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

Высшая школа технологий искусственного интеллекта

Направление: 02.03.01 Математика и компьютерные науки

Отчет о выполнении лабораторных работ №1-5
«Реализация алгоритмов на графе»
Дисциплина «Теория графов»
Вариант 8

Выполнил студент группы
№5130201/30001 __________ Мелещенко С.И.
Проверил __________ Востров А. В.

Санкт-Петербург, 2025

Содержание

Введение

Данный отчет содержит в себе описание выполнения 1 - 5 лабораторных работ.

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).

8) Парето.

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:

b) выполнить обход вершин графа поиском в ширину.

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.

i) алгоритм Беллмана-Форда.

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:

а) Краскала.

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:

е) нахождения максимального независимого множества ребер.

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).

Программа была реализована на языке программирования C++ в среде программирования Microsoft Visual Studio.

1 Математическое описание

1.1 Связный ациклический граф

Графом $G(V, E)$ называется совокупность двух множеств - непустого множества V (множества вершин) и множества E двухэлементных подмножеств множества V (E - множество ребер),

$$G(V,E) \stackrel{\text{Def}}{=} \langle V; E \rangle, \quad V \neq \varnothing, \quad E \subset 2^V \ \& \ \forall e \in E \ (|e| = 2)$$

Пусть v_1, v_2 - вершины, $e = (v_1, v_2)$ - соединяющее их ребро. Тогда вершина v_1 и ребро e инцидентны, ребро e и вершина v_2 также инцидентные. Два ребра, инцидентные одной вершине, называются смежными; две вершины, инцидентные одному ребру, также называются смежными.

Если элементы множества E являются упорядоченные пары, то граф называется ориентированным (орграфом). В этом случае элементы множества V - узлы, элементы множества E - дуги.

Граф $G'(V', E')$ называется подграфом графа $G(V, E)$, если $V' \subset V \& E' \subset E$. Если $V' = V$, то G' называетсяостовным подграфом G.

Маршрутом в графе называется чередующаяся последовательность вершин и рёбер, начинающаяся и кончающаяся вершиной $v_0, e_1, v_1, e_2, v_2, ..., e_k, v_k$, в которой любые два соседних элемента инцидентны, причём однородные элементы (вершины, рёбра) через один смежны или совпадают. Если $v_0{}^{-}v_k$, то маршрут замкнут, иначе — открыт. Если все рёбра различны, то маршрут называется цепью. Цикл - замкнутая цепь. Ациклический граф (орграф) - граф без циклов (контуров).

Две вершины в графе связаны, если существует соединяющая их цепь. Граф, в котором все вершины связаны, называется связным.

1.2 Распределение Парето

Распределение Парето, непрерывное распределение вероятностей с плотностью:

$$f(x) = \frac{\alpha}{x_0}\left(\frac{x_0}{x}\right)^{\alpha+1}, \quad x > x_0,$$

где x_0 — параметр положения, левая граница области возможных значений ($x_0 > 0$); α — параметр формы ($\alpha > 0$)

Функция распределения:

$$F(x) = 1 - \left(\frac{x_0}{x}\right)^{\alpha}, \quad x > x_0$$

Соотношения между распределениями.

1. Если случайная величина T имеет бета-распределение второго рода с параметрами $u = 1$, $v = \alpha$ (т.е. если $f(\tau) = \alpha(1+\tau)^{-(\alpha+1)}$, $0 < \tau < \infty$), то случайная величина $X = 1 + T$ подчиняется закону Парето с параметром положения $x_0 = 1$ и параметром формы α.

2. Распределение Парето с параметрами x_0, α представляет собой отсечение на интервале (x_0, ∞) распределения Парето с параметром положения 1 и параметром формы α.

3. Если случайная величина X имеет распределение Парето с параметрами x_0, α, то случайная величина $Y = \frac{X - x_0}{x_0}$ имеет бета-распределение второго рода с параметрами $u = 1$, $v = \alpha$.

4. Преобразованием $y = \frac{x_0}{x}$ распределение Парето с параметрами x_0, α сводится к бета-распределению первого рода с параметрами $u = \alpha$, $v = 1$.

5. В системе кривых Пирсона распределение Парето принадлежит к распределениям типа VI и XI.

Плотность вероятности и функция раска распределения Парето:

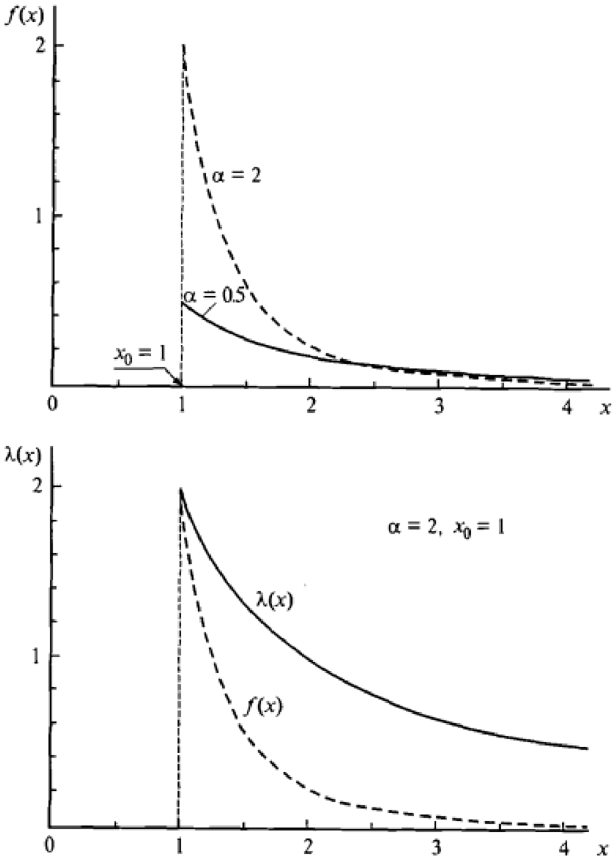


Рис. 1. Плотность вероятности и функция раска распределения Парето

Ниже приведена диаграмма вероятности распределения, построенная по 100 значениям (см. [Рис. 2]).

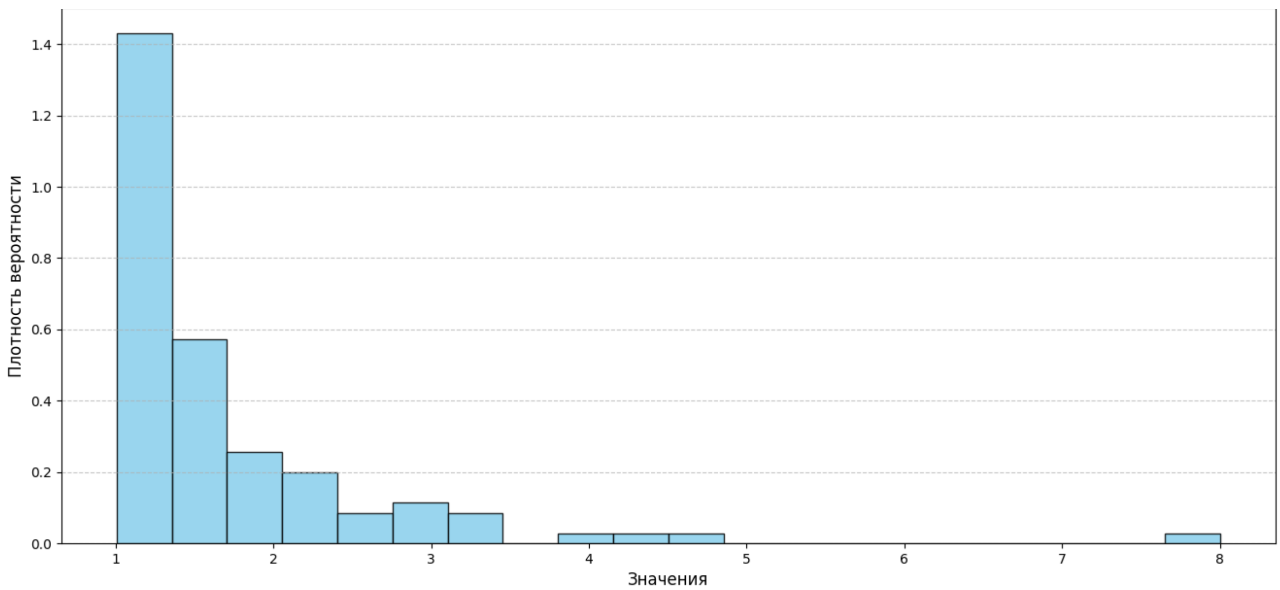


Рис. 2. Диаграмма вероятности распределения по 100 значениям

1.3 Матричное представление графов

Матрица смежности.

Определим матрицу смежности A = A(G) = $(a_{ij})_{n*n}$ графа G, полагая (a_{ij}) равным числу ребер, соединяющих вершины v_j и v_i, причем при i = j каждую петлю учитываем дважды.

Для обыкновенных графов матрица смежности бинарна, т.е. состоит из нулей и единиц, причем ее главная диагональ целиком состоит из нулей.

$$M[i,j] = \begin{cases} 1, & \text{если вершина } v_i \text{ смежна с вершиной } v_j, \\ 0, & \text{если вершины } v_i \text{ и } v_j \text{ не смежны.} \end{cases}$$

Матрица достижимости.

В качестве примера связи между орграфами и бинарными отношениями рассмотрим отношения частичного порядка с точки зрения теории графов. Узел и в орграфе G(V,E) достижим из узла v, если существует путь из v в и. Путь из v в и обозначим <v,u>. Отношение достижимости можно представить матрицей.

Матрица весов.

Вариант матрицы смежности для взвешенного графа, представляет собой квадратную матрицу, (i,j)-й элемент которой равен весу ребра/-дуги (v_i, v_j) если таковое имеется в графе; в противном случае (i,j)-й элемент полагается равным нулю или бесконечности.

Матрица стоимости.

Вариант матрицы смежности для графа, представляет собой квадратную матрицу, (i,j)-й элемент которой равен стоимости ребра/дуги (v_i, v_j) если таковое имеется в графе; в противном случае (i,j)-й элемент полагается равным нулю или бесконечности.

Матрица пропускноых способностей.

Вариант матрицы смежности для графа, представляет собой квадратную матрицу, (i,j)-й элемент которой равен пропускной способности ребра/дуги (v_i, v_j) если таковое имеется в графе; в противном случае (i,j)-й элемент полагается равным нулю или бесконечности.

Матрица Кирхгофа.

Матрица Кирхгофа — одно из представлений конечного графа с помощью матрицы. Матрица Кирхгофа представляет дискретный оператор Лапласа для графа. Она используется для подсчёта остовных деревьев данного графа.

$$B(G) = \begin{pmatrix} \deg v_1 & 0 & \dots & 0 \\ 0 & \deg v_2 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & \deg v_n \end{pmatrix} - A(G),$$

где $A(G)$ — матрица смежности графа G. Иными словами,

$$\beta_{ij} = \begin{cases} -1, & \text{если } i \neq j \text{ и } v_i \text{ смежна с } v_j, \\ 0, & \text{если } i \neq j \text{ и } v_i \text{ не смежна с } v_j, \\ \deg v_i, & \text{если } i = j. \end{cases}$$

Сумма элементов каждой строки (столбца) матрицы Кирхгофа равна нулю. Определитель матрицы Кирхгофа равен нулю.О является собственным значением матрицы (соответствующий собственный вектор — все единицы), кратность его равна числу связных компонент графа. Остальные собственные значения положительны.

1.4 Метод Шимбела

Это специальные операции над элементами матрицы смежности вершин, позволяющие находить кратчайшие или максимальные пути между вершинами, состоящие из заданного количества рёбер.

1. Операция умножения двух величин а и в при возведении матрицы в степень соответствует их алгебраической сумме.

$$\begin{cases} a * b = b * a \Rightarrow a + b = b + a, \\ a * 0 = 0 * a = 0 \Rightarrow a + 0 = 0. \end{cases}$$

2. Операция сложения двух величин a и b заменяется выбором из этих величин минимального (максимального) элемента.

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

нули при этом игнорируются. Минимальный или максимальный элемент выбирается из ненулевых элементов. Нуль в результате операции может быть получен лишь тогда, когда все элементы из выбираемых – ненулевые.

С помощью операций Шимбелла длины кратчайших или максимальных путей между всеми вершинами определяются возведением в степень весовой матрицы, содержащей веса рёбер.

1.5 Поиск количества маршрутов от одной вершины до другой

Пусть дан простой граф $G = (V, A)$, матрица смежности которого есть $\mathbf{A} = (a_{ij})_{n\times n}$, где $a_{ij} = 1 \Leftrightarrow (i, j) \in A$. Матрица смежности даёт информацию о всех путях длины 1 (то есть дугах) в орграфе. Для поиска путей длины 2 можно найти композицию отношения A с самим собой:

$A \circ A = \{(a, c) : \exists b \in V : (a, b), (b, c) \in A\}$.

По определению, матрица композиции отношений $A \circ A$ есть $\mathbf{A}^2 = \left(a^2{}_{ij}\right)_{n\times n} = (\sum_k a_{ik}a_{kj}) = ((a_{i1} \wedge a_{1j}) \vee (a_{i2} \wedge a_{2j}) \vee \ldots \vee (a_{in} \wedge a_{nj}))$, где $\wedge-$ конъюнкция, а $\vee-$ дизъюнкция.

После нахождения матриц $\mathbf{A}^k$ композиций $\underbrace{A \circ \ldots \circ A}_{k}$ для всех $k, 1 \leq k \leq n-1$ будет получена информация о всех путях длины от 1 до $n-1$.

1.6 Алгоритм Беллмана-Форда

Если веса – произвольные числа и граф G не содержит ориентированных циклов отрицательного веса, то минимальный путь может быть найден по алгоритму Беллмана – Форда, похожему на алгоритм Дейкстры и часто называемому алгоритмом корректировки меток.

Относится к классу динамического программирования.

Как и в алгоритме Дейкстры, всем вершинам приписываются метки, однако здесь нет деления меток на временные и постоянные. Корректировка меток осуществляется по старому правилу, однако вместо процедуры превращения временной метки в постоянную формируется очередь вершин, в которых нужно проанализировать по правилу возможности уменьшения меток вершин, не находящихся в данный момент в очереди, но достижимых из нее за один шаг. В процессе работы алгоритма одна и та же вершина может несколько раз вставать в очередь, а затем покидать ее.

Алгоритм Беллмана-Форда работает за O(|V|*|E|) времени.

Этап 1. Нахождение длин кратчайших путей от вершины 5 до всех остальных вершин графа.

Шаг 1. Присвоение начальных значений.

Шаг 2. Корректировка меток и очереди. Удаляем из очереди вершину, находящуюся в самом начале очереди. Для каждой вершины x_i, непосредственно достижимой из x, корректируем ее метку. Если при этом $d.(x_i) < d.(x_i)$, то корректируем очередь вершин, иначе продолжаем перебор вершин и корректировку временных меток.

Корректировка очереди. Если x_i, не была ранее в очереди и не находится в ней в данный момент, то вершину x, ставим в конец очереди. Если же x, уже была когда-нибудь ранее в очереди или находится там в данный момент, то переставляем ее в начало очереди.

Шаг 3. Проверка на завершение первого этапа. Если Q не равно 0, то возвращаемся к началу второго шага, если же Q=0, то первый этап закончен, то есть минимальные расстояния от s до всех вершин графа найдены.

Этап 2. Построение самого кратчайшего пути.

Шаг 5. Последовательный поиск дуг кратчайшего пути. Среди вершин, непосредственно предшествующих вершине x с постоянными метками, находим вершину x_i, удовлетворяющую соотношению $d(x) = d(x_i) + \omega(x_i, x)$ Включаем дугу $(_i,)$ в искомый путь и полагаем $=_i$.

Шаг 6. Проверка на завершение второго этапа. Если x = s, то кратчайший путь найден его образует последовательность дуг, полученных на пятом шаге и выстроенных в обратном порядке. В противном случае возвращаемся к пятому шагу.

1.7 Алгоритм обхода вершин в ширину

Сначала исследуются все вершины, смежные с начальной вершиной (вершина с которой начинается обход). Эти вершины находятся на расстоянии 1 от начальной. Затем исследуются все вершины на расстоянии 2 от начальной, затем все на расстоянии 3 и так далее. При этом для каждой вершины сразу находятся длина кратчайшего маршрута от начальной вершины.

Сложность: O(|V|+|E|).

Алгоритм.

Шаг 1. Всем вершинам графа присваивается значение не посещённой. Выбирается первая вершина и помечается как посещённая и заносится в очередь.

Шаг 2. Посещается первая вершина из очереди (если она не помечена как посещённая). Все её соседние вершины заносятся в очередь. После этого она удаляется из очереди.

Шаг 3. Повторяется шаг 2 до тех пор, пока очередь не станет пустой.

1.8 Алгоритм Форда-Фалкерсона

Алгоритм Форда - Фалкерсона решает задачу нахождения максимального потока в транспортной сети.

Изначально величине потока присваивается значение $0 : f(u, v) = 0$ для всех $u, v \in V$. Затем величина потока итеративно увеличивается посредством поиска увеличивающего пути (путь от источника s к стоку t, вдоль которого можно послать больший поток). Процесс повторяется, пока можно найти увеличивающий путь.

Сложность: O(p*q*U) U – наибольшая величина максимальной пропускной способности сети (оценки, содержащие U, являются слабо полиномиальными), p – количество вершин, q – количество ребер.

1. Обнуляем все потоки. Остаточная сеть изначально совпадает с исходной сетью.

2. В остаточной сети находим любой путь из источника в сток. Если такого пути нет, останавливаемся.

3. Пускаем через найденный путь (он называется *увеличивающим путём* или *увеличивающей цепью*) максимально возможный поток:

1. На найденном пути в остаточной сети ищем ребро с минимальной пропускной способностью $c_{\min}$.

2. Для каждого ребра на найденном пути увеличиваем поток на $c_{\min}$, а в противоположном ему - уменьшаем на $c_{\min}$.

3. Модифицируем остаточную сеть. Для всех рёбер на найденном пути, а также для противоположных (антипараллельных) им рёбер, вычисляем новую пропускную способность. Если она стала ненулевой, добавляем ребро к остаточной сети, а если обнулилась, стираем его.

4. Возвращаемся на шаг 2.

1.9 Алгоритм поиска потока минимальной стоимости

Рассмотрим задачу определения потока, заданной величины θ от s к t в сети $G = (S, U, \Omega, D)$, в которой каждая дуга характеризуется пропускной способностью и неотрицательной стоимостью $d\left(x_i, x_j\right) \in D$ пересылки единичного потока из x_i в x_j вдоль дуги $\left(x_i, x_j\right)$.

Если $\theta > \varphi_{\max}$, где $\varphi_{\max}$ - величина максимального потока в сети G от к t, то решения нет. Если же $\theta \leq \varphi_{\max}$, то может быть определено несколько различных потоков величины θ от ε к t.

Минимизировать

$$\sum_{(x_i,x_j)\cup\cup} d\left(x_i, x_j\right) \cdot \varphi\left(x_i, x_j\right),$$

где $\varphi\left(x_i, x_j\right)$ - поток по дуге $\left(x_i, x_j\right)$ при ограничениях:

1) $0 \leq \varphi\left(x_i, x_j\right) \leq dx_i, x_j) \exists \left(x_i, x_j\right) \in U$;

2) для начальной вершины $s \in S \sum_{x_i \in S} \varphi\left(s, x_i\right) - \sum_{x_j \in S} \varphi\left(x_i, s\right) = \theta$ - уравнение истока;

3) для конечной вершины $t \in S \sum_{x_i \in S} \varphi\left(t, x_i\right) - \sum_{x_i \in S} \varphi\left(x_i, t\right) = -\theta$ - уравнение стока;

4) для любой другой вершины $x_j \neq s, t \sum_{x_i \in S} \varphi\left(x_j, x_i\right) - \sum_{x_i \in S} \varphi\left(x_i, x_j\right) = 0$ - условие сохранения потока.

Модификация сети допускается только определенного порядка.

Алгоритм Форда-Беллмана работает за O(VE) , улучшение цикла за O(E) . Если обозначить максимальную стоимость потока как C , а максимальную пропускную способность как U , то алгоритм совершит O(ECU) итераций поиска цикла, если каждое улучшение цикла будет улучшать его на 1. Сложность: $O(VE^2CU + maxFlow), maxFlow -$.

1.10 Матричная теорема Кирхгофа

Матричная теорема о деревьях или теорема Кирхгофа даёт число остовных деревьев графа через определитель матрицы.

Пусть $G-$ связный помеченный граф с матрицей Кирхгофа M. Все алгебраические дополнения матрицы Кирхгофа M равны между собой и их общее значение равно количеству остовных деревьев графа G.

Алгебраическим дополнением элемента a_{ij} матрицы A называется число

$$A_{ij} = (-1)^{i+j} M_{ij}$$

где M_{ij} - дополнительный минор, определитель матрицы, получающейся из исходной матрицы A путём вычёркивания $i-$ й строки и j-го столбца.

1.11 Алгоритм Краскала

Алгоритм Краскала используется для нахождения кратчайшего остова графа. Перебираем рёбра в порядке возрастания их веса. Если ребро является безопасным, добавляем его.

Шаги реализации алгоритма Краскала.

1. Сортировать все ребра от малого веса до высокого.

2. Возьмите ребро с наименьшим весом и добавьте его в остовное дерево. Если добавление ребра создало цикл, то отклоните это ребро.

3. Продолжайте добавлять ребра, пока не достигнете всех вершин.

Алгоритм работает за $O(E(logE + a(V))) = O(ElogE)$.

1.12 Код Прюфера

Код Прюфера сопоставляет произвольному конечному дереву с n вершинами последовательность из $n-2$ чисел (от 1 до n) с возможными повторениями. Отношение между деревом с помеченными вершинами и кодом Прюфера является взаимно однозначным: каждому дереву соответствует уникальный код Прюфера, при этом номерам вершин сопоставляются элементы последовательности кода. Обратно, по заданному коду из $n-2$ чисел можно однозначно восстановить дерево с n вершинами.

Пусть T есть дерево с вершинами, занумерованными числами $\{1, 2, \ldots, n\}$. Построение кода Прюфера дерева T ведётся путем последовательного удаления вершин из дерева, пока не останутся только две вершины. При этом каждый раз выбирается концевая вершина с наименьшим номером, и в код записывается номер единственной вершины, с которой она соединена. В результате образуется последовательность $(p_1, \ldots, p_{n-2})$, составленная из чисел $\{1, 2, \ldots, n\}$, возможно с повторениями.

Для восстановления дерева по коду $p = (p_1, \ldots, p_{n-2})$ заготовим список номеров вершин $s = (1, \ldots, n)$. Выберем первый номер i_1, который не встречается в коде. Добавим ребро (i_1, p_1), после этого удалим i_1 из s и p_1 из p. Повторяем процесс до момента, когда код p становится пустым. В этот момент список s содержит ровно два числа i_{n-1} и n. Остаётся добавить ребро (i_{n-1}, n), и дерево построено.

1.13 Алгоритм нахождения максимального независимого множества ребер

В теории графов максимальным независимым множеством, максимальным устойчивым множеством, или максимальным стабильным множеством называется независимое множество, не являющееся подмножеством другого независимого множества. То есть это такое множество вершин S, что любое ребро графа имеет хотя бы одну конечную вершину, не принадлежащую S, и любая вершина не из S имеет хотя бы одну соседнюю в S.

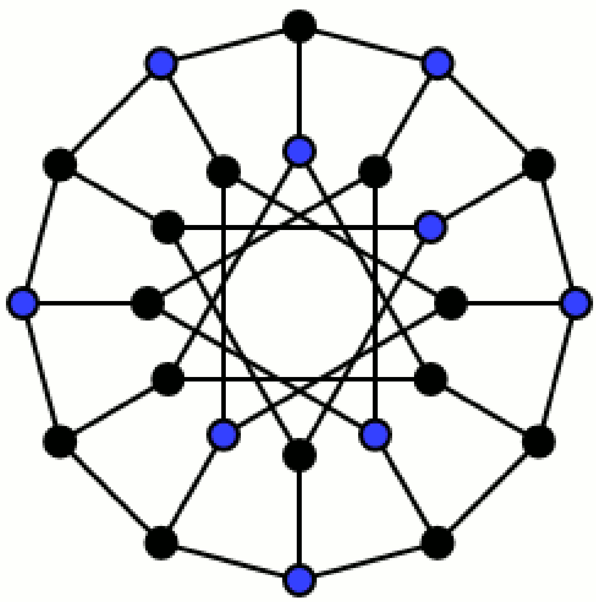


Рис. 3. Максимальное независимое множество

Множество вершин синего цвета — максимальное независимое множество.

Алгоритм.

1. Создаем пустой список для хранения выбранных рёбер.

2. Создаем массив отметок о использованных вершинах, изначально все false.

3. Для каждой вершины u от 0 до ver-1:

Если вершина u ещё не использована, то перебираем все вершины v от 0 до ver-1. Если между u и v есть ребро (matrix[u][v] != 0) и v ещё не использована - добавляем ребро (u,v) в matching. Помечаем u и v как использованные. Прерываем перебор для текущей вершины u

4. Возвращаем полученный список рёбер matching

Сложность: O(V^2).

1.14 Эйлеров граф

Эйлеров путь (эйлерова цепь) в графе - это путь, проходящий по всем рёбрам графа и притом только по одному разу.

Эйлеров цикл - эйлеров путь, являющийся циклом, то есть замкнутый путь, проходящий через каждое ребро графа ровно по одному разу.

Эйлеров граф - граф, в котором существует эйлеров цикл.

Согласно теореме, доказанной Эйлером, эйлеров цикл существует тогда и только тогда, когда граф связный или будет являться связным, если удалить из него все изолированные вершины, и в нём отсутствуют вершины нечётной степени.

Теорема: если граф G связен и нетривиален, то следующие утверждения эквивалентны:

1. G — эйлеров граф.

2. Каждая вершина G имеет чётную степень.

3. Множество рёбер G можно разбить на простые циклы.

1.15 Гамильтонов граф

Если граф имеет простой цикл, содержащий все вершины графа (по одному разу), то такой цикл называется гамилътоновым циклом, а граф называется гамилътоновым графом.

Гамильтонов цикл пе обязательно содержит все рёбра графа.

Любой граф G можно превратить в гамильтонов, добавив достаточное количество новых вершин и инцидентных им рёбер и не добавляя рёбер, инцидентных только старым вершинам.

Теорема (достаточное условие): если 6(G) >= p/2, то граф G является гамилътоновым, где 6(G) - минимальная степень вершин графа.

1.16 Задача коммивояжера

Рассмотрим следующую задачу, известную как задача коммивояжёра. Имеется p городов, расстояния между которыми известны. Коммивояжёр должен посетить все p городов по одному разу, вернувшись в тот, с которого начал. Требуется найти такой маршрут движения, при котором суммарное пройденное расстояние будет минимальным.

Требуется найти маршрут, проходящий через все вершины графа ровно по одному разу, такой что его длина (сумма длин рёбер маршрута) минимальна.

Задача коммивояжера – задача, в которой коммивояжер должен посетить N городов, добавляя в каждом из них ровно по одному разу на конец путешествия вернуться в тот город, с которого он начал. В какой последовательности коммивояжеру нужно объехать города, чтобы общая длина пути была наименьшей? Таким образом требуется

$$\sum_{(u,v)\in E} w(u,v) \to \min$$

при этом каждая вершина должна быть посещена ровно один раз. Для описания задачи коммивояжера используются следующие переменные:

- N – количество городов, которые нужно посетить;
- $V = \{v_1, v_2, \ldots, v_N\}$ – множество вершин графа, где каждая вершина представляет собой город, который нужно посетить;
- E – множество рёбер графа, которое определяется парами вершин (u, v), где $u, v \in V$ и $u \neq v$;
- $w(u, v)$ – вес ребра (u, v), который представляет собой длину пути между городами u и v.

Задачу коммивояжера можно представить в виде целевой функции:

Пусть $p_1, p_2, \ldots, p_{n-1}, p_n = p_0$ – номера городов, записанные в порядке обхода. То есть p_n – номер города, посещаемого на k-м месте, $k = 0, \ldots, n$, $p_0 = p_n$. Тогда пройденное расстояние можно представить в виде целевой функции:

$$\sum_{k=0}^{n-1} c[p_k][p_{k+1}]$$

Среди всех $p_1, p_2, \ldots, p_n \neq p_0$ по одному разу встречается каждое число из интервала $2..n$. Таким образом задача коммивояжера состоит в поиске перестановок чисел от 2 до n, при которой целевая функция минимальна.

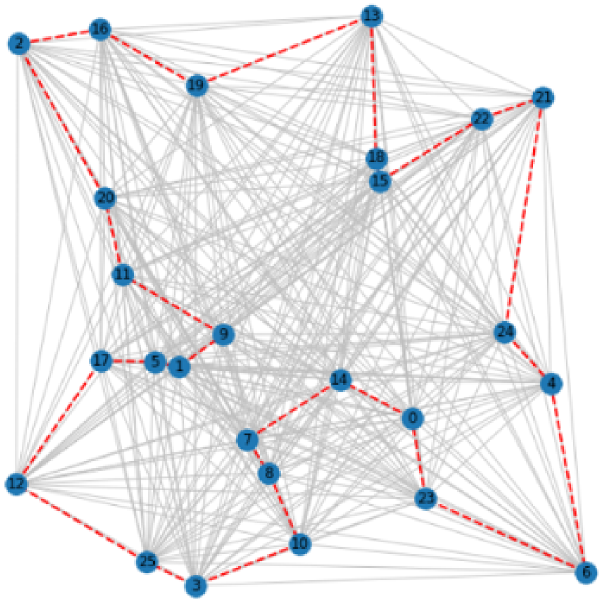


Рис. 4. Задача коммивояжера

2 Особенности реализации

2.1 Библиотечные классы

<iostream> – ввод/вывод (например, cin, cout, cerr).

<random> – генерация случайных чисел (рандомные распределения, генераторы).

<cmath> – математические функции (sin, sqrt, pow и т. д.).

<vector> – динамический массив (std::vector).

<queue> – очередь (std::queue) и приоритетная очередь (std::priority_queue).

<algorithm> – алгоритмы (sort, find, reverse и др.).

<set> – упорядоченное множество (std::set) и мультимножество (std::multiset).

<iomanip> – форматирование вывода (setw, fixed, setprecision).

<ctime> – работа со временем (time, clock, localtime).

<stack> – стек (std::stack).

<fstream> – работа с файлами (ifstream, ofstream).

2.2 Порядок вызова функций

1. Инициализация и ввод параметров.

double alpha = 1; // Параметр распределения Парето (влияет на равномерность степеней вершин).

int xmin = 1; // Минимальная степень вершины (смещение распределения) alpha определяет форму распределения степеней вершин (чем больше, тем более равномерно распределены степени).

xmin задаёт минимальную степень вершины (фиксировано как 1 для корректной работы).

2. Основной цикл программы.

while (continuef) // Ввод количества вершин // Генерация графа // Основное меню

Программа работает в цикле, пока пользователь не выберет выход (continuef = false).

3. Генерация графа.

a) Генерация степеней вершин (распределение Парето)

GenerateVertexDegreesPareto(degrees, ver, alpha, xmin);

Генерирует степени вершин в соответствии с распределением Парето.

Проверяет, что граф может быть построен (сумма степеней чётная, нет изолированных вершин и т. д.).

b) Построение матрицы смежности

vector<vector<int» admatrix = Createadmatrix(ver, degrees);

Создаёт ориентированный граф на основе сгенерированных степеней вершин.

Для каждой вершины случайным образом выбираются рёбра, учитывая её степень.

c) Генерация весов рёбер

vector<vector<int» weightmatrix = GenerateWeightMatrix(ver, admatrix, alpha, xmin);

Веса рёбер генерируются также по распределению Парето.

Если выбран режим с отрицательными весами (sign == 2), некоторые веса умножаются на -1.

d) Матрицы

Матрица достижимости (res_dostmatrix).

Матрица Кирхгофа (kirhgof_matrix).

Неориентированный граф (undirected_admatrix) создаётся путём симметризации матрицы смежности.

4. Главное меню.

Программа предоставляет 19 функций для работы с графом.

Будет рассмотрено далее.

5. Выход из программы

cout « "ДО НОВЫХ ВСТРЕЧ!«< endl;

return 0;

2.3 Функция GenerateVertexDegreesPareto

Вход: degrees - ссылка на вектор, куда будут записаны степени вершин.

ver - количество вершин в графе.

alpha - параметр формы распределения Парето. xmin - минимальное значение степени вершины.

Выход: заполненный вектор degrees, вывод в консоль степеней всех вершин.

Функция назначает веса рёбрам ориентированного графа, используя распределение Парето.

Строится массив вероятностей для степеней от xmin до ver, используется формула из справочника. После вычисления, вероятности нормализуются (суммируются и делятся на сумму).

Для создания дискретного распределения используется

std::discrete_distribution для генерации степеней в соответствии с рассчитанными вероятностями.

Генерируются степени для всех вершин. Вектор сортируется по убыванию. У самой последней вершины степень принудительно обнуляется.

Выполняются проверка на существование графа. Каждая степень не больше допустимого числа оставшихся вершин. Общая сумма степеней не превышает максимально возможную для ориентированного графа без петель. Только одна вершина может иметь степень ver - 1.

После генерации выводятся степени каждой вершины в консоль.

Листинг 1. GenerateVertexDegreesPareto

```
void GenerateVertexDegreesPareto(vector<int>& degrees, int ver,
    double alpha, int xmin) {
     vector<double> probability(ver);

     static random_device rd;
     static mt19937 gen(rd());

     double sum_prob = 0.0;
     for (int k = xmin; k <= ver; ++k) {
          double prob = pow(k, -(alpha + 1));
          probability[k - 1] = prob;
          sum_prob += prob;
     }

     for (int k = 0; k < ver; ++k) {
          probability[k] /= sum_prob;
     }
     discrete_distribution<> distribution(probability.begin(),
        probability.end());
```

```
    bool flag = true;
    while (flag) {
        for (int i = 0; i < ver; ++i) {
            degrees[i] = distribution(gen) + xmin;
        }
        sort(degrees.begin(), degrees.end());
        reverse(degrees.begin(), degrees.end());
        degrees[ver - 1] = 0;
        int pr_sum = 0;
        int sumVertexDegrees = 0;
        int max_deg = 0;
        for (int i = 0; i < ver; i++) {
            if (degrees[i] <= ver - i - 1) {
                pr_sum += 1;
                sumVertexDegrees += degrees[i];
            }
            else if (degrees[i] == ver - 1) max_deg += 1;
        }
        if (max_deg == 0) {
            degrees[0] = ver - 1;
            max_deg += 1;
        }
        if (pr_sum == ver && sumVertexDegrees <= (ver * ver - ver) / 2 && max_deg == 1) flag = false;
    }
    for (int i = 0; i < ver; i++) cout << "Степень вершины " << i << " = " << degrees[i] << endl;
}
```

2.4 Функция Createadmatrix

Вход: ver - количество вершин в графе.

degrees - вектор заданных степеней исходящих рёбер для каждой вершины.

Выход: матрица смежности.

Функция создает матрицу смежности.

Создаётся матрица смежности admatrix заполненная нулями. Копируется вектор степеней degrees в copy_degrees. Для каждой вершины i, согласно её степени copy_degrees[i], добавляется соответствующее число исходящих рёбер.

Случайным образом выбирается вершина j, куда можно провести ребро из i.

Проверяется, что:

1) i не равен j (исключаются петли),

2) admatrix[i][j] == 0 (ребро ещё не существует),

3) j > i (предотвращение симметрии).

После успешной генерации ребра admatrix[i][j] = 1, цикл завершает итерацию.

Функция возвращает полученную матрицу смежности.

Листинг 2. Createadmatrix

```
vector<vector<int>> Createadmatrix(int ver, const vector<int>&
    degrees) {
    vector<vector<int>> admatrix(ver, vector<int>(ver, 0));
    vector<int> copy_degrees(degrees);
    srand(time(0));
    for (int i = 0; i < ver; i++) {
        for (int d = 0; d < copy_degrees[i]; d++) {
            bool n_addded_edge = true;
            while (n_addded_edge) {
                double R = static_cast<double>(rand()) /
                    RAND_MAX;
                int j = static_cast<int>((ver)*R);
                if (admatrix[i][j] != 1 && i != j &&
                    copy_degrees[i] > 0 && j > i) {
                    admatrix[i][j] = 1;
                    n_addded_edge = false;
                }
            }
        }
    }
    return admatrix;
}
```

2.5 Функция GenerateWeightMatrix

Вход: ver - количество вершин в графе.

admatrix - матрица смежности вершин.

alpha - параметр распределения Парето.

xmin -нижний порог Парето.

Выход: матрица весов.

Функция генерирует матрицу весов по распределению Парето.

vector<vector<int» weightmatrix(ver, vector<int>(ver, 0)) задаёт нулевые веса по умолчанию.

Генератор случайных чисел:

static random_device rd;

static mt19937 gen(rd());

Для каждой пары вершин (i, j) проверяется, есть ли ребро в admatrix[i][j]. Если да — генерируется случайный вес по распределению Парето.

Возвращение готовой матрицы весов.

Листинг 3. GenerateWeightMatrix

```
vector<vector<int>> GenerateWeightMatrix(int ver, const vector<vector<int>>& admatrix, double alpha, int xmin) {
    vector<vector<int>> weightmatrix(ver, vector<int>(ver, 0));
    static random_device rd;
    static mt19937 gen(rd());
    for (int i = 0; i < ver; i++) {
        for (int j = 0; j < ver; j++) {
            if (admatrix[i][j] == 1) {
                weightmatrix[i][j] = generateParetoValue(alpha, xmin, gen);
            }
        }
    }
    return weightmatrix;
}
```

2.6 Функция AddMatrix

Вход: matrix1- первая матрица.

matrix2 - вторая матрица.

ver - размерность матриц.

Выход: матрица суммы: sum_matrix[i][j]

Функция Addmatrix выполняет поэлементное сложение двух квадратных матриц.

Создаётся пустая матрица sum_matrix размера ver x ver, инициализированная нулями.

vector<vector<int» sum_matrix(ver, vector<int>(ver, 0)); Два вложенных цикла пробегают по каждой ячейке. Каждый элемент матриц складывается поэлементно.

Возвращается результирующая матрица суммы.

Листинг 4. AddMatrix

```
vector<vector<int>> Addmatrix(vector<vector<int>> matrix1,
   vector<vector<int>> matrix2, const int ver) {
    vector<vector<int>> sum_matrix(ver, vector<int>(ver, 0));
    for (int i = 0; i < ver; i++)
        for (int j = 0; j < ver; j++) sum_matrix[i][j] = (
           matrix1[i][j] + matrix2[i][j]);
    return sum_matrix;
}
```

2.7 Функция determinant

Вход: ver- размерность матрицы.

kirhgof_matrix - матрица Кирхгофа, для которой считается определитель.

Выход: определитель матрицы.

Функция реализует разложение по строке для вычисления определителя.

Если ver == 2, то return ((a*d) - (b*c));

Если ver > 2:

Выбирается первая строка (i = 0) и поочерёдно исключается каждый столбец x.

Создаётся подматрица submatrix размером (ver - 1) $\times$ (ver - 1) без строки 0 и столбца x.

Рекурсивно вызывается determinant для этой подматрицы.

Суммируются с чередованием знаков: det += $(-1)^x$ * элемент * детерминант.

Листинг 5. determinant

```
int determinant(int ver, vector<vector<int>> kirhgof_matrix) {
    int det = 0;
    vector<vector<int>> submatrix(ver, vector<int>(ver, 0));

    if (ver == 2) {
```

```
        return ((kirhgof_matrix[0][0] * kirhgof_matrix[1][1]) - (kirhgof_matrix[1][0] * kirhgof_matrix[0][1]));
    }
    else {
        for (int x = 0; x < ver; x++) {
            int subi = 0;
            for (int i = 1; i < ver; i++) {
                int subj = 0;
                for (int j = 0; j < ver; j++) {
                    if (j == x) {
                        continue;
                    }
                    submatrix[subi][subj] = kirhgof_matrix[i][j];
                    subj++;
                }
                subi++;
            }
            det = det + (pow(-1, x) * kirhgof_matrix[0][x] * determinant(ver - 1, submatrix));
        }
    }
    return det;
}
```

2.8 Функция generateParetoValue

Вход: alpha - параметр формы распределения Парето.

xmin - минимальное возможное значение (смещение).

gen - генератор случайных чисел.

Выход: целое значение, сгенерированное по закону распределения Парето и ограниченное от 1 до 100.

Функция generateParetoValue реализует генерацию случайных значений по распределению Парето.

uniform_real_distribution<> dis(0.0, 1.0);

double u = dis(gen);

Генерируется случайное значение u (0, 1).

По формуле, где U — равномерное число от 0 до 1, — параметр формы, x_min — минимальное значение, вычисляется value.

return clamp(value, 1, 100);

Значение ограничивается в диапазоне от 1 до 100, чтобы избежать слишком больших или нулевых значений.

Листинг 6. generateParetoValue

```
int generateParetoValue(double alpha, int xmin, mt19937& gen) {
uniform_real_distribution<> dis(0.0, 1.0);
double u = dis(gen);
int value = static_cast<int>(xmin / pow(u, 1.0 / alpha));
return clamp(value, 1, 100);}
```

2.9 Функция Createdostmatrix

Вход: admatrix — матрица смежности Представляет связи между вершинами графа.

matrix — произвольная матрица.

ver - количество вершин в графе.

Выход: матрица достижимости.

Функция необходима для получения матрицы достижимости.

Эта функция реализует умножение двух квадратных матриц (matrix × admatrix).

Для каждой пары вершин (i, j) вычисляется сумма произведений соответствующих элементов строки i и столбца j.

Листинг 7. Createdostmatrix

```
vector<vector<int>> Createdostmatrix(vector<vector<int>>
   admatrix, vector<vector<int>> matrix, const int ver) {
    vector<vector<int>> new_matrix(ver, vector<int>(ver, 0));
    for (int i = 0; i < ver; i++) {
        for (int j = 0; j < ver; j++) {
            vector<int> vect;
            for (int k = 0; k < ver; k++) new_matrix[i][j] += (
               matrix[i][k] * admatrix[k][j]);
        }
    }
    return new_matrix;
}
```

2.10 Функция GenerateCostMatrix

Вход: ver – количество вершин в графе.

admatrix – матрица смежности графа.

alpha – параметр распределения Парето.

xmin – минимальное значение, которое может быть использовано при расчете стоимости, параметр распределения Парето.

Выход: заполненная матрица стоимости.

Функция GenerateCostMatrix просто вызывает функцию, GenerateWeightMatrix, передавая ей те же параметры, что и сама получила, так как внешне матрицы весов, стоимости и пропускных способностей одинаковы.

Листинг 8. GenerateCostMatrix

```
vector<vector<int>> GenerateCostMatrix(int ver, const vector<
   vector<int>>& admatrix, double alpha, int xmin) {
    return GenerateWeightMatrix(ver, admatrix, alpha, xmin);
}
```

2.11 Функция GenerateCapacityMatrix

Вход: ver – количество вершин в графе.

admatrix – матрица смежности графа.

alpha – параметр распределения Парето.

xmin – минимальное значение, которое может быть использовано при расчете стоимости, параметр распределения Парето.

Выход: заполненная матрица пропускных способносей.

Функция GenerateCapacityMatrix просто вызывает функцию, GenerateWeightMat передавая ей те же параметры, что и сама получила, так как внешне матрицы весов, стоимости и пропускных способностей одинаковы.

Листинг 9. GenerateCapacityMatrix

```
vector<vector<int>> GenerateCapacityMatrix(int ver, const vector
   <vector<int>>& admatrix, double alpha, int xmin) {
    return GenerateWeightMatrix(ver, admatrix, alpha, xmin);
}
```

2.12 Функция PrintMatrix

Вход: matrix – матрица.

Выход: вывод матрицы в консоль.

Функция для вывода матриц.

for (const auto& row : matrix) — этот цикл перебирает все строки матрицы matrix. В каждой итерации переменная row содержит ссылку на текущую строку матрицы.

for (int val : row) — этот вложенный цикл перебирает все элементы в текущей строке row и выводит их в консоль.

cout « setw(7) « val « setw(5); — каждая ячейка матрицы выводится с шириной 7 символов для каждого значения, для выравнивания.

Листинг 10. PrintMatrix

```
void PrintMatrix(vector<vector<int>> matrix) {
    for (const auto& row : matrix) {
        for (int val : row) cout << setw(7) << val << setw(5);
        cout << "\n";
    }
    cout << "\n";
}
```

2.13 Функция PrintPath

Вход: from – вектор, где для каждой вершины хранится предыдущая вершина в пути.

finish – индекс вершины, до которой нужно восстановить путь.

Выход: возвращается вектор path, который содержит вершины пути в правильном порядке.

Функция PrintPath предназначена для восстановления пути от начальной вершины до конечной вершины в графе.

path — создается пустой вектор path, который будет содержать вершины пути от начальной вершины до конечной.

for (int v = finish; v != -1; v = from[v]) path.push_back(v) — начинаем с вершины finish и, используя информацию из вектора from, восстанавливаем путь по цепочке. Для каждой вершины v в пути добавляется сама вершина в вектор path, а затем мы переходим к предыдущей вершине, используя v = from[v]. Это продолжается до тех пор, пока не дойдем до вершины с индексом -1, что обычно означает начало пути.

reverse(path.begin(), path.end()) — так как путь восстанавливается от конечной вершины к начальной, нужно перевернуть вектор, чтобы порядок вершин соответствовал пути от начальной до конечной вершины.

Листинг 11. PrintPath

```
vector<int> PrintPath(vector<int> from, int finish) {
    vector<int> path;
    for (int v = finish; v != -1; v = from[v]) path.push_back(v)
        ;
    reverse(path.begin(), path.end());
    return path;
}
```

2.14 Функция PrintWay

Вход: ver — количество вершин в графе.

dist — вектор расстояний от начальной вершины до каждой другой вершины.

from — индекс начальной вершины.

where_ — индекс конечной вершины.

path — вектор, содержащий путь от начальной вершины до конечной. Путь представлен в виде последовательности вершин.

Выход: выводит маршрут и длину маршрута.

Функция сначала выводит Вектор расстояний.

Для вершин с действительными расстояниями выводится их значение, разделённое табуляцией.

Если расстояние до конечной вершины не равно значениям trash1 или -trash1, то выводится информация о длине маршрута.

После этого выводится сам маршрут, то есть путь от начальной вершины до конечной, который содержится в векторе path. Вершины маршрута выводятся через стрелки ->.

Если путь не существует, выводится сообщение "Такой маршрут построить нельзя".

Листинг 12. PrintWay

```
void PrintWay(int ver, vector<int> dist, int from, int where_,
    vector<int> path) {
    cout << "Вектор расстояний: " << endl;
    for (int i = 0; i < ver; i++) {
```

```
        if (dist[i] != trash1 and dist[i] != trash2 and dist[i] != -trash1) cout << dist[i] << "\t";
        else cout << "-" << "\t";
    }
    if (dist[where_] != trash1 and dist[where_] != -trash1) {
        cout << "\nДлина маршрута: " << dist[where_] << ".";
        cout << "\nМаршрут: ";
        for (int i = 0; i < path.size(); i++) {
            if (i == path.size() - 1) cout << path[i] << ".";
            else cout << path[i] << "->";
        }
    }
    else cout << "\nТакой маршрут построить нельзя" << endl;
}
```

2.15 Функция Shimbel

Вход: weightmatrix — матрица весов, где каждый элемент представляет вес ребра между вершинами.

matrix — матрица смежности.

ver — количество вершин в графе.

parametr — параметр, который определяет, как обрабатывать значения при вычислениях:

parametr == 1 — выбирается максимальное значение.

parametr == 2 — выбирается минимальное значение, исключая нули.

Выход: new_matrix итоговая матрица-результат.

vector<vector<int» new_matrix(ver, vector<int>(ver, 0)) — создается новая матрица размером ver x ver, и все её элементы инициализируются нулями. Это будет итоговая матрица, которая будет возвращена.

Запускается двойной цикл по всем вершинам.

Внешний цикл for (int i = 0; i < ver; i++) — перебирает все строки.

Вложенный цикл for (int j = 0; j < ver; j++) — перебирает все столбцы.

Для каждой пары (i, j) создается временный вектор vect, который будет хранить результаты вычислений для строки i и столбца j.

Внутренний цикл for (int k = 0; k < ver; k++) — перебирает все возможные промежуточные вершины k.

Для каждой вершины k, если существует ребро между вершинами i и k (матрица смежности matrix[i][k] != 0) и между вершинами k и j (матрица весов weightmatrix[k][j] != 0), то вычисляется сумма значений matrix[i][k] + weightmatrix[k][j], которая добавляется в вектор vect.

Если одно из значений равно нулю, то вектор заполняется нулями (не учитывая такие промежуточные вершины).

Вектор vect сортируется для упорядочивания всех значений.

Если все элементы вектора vect равны нулю (вектор состоит из одних нулей), то элемент в новой матрице new_matrix[i][j] остается равным нулю.

Если parametr == 1, то выбирается максимальное значение из вектора vect (последний элемент после сортировки).

Если parametr == 2, то исключается значение 0, и выбирается минимальное ненулевое значение (первый элемент в отсортированном векторе после удаления нулей).

Листинг 13. Shimbel

```
vector<vector<int>> Shimbel(vector<vector<int>> weightmatrix, vector<vector<int>> matrix, const int ver, const int parametr) {
    vector<vector<int>> new_matrix(ver, vector<int>(ver, 0));
    for (int i = 0; i < ver; i++) {
        for (int j = 0; j < ver; j++) {
            vector<int> vect;
            for (int k = 0; k < ver; k++) {
                if (matrix[i][k] == 0 or weightmatrix[k][j] == 0) vect.push_back(0);
                else vect.push_back(matrix[i][k] + weightmatrix[k][j]);
            }
            sort(vect.begin(), vect.end());
            if (vect.back() == vect.front() && vect.back() == 0) new_matrix[i][j] = 0;
            else {
                if (parametr == 1) new_matrix[i][j] = vect.back();
                else if (parametr == 2) {
                    erase(vect, 0);
                    new_matrix[i][j] = vect.front();
                }
            }
        }
    }
    return new_matrix;
```

```
}
```

2.16 Функция BFS

Вход: weightmatrix — матрица весов.

ver — количество вершин в графе.

start — индекс начальной вершины.

finish — индекс конечной вершины, до которой нужно найти кратчайший путь.

path — ссылка на вектор, в который будет записан найденный путь.

iter_BFS — ссылка на переменную, в которой будет храниться количество итераций, выполненных в процессе работы BFS.

Выход: вектор distance, который содержит минимальные расстояния, iter_BFS с определенным количеством операций.

Функция BFS реализует алгоритм поиска в ширину (Breadth-First Search) с использованием очереди для нахождения кратчайшего пути от начальной вершины start до конечной вершины finish.

vector<int> from(ver, -1); — вектор from хранит для каждой вершины информацию о предыдущей вершине на пути (если вершина v достигается через вершину u, то from[v] = u). Изначально все вершины имеют значение -1, что означает отсутствие пути.

vector<int> distance(ver, trash1); — вектор расстояний от начальной вершины до других. Изначально все расстояния установлены в значение trash1 (значение, указывающее на то, что вершина ещё не посещена).

queue<int> q; — очередь для BFS, в которую помещаются вершины для дальнейшего обхода.

distance[start] = 0; — начальная вершина имеет расстояние 0 от самой себя.

q.push(start); — начальная вершина помещается в очередь для начала обхода.

while (!q.empty()) — пока очередь не пуста, продолжаем обход графа.

int vert = q.front(); q.pop(); — извлекаем вершину из очереди и обрабатываем её.

Вложенный цикл for (int to = 0; to < ver; to++) — перебираем все возможные вершины, которые могут быть соседями для текущей вершины vert.

Если между вершинами vert и to существует ребро (то есть weightmatrix[vert][to] != 0), проверяется, можно ли обновить расстояние до вершины to. Если да, то:

distance[to] = distance[vert] + weightmatrix[vert][to]; — обновляем расстояние до вершины to.

q.push(to); — добавляем вершину to в очередь для дальнейшего обхода.

from[to] = vert; — запоминаем, что до вершины to можно добраться через вершину vert.

После завершения обхода вызывается функция PrintPath(from, finish), которая восстанавливает путь от начальной вершины start до конечной вершины finish с использованием массива from.

Листинг 14. BFS

```
vector<int> BFS(vector<vector<int>> weightmatrix, int ver, int
    start, int finish, vector<int>& path, int& iter_BFS) {
    vector<int> from(ver, -1);
    iter_BFS = 0;
    vector<int> distance(ver, trash1);
    queue<int> q;
    distance[start] = 0;
    q.push(start);

    while (!q.empty()) {
        int vert = q.front();
        q.pop();
        for (int to = 0; to < ver; to++) {
            if (weightmatrix[vert][to] != 0) {
                if (distance[to] > distance[vert] + weightmatrix
                    [vert][to]) {
                    distance[to] = distance[vert] + weightmatrix
                        [vert][to];
                    q.push(to);
                    from[to] = vert;
                }
            }
            iter_BFS++;
        }
    }
    path = PrintPath(from, finish);
    return distance;
}
```

2.17 Функция BellmanFord

Вход: weightmatrix — матрица весов.

ver — количество вершин в графе.

start — индекс начальной вершины, от которой нужно найти кратчайшие пути.

finish — индекс конечной вершины, до которой нужно восстановить путь.

path — ссылка на вектор, в который будет записан найденный путь от начальной вершины до конечной.

iter_Bell — ссылка на переменную, в которой будет храниться количество итераций, выполненных в процессе работы алгоритма.

Выход: iter_Bell - количество итераций алгоритма, вектор dist, который содержит минимальные расстояния от начальной вершины до всех остальных.

vector<int> dist(ver, trash1); — инициализируем вектор dist, который будет хранить минимальное расстояние от начальной вершины до каждой из вершин. Изначально все расстояния равны trash1 (значение, которое обозначает, что вершина еще не посещена или не обновлялась).

vector<int> from(ver, -1); — вектор from используется для восстановления пути. Он хранит для каждой вершины индекс предыдущей вершины на кратчайшем пути.

dist[start] = 0; — начальная вершина имеет расстояние 0 от самой себя.

В каждом внешнем цикле for алгоритм обновляет расстояния до всех соседних вершин.

Для каждой пары рёбер (u, v) проверяется во внутреннем цикле, можно ли улучшить расстояние до вершины v, используя ребро (u, v). Если да, то обновляется значение dist[v] и записывается вершина u в вектор from[v].

Если за одну итерацию не было обновлений расстояний, это означает, что кратчайшие пути уже найдены, и можно прервать выполнение алгоритма досрочно (if (!updated) break;).

В каждой итерации увеличивается счётчик iter_Bell, чтобы отслеживать количество шагов, выполненных в процессе работы алгоритма.

После выполнения основного цикла алгоритма вызывается функция

PrintPath(from, finish), которая восстанавливает путь от начальной вершины до конечной.

Листинг 15. BellmanFord

```
vector<int> BellmanFord(vector<vector<int>>& weightmatrix, int
   ver, int start, int finish, vector<int>& path, int& iter_Bell
   ) {
    vector<int> dist(ver, trash1);
    vector<int> from(ver, -1);
    dist[start] = 0;
    iter_Bell = 0;
    for (int i = 0; i < ver - 1; ++i)
    {
        bool updated = false;
        for (int u = 0; u < ver; ++u) {
            for (int v = 0; v < ver; ++v) {
                if (weightmatrix[u][v] != 0) {
                    if (dist[v] > dist[u] + weightmatrix[u][v])
                       {
                        dist[v] = dist[u] + weightmatrix[u][v];
                        from[v] = u;
                        updated = true;
                    }
                }
                iter_Bell++;
            }
        }
        if (!updated) break;
    }

    path = PrintPath(from, finish);
    return dist;
}
```

2.18 Функция Ford_Fulkerson

Вход: bandwidth_matrix — матрица пропускных способностей.

start — индекс начальной вершины (источник потока).

finish — индекс конечной вершины (сток потока).

ver — количество вершин в графе.

Выход: максимальный поток в сети.

Функция Ford_Fulkerson реализует алгоритм Форда-Фалкерсона для нахождения максимального потока в сети на основе матрицы пропускных способностей.

while (min_cap = MinPropusk(bandwidth_matrix, start, finish, ver, parent)) — в цикле ищется путь увеличения с помощью функции MinPropusk. Если найден путь, то обновляется минимальная пропускная способность.

max_flow += min_cap; — увеличиваем общий поток на минимальную пропускную способность найденного пути.

Затем выводится промежуточная матрица пропускных способностей.

Восстанавливаем путь из конечной вершины finish к начальной вершине start через массив parent.

Обновляем матрицу пропускных способностей: уменьшаем пропускную способность для рёбер, по которым прошел поток, и увеличиваем пропускную способность для обратных рёбер.

Листинг 16. Ford_Fulkerson

```
int Ford_Fulkerson(vector<vector<int>> bandwidth_matrix, int start, int finish, int ver) {
    vector<int> parent(ver, -1);
    int max_flow = 0;
    int min_cap = 0;
    while (min_cap = MinPropusk(bandwidth_matrix, start, finish, ver, parent)) {
        max_flow += min_cap;
        cout << "Промежуточная матрица пропускных способностей:" << endl;
        for (int i = 0; i < ver; i++) {
            for (int j = 0; j < ver; j++) cout << bandwidth_matrix[i][j] << "   ";
            cout << endl;
        }
        cout << "Промежуточный путь с пропускной способностью: " << min_cap << endl;
        cout << "Маршрут: ";
        int v_path = finish;
        vector<int> path;
        while (v_path != start) {
            path.push_back(v_path);
            v_path = parent[v_path];
        }
        path.push_back(start);
        reverse(path.begin(), path.end());
        for (int p : path) cout << p << "   ";
        int v = finish;
```

```
        while (v != start) {
            int u = parent[v];
            bandwidth_matrix[u][v] -= min_cap;
            bandwidth_matrix[v][u] += min_cap;
            v = u;
        }
    }

    cout << "Промежуточная матрица пропускных способностей:" <<
       endl;
    for (int i = 0; i < ver; i++) {
        for (int j = 0; j < ver; j++) cout << bandwidth_matrix[i
           ][j] << "    ";
        cout << endl;
    }
    return max_flow;
}
```

2.19 Функция MinPropusk

Вход: bandwidth_matrix — матрица пропускных способностей.

start — индекс начальной вершины, с которой начинается поиск.

finish — индекс целевой вершины, до которой нужно найти минимальный пропуск.

ver — количество вершин в графе.

parent — вектор, который будет использоваться для отслеживания родителей вершин в процессе поиска.

Выход: минимальная пропускная способность пути между вершинами start и finish.

Вектор parent заполняется значениями -1, чтобы указать, что вершины еще не были посещены. Для начальной вершины start значение в parent устанавливается в -2.

Создается очередь q, в которую помещается пара: начальная вершина и максимально возможная пропускная способность.

Пока очередь не пуста, выполняется извлечение вершины u из очереди.

Для каждой вершины v, которая связана с вершиной u и еще не посещена устанавливается родитель для вершины v как вершина u. Рассчитывается минимальная пропускная способность между текущим путем и путем, проходящим через вершину u, через min(cap, bandwidth_matrix[u][v]).

Если вершина v является целевой вершиной (finish), то алгоритм сразу возвращает минимальную пропускную способность.

Если это не целевая вершина, то вершина v добавляется в очередь для дальнейшего обхода.

Если очередь пуста, то путь между начальной и целевой вершинами не был найден, и возвращается 0.

Листинг 17. MinPropusk

```
int MinPropusk(vector<vector<int>> bandwidth_matrix, int start, int finish, int ver, vector<int>& parent) {
    fill(parent.begin(), parent.end(), -1);
    parent[start] = -2;
    queue<pair<int, int>> q;
    q.push({ start, trash1 });
    while (!q.empty()) {
        int u = q.front().first;
        int cap = q.front().second;
        q.pop();
        for (int v = 0; v < ver; v++) {
        [v] != 0), и вершина v еще не была посещена (parent[v] == -1)
            if (u != v && bandwidth_matrix[u][v] != 0 && parent[v] == -1) {
                parent[v] = u;
                int min_cap = min(cap, bandwidth_matrix[u][v]);.
                if (v == finish) return min_cap;
                q.push({ v, min_cap });
            }
        }
    }
    return 0;
}
```

2.20 Функция MinCost

Вход: cost_matrix — матрица стоимостей.

bandwidth_matrix — матрица пропускных способностей.

s — индекс начальной вершины (источник потока).

t — индекс целевой вершины (сток).

ver — количество вершин в графе.

need_flow — величина потока.

Выход: величина стоимости потока.

Вектор flow_matrix используется для отслеживания потока на каждом ребре. Изначально все значения в нем равны 0. Матрица residual_cap — остаточные пропускные способности, которая в начале равна матрице пропускных способностей. Вектор dist хранит кратчайшие расстояния от начальной вершины до других вершин, инициализируется значением trash1 = INT_MAX.

Вектор parent отслеживает путь от вершины t до вершины s.

Пока общий поток меньше требуемого (flow < need_flow), выполняется следующее:

Матрица residual_cost обновляется, учитывая уже существующие потоки (для обратных ребер стоит отрицательная стоимость).

Для каждой вершины и ребра выполняется поиск кратчайшего пути с помощью алгоритма Беллмана-Форда. В процессе обновляются расстояния dist[v] и устанавливаются родители parent[v].

Если в целевую вершину t не удалось дойти, алгоритм завершает выполнение.

После нахождения кратчайшего пути, определяется минимальный поток (delta) по этому пути.

Обновляются потоки по всем ребрам пути, а также пропускные способности.

Стоимость потока добавляется в общий итог, а остаточные пропускные способности и стоимости соответствующих ребер обновляются.

Листинг 18. MinCost

```
int MinCost(vector<vector<int>>& cost_matrix,
    vector<vector<int>>& bandwidth_matrix,
    int s, int t,
    int ver,
    int need_flow) {
    int total_cost = 0;
    int flow = 0;
    vector<vector<int>> flow_matrix(ver, vector<int>(ver, 0));
    vector<vector<int>> residual_cap = bandwidth_matrix;
    while (flow < need_flow) {
        vector<vector<int>> residual_cost(ver, vector<int>(ver,
            0));
```

```
        for (int i = 0; i < ver; ++i) {
            for (int j = 0; j < ver; ++j) {
                residual_cost[i][j] = cost_matrix[i][j];
                if (flow_matrix[i][j] > 0) {
                    residual_cap[j][i] = flow_matrix[i][j];
                    residual_cost[j][i] = -cost_matrix[i][j];
                }
            }
        }
        vector<int> dist(ver, trash1);
        vector<int> parent(ver, -1);
        dist[s] = 0;
        for (int i = 0; i < ver - 1; ++i) {
            for (int u = 0; u < ver; ++u) {
                for (int v = 0; v < ver; ++v) {
                    if (residual_cap[u][v] > 0 && dist[u] !=
                        trash1 &&
                        dist[v] > dist[u] + residual_cost[u][v])
                            {
                        dist[v] = dist[u] + residual_cost[u][v];
                        parent[v] = u;
                    }
                }
            }
        }
        if (parent[t] == -1) break;
        int delta = need_flow - flow;
        for (int v = t; v != s; v = parent[v]) {
            delta = min(delta, residual_cap[parent[v]][v]);
        }
        for (int v = t; v != s; v = parent[v]) {
            int u = parent[v];
            flow_matrix[u][v] += delta;
            flow_matrix[v][u] -= delta;
            total_cost += delta * cost_matrix[u][v];
            residual_cap[u][v] -= delta;
            residual_cap[v][u] += delta;
        }
        flow += delta;
    }
    cout << "Распределение потока:" << endl;
    for (int i = 0; i < ver; ++i) {
        for (int j = 0; j < ver; ++j) {
            if (flow_matrix[i][j] > 0) {
                cout << i << "->" << j << ": " << flow_matrix[i
                    ][j]
                    << " (стоимость " << cost_matrix[i][j] << ")
                        " << endl;
```

```
            }
        }
    }
    return total_cost;
}
```

2.21 Функция Kruskal

Вход: weightmatrix — матрица весов.

ver — количество вершин в графе.

mst_weight — переменная для хранения общей стоимости минимального остовного дерева.

iter_kruskal — переменная для отслеживания количества итераций.

Выход: возвращается вектор рёбер, которые входят в минимальное остовное дерево, обновляются значения переменных mst_weight (общая стоимость минимального остовного дерева) и iter_kruskal (количество итераций).

Алгоритм Краскала предназначен для нахождения минимального остовного дерева (MST) графа с использованием жадного подхода.

Проходит по матрице весов рёбер и собирает все рёбра, у которых вес не равен 0.

Каждое ребро представлено как структура edge, содержащая вес (cost), вершину откуда (from) и вершину куда (where_).

Рёбра сортируются по возрастанию их веса с помощью функции compare.

Создаётся вектор parent, который будет использоваться для отслеживания корней вершин. В начале каждой вершине назначается свой собственный родитель (то есть, она является корнем).

Алгоритм обрабатывает рёбра по порядку их веса, начиная с минимального. Для каждого ребра выполняется проверка, образует ли оно цикл в текущем остовном дереве. Это делается с помощью функции find_root, которая находит корень вершины.

Если два рёбра соединяют разные компоненты, то оно добавляется в остовное дерево, и соединяем эти компоненты с помощью функции union_.

Листинг 19. Kruskal

```cpp
vector<edge> Kruskal(vector<vector<int>> weightmatrix, int ver, int& mst_weight, int& iter_kruskal) {
    vector<edge> edges;
    for (int i = 0; i < ver; i++) {
        for (int j = 0; j < ver; j++) {
            if (weightmatrix[i][j] != 0) {
                edges.push_back({ weightmatrix[i][j], i, j });
            }
        }
    }
    sort(edges.begin(), edges.end(), compare());
    cout << "\nОтсортированные рёбра: " << endl;
    for (const auto& e : edges) {
        cout << e.from << "-" << e.where_ << " Вес: " << e.cost << endl;
    }
    vector<int> parent(ver);
    for (int i = 0; i < ver; i++) {
        parent[i] = i;
    }
    vector<edge> mst;
    while (mst.size() != ver - 1) {
        edge next_edge = edges.back();
        edges.pop_back();
        int f_p = find_root(next_edge.from, parent);
        int s_p = find_root(next_edge.where_, parent);
        if (f_p != s_p) {
            mst.push_back(next_edge);
            mst_weight += next_edge.cost;
            union_(f_p, s_p, parent);
        }

        iter_kruskal++;
    }
    return mst;
}
```

2.22 codePrufer

Вход: mst — вектор рёбер минимального остовного дерева.

ver — количество вершин в графе.

Выход: код Прюфера (вектор пар, где каждая пара представляет вершину и соседнюю с ней вершину и веса).

Сначала создаётся матрица смежности mst_matrix, где для каждого ребра в mst обновляется значение в ячейках матрицы (для обоих направлений, так как граф неориентированный).

Степень каждой вершины считается как количество рёбер, инцидентных этой вершине. Для этого обходят все элементы матрицы смежности.

Пока не останется только две вершины, продолжается процесс удаления рёбер, соединяющих вершины с минимальной степенью.

Для каждой вершины, степень которой равна 1, выбирается её сосед (в графе будет только одно такое ребро). В код Прюфера добавляется индекс соседа этой вершины. Уменьшаются степени обеих вершин и удаляется ребро.

Когда степени всех вершин, кроме двух, будут равны 1, процесс прекращается. Оставшиеся две вершины не включаются в код Прюфера, так как они являются последним ребром в дереве.

Листинг 20. codePrufer

```
vector<pair<int, int>> codePrufer(vector<edge> mst, int ver) {
    vector<vector<int>> mst_matrix(ver, vector<int>(ver, 0));
    for (int k = 0; k < mst.size(); k++) {
        int i = mst[k].from;
        int j = mst[k].where_;
        int w = mst[k].cost;
        mst_matrix[i][j] = w;
        mst_matrix[j][i] = w;
    }

    vector<pair<int, int>> prufer;
    vector<int> degree_vert(ver);
    for (int i = 0; i < ver; i++) {
        for (int j = 0; j < ver; j++) {
            if (mst_matrix[i][j] != 0) {
                degree_vert[i]++;
            }
        }
    }

    int r = 0;
    for (int i = 0; i < ver; i++) {
        r += degree_vert[i];
    }
    while (r != 0) {
        for (int i = 0; i < ver; i++) {
            if (degree_vert[i] == 1) {
                for (int j = 0; j < ver; j++) {
```

```
                    if (mst_matrix[i][j] != 0) {
                        prufer.push_back(make_pair(mst_matrix[i][j], j));
                        degree_vert[i]--;
                        degree_vert[j]--;
                        mst_matrix[i][j] = 0;
                        mst_matrix[j][i] = 0;
                        r -= 2;
                    }
                }
            }
        }
    }

    return prufer;
}
```

2.23 decodePrufer

Вход: prufer — последовательность пар, представляющих код Прюфера для дерева с весами.

Выход: матрица смежности восстановленного дерева и веса.

Сначала создается матрица смежности ans_matrix. Эта матрица будет хранить веса рёбер. Создается вектор vertexes, который отслеживает степень каждой вершины. В начале все вершины имеют степень 0. Степень вершины увеличивается на 1 для каждой вершины, которая появляется в коде Прюфера.

Для каждой вершины, которая имеет степень 0, выбирается минимальная вершина (если степень вершины ещё не была уменьшена до 0). Восстанавливается соответствующее ребро, и степени обеих вершин обновляются. Алгоритм заканчивается, когда все рёбра восстановлены.

Когда восстанавливаются все рёбра, остальная единственная пара вершин, которые ещё не были соединены, образует последний ребро.

Листинг 21. decodePrufer

```
vector<vector<int>> decodePrufer(vector<pair<int, int>> prufer) {
    int ver = prufer.size() + 1;
    vector<vector<int>> ans_matrix(ver, vector<int>(ver, 0));
    vector<int> vertexes(ver, 0);
    for (int i = 0; i < prufer.size(); i++) vertexes[prufer[i].second] += 1;
```

```

    int j = 0;
    int num = 0;
    for (int i = 0; i < ver - 1; i++) {
        for (j = 0; j < ver; j++) {
            if (i == ver - 2) {
                if (vertexes[j] == 0) {
                    vertexes[j] = -1;
                    ans_matrix[j][prufer[i].second] = prufer[i].first;
                    ans_matrix[prufer[i].second][j] = prufer[i].first;
                    vertexes[prufer[i].second]--;
                    num = j;
                    break;
                }
            }
            else {
                if (vertexes[j] == 0 && num <= j) {
                    vertexes[j] = -1;
                    ans_matrix[j][prufer[i].second] = prufer[i].first;
                    ans_matrix[prufer[i].second][j] = prufer[i].first;
                    vertexes[prufer[i].second]--;
                    num = j;
                    break;
                }
            }
        }
    }
    return ans_matrix;
}
```

2.24 printAllEdges

Вход: matrix — матрица смежности графа.

ver — количество вершин в графе.

Выход: вывод в консоль ребер.

Внешний цикл проходит по всем вершинам графа. Внутренний цикл перебирает вершины, которые идут после текущей вершины в списке (чтобы не выводить рёбра дважды, например, для рёбер u <-> v и v <-> u). Если вес ребра между вершинами u и v не равен 0, выводится соответствующее сообщение.

Листинг 22. printAllEdges

```
void printAllEdges(const vector<vector<int>>& matrix, int ver) {
    cout << "Все рёбра графа:" << endl;
    for (int u = 0; u < ver; ++u) {
        for (int v = u + 1; v < ver; ++v) {
            if (matrix[u][v] != 0) {
                cout << u << " <-> " << v << " (Вес: " << matrix
                    [u][v] << ")" << endl;
            }
        }
    }
}
```

2.25 MaxIndependentEdgeSet

Вход: matrix — матрица смежности графа.

ver — количество вершин в графе.

Выход: пары вершин.

Используется массив used, который отслеживает, использованы ли вершины в рёбрах, включённых в независимое множество. Внешний цикл проходит по всем вершинам. Если вершина ещё не использована (!used[u]), внутренний цикл ищет вершину v, с которой существует ребро, и обе вершины (u и v) ещё не использованы. Как только найдено такое ребро, оно добавляется в список рёбер, и обе вершины помечаются как использованные.

Листинг 23. MaxIndependentEdgeSet

```
vector<pair<int, int>> MaxIndependentEdgeSet(vector<vector<int
    >>& matrix, int ver) {
    vector<pair<int, int>> matching;
    vector<bool> used(ver, false);
    for (int u = 0; u < ver; ++u) {
        if (!used[u]) {
            for (int v = 0; v < ver; ++v) {
                if (matrix[u][v] != 0 && !used[v]) {
                    matching.push_back({ u, v });
                    used[u] = true;
                    used[v] = true;
                    break;
                }
            }
```

```
        }
    }

    return matching;
}
```

2.26 Euler

Вход: ver — количество вершин в графе.

undirected_weight_matrix — матрица смежности графа.

undirected_degrees — вектор степеней вершин графа.

Выход: вывод в консоль, новая матрица весов эйлерова графа.

Функция проверяет, является ли граф Эйлеровым, то есть если все вершины имеют чётную степень. Если граф не является Эйлеровым, попытаться добавить или удалить рёбра таким образом, чтобы все вершины имели чётную степень, что обеспечит существование Эйлерова цикла.

Вначале выводится матрица весов рёбер и степени каждой вершины. Алгоритм проверяет, имеет ли граф Эйлеров цикл, то есть, если все вершины имеют чётную степень.

Если граф не является Эйлеровым, алгоритм пытается добавить рёбра между вершинами с нечётными степенями, пока все степени не станут чётными. Если не удаётся добавить ребро, алгоритм удаляет существующие рёбра, чтобы сбалансировать степени вершин.

По завершении модификации графа, выводится изменённая матрица весов рёбер.

Листинг 24. Euler

```
vector<vector<int>> Euler(int ver, vector<vector<int>>
    undirected_weight_matrix, vector<int> undirected_degrees) {
    vector<vector<int>> eulergraph(undirected_weight_matrix);
    int countevenvert = 0;
    bool flagEuler = false;
    cout << "\nМатрица весов: " << endl;
    PrintMatrix(eulergraph);
    for (int i = 0; i < ver; i++) cout << "Степень вершины "
        << i << " = " << undirected_degrees[i] << endl;
    for (int i = 0; i < ver; i++) {
        if (undirected_degrees[i] % 2) countevenvert++;
    }
    if (countevenvert == ver) {
```

```
        cout << "Граф является Эйлеровым\n" << endl;
        flagEuler = true;
    }
    else {
        cout << "\nГраф не является Эйлеровым. Модифицируем граф.\n" << endl;
        while (!flagEuler) {
            bool flag = 0;
            for (int i = 0; i < ver; i++) {
                for (int j = 0; j < ver; j++) {
                    if (undirected_degrees[i] % 2 != 0 and undirected_degrees[j] % 2 != 0 and eulergraph[i][j] == 0 and i != j) {
                        flag = true;
                        int r = rand() % (10 - (1) + 1) + 1;
                        eulergraph[i][j] = r;
                        eulergraph[j][i] = r;
                        undirected_degrees[i]++;
                        undirected_degrees[j]++;
                        cout << "Добавили ребро: " << i << "<->" << j << " с весом: " << r << endl;
                        break;
                    }
                }
                if (flag) break;
            }
            if (!flag) {
                for (int i = 0; i < ver; i++) {
                    for (int j = 0; j < ver; j++) {
                        if (undirected_degrees[i] % 2 != 0 and undirected_degrees[j] % 2 != 0 and eulergraph[i][j] != 0 and i != j) {
                            flag = true;
                            eulergraph[i][j] = 0;
                            eulergraph[j][i] = 0;
                            undirected_degrees[i]--;
                            undirected_degrees[j]--;
                            cout << "Удалили ребро: " << i << "<->" << j << endl;
                            break;
                        }
                    }
                    if (flag) break;
                }
            }
            int countevenvert1 = 0;
            for (int i = 0; i < ver; i++) {
                if (undirected_degrees[i] % 2 == 0)
```

```
                        countevenvert1++;
                }
                if (countevenvert1 == ver) flagEuler = true;
            }
            cout << "**********************" << endl;
            cout << "Модифицированный эйлеров граф." << endl;
            cout << "Матрица весов: " << endl;
            PrintMatrix(eulergraph);
            for (int i = 0; i < ver; i++) cout << "Степень вершины
                 " << i << " = " << undirected_degrees[i] << endl;
            cout << "**********************" << endl;
    }
    return eulergraph;
}
```

2.27 FindEuler

Вход: ver — количество вершин в графе.

eulergraph — матрица смежности эйлерова графа.

Выход: эйлеров цикл.

Находится начальная вершина, которая имеет хотя бы одно ребро. Это делается через перебор всех вершин, пока не найдется вершина с хотя бы одним исходящим рёбром. Используется стек для хранения текущих вершин, которые нужно посетить, и вектор для хранения самого цикла.

Пока в стеке есть вершины:

Берется вершина из стека (это текущая вершина, с которой будем работать). Ищется соседняя вершина, с которой есть ребро. Если такая вершина найдена, она добавляется в стек, а ребро между текущей и соседней вершинами удаляется из графа (путем зануления элементов матрицы смежности).

Если у текущей вершины больше нет рёбер, она добавляется в цикл и удаляется из стека.

Листинг 25. FindEuler

```
void FindEuler(int ver, vector<vector<int>> eulergraph) {
    stack<int> vertexes;
    vector<int> cycle;
    int vert = 0;
    for (vert = 0; vert < ver; vert++) {
```

```
        for (int j = 0; j < ver; j++) {
            if (eulergraph[vert][j] != 0) {
                break;
            }
        }
        if (vert < ver) {
            break;
        }
    }
    vertexes.push(vert);
    while (!vertexes.empty()) {
        vert = vertexes.top();
        int i;
        for (i = 0; i < ver; i++) {
            if (eulergraph[vert][i] != 0) {
                break;
            }
        }
        if (i == ver) {
            cycle.push_back(vert);
            vertexes.pop();
        }
        else {
            vertexes.push(i);
            eulergraph[vert][i] = 0;
            eulergraph[i][vert] = 0;
        }
    }
    cout << "\nЭйлеров цикл:" << endl;
    for (int i = 0; i < cycle.size(); i++) {
        cout << cycle[i];
        if (i != cycle.size() - 1) {
            cout << " -> ";
        }
    }
}
```

2.28 CanAddV

Вход: v - текущая вершина, которую пытаемся добавить в путь.

gamiltongraph - матрица смежности графа.

path - массив текущего пути, в котором хранятся вершины, уже добавленные в путь.

p - индекс вершины, которую проверяют для добавления в путь.

Выход: true, если вершину можно добавить в путь.

false, если вершину нельзя добавить в путь.

Функция CanAddV проверяет, можно ли добавить вершину v в текущий путь в задаче о гамильтоновом пути.

Если между последней вершиной в пути и вершиной v нет ребра (gamiltongraph[path[p - 1]][v] == 0), то возвращается false.

Далее проверяется, встречалась ли вершина v ранее в пути. Если она уже присутствует в массиве path, то возвращается false.

Если оба условия не нарушаются, возвращается true.

Листинг 26. CanAddV

```
bool CanAddV(int v, vector<vector<int>>& gamiltongraph, vector<int>& path, int p) {
    if (gamiltongraph[path[p - 1]][v] == 0) return false;
    for (int i = 0; i < p; i++) {
        if (path[i] == v) return false;
    }
    return true;
}
```

2.29 Gamilton

Вход: ver — количество вершин в графе.

undirected_weight_matrix — матрица весов для неориентированного графа.

Выход: модифицированная матрица весов (gamiltongraph).

Копируется исходная матрица весов в переменную gamiltongraph. Выводится исходная матрица весов.

Инициализируется массив path размером ver, где -1 означает, что вершина еще не посещена. Начальная вершина (первый элемент) ставится равной 0.

Функция PrintAllHamiltonCyclesWithWeights пытается найти все возможные гамильтоновы циклы в графе, начиная с вершины 0, и выводит найденные циклы. Если хотя бы один цикл найден, то граф считается гамильтоновым, и информация об этом выводится. В таком случае возвращается текущий граф.

Создается случайная перестановка вершин (массив vect), чтобы изменить порядок вершин. Для каждой пары соседних вершин в перестановке проверяется наличие ребра. Если ребра нет, оно добавляется с случайным весом от 1 до 10. После модификации графа снова проверяется наличие гамильтонова цикла.

Если после модификации гамильтонов цикл найден, выводится соответствующее сообщение, и граф считается гамильтоновым.

Листинг 27. Gamilton

```
vector<vector<int>> Gamilton(int ver, vector<vector<int>> undirected_weight_matrix) {
    vector<vector<int>> gamiltongraph(undirected_weight_matrix);
    cout << "\nМатрица весов: " << endl;
    PrintMatrix(gamiltongraph);
    vector<int> path(ver, -1);
    path[0] = 0;
    int cycle_count = 0;
    cout << "\nВывод найденных гамильтоновых циклов." << endl;
    PrintAllHamiltonCyclesWithWeights(gamiltongraph, path, 1, cycle_count, undirected_weight_matrix);
    if (cycle_count > 0) {
        cout << "**********************" << endl;
        cout << "\nГраф является гамильтоновым. Найдено циклов: " << cycle_count << endl;
        cout << "**********************" << endl;
        return gamiltongraph;
    }
    else {
        cout << "\nГраф не является гамильтоновым. Модифицируем граф." << endl;
        vector<int> vect(ver, -1);
        for (int i = 0; i < ver; i++) {
            int r = rand() % ver;
            while (find(vect.begin(), vect.end(), r) != vect.end()) {
                r = rand() % ver;
            }
            vect[i] = r;
        }
        for (int i = 0; i < ver; i++) {
            if (gamiltongraph[vect[i]][vect[(i + 1) % ver]] == 0) {
                int r = rand() % 10 + 1;
                gamiltongraph[vect[i]][vect[(i + 1) % ver]] = r;
                gamiltongraph[vect[(i + 1) % ver]][vect[i]] = r;
                cout << "Добавлено ребро: " << vect[i] << " <-> " << vect[(i + 1) % ver]
```

```
                    << " с весом: " << r << endl;
            }
        }
        cycle_count = 0;
        fill(path.begin(), path.end(), -1);
        path[0] = 0;
        cout << "\nПроверка после модификации." << endl;
        PrintAllHamiltonCyclesWithWeights(gamiltongraph, path, 1, cycle_count, undirected_weight_matrix);
        if (cycle_count > 0) {
            cout << "**********************" << endl;
            cout << "\nТеперь граф гамильтонов. Найдено циклов: " << cycle_count << endl;
            cout << "**********************" << endl;
        }
    }
    cout << "\nМатрица весов:" << endl;
    PrintMatrix(gamiltongraph);
    return gamiltongraph;
}
```

2.30 PrintAllHamiltonCyclesWithWeights

Вход: gamiltongraph — матрица смежности графа.

path — вектор, представляющий текущий путь.

position — текущая позиция в пути, на которой мы ищем следующую вершину для добавления.

cycle_count — счётчик количества найденных гамильтоновых циклов.

weight_matrix — матрица весов графа.

Выход: найденный цикл.

Если текущая позиция в пути равна количеству вершин (position == ver), это значит, что мы заполнили путь. Далее проверяется, существует ли ребро между последней и первой вершинами (gamiltongraph[path[position - 1]][path[0]] != 0). Если оно существует, то найден гамильтонов цикл. Суммируются веса всех рёбер между соседними вершинами в пути, а также добавляется вес ребра между последней и первой вершинами.

Выводится сам цикл с перечислением всех вершин, а также суммарный вес цикла. Для каждой вершины v в графе проверяется, можно ли добавить её в текущий путь с помощью функции CanAddV(v, gamiltongraph, path, position). Если добавление возможно, то:

Вершина v добавляется в путь на текущую позицию position. Рекурсивно вызывается функция для следующей позиции в пути (position + 1). После рекурсивного вызова текущая вершина снова сбрасывается в -1, чтобы попытаться добавить другие вершины.

Когда весь путь построен и цикл найден, функция возвращает управление назад, продолжая искать другие возможные циклы в графе.

Листинг 28. PrintAllHamiltonCyclesWithWeights

```
void PrintAllHamiltonCyclesWithWeights(vector<vector<int>>&
   gamiltongraph, vector<int>& path, int position, int&
   cycle_count, vector<vector<int>>& weight_matrix) {
    int ver = gamiltongraph.size();
    if (position == ver) {
        if (gamiltongraph[path[position - 1]][path[0]] != 0) {
            int total_weight = 0;
            for (int i = 0; i < ver - 1; i++) {
                total_weight += weight_matrix[path[i]][path[i +
                   1]];
            }
            total_weight += weight_matrix[path[ver - 1]][path
               [0]];
            cout << "Цикл " << ++cycle_count << ": ";
            for (int i = 0; i < ver; i++) {
                cout << path[i] << (i < ver - 1 ? " -> " : "");
            }
            cout << " -> " << path[0] << " (Суммарный вес: " <<
               total_weight << ")" << endl;
        }
        return;
    }
    for (int v = 0; v < ver; v++) {
        if (CanAddV(v, gamiltongraph, path, position)) {
            path[position] = v;
            PrintAllHamiltonCyclesWithWeights(gamiltongraph,
               path, position + 1, cycle_count, weight_matrix);
            path[position] = -1;
        }
    }
}
```

2.31 FindGamilton

Вход: gamiltongraph — матрица смежности графа.

path — вектор, представляющий текущий путь.

p — текущая позиция в пути, на которой мы ищем следующую вершину для добавления.

Выход: true: если найден гамильтонов цикл.

false: если гамильтонов цикл не найден.

Если позиция в пути равна количеству вершин (p == gamiltongraph.size()), это значит, что путь заполнился. Далее проверяется, существует ли ребро между последней вершиной пути и первой (gamiltongraph[path[p - 1]][path[0]] != 0). Если оно существует, то найден гамильтонов цикл. Возвращается true.

Для каждой вершины v (начиная с вершины 1, чтобы не начинать путь с самой первой вершины), проверяется, можно ли добавить её в текущий путь с помощью функции CanAddV(v, gamiltongraph, path, p). Эта функция проверяет, существует ли ребро между последней вершиной пути и вершиной v, а также не была ли вершина v уже посещена в текущем пути. Вершина v добавляется на текущую позицию пути path[p] = v. Рекурсивно вызывается функция для следующей позиции в пути (p + 1).

Если рекурсивный вызов возвращает true, это означает, что найден гамильтонов цикл, и функция возвращает true. Если путь не ведет к решению, то текущая вершина сбрасывается (path[p] = -1), и пробуется следующая вершина.

Если на текущем шаге не удалось найти подходящий путь, функция возвращает false, что означает, что гамильтонов цикл для данного графа не найден.

Листинг 29. FindGamilton

```
bool FindGamilton(vector<vector<int>>& gamiltongraph, vector<int>& path, int p) {
    if (p == gamiltongraph.size()) return gamiltongraph[path[p - 1]][path[0]] != 0;
    for (int v = 1; v < gamiltongraph.size(); v++) {
        if (CanAddV(v, gamiltongraph, path, p)) {
            path[p] = v;
            if (FindGamilton(gamiltongraph, path, p + 1)) return true;
            path[p] = -1;
        }
    }
    return false;
```

```
}
```

2.32 ExistGamiltonCycle

Вход: gamiltongraph — матрица смежности графа .

Выход: true: если гамильтонов цикл существует в графе.

false: если гамильтонов цикл не существует.

Создается вектор path, который будет хранить текущий путь для поиска гамильтонова цикла. Он инициализируется значениями -1 (незанятыми вершинами), за исключением первой вершины, которая устанавливается в 0 (path[0] = 0).

Функция вызывает FindGamilton, которая пытается найти гамильтонов цикл с текущим состоянием пути. Вектор path передается в функцию FindGamilton с начальной позицией 1 (так как вершина 0 уже добавлена в путь). Если FindGamilton возвращает true, это означает, что гамильтонов цикл был найден.

Листинг 30. ExistGamiltonCycle

```
bool ExistGamiltonCycle(vector<vector<int>>& gamiltongraph) {
    vector<int> path(gamiltongraph.size(), -1);
    path[0] = 0;
    if (!FindGamilton(gamiltongraph, path, 1)) return false;
    return true;
}
```

2.33 Komivoyadger

Вход: gamiltongraph — матрица смежности графа.

ver — количество вершин в графе.

Выход: путь коммивояжера, минимальный вес пути.

Создается вектор visited, чтобы отслеживать, какие вершины были посещены. path — вектор для хранения текущего пути.

min_cost — минимальный вес пути.

min_path — вектор для хранения пути с минимальным весом.

Вызывается функция FindGamiltonKomivvv, которая рекурсивно перебирает все возможные гамильтоновы циклы, считая их веса. Функция

FindGamiltonKomivvv использует жадный метод для поиска цикла, минимизируя суммарный вес пути, обходя все вершины.

После выполнения рекурсивного поиска, выводится минимальный найденный цикл (путь) и его вес.

Листинг 31. Komivoyadger

```
void Komivoyadger(vector<vector<int>>& gamiltongraph, int ver) {
    vector<bool> visited(ver);
    vector<int> path;
    visited[0] = true;
    int min_cost = trash1;
    vector<int> min_path;
    cout << "**********************" << endl;
    cout << "Все возможные гамильтоновы циклы:" << endl;

    FindGamiltonKomivvv(gamiltongraph, visited, path, 0, ver, 1,
        0, min_cost, min_path);

    cout << "**********************" << endl;
    cout << "Цикл коммивояжера: " << endl;
    for (int i = 0; i < min_path.size(); i++) cout << min_path[i
        ] << " -> ";
    cout << min_path[0];
    cout << "\nВес: " << min_cost << endl;
}
```

2.34 FindGamiltonKomivvv

Вход: gamiltongraph — матрица смежности графа.

visited — вектор логических значений, который отслеживает, были ли посещены вершины.

path — вектор для хранения текущего пути.

vert — текущая вершина, которая рассматривается в данный момент.

ver — общее количество вершин в графе.

count — количество вершин, которые уже включены в текущий путь.

cost — текущая стоимость пути.

min_cost — минимальная стоимость пути, которая обновляется при нахождении лучшего пути.

min_path — путь, который соответствует минимальной стоимости.

Выход: min_path — минимальный гамильтонов цикл.

min_cost — минимальный вес этого цикла.

Вершина vert добавляется в путь path.

Когда количество вершин в пути (count) равно количеству вершин в графе (ver), проверяется, можно ли вернуться в исходную вершину (первую вершину). Если это возможно (вес ребра между текущей вершиной и первой вершиной не равен 0), то найден полный цикл. Выводится путь, его вес, и если найденный путь имеет меньший вес, чем текущий минимальный (min_cost), то обновляются min_cost и min_path.

Для каждой вершины графа проверяется, была ли она посещена (если не была, то она доступна для добавления в путь). Если вершина не посещена и существует ребро между текущей вершиной и рассматриваемой вершиной, то эта вершина добавляется в путь, и для нее рекурсивно вызывается FindGamiltonKomivvv.

После того как все возможные пути с текущей вершиной были исследованы, она удаляется из пути с помощью path.pop_back().

Листинг 32. FindGamiltonKomivvv

```
void FindGamiltonKomivvv(vector<vector<int>>& gamiltongraph, vector<bool>& visited, vector<int>& path, int vert, int ver, int count, int cost, int& min_cost, vector<int>& min_path) {
    path.push_back(vert);
    if (count == ver and gamiltongraph[vert][0]) {
        for (int i = 0; i < path.size(); i++) cout << path[i] << " -> ";
        cout << path[0];
        cout << "\tВес цикла: " << gamiltongraph[vert][0] + cost << endl;
        if (cost + gamiltongraph[vert][0] < min_cost) {
            min_cost = cost + gamiltongraph[vert][0];
            min_path = path;
        }
        path.pop_back();
        return;
    }
    for (int i = 0; i < ver; i++) {
        if (!visited[i] and gamiltongraph[vert][i] != 0) {
            visited[i] = true;
            FindGamiltonKomivvv(gamiltongraph, visited, path, i, ver, count + 1, cost + gamiltongraph[vert][i], min_cost, min_path);
            visited[i] = false;
        }
    }
    path.pop_back();
```

```
}
```

3 Результаты работы программы

При запуске программы мы видим стартовое окно, предлагающее ввести количество вершин (см. [Рис. 5]).

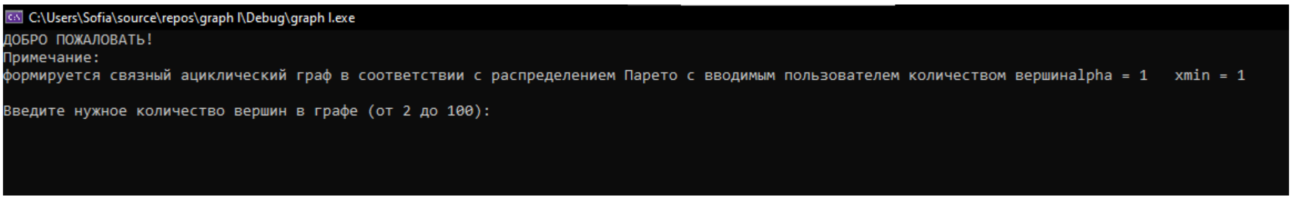


Рис. 5. Стартовое окно

Предусмотрена защита от некорректного пользовательского ввода (см. [Рис. 6]).

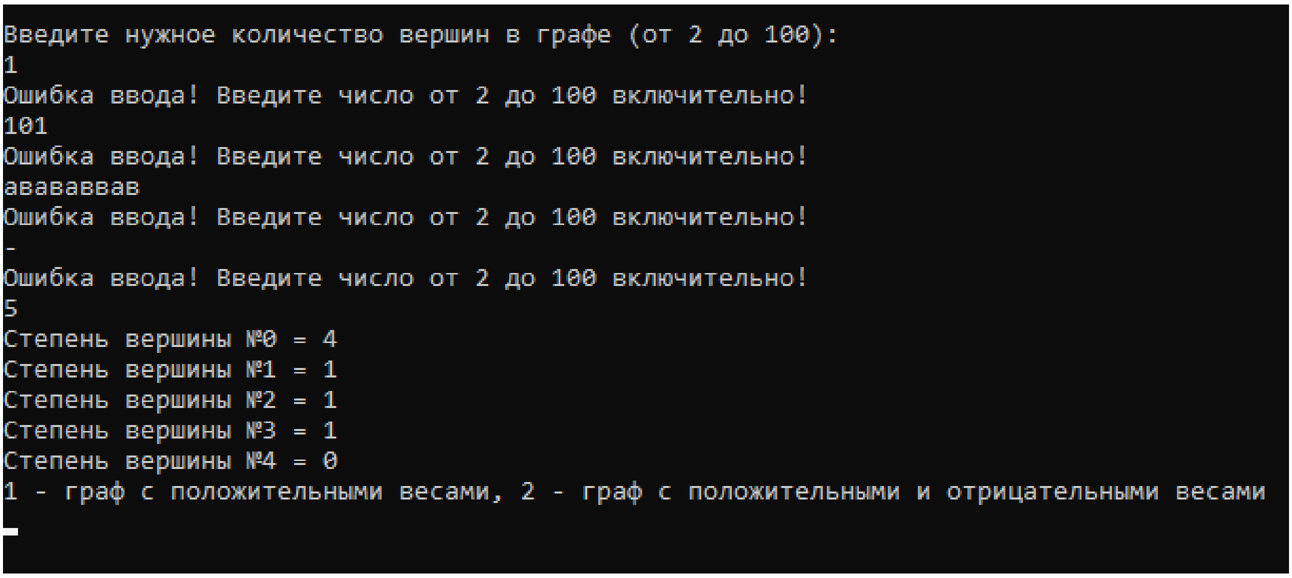
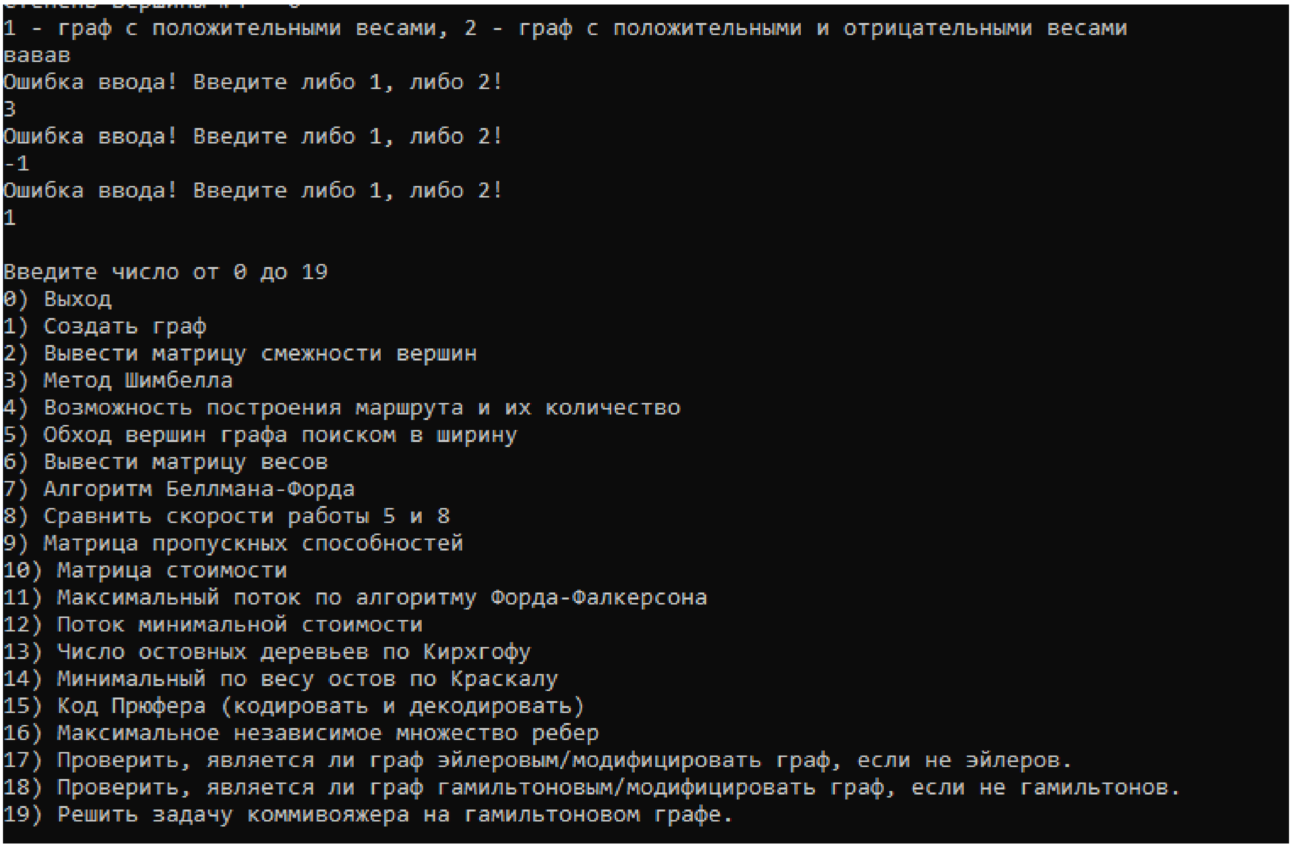


Рис. 6. Ввод количества вершин

После корректного ввода программа предлагает выбрать один из вариантов: граф с положительными весами или граф с положительными и отрицательными весами. Так же предусмотрена защита от некорректного ввода (см. [Рис. 7]).



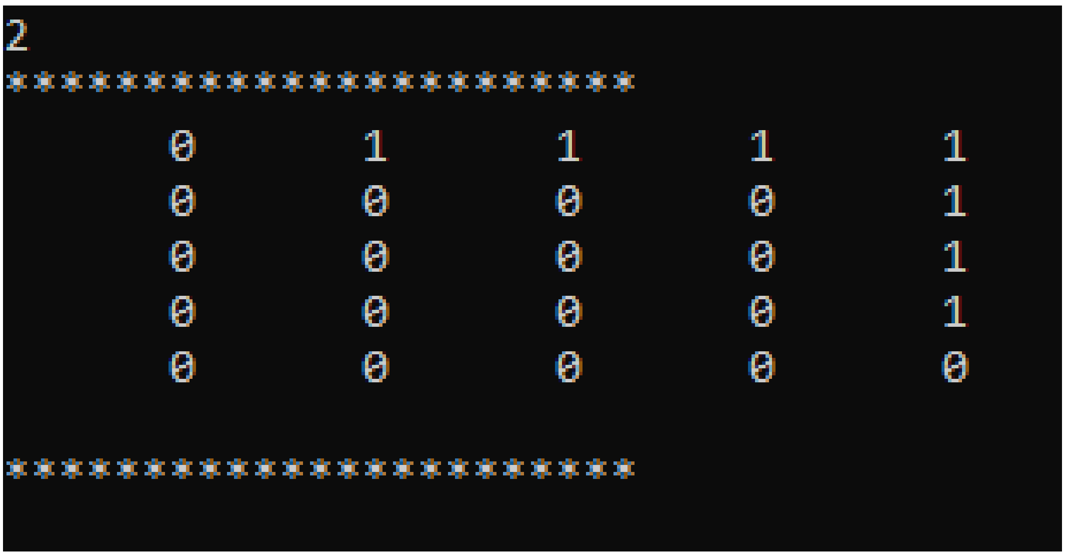
```
1 - граф с положительными весами, 2 - граф с положительными и отрицательными весами
вавав
Ошибка ввода! Введите либо 1, либо 2!
3
Ошибка ввода! Введите либо 1, либо 2!
-1
Ошибка ввода! Введите либо 1, либо 2!
1

Введите число от 0 до 19
0) Выход
1) Создать граф
2) Вывести матрицу смежности вершин
3) Метод Шимбелла
4) Возможность построения маршрута и их количество
5) Обход вершин графа поиском в ширину
6) Вывести матрицу весов
7) Алгоритм Беллмана-Форда
8) Сравнить скорости работы 5 и 8
9) Матрица пропускных способностей
10) Матрица стоимости
11) Максимальный поток по алгоритму Форда-Фалкерсона
12) Поток минимальной стоимости
13) Число остовных деревьев по Кирхгофу
14) Минимальный по весу остов по Краскалу
15) Код Прюфера (кодировать и декодировать)
16) Максимальное независимое множество ребер
17) Проверить, является ли граф эйлеровым/модифицировать граф, если не эйлеров.
18) Проверить, является ли граф гамильтоновым/модифицировать граф, если не гамильтонов.
19) Решить задачу коммивояжера на гамильтоновом графе.
```

Рис. 7. Выбор способа задания весов

После ввода, генерируются степени вершин, создается граф, пользователю демонстрируется меню.

При вводе 2 - вывести матрицу смежности вершин пользователю выводится матрица (см. [Рис. 8]).



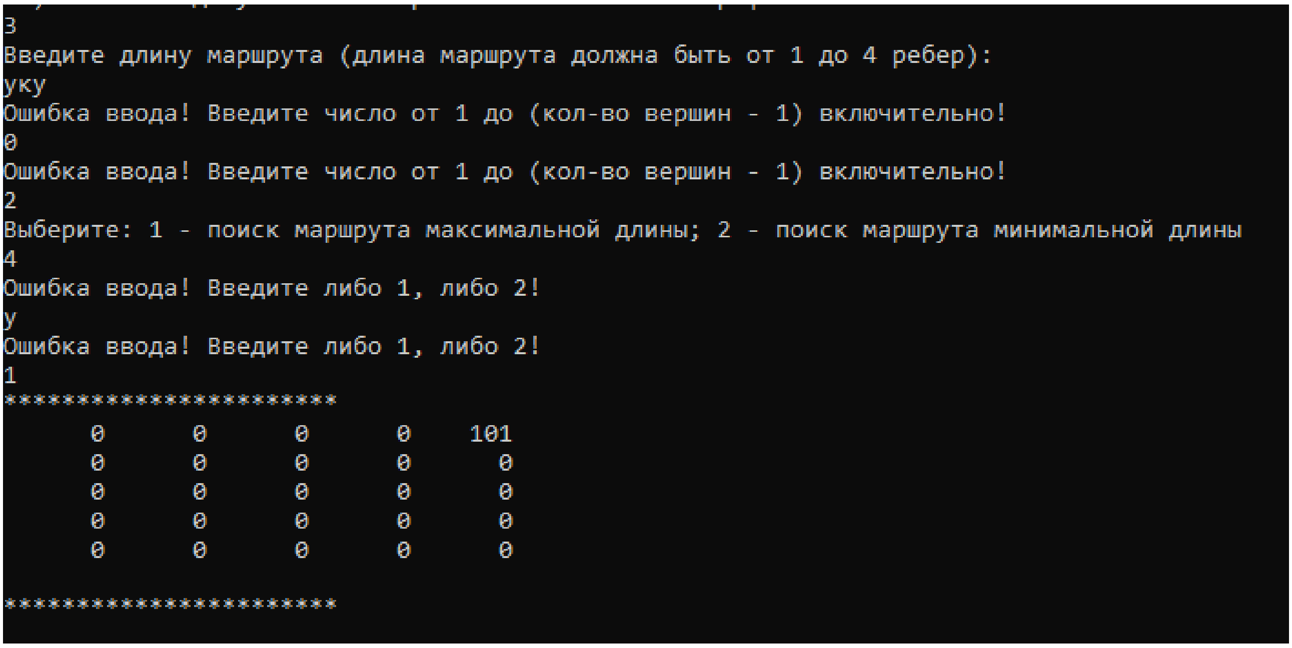
```
2
***********************
      0      1      1      1      1
      0      0      0      0      1
      0      0      0      0      1
      0      0      0      0      1
      0      0      0      0      0

***********************
```

Рис. 8. Функция 2 - матрица смежности вершин

А затем снова выводится меню.

При вводе 3 - метод Шимбела, пользователю нужно ввести длину маршрута, а затем выбрать интересует его маршрут максимальной длины или минимальной. Только тогда он получит матрицу (см. [Рис. 9]).

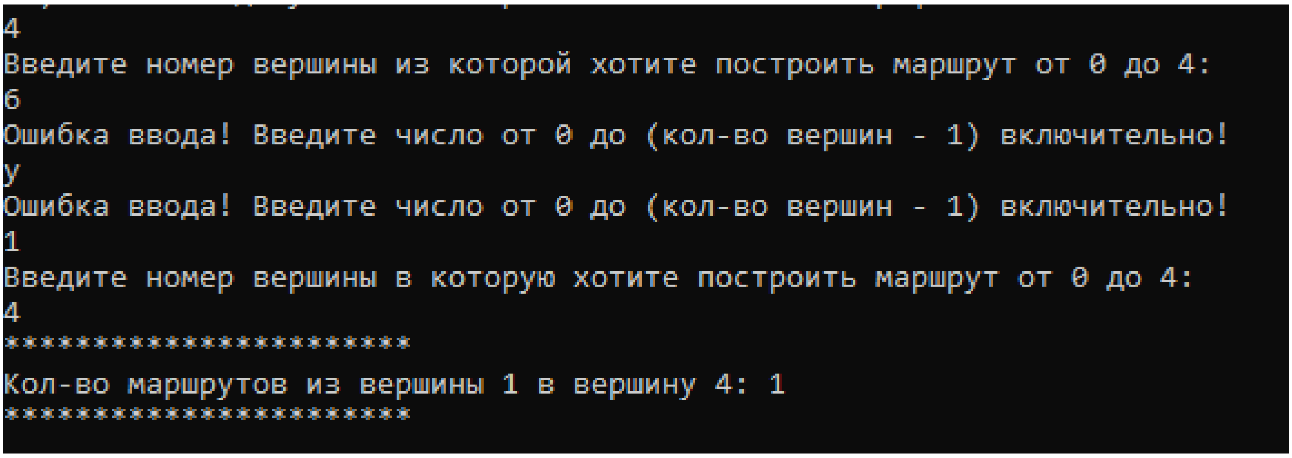


```
3
Введите длину маршрута (длина маршрута должна быть от 1 до 4 ребер):
уку
Ошибка ввода! Введите число от 1 до (кол-во вершин - 1) включительно!
0
Ошибка ввода! Введите число от 1 до (кол-во вершин - 1) включительно!
2
Выберите: 1 - поиск маршрута максимальной длины; 2 - поиск маршрута минимальной длины
4
Ошибка ввода! Введите либо 1, либо 2!
у
Ошибка ввода! Введите либо 1, либо 2!
1
***********************
      0      0      0      0    101
      0      0      0      0      0
      0      0      0      0      0
      0      0      0      0      0
      0      0      0      0      0

***********************
```

Рис. 9. Функция 3 - метод Шимбела

При вводе 4 - определяется возможность построения маршрута и их количество. Пользователь вводит откуда и куда ему нужно проверить маршрут (см. [Рис. 10]).



```
4
Введите номер вершины из которой хотите построить маршрут от 0 до 4:
6
Ошибка ввода! Введите число от 0 до (кол-во вершин - 1) включительно!
у
Ошибка ввода! Введите число от 0 до (кол-во вершин - 1) включительно!
1
Введите номер вершины в которую хотите построить маршрут от 0 до 4:
4
***********************
Кол-во маршрутов из вершины 1 в вершину 4: 1
***********************
```

Рис. 10. Функция 4 - возможность построения маршрута и их количество

Если такого маршрута нет, то вывод будет следующий (см. [Рис. 11]).

```
4
Введите номер вершины в которую хотите построить маршрут от 0 до 4:
1
************************
Такой маршрут построить нельзя
************************
```

Рис. 11. Функция 4 - возможность построения маршрута и их количество

Функция обход вершин графа поиском в ширину предлагает пользователю ввести откуда он хочет начать (см. [Рис. 12]).

```
5
Введите номер вершины из которой хотите построить маршрут от 0 до 4:
8
Ошибка ввода! Введите число от 0 до (кол-во вершин - 1) включительно!
п
Ошибка ввода! Введите число от 0 до (кол-во вершин - 1) включительно!
2
************************
Вектор расстояний:
-       -       0       -       4
Длина маршрута: 4.
Маршрут: 2->4.
************************
```

Рис. 12. Функция 5 - обход вершин графа поиском в ширину

В данном случе выводится маршрут с минимальной длиной.

Если ввести номер последней вершины, то вывод будет следующим (см. [Рис. 13]).

```
5
Введите номер вершины из которой хотите построить маршрут от 0 до 4:
4
************************
Вектор расстояний:
-       -       -       -       0
Длина маршрута: 0.
Маршрут: 4.
************************
```

Рис. 13. Функция 5 - обход вершин графа поиском в ширину

Пункт 6 меню выводит матрицу весов (см. [Рис. 14]).

```
6
***********************
       0       1      13       1       5
       0       0       0       0       2
       0       0       0       0       4
       0       0       0       0     100
       0       0       0       0       0

***********************
```

Рис. 14. Функция 6 - вывод матрицы весов

Пункт 7 реализует алгоритм Беллмана-Форда для этого пользователю нужно ввести начало и конец (см. [Рис. 15] [Рис. 16]).

```
7
Введите номер вершины из которой хотите построить маршрут от 0 до 4:
0
Введите номер вершины в которую хотите построить маршрут от 0 до 4:
4
***********************
Вектор расстояний:
0       1       13      1       3
Длина маршрута: 3.
Маршрут: 0->1->4.***********************
```

Рис. 15. Функция 7 - алгоритм Беллмана-Форда

```
7
Введите номер вершины из которой хотите построить маршрут от 0 до 4:
4
Введите номер вершины в которую хотите построить маршрут от 0 до 4:
2
***********************
Вектор расстояний:
-       -       -       -       0
Такой маршрут построить нельзя
***********************
```

Рис. 16. Функция 7 - алгоритм Беллмана-Форда

Пункт 8 сравнивает скорости обхода в вершин в ширину и алгоритм Беллмана-Форда (см. [Рис. 17]).

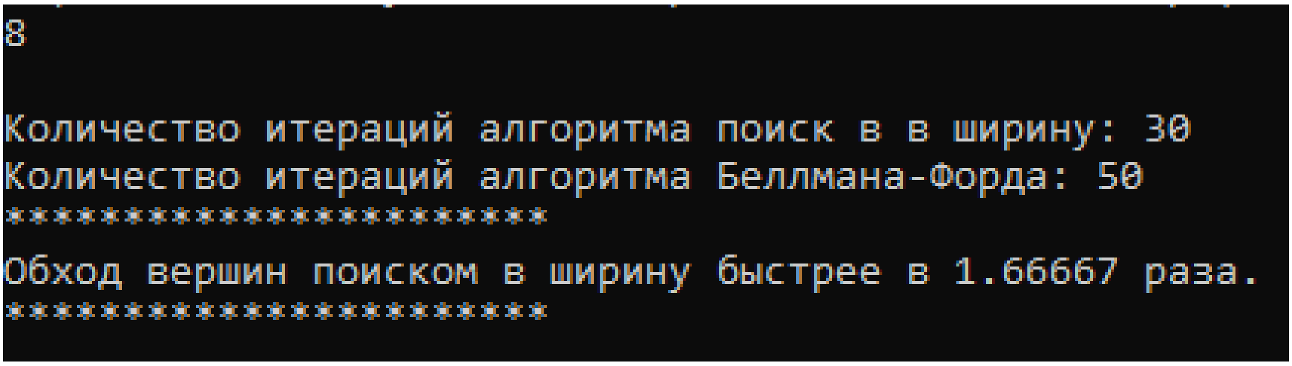


Рис. 17. Функция 8 - сравнение алгоритмов

Ввод 9 выводит матрицу пропускных способностей (см. [Рис. 18]).

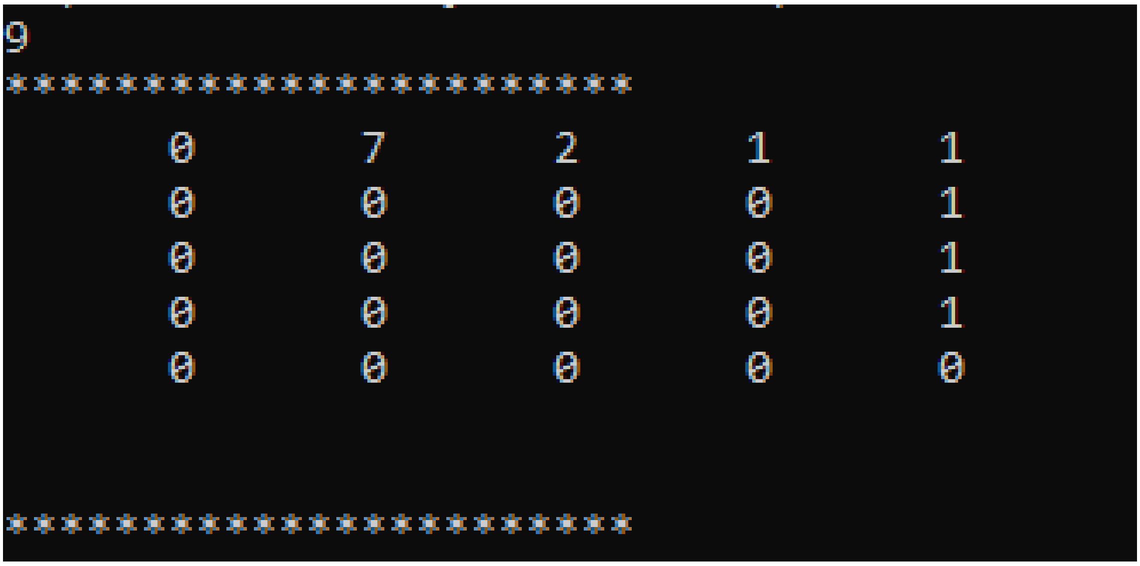


Рис. 18. Функция 9 - вывод матрицы пропускных способностей

Пункт 10 - вывод матрицы стоимостей (см. [Рис. 19]).

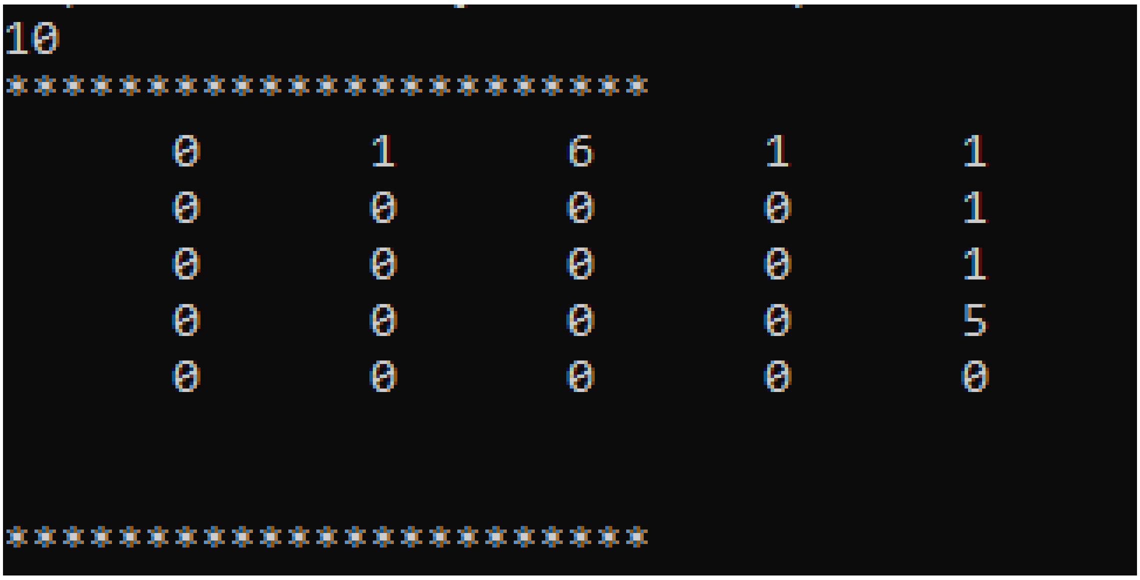


Рис. 19. Функция 10 - матрица стоимости

При вводе 11 мы получаем ход решения и максимальный поток по алгоритму Форда-Фалкерсона (см. [Рис. 20]).

```
-----------------
Промежуточный путь с пропускной способностью: 1
Маршрут: 0   4
Промежуточная матрица пропускных способностей:
0   7   2   1   0
0   0   0   0   1
0   0   0   0   1
0   0   0   0   1
1   0   0   0   0
-----------------
Промежуточный путь с пропускной способностью: 1
Маршрут: 0   1   4
Промежуточная матрица пропускных способностей:
0   6   2   1   0
1   0   0   0   0
0   0   0   0   1
0   0   0   0   1
1   1   0   0   0
-----------------
Промежуточный путь с пропускной способностью: 1
Маршрут: 0   2   4
Промежуточная матрица пропускных способностей:
0   6   1   1   0
1   0   0   0   0
1   0   0   0   0
0   0   0   0   1
1   1   1   0   0
-----------------
Промежуточный путь с пропускной способностью: 1
Маршрут: 0   3   4
Промежуточная матрица пропускных способностей:
0   6   1   0   0
1   0   0   0   0
1   0   0   0   0
1   0   0   0   0
1   1   1   1   0
-----------------
***********************
Максимальный поток: 4
***********************
```

Рис. 20. Функция 11 - максимальный поток по алгоритму Форда-Фалкерсона

12 - поток минимальной стоимости по определенному целевому потоку (см. [Рис. 21]).

```
12
Величина потока [2/3*max]: 2
Распределение потока:
0->1: 1 (стоимость 1)
0->4: 1 (стоимость 1)
1->4: 1 (стоимость 1)
*************************

Минимальная стоимость потока: 3
*************************
```

Рис. 21. Функция 12 - поток минимальной стоимости

При вводе 13 получаем сначала матрицу Кирхгофа, а затем и число остовных деревьев по Кирхгофу (см. [Рис. 22]).

```
13
Создадим неориентированный граф из ориентированного.
Матрица смежности:
      0      1      1      1      1
      1      0      0      0      1
      1      0      0      0      1
      1      0      0      0      1
      1      1      1      1      0

Матрица Кирхгофа:
      4     -1     -1     -1     -1
     -1      2      0      0     -1
     -1      0      2      0     -1
     -1      0      0      2     -1
     -1     -1     -1     -1      4


***********************
Число остовных деревьев по Кирхгофу: 20
***********************
```

Рис. 22. Функция 13 - число остовных деревьев по Кирхгофу

14 - выводит все ребра графа с весами для провекри и находит минимальный по весу остов по Краскалу (см. [Рис. 23]).

```
14

Отсортированные рёбра:
3-4 Вес: 100
4-3 Вес: 100
0-2 Вес: 13
2-0 Вес: 13
0-4 Вес: 5
4-0 Вес: 5
2-4 Вес: 4
4-2 Вес: 4
1-4 Вес: 2
4-1 Вес: 2
0-1 Вес: 1
0-3 Вес: 1
1-0 Вес: 1
3-0 Вес: 1
***********************
Минимальный по весу остов: 8
***********************
3-0 Вес: 1
1-0 Вес: 1
4-1 Вес: 2
4-2 Вес: 4
```

Рис. 23. Функция 14 - минимальный по весу остов по Краскалу

Функция 15 выводит все ребра графа, затем выводит код Прюфера и декодирует код Прюфера (см. [Рис. 24]).

```
15

Отсортированные рёбра:
3-4 Вес: 100
4-3 Вес: 100
0-2 Вес: 13
2-0 Вес: 13
0-4 Вес: 5
4-0 Вес: 5
2-4 Вес: 4
4-2 Вес: 4
1-4 Вес: 2
4-1 Вес: 2
0-1 Вес: 1
0-3 Вес: 1
1-0 Вес: 1
3-0 Вес: 1
Остов: 8
3-0 Вес: 1
1-0 Вес: 1
4-1 Вес: 2
4-2 Вес: 4

***********************
Код Прюфера:
номер вершины: 4 вес ребра: 4
номер вершины: 0 вес ребра: 1
номер вершины: 1 вес ребра: 2
номер вершины: 1 вес ребра: 1
***********************
***********************

Декодированный код Прюфера:
      0      1      0      1      0
      1      0      0      0      2
      0      0      0      0      4
      1      0      0      0      0
      0      2      4      0      0

***********************
```

Рис. 24. Функция 15 - Код Прюфера (кодировать и декодировать)

Предлагается выбрать с чем мы работаем: с исходным графом или остовом. Выводится матрица весов и все ребра графа, а также ответ (см. [Рис. 25]).

```
16

Вы выбрали функцию 20 - Найти максимальное независимое множество ребер в графе или остове
Выберите: 1 - исходный граф, 2 - остов (по алгоритму Краскала): a
Ошибка ввода! Введите либо 1, либо 2!
3
Ошибка ввода! Введите либо 1, либо 2!
1
Исходный граф:
      0      1     13      1      5
      1      0      0      0      2
     13      0      0      0      4
      1      0      0      0    100
      5      2      4    100      0

Все рёбра графа:
0 <-> 1 (Вес: 1)
0 <-> 2 (Вес: 13)
0 <-> 3 (Вес: 1)
0 <-> 4 (Вес: 5)
1 <-> 4 (Вес: 2)
2 <-> 4 (Вес: 4)
3 <-> 4 (Вес: 100)
Максимальное независимое множество ребер (паросочетание):
0 <-> 1
2 <-> 4
***********************
Размер паросочетания: 2
***********************
```

Рис. 25. Функция 16 - Максимальное независимое множество ребер

Функция 17 служит для проверки графа является ли граф эйлеровым и преобразование его к эйлерову если он не такой (см. [Рис. 27], [Рис. 26]).

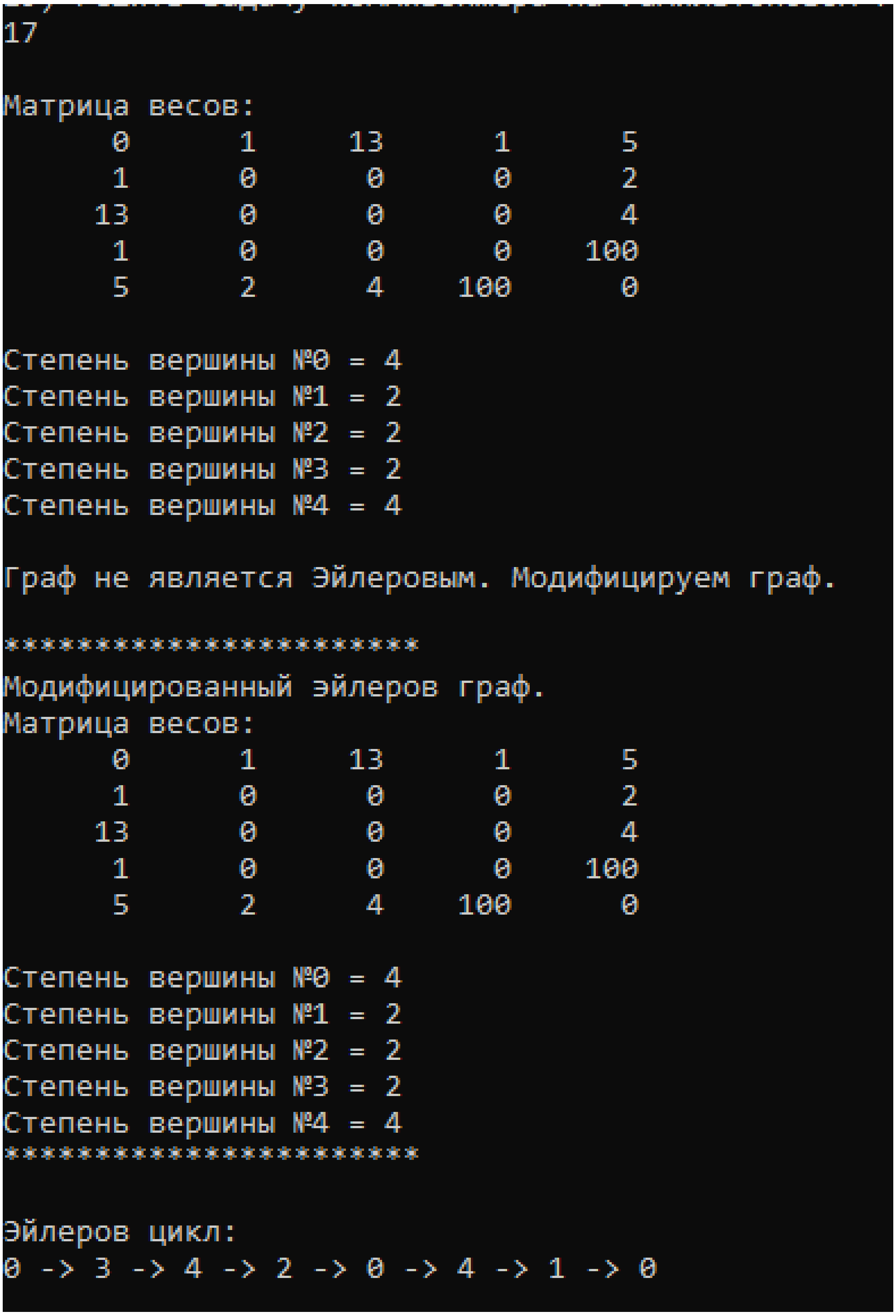
```
17

Матрица весов:
      0      1      2      1
      1      0      1      1
      2      1      0      6
      1      1      6      0

Степень вершины №0 = 3
Степень вершины №1 = 3
Степень вершины №2 = 3
Степень вершины №3 = 3
Граф является Эйлеровым


Эйлеров цикл:
1 -> 2 -> 3 -> 0 -> 2 -> 1 -> 0
```

Рис. 26. Функция 17 - Проверить, является ли граф эйлеровым/модифицировать граф, если не эйлеров



```
17

Матрица весов:
      0      1     13      1      5
      1      0      0      0      2
     13      0      0      0      4
      1      0      0      0    100
      5      2      4    100      0

Степень вершины №0 = 4
Степень вершины №1 = 2
Степень вершины №2 = 2
Степень вершины №3 = 2
Степень вершины №4 = 4

Граф не является Эйлеровым. Модифицируем граф.

***********************
Модифицированный эйлеров граф.
Матрица весов:
      0      1     13      1      5
      1      0      0      0      2
     13      0      0      0      4
      1      0      0      0    100
      5      2      4    100      0

Степень вершины №0 = 4
Степень вершины №1 = 2
Степень вершины №2 = 2
Степень вершины №3 = 2
Степень вершины №4 = 4
***********************

Эйлеров цикл:
0 -> 3 -> 4 -> 2 -> 0 -> 4 -> 1 -> 0
```

Рис. 27. Функция 17 - Проверить, является ли граф эйлеровым/модифицировать граф, если не эйлеров

Функция 18 служит для проверки графа является ли граф гамильтоновым и преобразование его к гамильтонову если он не такой (см. [Рис. 28], [Рис. 29]).

```
18

Матрица весов:
      0      1      2      8
      0      0      6      0
      0      0      0      1
      0      0      0      0


Вывод найденных гамильтоновых циклов.

Граф не является гамильтоновым. Модифицируем граф.
Добавлено ребро: 3 <-> 1 с весом: 6
Добавлено ребро: 2 <-> 0 с весом: 4

Проверка после модификации.
Цикл 1: 0 -> 3 -> 1 -> 2 -> 0 (Суммарный вес: 14)
***********************


Теперь граф гамильтонов. Найдено циклов: 1
***********************


Итоговая матрица весов:
      0      1      4      8
      0      0      6      6
      4      0      0      1
      0      6      0      0
```

Рис. 28. Функция 18 - ППроверить, является ли граф гамильтоновым/модифицировать граф, если не гамильтонов.

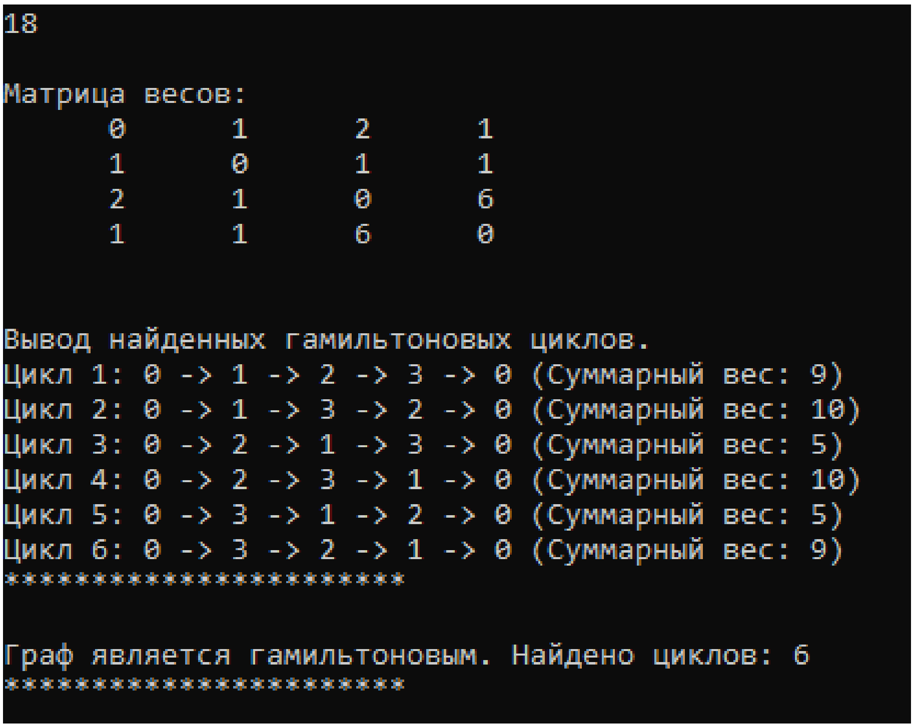


Рис. 29. Функция 18 - Проверить, является ли граф гамильтоновым/модифицировать граф, если не гамильтонов.

При вводе 18 выводится матрица весов. При необходимости граф преобразуется к гамильтонову, выводятся все циклы и ответ (см. [Рис. 30]).

```
19

Матрица весов:
      0      1      2      8
      1      0      6      0
      2      6      0      1
      8      0      1      0


Вывод найденных гамильтоновых циклов.
Цикл 1: 0 -> 1 -> 2 -> 3 -> 0 (Суммарный вес: 16)
Цикл 2: 0 -> 3 -> 2 -> 1 -> 0 (Суммарный вес: 16)
***********************

Граф является гамильтоновым. Найдено циклов: 2
***********************
***********************
Все возможные гамильтоновы циклы:
0 -> 1 -> 2 -> 3 -> 0   Вес цикла: 16
0 -> 3 -> 2 -> 1 -> 0   Вес цикла: 16
***********************
Цикл коммивояжера:
0 -> 1 -> 2 -> 3 -> 0
Вес: 16
```

Рис. 30. Функция 19 - Решить задачу коммивояжера на гамильтоновом графе.

В меню предусмотрена защита от некорректного пользовательского ввода (см. [Рис. 31]).

```
уукук
Ошибка ввода! Число от 0 до 19 включительно!
-3
Ошибка ввода! Число от 0 до 19 включительно!
20
Ошибка ввода! Число от 0 до 19 включительно!
```

Рис. 31. Защита от некорректного пользовательского ввода

При выборе пункта 1 пользователь заново создает граф так же как при запуске программы (см. [Рис. 32]).

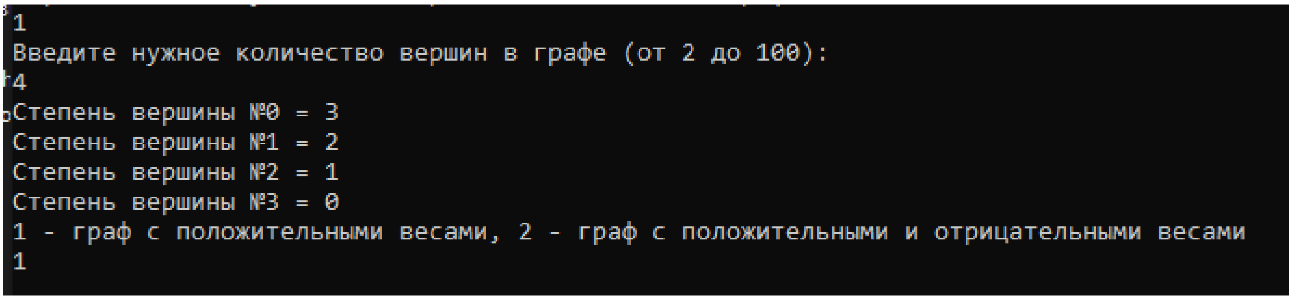


Рис. 32. Пересоздание графа

При вводе нуля выводится прощальное окно и программа завершает работу (см. [Рис. 33]).

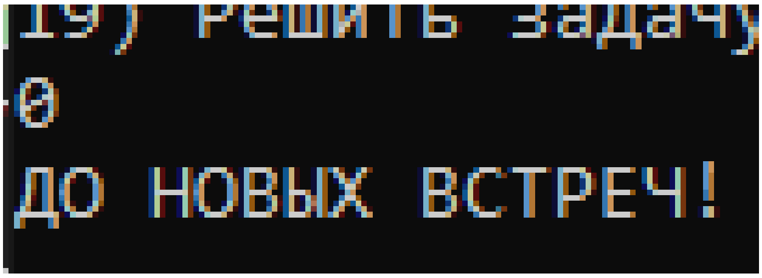


Рис. 33. Выход из программы

Ниже приведены примеры работы программы с графом с двумя вершинами (см. [Рис. 34]).

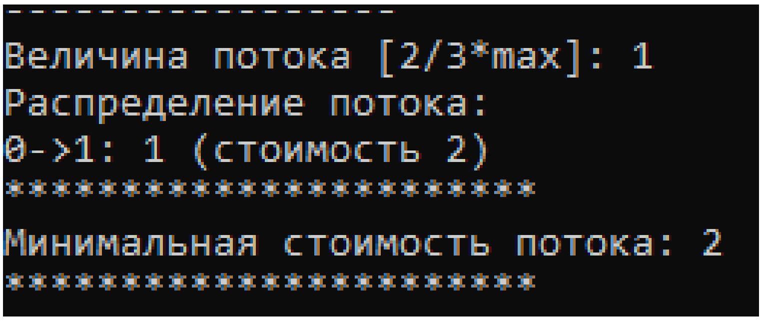


Рис. 34. Минимальная стоимость потока

При вызове пунктов 17, 18, 19 получаем предупреждение о невозможности модификации графа (см. [Рис. 35]).

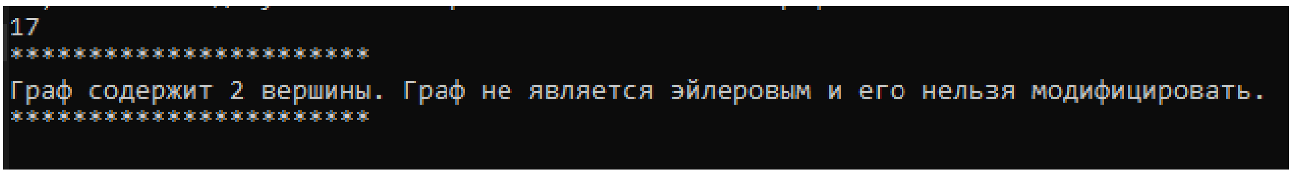


Рис. 35. Минимальная стоимость потока

Заключение

В результате работы была реализована программа позволяющая создавать связный ациклический граф по распределению Парето. Была реализована серия алгоритмов для работы с графами. В первой лабораторной работе был сгенерирован случайный связный ациклический граф с использованием распределения Парето, после чего реализован метод Шимбелла для вычисления минимальных и максимальных путей в сгенерированной весовой матрице. Также была определена возможность построения маршрутов между заданными вершинами с подсчетом их количества. Во второй лабораторной работе был реализован алгоритм поиска в ширину для обхода вершин графа. Для заданных весовых матриц был найден кратчайший путь между двумя вершинами с использованием алгоритма Беллмана-Форда, с выводом как расстояний, так и самих путей. Также был проведен анализ и сравнена скорость работы алгоритмов по количеству итераций. В третьей лабораторной работе был сгенерирован связный ациклический граф с матрицами пропускных способностей и стоимости, после чего был реализован алгоритм Форда-Фалкерсона для нахождения максимального потока в графе. Кроме того, был вычислен поток минимальной стоимости с использованием алгоритмов Дейкстры и Беллмана-Форда. В четвертой лабораторной работе использовалась матричная теорема Кирхгофа для вычисления числа остовных деревьев, построен минимальный остов с использованием алгоритма Краскала, а также реализовано кодирование и декодирование остова с помощью кода Прюфера. Были также реализованы алгоритмы нахождения максимального независимого множества ребер для исходного графа и полученного остова. В пятой лабораторной работе проверялось, является ли граф эйлеровым и гамильтоновым. Графы, которые не были эйлеровыми, были модифицированы, и построены эйлеровы циклы. Также была решена задача коммивояжера на гамильтоновом графе.

Присутствует ограничение на вводимое количество вершин - не более 100, так как даже создание графа со ста вершинами занимает 15 секунд, что достаточно много.

Недостатком данной программы является большое количество функций, которые можно было бы объединить в несколько. Например, одна функция для генерации распределения на степени верши, весовую матрицу, матрицу стоимости и пропускной способности. При генерации графа требуется, чтобы ровно одна вершина имела степень N-1, что не всегда соответствует реальным задачам.

Преимуществом программы является вывод промежуточных результатов, что помогает понять ход работы программы и упрощают проверку вывода. Также реализован поиск кратчайшего пути через обход в ширину. После создания гамильтонова графа, он сохраняется, что гарантирует, что при выборе пункта с задачей Коммивояжера не придется еще раз генерировать граф. Это ускоряет работу программы.

Дополнительно может быть реализована визуализация графов с помощью графических библиотек, дополнительные алгоритмы работы с графами (A*, Дейкстры и другие). Могут быть оптимизированы существующие алгоритмы для работы с большими графами. Например, в алгоритме Краскала заменить сортировку рёбер на более эффективную - быструю, SPFA (Shortest Path Faster Algorithm) – улучшенный Беллман-Форд.

Список использованной литературы

[1] Графы и алгоритмы. Новиков Ф. А. Дискретная математика для программистов. - Санкт-Петербург : Питер Пресс, 2000 - 304 с.

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

[3] Связь вершинного покрытия и независимого множества.

// URL: https://neerc.ifmo.ru/wiki/

(дата обращения: 24.04.2025).