МИНОБРНАУКИ РОССИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ

**«САНКТ-ПЕТЕРБУРГСКИЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ ПЕТРА ВЕЛИКОГО»**

Институт Прикладной математики и механики

Высшая школа прикладной математики и вычислительной физики

Секция: «Телематика»

Направление 02.03.01 Математика и компьютерные науки

Теория графов

# Отчёт

по лабораторным работам №1-5

*«Реализация различных алгоритмов на графах»*

Студент: _____ группа 3630201/90002 Коренова А.Д.

Преподаватель: _____ Востров А.В.

« ____»_____ 20__

Санкт-Петербург — 2021

# Содержание

# Введение

В данном отчете представлено описание особенностей выполнения пяти лабораторных работ по дисциплине *«Теория графов»*.

Лабораторные работы включали в себя программную реализацию различных алгоритмов. Все алгоритмы были применены на связном ациклическом графе. Матрица смежности графа сформирована в соответствии с **геометрическим распределением 2 (распределением Фарри)**.

Все лабораторные работы выполнены на языке программирования C++.

# 1 Постановка задачи

**Основная задача:** построить случайный связный ациклический граф в соответствии с заданным распределением (**геометрическое распределение 2**). На графе необходимо реализовать:

- поиск максимальных (или минимальных) путей с помощью **алгоритма Шимбелла**;
- определение **возможности построения маршрута** от одной заданной точки до другой и указание количества таких маршрутов;
- нахождение кратчайшего пути от одной точки к другой, используя **алгоритм Дейкстры**;
- нахождение кратчайшего пути от одной точки к другой, используя **алгоритм Беллмана-Форда**;
- нахождение кратчайшего пути от одной точки к другой, используя **алгоритм Флойда**;
- сравнение скорости работы трех вышеуказанных алгоритмов на нахождение кратчайшего пути;
- построение минимального по весу остова, используя **алгоритм Прима**;
- построение минимального по весу остова, используя **алгоритм Краскала**;
- найти число остовных деревьев в графе, используя **матричную теорему Кирхгофа**;
- закодировать полученный остов с помощью **кода Прюфера** (а также проверить верность кодирования декодированием);
- нахождение максимального потока по алгоритму **Форда-Фалкерсона**;
- вычисление **потока минимальной стоимости**;
- построение **Эйлерова цикла**;
- дополнение исходного графа до Эйлерова и/или Гч амильтонова;
- решение **задачи коммивояжера**.

# 2 Математическое описание

## 2.1 Определение графа

**Определение:** *графом* $G(V, E)$ называется совокупность двух элементов — непустого множества $V$ (множества *вершин*) и множества $E$ двухэлементных подмножеств множества $V$ ($E$ — множество *ребер*),

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

Число вершин графа $G$ обозначим $p$, а число ребер — $q$:

$$p \stackrel{def}{=} p(G) \stackrel{def}{=} |V|, \quad q \stackrel{def}{=} q(G) \stackrel{def}{=} |E|.$$

**Определение:** две вершины в графе *связаны*, если существует соединяющая их (простая) цепь.

**Определение:** граф, в котором все вершины связаны, называется *связным*.

**Определение:** *ациклический граф* — это граф, который не содержит циклов.[1]

## 2.2 Геометрическое распределение 2 (распределение Фарри)

В работе для генерации графа было использовано **геометрическое распределение 2** (распределение Фарри). Оно моделирует распределение дискретной случайной величины — число независимых испытаний Бернулли до первого успеха (включая и первый успех); $p$ — вероятность успеха; $q = 1 - p$ — вероятность неудачи в одном испытании.[3]

Генерирование случайных чисел в соответствии с этим распределением происходит по алгоритму, представленному в блок-схеме (см. рис. 1).

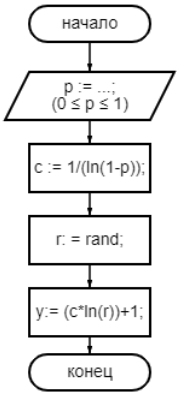


Рис. 1: Блок-схема алгоритма генерации случайных чисел

Полученная последовательность чисел характеризует распределение степеней вершин графа $G(V, E)$.

## 2.3 Метод Шимбелла

**Алгоритм (или же метод) Шимбелла** предназначен для определения *экстремальных* (минимальных или максимальных) расстояний между всеми парами вершин взвешенного графа и учитывает число ребер, входящих в соответствующие простые цепи. Матрица весов в исходном виде задает стоимость переходов между вершинами графа по цепям, состоящим из одного звена. [1]

Для нахождения экстремальных путей используются *специальные операции*:

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

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

- **операция сложения** двух величин $a$ и $b$ заменяется выбором из этих величин максимального (минимального) элемента, то есть:

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

  нули при этом игнорируются. Минимальный или максимальный элемент выбирается из ненулевых элементов. Ноль в результате *операции сложения* может быть получен лишь тогда, когда все элементы из выбираемых — нулевые.

С помощью этих операций длины экстремальных путей определяются возведением в степень $x$ весовой матрицы $\Omega$, где $x$ — это количество ребер в пути. Элемент $\Omega^x[i][j]$ покажет длину экстремального пути длиной в $x$ ребер из вершины $i$ в вершину $j$.

## 2.4 Алгоритм Дейкстры

**Алгоритм Дейкстры** находит кратчайший путь между двумя данными вершинами (узлами) в (ор)графе, если длины дуг неотрицательны.[1]

**Сложность алгоритма:** $O(n^2)$.

### 2.4.1 Описание работы алгоритма

В простейшей реализации для хранения чисел $d[i]$ можно использовать массив чисел, а для хранения принадлжености элемента множеству $U$ (множество посещенных вершин) — массив булевых переменных.

В начале алгоритма расстояние для начальной вершины полагается равным нулю, а все остальные расстояния заполняются большим положительным числом (число должно быть больше, чем максимальный возможный путь в графе). Массив флагов заполняется нулями. Затем запускается основной цикл.

На каждом шаге цикла мы ищем вершину $v$ с минимальным расстоянием и флагом равным нулю. Затем мы устанавливаем в ней флаг 1 и проверяем все соседние с ней вершины $\mu$. Если в них (в $\mu$) расстояние больше, чем сумма расстояний до текущей вершины и длины ребра, то уменьшаем его. Цикл завершается, когда флаги всех вершин становятся равны 1, либо когда у всех вершин с флагом 0 $d[i] = \infty$. Последний случай возможен тогда и только тогда, когда граф $G$ — несвязный.

## 2.5 Алгоритм Беллмана-Форда

**Алгоритм Беллмана-Форда** находит кратчайшие пути во взвешенном графе, не содержащем циклы с отрицательным суммарным весом, от одной вершины до всех остальных[2].

При этом алгоритм Беллмана-Форда позволяет определить наличие циклов отрицательного веса, достижимых из начальной вершины.

**Сложность алгоритма:** $O(n*m)$, где $n$ — количество вершин, $m$ — количество ребер.

## 2.6 Алгоритм Флойда-Уоршелла

**Алгоритм Флойда-Уоршелла** находит кратчайшие пути между всеми парами вершин (узлов) в (ор)графе. Веса ребер могут быть как положительными, так и отрицательными. Для нахождения кратчайших путей между всеми вершинами графа используется *восходящее динамическое программирование*, то есть все подзадачи, которые впоследсвии понадобятся для решения исходной задачи, просчитываются заранее, а затем используются[1].

**Идея алгоритма** — разбиение процесса поиска кратчайших путей на фазы.

Перед $k-$ой фазой величина $d[i][j]$ равна длине кратчайшего пути из вершины $i$ в вершину $j$, если этому пути разрешается заходить только в вершины с номерами меньшими $k$. На $k-$ой фазе мы попытаемся улучшить путь $i \Rightarrow j$, пройдя через вершину $k : d[i][j] = \min(d[i][j], d[i][k] + d[k][j])$. Матрица $d$ является искомой матрицей расстояний.

**Сложность алгоритма:** $O(n^3)$.

## 2.7 Минимальный остов

**Остовным деревом** или **остовом** графа $G(V, E)$ называется связный подграф без циклов, содержащий все вершины исходного графа. Подграф содержит часть или все ребра исходного графа.

**Минимальное остовное дерево** — это остовное дерево, сумма весов ребер которого минимальна.

## 2.8 Алгоритм Прима

**Алгоритм Прима** — алгоритм построения минимального остовного дерева взвешенного связного неориентированного графа[1].

В данном алгоритме кратчайший остов порождается в процессе разрастания одного дерева, к которому присоединяются ближайшие одиночные вершины. Каждая одиночная вершина является деревом.

*На выходе* мы получаем множество ребер, которые лежат в минимальном остовном дереве.

**Сложность алгоритма:** $O(n^2)$.

### 2.8.1 Описание работы алгоритма

- Изначально остов полагается состоящим из единственной вершины (её можно выбрать произвольно).
- Затем выбирается ребро минимального веса, исходящее из этой вершины, и добавляется в минимальный остов.
- После этого остов содержит уже две вершины, и теперь ищется и добавляется ребро минимального веса, имеющее один конец в одной из двух выбранных вершин, а другой — наоборот, во всех остальных, кроме этих двух. И так далее, т.е. всякий раз ищется минимальное по весу ребро, один конец которого — уже взятая в остов вершина, а другой конец — ещё не взятая, и это ребро добавляется в остов (если таких рёбер несколько, можно взять любое).
- Этот процесс повторяется до тех пор, пока остов не станет содержать все вершины.
- В итоге будет построен остов, являющийся минимальным. Если граф был изначально не связен, то остов найден не будет (количество выбранных рёбер останется меньше n - 1).

## 2.9 Алгоритм Краскала

**Алгоритм Краскала** — это эффективный алгоритм построения минимального остовного дерева взвешенного связного неориентированного графа[1].

*На выходе* получится множество ребер, которые лежат в минимальном остовном дереве.

**Сложность алгоритма:** $O(n \log n)$.

### 2.9.1 Описание работы алгоритма

- Все дуги удаляются из дерева.
- Все рёбра сортируются по весу (в порядке неубывания).

- Затем начинается процесс **объединения**: перебираются все рёбра от первого до последнего (в порядке сортировки), и если у текущего ребра его концы принадлежат разным поддеревьям, то эти поддеревья объединяются, а ребро добавляется к ответу. По окончании перебора всех рёбер все вершины окажутся принадлежащими одному поддереву, и ответ найден.

## 2.10 Матричная теорема Кирхгофа

**Матрица Кирхгофа** — матрица размера $n \times n$, где $n$ — количество вершин графа[2].

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

**Матричная теорема Кирхгофа:** число остовных деревьев в связном графе $G$ порядка $n \geq 2$ равно алгебраическому дополнению любого элемента матрицы Кирхгофа.

### 2.10.1 Свойства матрицы Кирхгофа

- Суммы элементов в каждой строке и столбце матрицы равны 0.
- Алгебраические дополнения всех элементов матрицы равны между собой.
- Определитель матрицы Кирхгофа равен 0.

## 2.11 Код Прюфера

**Код Прюфера** — это способ взаимно однозначного кодирования помеченных деревьев с $n$ вершинами с помощью последовательности $n-2$ целых чисел в отрезке $[1; n]$.[2]

Иными словами **код Прюфера** — это биекция между всеми остовными деревьями полного графа и числовыми последовательностями.

### 2.11.1 Алгоритм построения кода Прюфера

**Вход:** дерево с $n$ вершинами.

**Выход:** код Прюфера длины $n-2$.

Происходит $n-2$ повторений следующих действий:

- выбрать вершину $v$ — лист дерева с наименьшим номером;
- добавить номер единственного соседа $v$ в последовательность кода Прюфера;
- удалить вершину $v$ из дерева.

## 2.12 Максимальный поток в сети

Пусть $G(V, E)$ — сеть, $s, t$, соответственно, **источник** и **сток** сети. Дуги сети нагружены неотрицательными вещественными числами: $f : E \to \mathbb{R}^+$. Если $u$ и $v$ — узлы сети, то число $c(u, v)$ называется *пропускной способностью* дуги $(u, v)$.[1]

**Дивергенцией** функции $f$ в узле $v$ называется число $div(f, v)$, которое определяется следующим образом:

$$div(f, u) \stackrel{def}{=} \sum_{v|(u,v)\in E} f(u, v) - \sum_{v|(u,v)\in E} f(v, u).$$

Функция $f : E \to \mathbb{R}$ называется **потоком** в сети $G$, если:

- $\forall (u, v) \in E (0 \leq f(u, v) \leq c(u, v))$, то есть поток через дугу неотрицателен и не превосходит пропускной способности дуги.
- $\forall u \in V \quad s, t (div(f, u) = 0)$, то есть дивергенция потока равна нулю во всех узлах кроме источника и стока.

**Величина потока** в сети $G$ — сумма всех потоков, выходящих из истока, то $\omega(f) = div(f, s)$.

## 2.13 Алгоритм Форда-Фалкерсона

**Теорема Форда-Фалкерсона:** максимальный поток в сети равен минимальной пропускной способности разреза, то есть существует поток $f^*$, такой, что $\omega(f^*) = \max\limits_f \omega(f) = \min\limits_P C(P)$.

На основе данной теоремы реализуется **алгоритм Форда-Фалкерсона** для определения максимального потока в сети, заданной матрицей пропускных способностей дуг.

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

Для поиска потока минимальной стоимости заднной величины использовался ранее реализованный **алгоритм Беллмана-Форда**, возвращающий последовательность вершин, по которым можно восстановить путь минимальной стоимости. После нахождения такого пути по нему пускается максимальный поток. Если величина потока достигла заданной величины, то алгоритм завершается. За величину потока бралось значение, равное 2/3 величины максимального потока[1].

## 2.14 Поиск максимального потока минимальной стоимости

**Задача:** дана сеть $G(V, E)$. $s, t \in V$ — источник и сток соответственно. Ребра $(u, v) \in E$ имеют пропускную способность $c(u, v)$, поток $f(u, v)$ — и цену за единицу потока $a(u, v)$. Требуется найти максимальный поток, суммарная стоимость которого минимальна.

## 2.15 Эйлеров граф

Если граф имеет цикл, содержащий все ребра графа ровно один раз, то такой цикл называется **Эйлеровым циклом**, а граф называется **Эйлеровым графом**.

Эйлеров цикл содержит не только все ребра (по одному разу), но и все вершины графа (возможно, по несколько раз). Эйлеровым может быть только связный граф.

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

- $G$ — Эйлеров граф;
- каждая вершина $G$ имеет четную степень;
- множество ребер $G$ можно разбить на простые циклы.

Значит, для проверки наличия Эйлерова цикла в графе достаточно убедиться, что степени всех вершин *четны*[1].

## 2.16 Гамильтонов граф

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

Гамильтонов цикл не обязательно содержит все ребра графа. Гамильтоновым графом может быть только связный граф.

**Теорема** *(достаточное условие гамильтоновости графа)*: если $\delta(G) \geq p/2$ (где $\delta(G)$ — минимальная степень вершин графа), то граф $G$ является Гамильтоновым.[1]

## 2.17 Задача коммивояжера

**Постановка задачи:** задача, в которой коммивояжер должен посетить N городов, побывав в каждом из них по одному разу и завершив путешествие в том городе, с которого он начал. *В какой последовательности ему нужно обходить города, чтобы общая длина его пути была наименьшей?*[1]

Таким образом, требуется:

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

при этом каждая вершина должна быть посещена ровно один раз.

Иными словами, **задача коммивояжера** — задача отыскания кратчайшего гамильтонова цикла в нагруженном полном графе.

Задачу коммивояжера можно представить в виде **целевой функции**.

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

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

Среди чисел $p_1, p_2, \ldots, p_{n-1}$ по одному разу встречается каждое число из интервала $2, \ldots, n$. *Таким образом, задача коммивояжера состоит в поиске перестановки целых чисел от 2 до n, при которой целевая функция минимальна.*

# 3 Особенности реализации

## 3.1 Структура данных

В данном разделе будут перечислены все структуры данных, в которых хранятся: матрица смежности, матрицы с положительными и отрицательными весами, матрица пропускных способностей и др.

- `vector<vector<int>> AdjacencyMatrix` — двумерный вектор, в котором хранится матрица смежности. Иными словами, эта матрица и есть граф.
- `vector<vector<int>> MatrixVes` — двумерный вектор, в котором хранится матрица положительных весов.
- `vector<vector<int>> MatrixNegVes` — двумерный вектор, в котором хранится матрица отрицательных весов.
- `vector<vector<int>> AdjacencyMatrixInf` — двумерный вектор, в котором хранится матрица смежности, но нули заменены бесконечностями (для алгоритмов Прима и Краскала).
- `vector<vector<int>> GraphEuler` — двумерный вектор, в котором хранится граф, являющийся дополнением исходного графа до эйлерова графа.
- `vector<vector<int>> GraphGamilton` — двумерный вектор, в котором хранится граф, являющийся дополнением исходного графа до гамильтонова графа.
- `int** CapacityMatrix` — двумерный динамический массив, в котором хранится матрица пропускных способностей.
- `int** CostMatrix` — двумерный динамический массив, в котором хранится матрица стоимостей.
- `int** Potok` — двумерный динамический массив, в котором хранится матрица максимального потока.
- `vector<set<int>>* MinOstov` — вектор множеств, который хранит минимальный остов графа.
- `vector<int> PruferCode` — вектор, в котором хранится код Прюфера.
- `int numberVert` — глобальная целочисленная переменная для хранения количества вершин графа.
- `int it1, it2, it3, itP, itK, m` — глобальные целочисленные переменные для хранения количества итераций таких алгоритмов, как алгоритм Дейкстры, алгоритм Флойда-Уоршелла и др.

## 3.2 Генерация графа. Построение маршрута

### 3.2.1 Генерация графа

**Функция:** `void GraphCreation()`.

**Вход функции:** на вход функции подается двумерный вектор `vector<vector<int>>AdjacencyMatrix`, в котором хранится матрица смежности сгенерированного графа.

**Выход функции:** заполненная матрица смежности.

**Пример операции:** см. рисунок 3.

**Описание работы функции:** с помощью полного прохода по матрице смежности функция заполняет ее нулями и единицами на основе заданного распределения . Случайные величины распределения рассчитываются с помощью функции `int RandomGen()`. Кроме того при заполнении матрицы смежности проверяется выполняется ли условие связности графа.

```
void GraphCreation()
   {
       AdjacencyMatrix.clear();
       for (int i = 0; i < numberVert; i++)
               AdjacencyMatrix.push_back(vector<int>(numberVert, 0));

       for (int i = 0; i < numberVert; i++)
       {
               for (int j = i + 1; j < numberVert; j++)
               {
                       if (j == i + 1) // вот здесь проверить выполняется ли связность графа
                               AdjacencyMatrix.at(i).at(j) = 1;
                       else
                               AdjacencyMatrix.at(i).at(j) = abs(RandomGen() % 2);
               }
       }
       CostMatrix = nullptr;
       CapacityMatrix = nullptr;
       Potok = nullptr;
   }
```

### 3.2.2 Метод Шимбелла

**Функция:** `void Shimbell()`.

**Вход функции:** на вход функции подаются двумерные вектора `vector<vector<int>>MatrixVes` и `vector<vector<int>>MatrixNegVes`, в которых хранится матрица с положительными весами и матрица с отрицательными весами соответственно; а также номер вершины, из которой будет происходить поиск экстремального пути.

**Выход функции:** матрица Шимбелла экстремальных путей из заданной вершины.

**Пример операции:** см. рисунок 4.

**Описание работы функции:** изначально пользователю предоставляется выбор: начальной вершины, какая матрица будет использоваться для работы (с отрицательными или с положительными весами), а также поиск минимального или максимального пути. Далее происходит рекурсивное возведение в степень выбранной матрицы с помощью функции `void Matrix(vector<vector<int>>& vec1, vector<vector<int>>& vec2, int path)`. Выход из рекурсии происходит после того, когда матрица возведена в нужную степень.

```
void Shimbell()
   {
       printf("\n\n***Shimbel's method***\n\n");
       int count = 0;
       int path = 0;
       int matr = 0;
       count = inputNumber("Enter the number of verge:", 1, AdjacencyMatrix.size() - 1);
       printf("\n***MENU***\n");
       printf("Select an matrix:\n");
       printf("[1] - positive length matrix;\n");
       printf("[2] - negative length matrix.\n");
       matr = inputNumber("\nEntered value:", 1, 2);

       printf("\nSelect an action:\n");
       printf("[1] - search for maximum paths;\n");
       printf("[2] - search for minimum paths.\n");

       path = inputNumber("\nEntered value:", 1, 2);
       path -= 1;

       vector<vector<int>> buff_buff;
       if (matr == 1)
       {
               buff_buff = MatrixVes;
               for (int i = 1; i < count; i++)
               {
                       Matrix(buff_buff, MatrixVes, path);
               }
       }
       else
       {
               buff_buff = MatrixNegVes;
               for (int i = 1; i < count; i++)
               {
                       Matrix(buff_buff, MatrixNegVes, path);
               }
       }
       if (path == 1)
               printf("\nShortest paths with %d edges.", count);
       else
               printf("\nLongest paths with %d edges.", count);

       PrintShimbell(buff_buff);
   }
```

### 3.2.3 Определение возможности построения маршрута

**Функция:** `void PathExist()`.

**Вход функции:** на вход функции подается вершина, с которой нужно начать поиск пути, а также вершина, которой нужно закончить поиск пути. Кроме того понадобится матрица смежности графа.

**Выход функции:** выводится строка может или не может существовать путь, а также количество путей между заданными вершинами.

**Пример операции:** см. рисунок 5.

**Описание работы функции:** пользователь вводит желаемые вершины. Для поиска пути была реализована дополнительная функция `bool dfs(int u, int t, vector<bool>& visited, vector<vector<int>>& arr)`. Это функция обхода графа в глубину. Алгоритм обхода в глубину позволяет решать множество различных задач, в том числе и задачу определения количества маршрутов, состоящих из $k$ ребер. Глубина рекурсии, в которой мы находимся, не превышает общего числа вызовов функции `dfs` — числа вершин. То есть, объем памяти, необходимый для работы алгоритма, равен $O(V)$.

```
void PathExist()
    {
        printf("\n\n***Build a route***\n");
        path_exist = 0;
        int n1 = 0;
        int n2 = 0;
        n1 = inputNumber("\nEnter the number of the first vertex: ", 1, AdjacencyMatrix.size());
        n2 = inputNumber("\nEnter the number of the second vertex: ", 1, AdjacencyMatrix.size());
        vector<bool> visited(AdjacencyMatrix.size(), false);
        if (dfs(n1 - 1, n2 - 1, visited, AdjacencyMatrix))
                printf("\nA path from vertex %d to vertex %d exists", n1, n2);
        else
                printf("\nA path from vertex %d to vertex %d doesn't exists", n1, n2);

        NumberPath(n1 - 1, n2 - 1);
        printf("\n\nNumber of paths: %d\n\n", path_exist);
    }
```

## 3.3 Нахождение кратчайшего пути

### 3.3.1 Алгоритм Дейкстры

**Функция:** `void AlgDijkstra()`.

**Вход функции:** на вход функции подается вершина, с которой нужно начать поиск пути, а также матрица весов, в которой вместо нулей стоят бесконечности.

**Выход функции:** выводится вектор расстояний до каждой вершины графа от заданной пользователем вершины.

**Пример операции:** см. рисунок 6.

**Описание работы функции:** алгоритм Дейкстры состоит из двух этапов.

- **Первый этап.** *Алгоритм нахождения длины кратчайшего пути.* Для каждой вершины будем хранить текущую длину `Dijkstr[i]` кратчайшего пути, для этого создается вектор `vector<int>Dijkstr(numberVert, 0)`. На первом этапе текущая длина только для начальной вершины равна 0, для остальных вершин эта величина равна $\infty$. Каждой вершине графа сопоставляется метка — минимальное известное расстояние до данной вершины. Первый шаг алгоритма заключается в поиске вершины с наименьшим значением метки, будем считать эту вершину текущей.

  На втором шаге осуществляется проход по оставшимся вершинам и производится обновление метки, если найден более короткий путь через текущую вершину.

  Далее происходит превращение меток из временных в постоянные.

- **Второй этап.** *Построение кратчайших путей.* Вершины хранятся в структуре данных `vector<int>Dijkstr(numberVert, 0)` и для поиска минимума используется алгоритм линейного поиска.

**Сложность алгоритма:** $O(n^2)$, где $n$ — количество вершин.

```
void AlgDijkstra()
    {
        printf("\n\n***Dijkstra's algorithm***\n\n");

        it1 = 0;
        int start_vert = 0;
        start_vert = inputNumber("Input the start vertex: ", 1, numberVert);

        vector<bool> visited(numberVert, 0);
        vector<int> Dijkstr(numberVert, 0);

        for (int i = 0; i < numberVert; i++)
        {
                Dijkstr[i] = MatrixVesInf[start_vert-1][i];
                visited.at(i) = false;
        }
        Dijkstr.at(start_vert - 1) = 0;
        visited.at(start_vert - 1) = true;

        int index = 0;
        int dop_index = 0;
        for (int i = 0; i < numberVert; i++)
        {
                int min = INFINITY;
                for (int j = 0; j < numberVert; j++)
                {
                        if (!visited.at(j) && Dijkstr.at(j) < min)
                        {
```

```
                        min = Dijkstr.at(j);
                        index = j;
                }
                it1++;
        }
        dop_index = index;
        visited.at(dop_index) = true;

        for (int k = 0; k < numberVert; k++)
        {
                if (!visited.at(k) && MatrixVesInf.at(dop_index).at(k) != INFINITY && Dijkstr.at(dop_index)

                != INFINITY && (Dijkstr.at(dop_index) + MatrixVesInf.at(dop_index).at(k) < Dijkstr.at(k)))
                {
                        Dijkstr.at(k) = Dijkstr.at(dop_index) + MatrixVesInf.at(dop_index).at(k);
                }
        }
    }

    // вывод результатов работы функции
    printf("\nDijkstra's algorithm: \n");
    for (int i = 0; i < numberVert; i++)
    {
        if (Dijkstr.at(i) != INFINITY)
                printf("From the vert number %d to the vert number %d = %d\n", start_vert, i + 1, Dijkstr.at(i));
        else
                printf("From the vert number %d to the vert number %d = %s\n", start_vert, i + 1, "infinity");
    }

    printf("\nNumber of iterations: %d.\n", it1);
}
```

### 3.3.2 Алгоритм Беллмана-Форда

**Функция:** `void AlgBellmanFord()`.

**Вход функции:** на вход функции подается матрица с отрицательными весами, матрица с положительными весами, начальная вершина (которую выбирает пользователь).

**Выход функции:** выводится вектор расстояний до каждой вершины графа от заданной пользователем вершины.

**Пример операции:** см. рисунок 7.

**Описание работы функции:** на *первом шаге* инициализируются расстояния от исходной вершины до всех остальных вершин, как бесконечные, а расстояние до начальной вершины принимается равным 0. Создается вектор `vector<pair<int, int>> distance (numberVert, pair<int, int>(0, INFINITY))` со всеми значениями равными бесконечности, кроме элемента `distance[start_vert]`, где `start_vert` — начальная вершина.

На *втором шаге* вычисляются самые короткие расстояния. Эти шаги нужно выпол-

нить `numberVert - 1`.

На *третьем шаге* сообщается присутствует ли в графе цикл отрицательного веса. Для каждого ребра необходимо выполнить следующее: если `distance[j].second + MatrixNegVes[j][k] < distance[k].second`, то графе присутствует цикл отрицательного веса.

**Идея** третьего шага заключается в том, что второй шаг гарантирует кратчайшее расстояние, если граф не содержит цикла отрицательного веса. Если мы снова переберем все ребра и получим более короткий путь для любой из вершин, то это будет сигналом присутствия цикла отрицательного веса.

**Сложность алгоритма:** $O(m*n)$, где $n$ — количество вершин графа, $m$ — количество ребер графа.

```
void AlgBellmanFord()
    {
        printf("\n\n***Bellman-Ford algorithm***\n\n");

        it2 = 0;
        int start_vert = 0; // начальная вершина
        start_vert = inputNumber("Enter the number of verge:", 1, AdjacencyMatrix.size());

        int matr = 0;
        printf("\nSelect an matrix:\n");
        printf("[1] - positive length matrix;\n");
        printf("[2] - negative length matrix.\n");
        matr = inputNumber("\nEntered value:", 1, 2);

        vector<pair<int, int>> distance(numberVert, pair<int, int>(0, INFINITY));
        if (matr == 1)
        {
                distance[start_vert - 1].first = 0;
                distance[start_vert - 1].second = 0;

                for (int i = 0; i < numberVert; i++)
                {
                        for (int j = 0; j < numberVert; j++)
                        {
                                for (int k = 0; k < numberVert; k++)
                                {
                                        if (MatrixVes[j][k] != 0)
                                        {
                                                if (distance[j].second + MatrixVes[j][k] < distance[k].second)
                                                {
                                                        distance[k].first = j;
                                                        if (distance[j].second < INFINITY)
                                                                distance[k].second = distance[j].second +
                                                                MatrixVes[j][k];
                                                }
                                                it2++;
                                        }
                                }
                        }
                }
```

```
    }
    if (matr == 2)
    {
            distance[start_vert - 1].first = 0;
            distance[start_vert - 1].second = 0;

            for (int i = 0; i < numberVert; i++)
            {
                    for (int j = 0; j < numberVert; j++)
                    {
                            for (int k = 0; k < numberVert; k++)
                            {
                                    if (MatrixNegVes[j][k] != 0)
                                    {
                                            if (distance[j].second + MatrixNegVes[j][k] < distance[k].second)
                                            {
                                                    distance[k].first = j;
                                                    if (distance[j].second < INFINITY)
                                                            distance[k].second = distance[j].second
                                                            + MatrixNegVes[j][k];
                                            }
                                            it2++;
                                    }
                            }
                    }
            }
    }

    printf("\nBellman-Ford algorithm: \n");
    for (int i = 0; i < numberVert; i++)
    {
            if(distance[i].second == 0)
                    printf("To the verge number: %d - Rib weight: %d\n", i + 1, distance[i].second);
            else if (distance[i].second == INFINITY)
                    printf("To the verge number: %d - Rib weight: %s\n", i + 1, "infinity");
            else
                    printf("To the verge number: %d - Rib weight: %d\n", i + 1, distance[i].second);
    }

    printf("\nNumber of iterations: %d.\n", it2);
}
```

### 3.3.3 Алгоритм Флойда-Уоршелла

**Функция:** `void AlgFloydWarshall()`.

**Вход функции:** на вход функции подается матрица с отрицательными весами, матрица с положительными весами.

**Выход функции:** матрица расстояний между всеми вершинами.

**Пример операции:** см. рисунок 8.

**Описание работы функции:** *ключевая идея алгоритма* — разбиение процесса поиска кратчайших путей на фазы. Требуется, чтобы выполнялось `dopMatrix[i][j] == 0` для любых $i$. Если между какими-то вершинами ребра нет, то записывается $\infty$.

В цикле просматриваются все дуги графа. Если путь по дуге из вершины $i$ в вершину $j$ оказывается длиннее, чем путь из вершины $i$ в вершину $j$ через вершину $k$, то новый путь записывается, как наименьший.

**Сложность алгоритма:** $O(n^3)$, где $n$ — количество вершиин графа.

```
void AlgFloydWarshall()
    {
        printf("\n\n***Floyd-Warshall algorithm***\n\n");

        it3 = 0;
        int matr = 0;
        printf("Select an matrix:\n");
        printf("[1] - positive length matrix;\n");
        printf("[2] - negative length matrix.\n");
        matr = inputNumber("\nEntered value:", 1, 2);

        vector<vector<int>> dopMatrix;
        if (matr == 1)
                dopMatrix = MatrixVes;
        else if (matr == 2)
                dopMatrix = MatrixNegVes;

        for (int i = 0; i < numberVert; i++)
        {
                for (int j = 0; j < numberVert; j++)
                {
                        if (dopMatrix[i][j] == 0 && i != j)
                                dopMatrix[i][j] = INFINITY;
                }
        }

        for (int i = 0; i < numberVert; i++)
        {
                for (int j = 0; j < numberVert; j++)
                {
                        for (int k = 0; k < numberVert; k++)
                        {
                                if (dopMatrix[i][k] + dopMatrix[k][j] < dopMatrix[i][j])
                                        dopMatrix[i][j] = dopMatrix[i][k] + dopMatrix[k][j];
                                it3++;
                        }
                }
        }

        printf("\nFloyd-Warshall algorithm: \n");
        for (int i = 0; i < numberVert; i++)
        {
                printf("(");
                for (int j = 0; j < numberVert; j++)
                {
```

```
                        if (j != numberVert)
                        {
                                if (dopMatrix[i][j] >= 900)
                                        printf("%d\t", 0);
                                else
                                        printf("%d\t", dopMatrix[i][j]);
                        }
                        else
                        {
                                if (dopMatrix[i][j] >= 900)
                                        printf("%d", 0);
                                else
                                        printf("%d", dopMatrix[i][j]);
                        }
                }
                printf(")\n");
        }

        printf("\nNumber of iterations: %d.\n", it3);
    }
```

## 3.4 Поиск минимального остова

### 3.4.1 Алгоритм Прима

**Функция:** `void AlgPrim()`.

**Вход функции:** матрица весов с бесконечностями вместо нулей.

**Выход функции:** остовное дерево минимальной стоимости.

**Пример операции:** см. рисунок 10.

**Описание работы функции:** берется произвольная вершина и находится ребро, инцидентное данной вершине и обладающее наименьшей стоимостью. Найденное ребро и соединяемые им две вершины образуют дерево.

Рассматриваются ребра графа, один конец которых — это уже принадлежащая дереву вершина, а другой — еще не принадлежащая. Из этих ребер выбирается ребро наименьшей стоимости.

Выбираемое на каждом шаге ребро присоединяется к дереву. Рост дерева происходит до тех пор, пока не будут исчерпаны все вершины исходного графа.

**Сложность алгоритма:** $O(n^2)$.

```
void AlgPrim()
    {
        printf("\n\n***Prim's algorithm***\n\n");

        itP = 0;
```

```
    if (MinOstov != nullptr) // для не первого запуска очищаем MinOstov
            if (!MinOstov->empty())
                    MinOstov->clear();
    // алгоритм делает numberVert шагов, на каждом из которых выбирает вершину с наименьшей меткой min_e,
    // помечает ее, как used, а затем просматривает все ребра из вершины

    int cost = 0;
    vector<bool> used(numberVert);
    vector<int> min_e(numberVert, INFINITY); // вес наименьшего допустимого ребра из вершины
    vector<int> sel_e(numberVert, -1); // конец этого наименьшего ребра

    min_e[0] = 0;
    MinOstov = new vector<set<int>>(numberVert);
    for (int i = 0; i < numberVert; ++i)
    {
            int v = -1;
            for (int j = 0; j < numberVert; ++j)
            {
                    if (!used[j] && (v == -1 || min_e[j] < min_e[v]))
                            v = j;
            }
            if (min_e[i] == INFINITY)
            {
                    printf("\nNo minimum spanning tree!\n");
                    break;
            }

            used[v] = true; // означает, что вершина включена в остов

            for (int to = 0; to < numberVert; ++to)
            {
                    if (MatrixVesInf[v][to] < min_e[to])
                    {
                            min_e[to] = MatrixVesInf[v][to];
                            sel_e[to] = v;
                    }
                    itP++;
            }

            if (sel_e[v] != -1)
            {
                    MinOstov->at(sel_e[v]).insert(v);
                    MinOstov->at(v).insert(sel_e[v]);
                    if ((sel_e[v] + 1) > (v + 1))
                            printf("%d -> %d, weight: %d\n", v + 1, sel_e[v]+1, MatrixVes[sel_e[v]][v]);
                    else
                            printf("%d -> %d, weight: %d\n", sel_e[v]+1, v+1, MatrixVes[sel_e[v]][v]);
                    cost += MatrixVesInf[sel_e[v]][v];
            }
    }

    printf("\nMinimum spanning tree cost: %d\n", cost);
    printf("\nNumber of iterations: %d\n", itP);
}
```

### 3.4.2 Алгоритм Краскала

**Функция:** `void AlgKruskal()`.

**Вход функции:** матрица смежности.

**Выход функции:** матрица смежности минимального остова.

**Пример операции:** см. рисунок 11.

**Описание работы функции:** алгоритм Краскала заключается в том, что каждую вершину мы помещаем в свое множество.

Из всех ребер, добавление которых к уже имеющемуся множетсву не вызовет появление в нем цикла, выбирается ребро минимального веса. Затем мы проверяем принадлежат ли вершины ребра одному множеству.

Если нет, то давбавляем данное ребро в наше дерево, после добавления мы добавляем все вершины, которые принадлежали тому же множеству, что и вторая вершина ребра, в множество первой вершины.

Если же вершины уже принадлежат одному множеству, то переходим к следующему этапу цикла.

Подграф данного графа, содержащий все его вершины и найденное множество ребер, является его остовным деревом минимального веса.

Алгоритм Краскала заключается в сортировке всех ребер в порядке возрастания длины, и поочередному добавлению их в минимальный остов, если они соединяют различные компоненты связности.

**Сложность алгоритма:** $O(n \log n)$.

```
void AlgKruskal()
{
        printf("\n\n***Kruskal's algorithm***\n\n");

        itK = 0;
        int m = 0;
        if (MinOstov != nullptr)
                if (!MinOstov->empty())
                        MinOstov->clear();
        int count = 0;

        for (int i = 0; i < numberVert; i++)
                for (int j = 0; j < numberVert; j++)
                {
                        if (AdjacencyMatrixInf[i][j] > 0 && AdjacencyMatrixInf[i][j] < (INFINITY / 2))
                                m++;
```

```
        }
m = m / 2;

vector<pair<int, pair<int, int>>> g(m);
for (int i = 0; i < numberVert; i++)
{
        for (int j = i; j < numberVert; j++)
        {
                if (MatrixVesInf[i][j] < 500)
                {
                        g.at(count).first = MatrixVesInf[i][j];
                        g.at(count).second.first = i;
                        g.at(count).second.second = j;
                        count++;
                }
        }
}

int cost = 0;
vector<pair<int, int>> res;

sort(g.begin(), g.end());

vector<int> tree_id(numberVert);
for (int i = 0; i < numberVert; i++)
{
        tree_id[i] = i;
}
for (int i = 0; i < m; i++)
{
        int a = g[i].second.first;
        int b = g[i].second.second;
        int l = g[i].first;
        if (tree_id[a] != tree_id[b]) // для каждого ребра проверяем принадлежат ли его концы разным деревьям
        {
                cost += l;
                res.push_back(make_pair(a, b));

                int old_id = tree_id[a];
                int new_id = tree_id[b];

                for (int j = 0; j < numberVert; j++)
                {
                        if (tree_id[j] == old_id)
                                tree_id[j] = new_id;
                        itK++;
                }
        }
}

MinOstov = new vector<set<int>>(numberVert);
for_each(res.begin(), res.end(), [](pair<int, int> x)
        {
                MinOstov->at(x.first).insert(x.second);
                MinOstov->at(x.second).insert(x.first);
                if (x.first > x.second)
```

```
                                printf("%d -> %d, weight: %d\n", x.second + 1,
                                x.first + 1, MatrixVes[x.first][x.second]);
                        else
                                printf("%d -> %d, weight: %d\n", x.first + 1,
                                x.second + 1, MatrixVes[x.first][x.second]);
                });

        printf("\nMinimum spanning tree cost: %d\n", cost);
        printf("\nNumber of iterations: %d\n", itK);

}
```

### 3.4.3 Матричная теорема Кирхгофа

**Функция:** `void ThMatrixKirchhoff()`.

**Вход функции:** матрица смежности неорграфа.

**Выход функции:** число остовных деревьев.

**Пример операции:** см. рисунок 12.

**Описание работы функции:** в матрице смежности неориентированного графа заменяем каждый элемент на противоположный, а на диагонали ставим степень вершины. Тогда, согласно матричной теореме Кирхгофа, все алгебраические дополнения этой матрицы равны между собой, а также равны количеству остовных деревьев этого графа. Далее удаляем первую строку и первый столбец этой матрицы, и модуль ее определителя будет равен искомому количеству.

```
void ThMatrixKirchhoff()
    {
        printf("\n\n***Kirchhoff's matrix theorem***\n\n");

        if (numberVert > 13)
                printf("Execution time is too long!\n");

        int count = 0;
        vector<vector<int>> buffer(numberVert);
        for (int i = 0; i < numberVert; i++)
        {
                buffer[i] = vector<int>(numberVert);
                for (int j = 0; j < numberVert; j++)
                {
                        if (AdjacencyMatrixInf[i][j] == INFINITY)
                                buffer[i][j] = 0;
                        else
                        {
                                buffer[i][j] = -AdjacencyMatrixInf[i][j];
                                count++;
                        }
                }
                buffer[i][i] = count;
                count = 0;
```

```
    }

    printf("Kirchhoff matrix:\n");
    for (int i = 0; i < numberVert; i++)
    {
            printf("(");
            for (int j = 0; j < numberVert; j++)
            {
                    if(j != numberVert)
                            printf("%d\t", buffer[i][j]);
                    else
                            printf("%d", buffer[i][j]);
            }
            printf(")\n");
    }

    int minor = 1;
    vector<vector<int>> arr(numberVert - 1);
    for (int i = 0; i < numberVert - 1; i++)
    {
            arr[i] = vector<int>(numberVert - 1);
    }

    for (int i = 1; i < numberVert; i++)
    {
            for (int j = 1; j < numberVert; j++)
            {
                    arr[i - 1][j - 1] = buffer[i][j];
            }
    }

    for (int i = 0; i < numberVert - 1; i++)
            minor += arr[0][i] * pow(-1, (1 + i + 1)) * findMinor(arr, 0, i, numberVert - 1);

    if (numberVert != 2)
    {
            minor -= 1;
    }
    printf("\nThe number of spanning trees by Kirchhoff's theorem: %d\n", abs(minor));
}
```

## 3.5 Код Прюфера

### 3.5.1 Кодирование Прюфера

**Функция:** `void EnPrufer()`.

**Вход функции:** минимальное остовное дерево.

**Выход функции:** код Прюфера.

**Пример операции:** см. рисунок 13.

**Описание работы функции:** в цикле выбирается лист с наименьшим номером, в код

Прюфера добавляется номер вершины, связанной с этим листом. Вся процедура повторяется $n - 2$ раза.

Кроме того, код Прюфера не может быть пустым, поэтому для двух вершин выбрасывается исключение, в котором единственное значение, которое хранится в коде, равно 0.

```
void EnPrufer()
    {
        printf("\n\n***Prufer encoding***\n\n");

        if (MinOstov == nullptr)
                printf("Error! No minimum spanning tree found! Search for a spanning tree!\n");

        if (MinOstov->size() == 2)
        {
                printf("The graph has two vertices. Prufer's code is not possible!\n");
                printf("Code: %d\n\n", 0);
        }


        if (MinOstov->size() != 2)
        {
                if (PruferCode.size() == (numberVert - 2))
                {
                        printf("Code: ");
                        for (int i = 0; i < numberVert - 2; i++)
                        {
                                printf("%d ", PruferCode.at(i) + 1);
                        }
                        printf("\n");
                }
        }

        vector<bool> visited(numberVert);

        while (PruferCode.size() != numberVert - 2)
        {
                for (int i = 0; i < numberVert; i++)
                {
                        if (!visited[i])
                        {
                                if (MinOstov->at(i).size() == 1)
                                {
                                        visited[i] = true;
                                        for (int j = 0; j < numberVert; j++)
                                        {
                                                MinOstov->at(i).clear();
                                                auto iter = MinOstov->at(j).find(i);
                                                if (iter != MinOstov->at(j).end())
                                                {
                                                        MinOstov->at(j).erase(*iter);
                                                        PruferCode.push_back(j);
                                                        break;
                                                }
```

```
                                        }
                                        break;
                                }
                        }
                }
        }

        if (MinOstov->size() != 2)
        {
                printf("Code: ");
                for (int i = 0; i < numberVert - 2; i++)
                {
                        printf("%d ", PruferCode.at(i) + 1);
                }
                printf("\n");
        }
    }
```

### 3.5.2 Декодирование Прюфера

**Функция:** `void DePrufer()`.

**Вход функции:** код Прюфера.

**Выход функции:** минимальное остовное дерево.

**Пример операции:** см. рисунок 14.

**Описание работы функции:** по построенному коду Прюфера можно восстановить исходное остовное дерево по следующему алгоритму: берется первый элемент кода Прюфера и по всем вершинам дерева производится поиск наименьшей вершины, не содержащейся в коде. Найденная вершина и текущий элемент кода составляют ребро дерева.

```
void DePrufer()
    {
        printf("\n\n***Prufer decoding***\n\n");

        if (MinOstov == nullptr)
                printf("Error! No minimum spanning tree found! Search for a spanning tree!\n");

        if (MinOstov->size() == 2)
                printf("Decode:\n%d -> %d\n", 1,2);

        if (PruferCode.size() < (numberVert - 2))
                printf("Error! Make Prufer coding!\n");

        if (MinOstov->size() != 2)
        {
                deque<int> v;
                bool b = false;

                printf("Decode:\n");
```

```
            for (int i = 0; i < numberVert; i++)
            {
                    v.push_back(i);
            }

            int index = 0;
            while (v.size() != 2)
            {
                    bool b = false;
                    for (int i = 0; i < PruferCode.size(); i++)
                    {
                            if (PruferCode.at(i) == v.at(index))
                                    b = true;
                    }
                    if (b)
                    {
                            index++;
                            b = false;
                    }
                    else
                    {
                            auto iter = v.begin() + index;
                            printf("%d -> %d\n", v.at(index) + 1, PruferCode.at(0) + 1);
                            v.erase(iter);
                            PruferCode.erase(PruferCode.begin());
                            index = 0;
                            b = false;
                    }
            }
            printf("%d -> %d\n", v.at(0) + 1, v.at(1) + 1);
    }
}
```

## 3.6 Нахожение максимального потоков и потока минимальной стоимости

### 3.6.1 Алгоритм Форда-Фалкерсона

**Функция:** `int FordFulkerson(int s, int t)`.

**Вход функции:** матрица пропускных способностей, сток и исток сети.

**Выход функции:** максимальный поток.

**Пример операции:** см. рисунок 16.

**Описание работы функции:** основная задача алгоритма Форда-Фалкерсона — нахождение максимального потока в транспортной сети. Идея алгоритма заключается в следующем: 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.

```
int FordFulkerson(int s, int t)
    {
        if ((CostMatrix == nullptr) || (CapacityMatrix == nullptr)) {
                printf("No capacity and cost matrices have been created!\n");
                return 0;
        }
        else {
                int u, v;

                Potok = new int* [numberVert];

                for (u = 0; u < numberVert; u++) {
                        Potok[u] = new int[numberVert];
                        for (v = 0; v < numberVert; v++) {
                                Potok[u][v] = 0;
                        }
                }


                int** rGraph = new int* [numberVert];

                for (u = 0; u < numberVert; u++) {
                        rGraph[u] = new int[numberVert];
                        for (v = 0; v < numberVert; v++) {
                                if (CapacityMatrix[u][v] > (INFINITY / 2)) {
                                        rGraph[u][v] = 0;
                                }
                                else {
                                        rGraph[u][v] = CapacityMatrix[u][v];
                                }
                        }
                }

                int* parent = new int[numberVert];

                int max_flow = 0;

                while (bfs(rGraph, s, t, parent))
                {

                        int path_flow = INT_MAX;
                        for (v = t; v != s; v = parent[v])
                        {
                                u = parent[v];
                                path_flow = min(path_flow, rGraph[u][v]);

                        }

                        for (v = t; v != s; v = parent[v])
                        {
                                u = parent[v];
                                rGraph[u][v] -= path_flow;
                                rGraph[v][u] += path_flow;
                                if (Potok[u][v] != 0)
                                        Potok[u][v] += path_flow;
                                else
```

```
                                Potok[u][v] = path_flow;
                        }

                        max_flow += path_flow;
                }


                if (FFl)
                {
                        printf("\nFlow matrix\n");
                        for (int i = 0; i < numberVert; i++)
                        {
                                printf("(");
                                for (int j = 0; j < numberVert; j++)
                                {
                                        if(j != numberVert -1)
                                                printf("%d\t", Potok[i][j]);
                                        else
                                                printf("%d", Potok[i][j]);
                                }
                                printf(")\n");
                        }
                }

                FFl = false;

                for (int i = 0; i < numberVert; i++) {
                        delete[] rGraph[i];
                        delete[] Potok[i];
                }
                delete[] rGraph;
                delete[] Potok;
                delete[] parent;

                return max_flow;
        }
}
```

### 3.6.2 Поиск заданного потока минимальной стоимости

**Функция:** `void MinCostMaxFlow(int b)`.

**Вход функции:** матрица пропускных способностей, максимальный поток.

**Выход функции:** матрица потоков.

**Пример операции:** см. рисунок 17.

**Описание работы функции:** заданное значение по условию устанавливаетя, как $\frac{2}{3}$ от максимального потока в сети.

Сначала ищется минимальный по весу путь из истока в сток с помощью алгоритма Беллмана-Форда. Потом вычисляется максимальный поток по найденному пути. Полу-

ченное значение прибавляется к значению потока каждой дуги найденного увеличивающего пути и отнимается от заданного значения потока. И так алгоритм повторяется, пока заданное значение потока не станет равным нулю.

```
void MinCostMaxFlow(int b)
{
	FFl = false;

	int** matrPotok = new int* [numberVert];
	for (int i = 0; i < numberVert; i++)
	{
		matrPotok[i] = new int[numberVert];
		for (int j = 0; j < numberVert; j++)
		{
			matrPotok[i][j] = 0;
		}
	}

	if ((CostMatrix == nullptr) || (CapacityMatrix == nullptr))
	{
		printf("No capacity and cost matrices have been created!\n");
		return;
	}
	else
	{
		int maxFlow = 0;
		int flow = 0;
		int sigma = INFINITY;
		int result = 0;
		int** modifiedCostArr;

		map<int, pair<vector<int>, int>> tempMap;
		map<vector<int>, pair<int, int>> temporary;
		set<pair<int, int>> resultVert;
		vector<vector<int>> f(numberVert);
		vector<int> k;

		for (int i = 0; i < numberVert; i++)
		{
			for (int j = 0; j < numberVert; j++)
			{
				if (AdjacencyMatrixInf[i][j] > 0 && AdjacencyMatrixInf[i][j] < (INFINITY / 2))
					m++;
			}
		}

		modifiedCostArr = new int* [numberVert];
		for (int i = 0; i < numberVert; i++)
		{
			modifiedCostArr[i] = new int[numberVert];
			for (int j = 0; j < numberVert; j++)
			{
				f.at(i).push_back(0);
				modifiedCostArr[i][j] = CostMatrix[i][j];
			}
		}
```

```
while (maxFlow < b)
{
        vector<int> path = BellmanFordPath(modifiedCostArr);
        for (int i = 0; i < path.size() - 1; i++)
        {
                sigma = min(sigma, CapacityMatrix[path.at(i)][path.at(i + 1)]);
                if (resultVert.find(pair<int, int>(path.at(i), path.at(i + 1))) == resultVert.end())
                {
                        resultVert.insert(pair<int, int>(path.at(i), path.at(i + 1)));
                }
        }

        flow = min(sigma, (b - maxFlow));
        k.push_back(flow);
        int tempCost = 0;

        for (int i = 0; i < path.size() - 1; i++)
        {
                tempCost += CostMatrix[path.at(i)][path.at(i + 1)];
        }

        if (temporary.find(path) == temporary.end())
        {
                temporary[path] = pair<int, int>(flow, tempCost);
        }
        else
        {
                temporary[path].first += flow;
        }

        for (int i = 0; i < path.size() - 1; i++)
        {
                f.at(path[i]).at(path[i + 1]) += flow;
                if (f.at(path[i]).at(path[i + 1]) == CapacityMatrix[path[i]][path[i + 1]])
                {
                        modifiedCostArr[path[i]][path[i + 1]] = INFINITY;
                        modifiedCostArr[path[i + 1]][path[i]] = INFINITY;
                }
                else
                {
                        if (f.at(path[i]).at(path[i + 1]) >= 0)
                        {
                                modifiedCostArr[path[i]][path[i + 1]] =
                                CostMatrix[path[i]][path[i + 1]];
                                modifiedCostArr[path[i + 1]][path[i]] =
                                -modifiedCostArr[path[i]][path[i + 1]];
                        }
                }
        }
        maxFlow += flow;
}
int min = INFINITY;
vector<pair<int, pair<vector<int>, int>>> tempVect;

for (auto i = temporary.begin(); i != temporary.end(); i++)
```

```
{
        tempVect.push_back(pair<int, pair<vector<int>, int>>(i->second.second,
        pair<vector<int>, int>(i->first, i->second.first)));
}

for (int i = 0; i < tempVect.size() - 1; i++)
{
        for (int j = 0; j < tempVect.size() - 1; j++)
        {
                if (tempVect[j].first > tempVect[j + 1].first)
                        swap(tempVect[j], tempVect[j + 1]);
        }
}

vector<int> vecPotok;
vector<int> vecMatr;
int ur = 0;

for (int i = 0; i < tempVect.size(); i++)
{
        printf("\n\nFlow path: ");
        for (int j = 0; j < tempVect[i].second.first.size() - 1; j++)
        {
                printf("%d - ", tempVect[i].second.first[j] + 1);
                vecMatr.push_back(tempVect[i].second.first[j]);
        }

        printf("%d", tempVect[i].second.first[tempVect[i].second.first.size() - 1] + 1);
        vecMatr.push_back(tempVect[i].second.first[tempVect[i].second.first.size() - 1]);

        printf("\n\nFlow throught the path found: %d\n", tempVect[i].second.second);
        for (int l = 0; l < vecMatr.size() - 1; l++)
        {
                if (vecMatr[l] < vecMatr[l + 1] && matrPotok[vecMatr[l]][vecMatr[l + 1]] == 0)
                        matrPotok[vecMatr[l]][vecMatr[l + 1]] = tempVect[i].second.second;
        }
        vecPotok.push_back(tempVect[i].second.second);
}

for_each(resultVert.begin(), resultVert.end(), [&result, &f](pair<int, int>x)
{result += f.at(x.first).at(x.second) * CostMatrix[x.first][x.second]; });

printf("\n\nGiven flow: %d", b);
printf("\n\nMinimum cost of a given stream: %d", result);
printf("\n\nCost matrix:\n");

for (int i = 0; i < numberVert; i++)
{
        printf("(");
        for (int j = 0; j < numberVert; j++)
        {
                if (j != numberVert - 1)
                {
                        if(CostMatrix[i][j] == INFINITY)
                                printf("%d\t", 0);
                        else
```

```
                                        printf("%d\t", CostMatrix[i][j]);
                                }
                                else
                                {
                                        if (CostMatrix[i][j] == INFINITY)
                                                printf("%d", 0);
                                        else
                                                printf("%d", CostMatrix[i][j]);
                                }
                        }
                        printf(")\n");
                }

                printf("\n\nCapacity matrix:\n");
                for (int i = 0; i < numberVert; i++)
                {
                        printf("(");
                        for (int j = 0; j < numberVert; j++)
                        {
                                if (j != numberVert - 1)
                                        printf("%d\t", matrPotok[i][j]);
                                else
                                        printf("%d", matrPotok[i][j]);
                        }
                        printf(")\n");
                }


                for (int i = 0; i < numberVert; i++) {
                        delete[] modifiedCostArr[i];
                        delete[] matrPotok[i];
                }
                delete[] modifiedCostArr;
                delete[] matrPotok;
        }
}
```

## 3.7 Эйлеровы и Гамильтоновы графы

### 3.7.1 Поиск Эйлерова цикла

**Функция:** `void EulerCycle()`.

**Вход функции:** матрица смежности исходного графа.

**Выход функции:** матрица смежности эйлерова графа.

**Пример операции:** см. рисунок 19.

**Описание работы функции:** изначально необходимо убедиться в том, что граф является Эйлеровым. Для этого проверяем степени вершин (*условие:* степени всех вершин должны быть четными). Если же граф не является Эйлеровым, то нужно его привести к такому посредством добавления и/или удаления ребер, связывающих те вершины, степень

которых нечетная.

Мы не проходим по ребру, если удаление этого ребра приводит к разбиению графа на две связные компоненты, то есть необходимо проверять является ли ребро мостом или нет.

```
void EulerCycle()
{
        int vert = numberVert;

        if (vert == 2)
        {
                printf("There are two vertices in the graph! Euler cycle is impossible!\n");
                return;
        }

        GraphEuler = vector<vector<int>>(vert, vector<int>(vert, 0));

        for (int i = 0; i < vert; i++)
        {
                for (int j = i; j < vert; j++)
                {
                        GraphEuler[i][j] = GraphEuler[j][i] = AdjacencyMatrix[i][j];
                }
        }

        int eU = IsItEuler(GraphEuler);

        bool flag = 0;

        while (!eU)
        {
                flag = 1;
                vector<int> degree1;
                for (int i = 0; i < numberVert; i++)
                {
                        int sst = 0;
                        for (int j = 0; j < numberVert; j++)
                        {
                                if (GraphEuler[i][j] != 0)
                                        sst++;
                        }
                        degree1.push_back(sst);
                }

                srand(time(NULL));

                bool isChanged = false;

                for (int i = 0; i < deg.size(); i++)
                {
                        for (int j = i + 1; j < deg.size(); j++)
                        {
                                if (GraphEuler[deg[i]][deg[j]] == 0)
                                {
                                        GraphEuler[deg[i]][deg[j]] = 1;
                                        GraphEuler[deg[j]][deg[i]] = 1;
```

```
            printf("Add a rib: %d - %d\n", deg[i]+1, deg[j]+1);
            isChanged = true;
            break;
        }

        if (GraphEuler[deg[i]][deg[j]] == 1)
        {
            GraphEuler[deg[i]][deg[j]] = 0;
            GraphEuler[deg[j]][deg[i]] = 0;

            printf("Remove a rib: %d - %d\n", deg[i]+1, deg[j]+1);

            if (degree1[deg[i]] == 1 && degree1[deg[j]] % 2 != 0 && degree1[deg[j]] != 1)
            {
                if (deg[i] != numberVert - 1)
                {
                    GraphEuler[deg[i]][numberVert - 1] = 1;
                    GraphEuler[numberVert - 1][deg[i]] = 1;
                    printf("Add a rib: %d - %d\n", deg[i]+1, numberVert - 1+1);
                }
                else
                {
                    GraphEuler[deg[i]][0] = 1;
                    GraphEuler[0][deg[i]] = 1;
                    printf("Add a rib: %d - %d\n", 0+1, deg[i]+1);
                }

                if (deg[i] != numberVert - 2)
                {
                    GraphEuler[deg[i]][numberVert - 2] = 1;
                    GraphEuler[numberVert - 2][deg[i]] = 1;
                    printf("Add a rib: %d - %d\n", deg[i]+1, numberVert - 2+1);
                }
                else
                {
                    GraphEuler[deg[i]][0] = 1;
                    GraphEuler[0][deg[i]] = 1;
                    printf("Add a rib: %d - %d\n", 0+1, deg[i]+1);
                }

            }

            if (degree1[deg[j]] == 1 && degree1[deg[i]] % 2 != 0 && degree1[deg[i]] != 1)
            {
                if (deg[j] != numberVert - 1 && GraphEuler[deg[i]][deg[j]] == 0)
                {
                    GraphEuler[deg[j]][numberVert - 1] = 1;
                    GraphEuler[numberVert - 1][deg[i]] = 1;
                    printf("Add a rib: %d - %d\n", deg[j]+1, numberVert - 1+1);
                }
                else
                {
                    GraphEuler[deg[j]][0] = 1;
                    GraphEuler[0][deg[j]] = 1;
                    printf("Add a rib: %d - %d\n", 0+1, deg[j]+1);
                }
```

```
                                        if (deg[j] != numberVert - 2 && GraphEuler[deg[i]][deg[j]] == 0)
                                        {
                                                GraphEuler[deg[j]][numberVert - 2] = 1;
                                                GraphEuler[numberVert - 2][deg[j]] = 1;
                                                printf("Add a rib: %d - %d\n", deg[j]+1, numberVert - 2+1);
                                        }
                                        else
                                        {
                                                GraphEuler[deg[j]][0] = 1;
                                                GraphEuler[0][deg[j]] = 1;
                                                printf("Add a rib: %d - %d\n", 0+1, deg[j]+1);
                                        }
                                }
                                isChanged = true;
                                break;
                        }
                }
                if (isChanged)
                        break;
        }
        deg.clear();

        eU = IsItEuler(GraphEuler);
    }

    if (flag)
    {
        printf("\n\nChanged adjacency matrix:\n");

        for (int i = 0; i < vert; i++)
        {
                printf("(");
                for (int j = 0; j < vert; j++)
                {
                        if (j != vert - 1)
                                printf("%d\t", GraphEuler[i][j]);
                        else
                                printf("%d", GraphEuler[i][j]);
                }
                printf(")\n");
        }
    }

    deg.clear();
}
```

### 3.7.2 Поиск Гамильтонова цикла

**Функция:** `void GamiltonCycle()`.

**Вход функции:** матрица смежности исходного графа.

**Выход функции:** матрица смежности Гамильтонова графа.

**Пример операции:** см. рисунок 20.

**Описание работы функции:** изначально необходимо убедиться в том, что граф является Гамильтоновым. Для этого производился поиск Гамильтонова цикла. Если в графе минимальная степень вершины не меньше двух, две вершины с минимальными степенями соединяются ребром и производится поиск Гамильтонова цикла.

```
void GamiltonCycle()
{
        int vert = numberVert;
        int c = 0;
        bool flagG = true;
        bool flag = false;
        bool add = false;
        bool first = 1;
        bool f = false;
        vector<int> v0;

        GraphGamilton = vector<vector<int>>(vert, vector<int>(vert, 0));

        if (numberVert < 3)
        {
                printf("The graph cannot be made Hamiltonian! Less than three vertices!\n");
                return;
        }

        for (int i = 0; i < vert; i++)
        {
                for (int j = i; j < vert; j++)
                {
                        GraphGamilton[i][j] = GraphGamilton[j][i] = AdjacencyMatrixInf[i][j];
                }
        }

        while (flagG)
        {

                if (IsItGamilton(GraphGamilton))
                {
                        printf("\nThe graph is hamiltonian!\n");
                        flagG = false;
                        flag = true;
                        break;
                }
                else
                {
                        printf("\nIt's necessary to complete the graph!\n");
                        add = false;
                        while (!flag)
                        {
                                for (int i = 0; i < numberVert; i++)
                                {
                                        if (path[i].second < numberVert / 2 && !add)
                                        {
                                                v0.push_back(path[i].first);
```

```
                        }
                    }

                    for (int i = 0; i < v0.size(); i++)
                    {
                        for (int j = 0; j < numberVert; j++)
                        {
                            if (!add)
                            {
                                if (GraphGamilton[v0[i]][j] == 999 && v0[i] != j)
                                {
                                    if (v0.size() == 2
                                    && GraphGamilton[v0[i]][v0[i + 1]] == INFINITY)
                                    {
                                        GraphGamilton[v0[i]][v0[i + 1]] = 1;
                                        GraphGamilton[v0[i + 1]][v0[i]] = 1;
                                        if (v0[i] + 1 < v0[i + 1] + 1)
                                            printf("\nAdd a rib: %d - %d\n",
                                            v0[i] + 1, v0[i + 1] + 1);
                                        else
                                            printf("\nAdd a rib: %d - %d\n",
                                            v0[i+1] + 1, v0[i] + 1);
                                        add = true;
                                        f = true;
                                    }
                                    else
                                    {
                                        GraphGamilton[v0[i]][j] = 1;
                                        GraphGamilton[j][v0[i]] = 1;
                                        if (v0[i] + 1 < j + 1)
                                            printf("\nAdd a rib: %d - %d\n",
                                            v0[i] + 1, j + 1);
                                        else
                                            printf("\nAdd a rib: %d - %d\n",
                                            j + 1, v0[i] + 1);
                                        add = true;
                                        f = true;
                                    }
                                }
                            }
                        }
                    }

                    flag = IsItGamilton(GraphGamilton);
                    add = false;
                    v0.clear();
                }
            }
        }

        if (f)
        {
            printf("\n\nChanged adjacency matrix:\n");

            for (int i = 0; i < vert; i++)
            {
```

```
                        printf("(");
                        for (int j = 0; j < vert; j++)
                        {
                                if (j != vert - 1)
                                {
                                        if(GraphGamilton[i][j] == 999)
                                                printf("%d\t", 0);
                                        else
                                                printf("%d\t", GraphGamilton[i][j]);
                                }
                                else
                                {
                                        if (GraphGamilton[i][j] == 999)
                                                printf("%d", 0);
                                        else
                                                printf("%d", GraphGamilton[i][j]);
                                }
                        }
                        printf(")\n");
                }
        }
}
```

### 3.7.3 Задача коммивояжера

**Функция:** `void TSP()`.

**Вход функции:** матрица смежности Гамильтонова графа.

**Выход функции:** минимальный Гамильтонов цикл.

**Пример операции:** см. рисунок 21.

**Описание работы функции:** задача коммивояжера решается полным перебором всех перестановок последовательности вершин графа. Очередная перестановка — это последовательность вершин в пути. Если такой путь есть в данном графе, то существует ребро из первой вершины во вторую вершину пути и ребро из последней вершины в первую, тогда эта последовательность вершин — Гамильтонов цикл — записывается в файл.

Из всех найденных Гамильтоновых циклов выбирается один с минимальной длиной.

```
void TSP()
    {
        printf("\n\n***Travelling salesman problem (TSP)***\n");
        GraphGamiltonVes = vector<vector<int>>(numberVert, vector<int>(numberVert, 0));
        for (int i = 0; i < numberVert; i++)
        {
                for (int j = i + 1; j < numberVert; j++)
                {
                        if (GraphGamilton[i][j] = 0)
                        {
                                GraphGamiltonVes[i][j] = 0;
                                GraphGamiltonVes[j][i] = 0;
```

```
                }
                else
                {
                        GraphGamiltonVes[i][j] = rand() % 9 + 1;
                        GraphGamiltonVes[j][i] = rand() % 8 + 1;
                }
        }
}

ofstream f;
f.open("Gamilton.txt", std::ofstream::out | std::ofstream::trunc);
f.close();

printf("\nAdjacency matrix of a hamiltonian graph:\n");
for (int i = 0; i < numberVert; i++)
{
        printf("(");
        for (int j = 0; j < numberVert; j++)
        {
                if (j != numberVert - 1)
                        printf("%d\t", GraphGamiltonVes[i][j]);
                else
                        printf("%d", GraphGamiltonVes[i][j]);
        }
        printf(")\n");
}

if (numberVert < 3)
{
        printf("Less than 3 vertex! It's impossible to solve the TSP!\n");
        return;
}

if (numberVert > 12)
{
        printf("Solving the TSP will take too much time!\n");
        return;
}

int cost = 0;
int minCost = INFINITY * numberVert;
deque<int> que;
deque<int> res;
//ofstream f;

f.open("Gamilton.txt", ofstream::out);

bool isGamiltonC = true;
for (int i = 0; i < numberVert; i++)
{
        que.push_back(i);
}

do
{
        isGamiltonC = true;
```

```
            cost = 0;
            if (GraphGamilton[que.at(numberVert - 1)][que.at(0)] > (INFINITY / 2))
            {
                    isGamiltonC = false;
            }
            else
            {
                    for (int i = 0; i < numberVert - 1; i++)
                    {
                            if (GraphGamilton[que.at(i)][que.at(i + 1)] > (INFINITY / 2))
                            {
                                    isGamiltonC = false;
                            }
                            else
                            {
                                    cost += GraphGamiltonVes[que.at(i)][que.at(i + 1)];
                            }
                    }
            }

            if (isGamiltonC)
            {
                    cost += GraphGamiltonVes[que.at(numberVert - 1)][que.at(0)];
                    f << "Hamiltonian cycle:" << endl;
                    f << que.at(0) + 1;
                    for (int i = 1; i < que.size(); i++)
                    {
                            f << " - " << que.at(i) + 1;
                    }

                    f << " - " << que.at(0) + 1;
                    f << "\n";
                    f << "\n\nWeight: " << cost << endl << endl;
                    if (cost < minCost)
                    {
                            res.clear();
                            res.resize(que.size());
                            copy(que.begin(), que.end(), res.begin());
                            minCost = cost;
                    }
            }
    } while (next_permutation(que.begin(), que.end()));

    f.close();

    printf("\nMinimum Hamiltonian cycle: ");
    printf("%d", res.at(0) + 1);
    for (int i = 1; i < res.size(); i++)
    {
            printf(" - %d", res.at(i) + 1);
    }
    printf(" - %d", res.at(0) + 1);
    printf("\n\nWeight: %d\n\n", minCost);
}
```

# 4 Результаты работы программы

- На рис. 2-21 представлена работа программы, состоящим из 4 вершин.

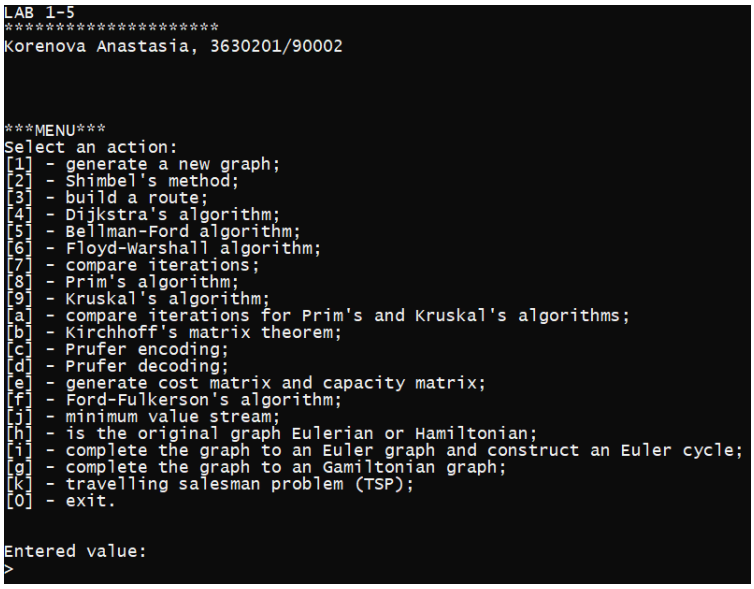


Рис. 2: Пользовательское меню

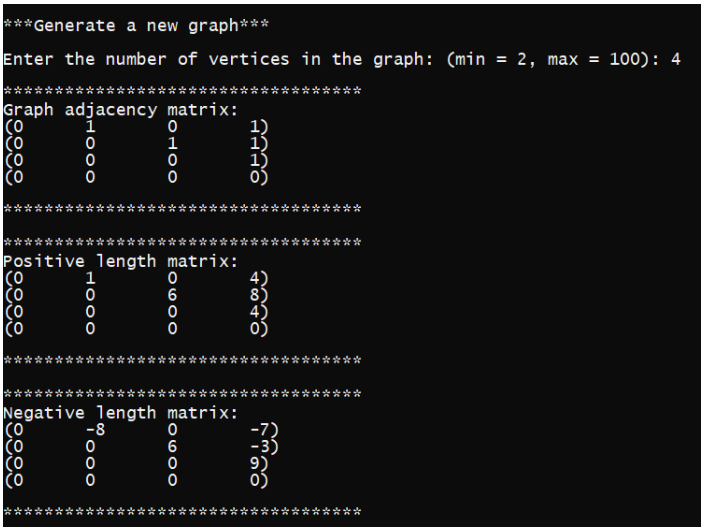


Рис. 3: Генерация нового графа, количество вершин задается пользователем

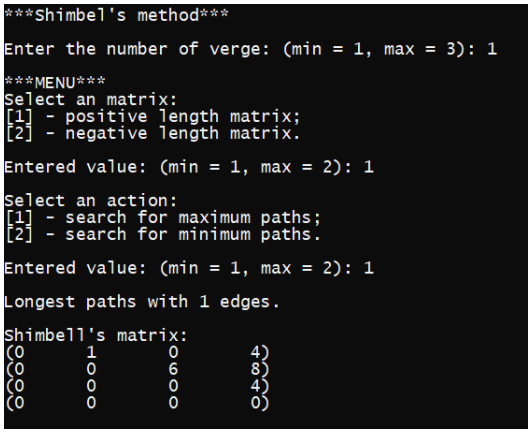


Рис. 4: Метод Шимбелла. Поиск максимального пути, использована матрица с положительными весами

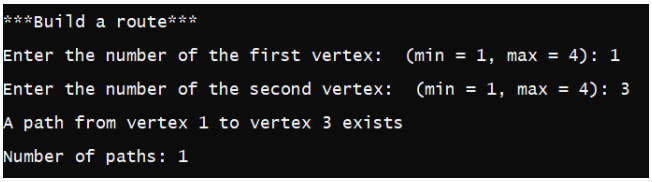


Рис. 5: Существование пути из вершины 1 в вершину 3

- Следующие три рисунка (рис.6 - рис.8) показывают алгоритмы работы по поиску кратчайших путей в графе.

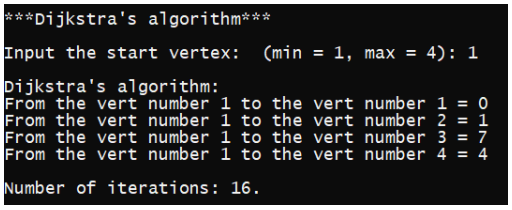


Рис. 6: Алгоритм Дейкстры

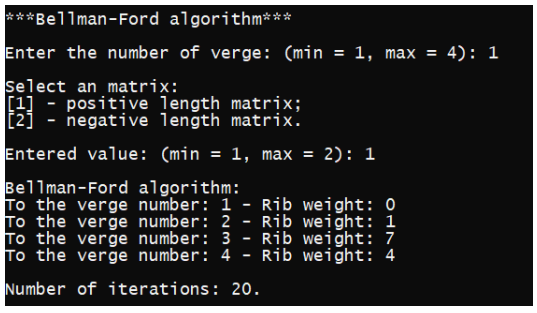


Рис. 7: Алгоритм Беллмана-Форда

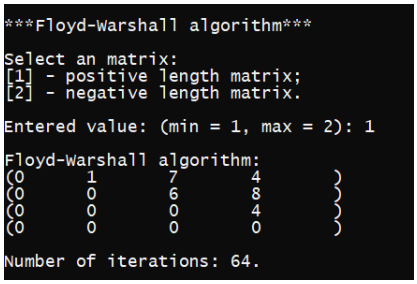


Рис. 8: Алгоритм Флойда-Уоршелла

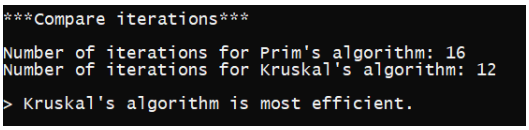


Рис. 9: Сравнение количества итераций алгоритмов Дейкстры, Беллмана-Форда и Флойда-Уоршелла

- Следующие два рисунка (рис.10 - рис.11) показывают работу алгоритмов по поиску минимального остова в графе. А также, поиск количества остовных деревьев по матричной теореме Кирхгофа.


Рис. 10: Алгоритм Прима

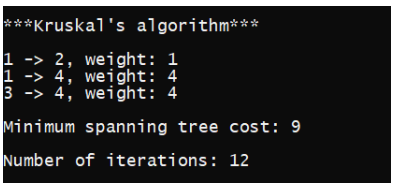


Рис. 11: Алгоритм Краскала

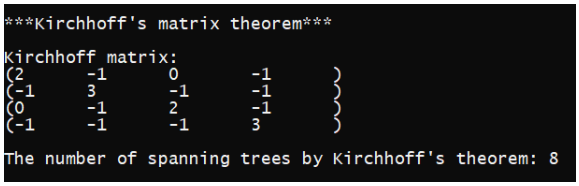


Рис. 12: Матричная теорема Кирхгофа

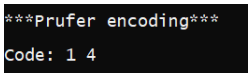


Рис. 13: Кодирование Прюфера

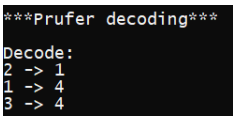


Рис. 14: Декодирование кода Прюфера

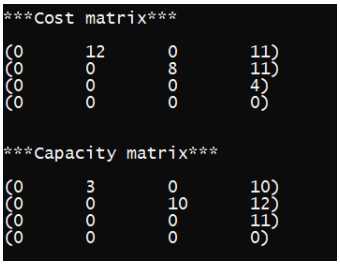


Рис. 15: Генерация матриц пропускной способности и стоимостей

- Рис.16 - рис.17 показывают работу программы по поиску максимального потока по алгоритму Форда-Фалкерсона, а также поиск максимального потока минимальной стоимости.

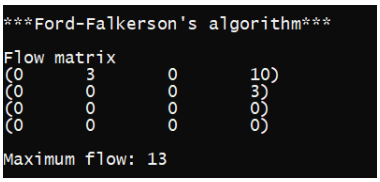


Рис. 16: Алгоритм Форда-Фалкерсона

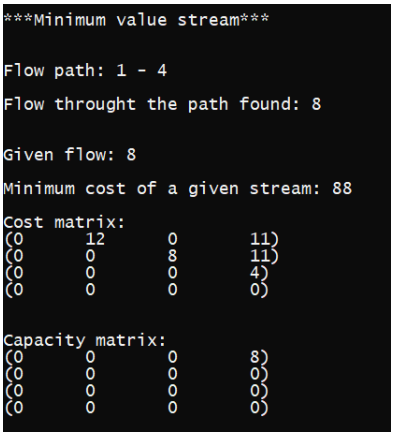


Рис. 17: Поиск максимального потока минимальной стоимости

- На рис.18 - рис.21 представлена работа программы с Эйлеровыми и Гамильтоновыми циклами.

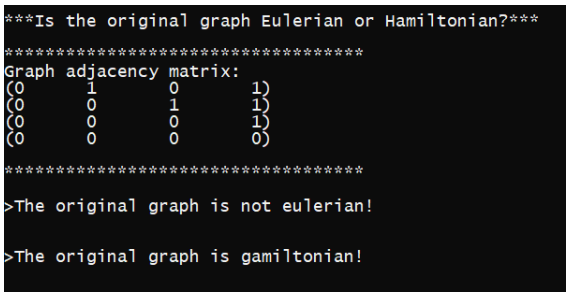


Рис. 18: Проверка является ли исходный граф гамильтоновым или эйлеровым

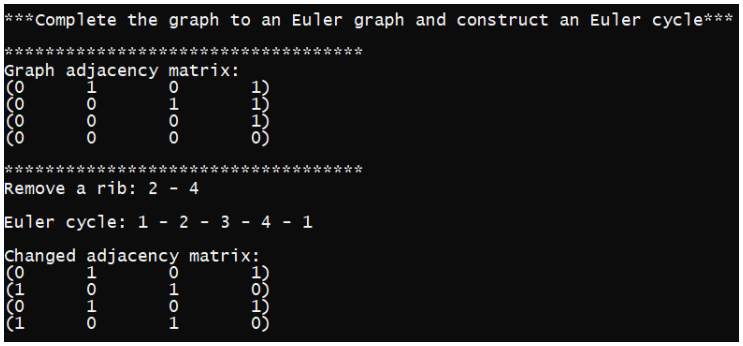


Рис. 19: Построение эйлерова цикла

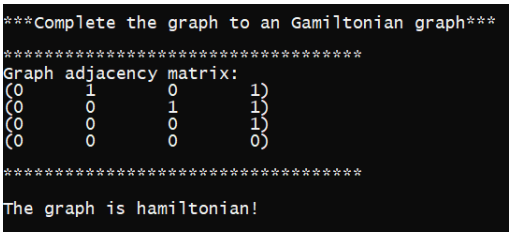


Рис. 20: Построение гамильтонова цикла

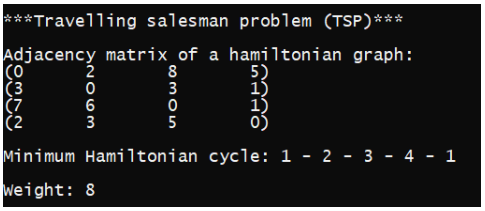


Рис. 21: Решение задачи коммивояжера

# Заключение

Результатом проделанной работы стала программа, выполняющая все поставленные задачи: происходит генерация графа в соответствии с заданным распределением; поиск экстремальных путей методом Шимбелла; определение возможности построения маршрута; поиск кратчайших путей с помощью алгоритма Дейкстры, Беллмана-Форда, Флойда-Уоршелла; построение минимального по весу остова с помощью алгоритмов Прима и Краскала; поиск числа остовных деревьев, основываясь на матричной теореме Кирхгофа; кодирование с помощью кода Прюфера; поиск максимального потока с помощью алгоритма Форда-Фалкерсона; поиск потока минимальной стоимости; построение Эйлеровых и Гамильтоновых циклов, а также решение задачи коммивояжера.

Кроме того, в программе реализованы несколько алгоритмов, выполняющих одну и ту же задачу.

- Поиск кратчайших путей (алгоритмы Дейкстры, Беллмана-Форда, Флойда-Уоршелла).
  - **Алгоритм Дейкстры** является самым быстрым (временная сложность ($O(n^2)$)), но не применяется при наличии отрицательных весов дуг.
  - **Алгоритм Беллмана-Форда** медленнее, чем алгоритм Дейкстры ($O(n * m)$), так как рёбер в графе обычно больше, чем вершин. Преимущество данного алгоритма заключается в том, что он может работать с рёбрами отрицательного веса, при условии, что в графе отсутствуют отрицательные циклы.
  - **Алгоритм Флойда-Уоршелла** имеет самую высокую временную сложность среди всех перечисленных ($O(n^3)$). Преимуществом данного алгоритма является то, что он может работать с ребрами как положительного, так и отрицательного веса.
- Построение минимального по весу остова (алгоритмы Прима и Краскала).
  - **Алгоритм Прима**. Недостаток: необходимость просматривать все оставшиеся ребра для поиска минимального. Сложнее в реализации, чем алгоритм Краскала.
  - **Алгоритм Краскала.** Недостаток: необходимость в реализации хранения и сортировки списка ребер.

**Достоинства программы:**

- в программе реализованы методы обхода в глубину и ширину и методы нахождения кратчайших путей, которые могут использоваться в других алгоритмах;
- при решении задачи коммивояжёра полным перебором для оптимизации алгоритма каждая новая перестановка сначала проходит проверку: если последний элемент меньше первого, значит эта перестановка с поменяными местами первым и последним элементами уже рассматривалась, и алгоритм сразу переходит к следующей итерации (такая оптимизация, очевидно, не работает, если перестановки не упорядочены по первому элементу).

**Недостатки программы:**

- задача коммивояжера решена полным перебором;
- проверка графа на гамильтоновость осуществляется с помощью поиска Гамильтонова цикла;
- повторяющийся в некоторых местах код.

**Масштабирование программы:**

- задачу коммивояжера можно решить более эффективными методами (метод ветвей и границ, муравьиный алгоритм, метод имитационного отжига);
- можно дополнить реализацию другими алгоритмами решения поставленных задач.

# Список литературы

[1] Ф.А.Новиков. Дискретная математика для программистов. СПб: Питер Пресс, 2009г. 364с.

[2] С.Д. Шапорев. Дискретная математика. СПб: БХВ - Петербург, 2006г. 369с.

[3] Р.Н. Вадзинский. Справочник по вероятностным распределениям. СПб: Наука, 2001г. 294с.