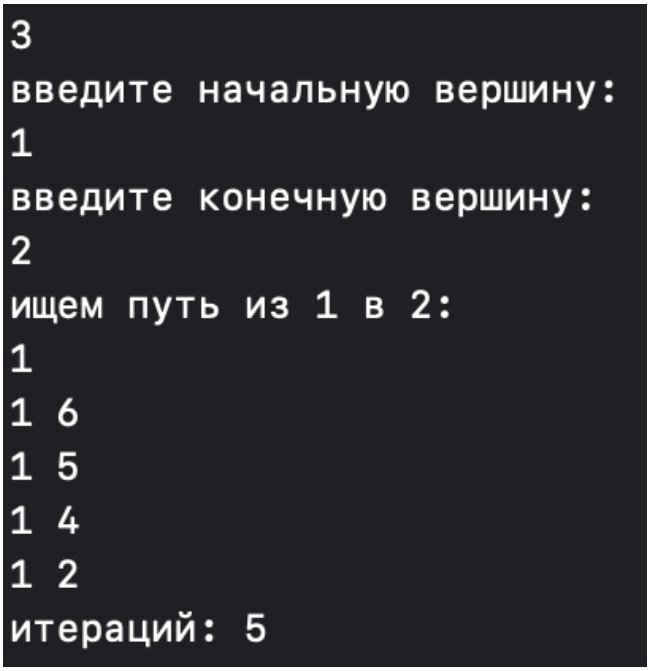

Рис. 7: Поиск в глубину

При выборе «4» пользователь вводит номера вершин и выводится кратчайшее расстояние, вес, вектор кратчайших расстояний – рис. 8.

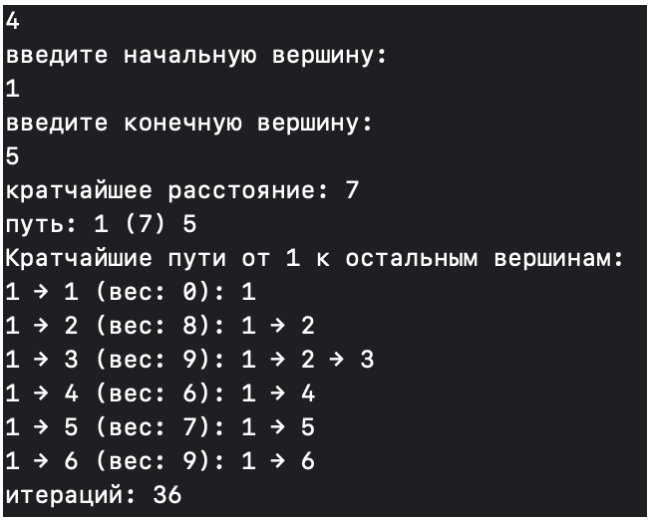

Рис. 8: алгоритм Дейкстры

При выборе «5» сравнивается количество итераций между поиском в глубину и алгоритмом Дейкстры – рис. 9.

```
5
введите начальную вершину:
1
введите конечную вершину:
2
алгоритм Дейкстры: итераций: 36
поиск в глубину: итераций: 5
```

Рис. 9: сравнение скоростей по количеству итераций

При выборе «6» выводится максимальный поток по алгоритму Форда-Фалкерсона, целевой поток, пути для потока минимальной стоимости, минимальная стоимость для целевого потока – рис. 10.

```
6
матрица пропускных способностей:
   0    3    0    2    9    5
   0    0    9    0    5    6
   0    0    0    3    4    4
   0    0    0    0    7    0
   0    0    0    0    0    2
   0    0    0    0    0    0
максимальный поток:
10
целевой поток: 6
матрица стоимости:
    0    4    0    4    7    2
    0    0    1    0    2    5
    0    0    0    3    3    7
    0    0    0    0    2    0
    0    0    0    0    0    6
    0    0    0    0    0    0
пути для потока минимальной стоимости:
    0    1    5 - 1 единиц потока, стоимость: 9
    0    5 - 5 единиц потока, стоимость: 10
минимальная стоимость для потока: 19
```

Рис. 10: алгоритм Форда-Фалкерсона, поток минимальной стоимости

При выборе «7» выводится остов, суммарный вес ребер, матрица Кирхгофа, число остовных деревьев, код Прюфера – рис. 11.

```
7
остовное дерево:
    0    0    0    6    0    0
    0    0    1    0    0    0
    0    1    0    6    0    4
    6    0    6    0    4    0
    0    0    0    4    0    0
    0    0    4    0    0    0
суммарный вес: 21
матрица Кирхгофа:
    4   -1    0   -1   -1   -1
   -1    4   -1    0   -1   -1
    0   -1    4   -1   -1   -1
   -1    0   -1    3   -1    0
   -1   -1   -1   -1    5   -1
   -1   -1   -1    0   -1    4
число остовных деревьев: 360
код Прюфера:
3 вес: 6
2 вес: 1
3 вес: 4
2 вес: 6
5 вес: 4
декодирование:
    0    0    0    6    0    0
    0    0    1    0    0    0
    0    1    0    6    0    4
    6    0    6    0    4    0
    0    0    0    4    0    0
    0    0    4    0    0    0
```

Рис. 11: число остовных деревьев, минимальный по весу остов, код Прюфера

При выборе «8» выбор минимальной раскраски для графа или остова и результат работы алгоритма – рис. 12.

```
8
остовное дерево:
    0    0    0    6    0    0
    0    0    1    0    0    0
    0    1    0    6    0    4
    6    0    6    0    4    0
    0    0    0    4    0    0
    0    0    4    0    0    0
граф:
   0    1    0    1    1    1
   1    0    1    0    1    1
   0    1    0    1    1    1
   1    0    1    0    1    0
   1    1    1    1    0    1
   1    1    1    0    1    0
вы хотите найти минимальную раскраску для исходного графа(1) или остова(2)?
1
Количество использованных цветов: 4
Цвет 1 – номера вершин: 1 3
Цвет 2 – номера вершин: 2 4
Цвет 3 – номера вершин: 5
Цвет 4 – номера вершин: 6
```

Рис. 12: Минимальная раскраска графа

При выборе «9» граф модифицируется до Эйлерова, а также выводится эйлеров цикл – рис. 13.

```
9
Граф не Эйлеров!
Граф был изменён:
удалено ребро между 3 и 4
Граф Эйлеров!
Эйлеров граф:
    0    8    0    6    7    9
    8    0    1    0    9    4
    0    1    0    6    7    4
    6    0    6    0    0    0
    7    9    7    0    0    6
    9    4    4    0    6    0

Эйлеров цикл:
0 5 4 2 5 1 4 0 3 2 1 0
```

Рис. 13: Эйлеров цикл

При выборе «10» граф проверяется на гамильтоновость и решается задача коммивояжера – рис. 14.

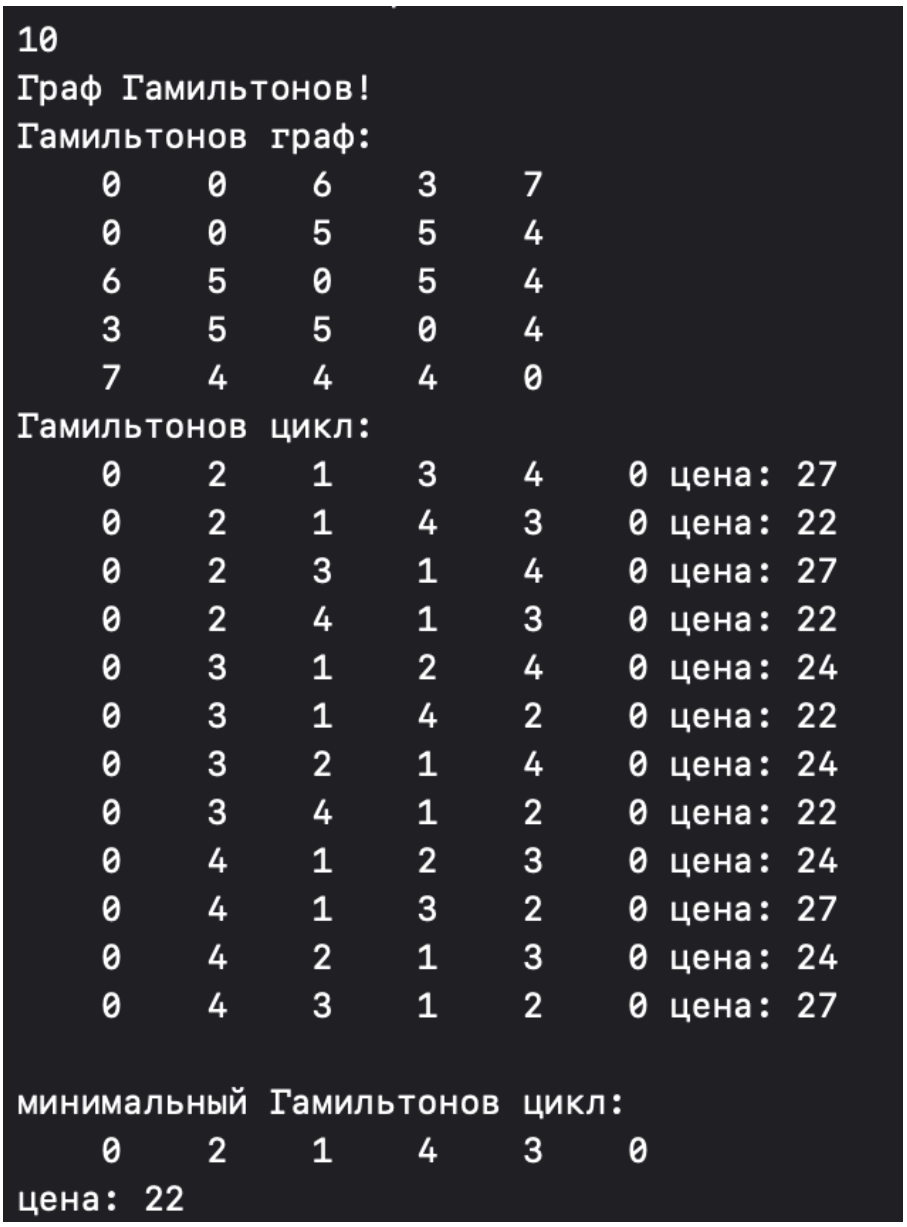

Рис. 14: Гамильтонов цикл

На рис. 15, 16, 17 представлены примеры обработки ограничений.

```
введите количество вершин:
0
введите количество вершин:
p
введите количество вершин:
4
```

Рис. 15: Некорректный ввод

```
матрица смежности:
   0    0    1
   0    0    1
   0    0    0
весовая матрица с положительными весами – 0, с отрицательными – 1
-1
матрица весов:
   0    0   -1
   0    0    6
   0    0    0

выберите следующее действие:
0 – генерировать новый граф
1 – метод Шимбелла
2 – количество маршрутов между двумя вершинами
3 – поиск в глубину
4 – алгоритм Дейкстры
5 – сравнение скоростей по количеству итераций
6 – алгоритм Форда–Фалкерсона, поток минимальной стоимости
7 – число остовных деревьев, минимальный по весу остов, код Прюфера
8 – минимальная раскраска графа или остова
9 – Эйлеров цикл
10 – Гамильтонов цикл
4
нельзя использовать алгоритм Дейкстры: отрицательные веса.
```

Рис. 16: использование алгоритма Дейкстры с отрицательной весовой матрицей

```
введите количество вершин:
2
матрица смежности:
   0    1
   0    0
весовая матрица с положительными весами – 0, с отрицательными – 1
0
матрица весов:
   0    8
   0    0

выберите следующее действие:
0 – генерировать новый граф
1 – метод Шимбелла
2 – количество маршрутов между двумя вершинами
3 – поиск в глубину
4 – алгоритм Дейкстры
5 – сравнение скоростей по количеству итераций
6 – алгоритм Форда–Фалкерсона, поток минимальной стоимости
7 – число остовных деревьев, минимальный по весу остов, код Прюфера
8 – минимальная раскраска графа или остова
9 – Эйлеров цикл
10 – Гамильтонов цикл
9
цикл Эйлера невозможен менее, чем на 3 вершинах
```

Рис. 17: цикл Эйлера на 2 вершинах

Заключение

По результатам выполнения лабораторных работ можно сделать следующие выводы:

В первой работе выполнены задачи по созданию случайного связного ациклического графа и применению метода Шимбелла к нему. Также реализована возможность определения возможности построения маршрута между заданными вершинами и подсчета количества таких маршрутов.

Во второй работе выполнено задание по генерации весовой матрицы для случайно сгенерированных графов из предыдущей работы. Реализован поиск кратчайшего пути между двумя заданными вершинами с использованием алгоритмов Дейкстры, а также поиск максимального пути для указанных вершин.

В третьей работе выполнено формирование связного ациклического графа и построение матриц пропускных способностей и стоимости на его основе. Реализован поиск максимального потока по алгоритму Форда-Фалкерсона и вычисление потока минимальной стоимости с использованием алгоритма Дейкстры.

В четвертой лабораторной работе найдено число остовных деревьев и построены минимальные остовы по весу с помощью алгоритма Прима. Выполнено кодирование полученного остова с помощью кода Прюфера и его последующее декодирование с сохранением весов.

В пятой лабораторной работе проведены проверки на эйлеровость и гамильтоновость графов, а также модификации графов и построение эйлеровых циклов. Выполнена задача коммивояжера на гамильтоновом графе с выводом всех гамильтоновых циклов.

Выполнение лабораторных работ позволило изучить и применить различные алгоритмы работы с графами и оценить их производительность и эффективность. Большая часть задач, таких как метод Шимбелла, алгоритм Дейкстры, была решена для ориентированных графов, но реализацию можно масштабировать на неориентированные графы. Также возможно реализовать графическое представление для данных алгоритмов.

Достоинством программы можно назвать то, что ориентированный граф построенный в лабораторной 1 автоматически достраивается до неориентированного в тех задачах, где это требуется. А также неориентированный граф достраивается до эйлерова и гамильтонова, если это необходимо. Используется контейнер STL vector, который обеспечивает отсутствие утечек памяти из-за использования динамики. Использование раннее реализованных алгоритмов для реализации других алгоритмов.

Недостатками программы можно назвать неоптимальный поиск гамильтонова цикла, так как перебираются все перестановки без использования эвристик и отсутствие разделения кода программы на файлы.

Лабораторные работы были выполнены на языке C++ в среде разработ-

ки Xcode.

Список литературы

[1] Секция "Телематика"/ текст : электронный / — URL: `https://tema.spbstu.ru/dismath/` (Дата обращения 18.05.2025).

[2] Новиков Ф. А. Дискретная математика для программистов. 3-е изд. — Санкт-Петербург: Питер Пресс, 2009. — 384 с.

[3] Вадзинский Р. Н. Справочник по вероятностным распределениям - Санкт-Петербург: Наука, 2001 - 295 с.