Министерство образования и науки
Санкт-Петербургский политехнический университет Петра Великого
Институт компьютерных наук и технологий
Высшая школа искусственного интеллекта
Направление 02.03.01 Математика и компьютерные науки

Л А Б О Р А Т О Р Н Ы Е Р А Б О Т Ы № 1 - 5

Реализация алгоритмов на графах

по дисциплине «Теория графов»

Выполнил студент гр. 3530201/10002 _____ Кондраев Дмитрий Евгеньевич
Проверил _____ Востров Алексей Владимирович

Санкт-Петербург
2023

Содержание

Введение

Постановка задачи

Лабораторная работа 1

1. Сформировать случайным образом связный ациклический граф, выходные степени вершин которого подчиняются распределению Пойя 1 (параметры распределения задаются как константы), с необходимым количеством вершин.
2. Реализовать метод Шимбелла на полученном графе (пользователь вводит количество ребер).
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) найти кратчайший путь для выбранных точек, используя алгоритмы Дейкстры, Беллмана-Форда, Флойда (пользователь вводит номера вершин начальной и конечной). Для алгоритмов Дейкстры и Беллмана-Форда рекомендуется выводить вектор расстояний, для Флойда-Уоршалла — матрицу расстояний. Обязательно выводить сам путь в виде последовательности вершин.
2. Сравнить скорости работы данных алгоритмов (по количеству итераций).

Лабораторная работа 3

1. Сформировать связный ациклический граф случайным образом в соответствии с заданным распределением. На его основе построить 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 $\frac{2}{3} f_{\max}$, где $f_{\max}$ — максимальный поток). Использовать ранее реализованные алгоритмы Дейкстры и/или Беллмана–Форда.

Лабораторная работа 4

1. Для заданных графов (случайно сгенерированных в первой работе) построить минимальный по весу остов, используя алгоритмы Прима и Краскала. Сравнить данные алгоритмы (итерации).
2. Используя матричную теорему Кирхгофа, найти число остовных деревьев в графе.
3. Полученный остов закодировать с помощью кода Прюфера (проверить правильность кодирования декодированием). Желательно сохранять веса при кодировании.

Лабораторная работа 5

1. Для заданных графов (случайно сгенерированных в первой работе) проверить, является ли граф эйлеровым и гамильтоновым. Если граф не является таковым, то отдельно модифицировать граф до эйлерова и отдельно до гамильтонова (до полного графа можно дополнять только в крайнем случае!).

2. Построить эйлеров цикл.
3. Решить задачу коммивояжёра на гамильтоновом графе (все гамильтоновы циклы с суммарным весом выводить либо на экран, если их мало, либо в файл).

1 Математическое описание

Неориентированным графом $G(V, E)$ называется совокупность двух множеств — непустого множества *вершин* V и множества *рёбер* E,

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

Ориентированным графом $G(V, E)$ называется совокупность двух множеств — непустого множества *узлов* V и множества *дуг* E,

$$G(V, E) \stackrel{\text{def}}{=} \langle V; E \rangle,$$
$$V \neq \varnothing, \quad E \subset V^2.$$

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

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

1.1 Связность

Маршрутом в графе называется чередующаяся последовательность вершин и рёбер, начинающаяся и кончающаяся вершиной, $v_0, e_1, v_1, e_2, v_2, \ldots, e_k, v_k$, 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*. Если все вершины (а значит, и рёбра) различны, то маршрут называется *простой цепью*.

Говорят, что две вершины в графе *связаны*, если существует соединяющая их (простая) цепь. Граф, в котором все вершины связаны, называется *связным*.

Отношение связанности вершин является эквивалентностью. Классы эквивалентности по отношению связанности называются *компонентами связности* графа.

Число компонентов связности графа G обозначается $k(G)$.

Граф G связный $\Longleftrightarrow k(G) = 1$.

Если $k(G) > 1$, то G — *несвязный* граф.

Замкнутая цепь называется *циклом*; замкнутая простая цепь называется *простым циклом*. Число циклов в графе G обозначается $z(G)$. Граф без циклов называется ациклическим ($z(G) = 0$).

1.2 Метод Шимбелла

Пусть граф задан матрицей $\Omega = (\omega_{ij})$ весов ребер, т.е.

$$\omega_{ij} = \begin{cases} 0, & \text{если вершины } i, j \text{ не смежны,} \\ \omega, & \text{если вес ребра } ij = \omega. \end{cases}$$

Введем специальные операции над элементами матрицы смежности вершин, позволяющие находить кратчайшие или максимальные пути между вершинами, состоящие из заданного количества ребер. Эти операции таковы.

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

$$\begin{cases} a \cdot b \stackrel{\text{def}}{=} a + b \\ a \cdot 0 = 0 \cdot a \stackrel{\text{def}}{=} 0 \end{cases}$$

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

$$\begin{cases} a + b \stackrel{\text{def}}{=} \min(\max)\{a, b\} \\ a + 0 = 0 + a \stackrel{\text{def}}{=} a \end{cases} \tag{1}$$

нули при этом игнорируются. Минимальный или максимальный элемент выбирается из ненулевых элементов. Нуль в результате операции (1) может быть получен лишь тогда, когда все элементы из выбираемых — нулевые.

С помощью этих операций длины кратчайших или максимальных путей между всеми вершинами определяется возведением в степень весовой матрицы Ω, содержащей веса ребер. Например, элементы матрицы $\Omega^2 = \left(\omega_{ij}^{(2)}\right)$ определяются следующим образом

$$\omega_{ij}^{(2)} = \min_k(\max_k)\left\{\omega_{ik}^{(1)} + \omega_{kj}^{(1)}\right\}.$$

Аналогично определяются элементы матрицы $\Omega^m = \left(\omega_{ij}^{(m)}\right)$. Длина кратчайшего или максимального пути из m ребер от вершины i до вершины j равна $\omega_{ij}^{(m)}$[5].

1.3 Распределение Пойа 1

Ряд распределения дискретной случайной величины X — совокупность всех ее возможных значений $x_1, \ldots, x_n$, и вероятностей $p_1, \ldots, p_n$, появления каждого из этих значений.

Примем как обозначение $\prod_{k=0}^{-1} f(k) = 1$ (как произведение пустого множества элементов).

Ряд распределение Пойа —

$$p(x) = C_n^x \frac{\prod_{k=0}^{x-1}(b + kc) \prod_{k=0}^{n-x-1}(r + kc)}{\prod_{k=0}^{n-1}(b + r + kc)}, \; x = 0, 1, 2, \ldots, n.$$

В частности,

$$p(0) = \prod_{k=0}^{n-1} \frac{r + kc}{b + r + kc}, \quad p(n) = \prod_{k=0}^{n-1} \frac{b + kc}{b + r + kc},$$

где $n > 0$, $b > 0$, $r > 0$, $c \in \mathbb{Z}$. Параметр c может быть отрицательным, однако он должен удовлетворять условию $b + r + c(n-1) > 0$[2].

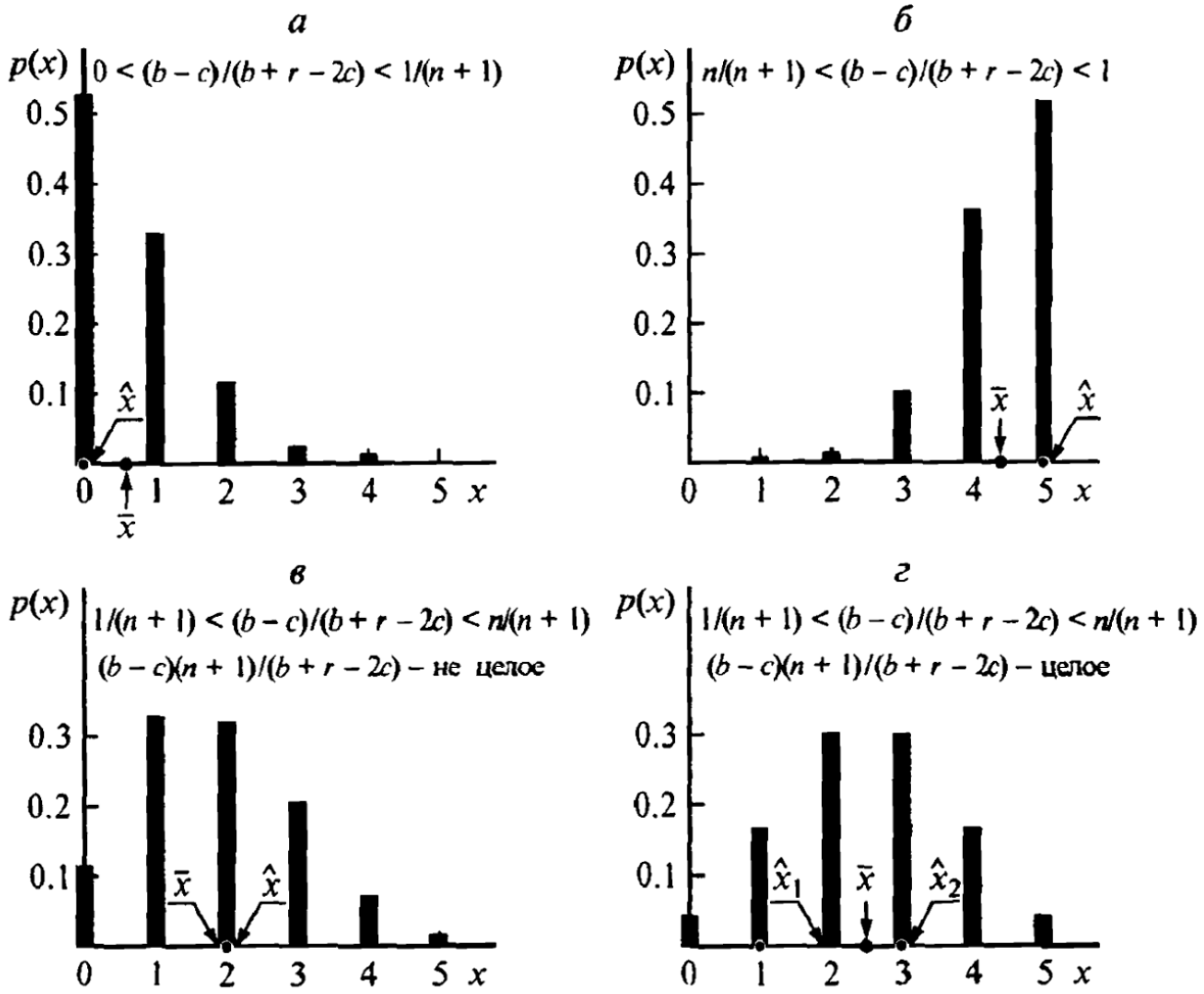

Рис. 1: Функция вероятности распределения Пойа 1.
Параметры распределения:
а — $n = 5$, $b = 3$, $r = 20$, $c = 1$;
б — $n = 5$, $b = 140$, $r = 20$, $c = 1$;
в — $n = 5$, $b = 12$, $r = 20$, $c = 1$;
г — $n = 5$, $b = 20$, $r = 20$, $c = 1$.

1.4 Вспомогательные процедуры

Во многих алгоритмах поиска кратчайших путей используются две процедуры. Процедура инициализации INIT(s) строит начальное состояние матрицы длин путей и матрицы предшествования.

Вход: узел s
Выход: заполненные матрицы T :**array**[1..p, 1..p] **of real** длин путей и
П: **array**[1..p, 1..p] **of** 0..p самих путей.

```
procedure INIT(s)
    for v := 1, p do
        T[v] := +∞, П[v] := 0
    end for
    T[s] := 0
end procedure
```

Процедура ослабления, или релаксации, — RELAX(s, v, u) проверяет, возможно ли улучшить известный путь из узла s в узел v, проведя новый путь через узел u, и обновляет пути, если

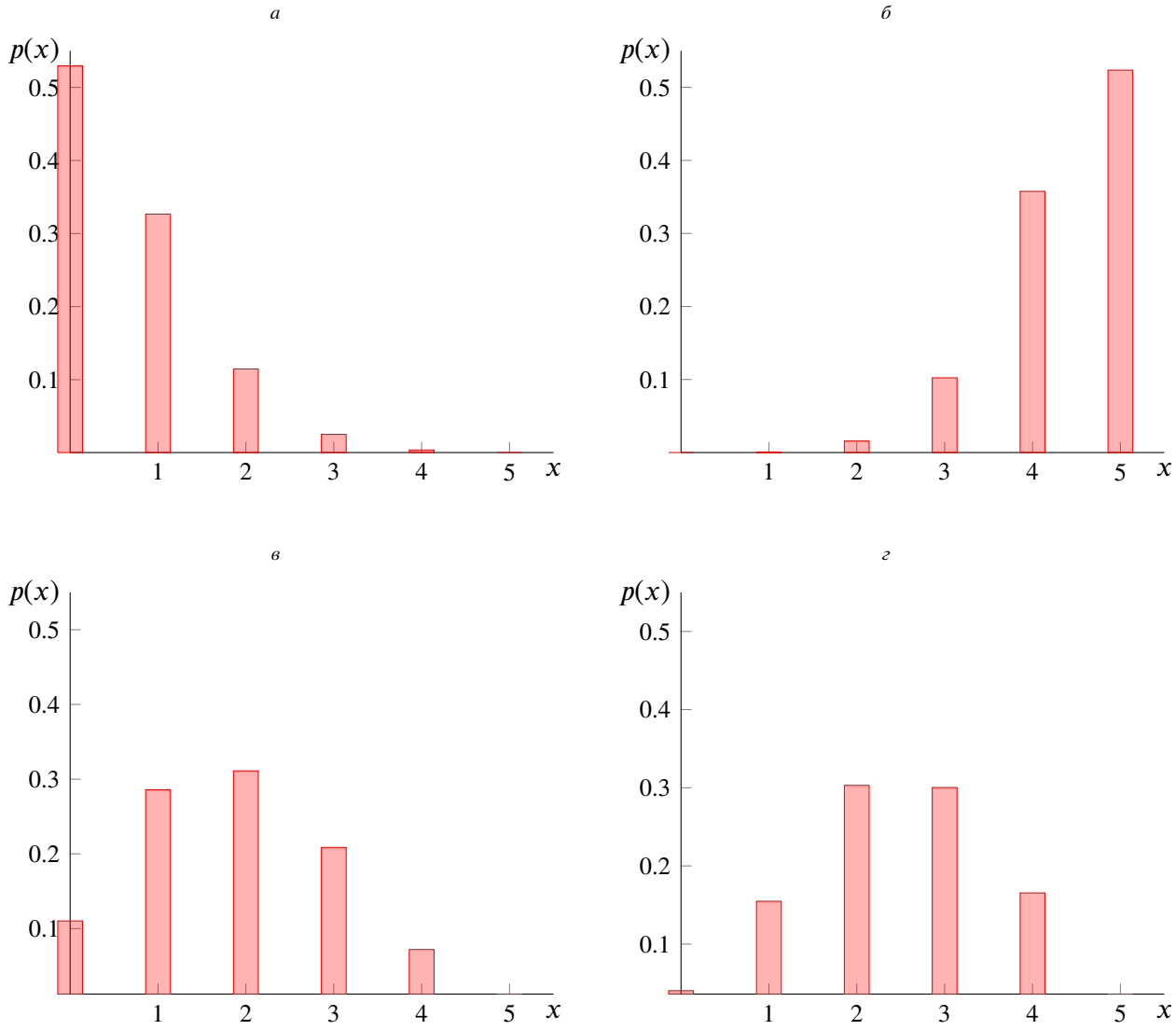

Рис. 2: Функция вероятности распределения (по выборке $N = 10\,000$).
Параметры распределения такие же, как на рис. 1.

Алгоритм 1 Генерации случайного числа, подчиняющегося распределению Пойа 1

Вход: $n > 0, b > 0, r > 0, c \in \mathbb{Z}$

Выход: случайное число из множества $\{0, 1, \ldots, n\}$

$p := p_0$ ▷ $p_0 = p(0)$

$r :=$ RAND ▷ $r \in [0, 1] \subset \mathbb{R}$, случайное с равномерным распределением

for $x := 0, n - 1$ **do**

  $r := r - p$

  if $r < 0$ **then**

    return x

  end if

  $p := p \cdot \alpha(x)$ ▷ $\alpha(x) = \dfrac{(n + 1 - x)\,(b + (x - 1)c)}{x\,(r + (n - x))\,c}$

end for

return n

это возможно.

Вход: узлы s, v, u
матрица W : **array**[1..p, 1..p] **of real** длин дуг,
матрица T : **array**[1..p, 1..p] **of real** длин путей,
матрица Π : **array**[1..p, 1..p] **of** 0..p самих путей.
Выход: обновленные матрицы T, Π
procedure RELAX(s, v, u)
if $T[s,v] > T[s,u] + W[u,v]$ **then**
$T[s,v] := T[s,u] + W[u,v]$; $\Pi[s,v] := u$ ▷ новый путь короче
end if
end procedure

Восстановление пути по матрице предшествования Π можно произвести так:

$w := v$; **yield** w ▷ последний узел
while $w \neq u$ **do**
$w := \Pi[u,w]$
yield w ▷ предыдущий узел
end while

Полученную последовательность можно обратить, если это потребуется, и получить прямой порядок узлов пути.

1.5 Алгоритм Дейкстры

Алгоритм Дейкстры 2 находит кратчайший путь между двумя данными вершинами (узлами) в (ор)графе, если длины дуг неотрицательны[4].

Сложность алгоритма: $\mathcal{O}((p+q)\log p)$ при условии, что Q — бинарная куча (сложность EXTRACTMIN — $\mathcal{O}(\log p)$).

Алгоритм 2 Дейкстры поиска кратчайших путей

Вход: взвешенный орграф $G(V, E)$,
матрица весов W : **array**[1..p, 1..p] **of real**,
источник s.
Выход: вектор T : **array**[1..p] **of real** длин кратчайших путей от источника,
вектор Π : **array**[1..p] **of** 0..p самих путей.
INIT(s) ▷ инициализация
$Q := V$ ▷ контейнер (очередь с приоритетами)
while $Q \neq \varnothing$ **do**
$u :=$ EXTRACTMIN(Q) ▷ извлечение узла с минимальным значением $T[u]$
if $T[u] = \infty$ **then stop**
end if ▷ остальные узлы недостижимы из s
for all $v \in \Gamma(u)$ **do**
if $v \in Q$ **then**
RELAX(s, v, u) ▷ релаксация дуги (u, v)
end if
end for
end while

1.6 Алгоритм Беллмана-Форда

Алгоритм Беллмана-Форда 3 находит кратчайшие пути во взвешенном графе, не содержащем циклы с отрицательным суммарным весом, от одной вершины до всех остальных[5].

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

Сложность алгоритма: $\mathcal{O}(p \cdot q)$, где p — количество вершин, q — количество ребер.

Алгоритм 3 Беллмана–Форда

Вход: взвешенный орграф $G(V, E)$, матрица весов W : **array**[1..p, 1..p] **of real**, источник s.
Выход: вектор T : **array**[1..p] **of real** длин кратчайших путей от источника,
вектор Π : **array**[1..p] **of** 0..p самих путей.

```
Init(s)                                          ▷ инициализация
for i := 1, p − 1 do
    for all (u, v) ∈ E do
        Relax(s, v, u)
    end for
end for
for all (u, v) ∈ E do                            ▷ проверка на отрицательные контуры
    if T[v] > T[u] + W[u, v] then stop            ▷ найден контур отрицательного веса
    end if
end for
```

1.7 Алгоритм Флойда-Уоршелла

Алгоритм Флойда-Уоршелла 4 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[4].

Алгоритм не всегда выдаёт решение, поскольку оно не всегда определено. Дополнительный цикл по j служит для прекращения работы в случае обнаружения в орграфе контура с отрицательным весом.

Идея алгоритма — разбиение процесса поиска кратчайших путей на фазы.

Сложность алгоритма: $\mathcal{O}(p^3)$.

1.8 Максимальный поток в сети

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

Дивергенцией функции f в узле v называется число $\operatorname{div}(f, v)$, которое определяется следующим образом:

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

Алгоритм 4 Флойда-Уоршалла поиска всех кратчайших путей

Вход: матрица W : **array**[1..p, 1..p] **of real** длин дуг.
Выход: матрица T: **array**[1..p, 1..p] **of real** длин путей и
матрица П: **array**[1..p, 1..p] **of** 0..p самих путей.

```
for i := 1, p do                                          ▷ инициализация
    for j := 1, p do
        T[i, j] := W[i, j]
        if W[i, j] = ∞ then
            П[i, j] := 0                                  ▷ нет дуги из i в j
        else
            П[i, j] := i                                  ▷ есть дуга из i в j
        end if
    end for
end for
for i := 1, p do
    for j := 1, p do
        for k := 1, p do
            if { i ≠ j ∧ T[j, i] ≠ ∞
               { i ≠ k ∧ T[i, k] ≠ ∞                      then
               { T[j, k] = ∞ ∨ T[j, k] > T[j, i] + T[i, k]
                T[j, k] := T[j, i] + T[i, k]              ▷ запомнить длину нового пути
                П[j, k] := i                              ▷ и сам путь
            end if
        end for
    end for
    for j := 1, p do
        if T[j, j] < 0 then stop                          ▷ Узел j входит в отрицательный контур
        end if
    end for
end for
```

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

1. $\forall (u, v) \in E (0 \leqslant f(u, v) \leqslant c(u, v))$, то есть поток через дугу неотрицателен и не превосходит пропускной способности дуги.
2. $\forall u \in V \setminus \{s, t\} (\mathrm{div}(f, u) = 0)$, 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 есть[4]

$$w(f) = \mathrm{div}(f, s).$$

Рассмотрим пару вершин $u, v \in V$. Определим *остаточную пропускную способность* $c_f(u, v)$ как

$$c_f(u, v) = \begin{cases} c(u, v) - f(u, v), & \text{если } (u, v) \in E, \\ f(v, u), & \text{если } (v, u) \in E, \\ 0 & \text{иначе.} \end{cases}$$

1.9 Алгоритм Форда-Фалкерсона

Теорема Форда-Фалкерсона Максимальный поток в сети равен минимальной пропускной способности разреза, то есть существует поток f^* такой, что

$$w(f^*) = \max_f w(f) = \min_P C(P).$$

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[4].

Приведенная реализация 5 метода Форда-Фалкерсона вычисляет максимальный поток в транспортной сети $G(V, E)$ путем обновления атрибута потока $(u, v).f$ каждого ребра $(u, v) \in E$. Если $(u, v) \notin E$, неявно предполагается, что $(u, v).f = 0$.

Алгоритм 5 Форда-Фалкерсона нахождения максимального потока в сети

Вход: Граф $G(V, E)$, источник s и сток t сети.
Пропускные способности $c(u, v)$, $c(u, v) = 0$ если $(u, v) \notin E$.
Выход:

```
for (u, v) ∈ E do
    (u, v).f = 0
end for
while существует путь p из s в t в остаточной сети G_f do
    c_f(p) := min{c_f(u, v) : (u, v) ∈ p}        ▷ c_f(p) — локальная переменная цикла
    for (u, v) ∈ p do
        if (u, v) ∈ E then (u, v).f := (u, v).f + c_f(p)
        else (v, u).f := (v, u).f − c_f(p)
        end if
    end for
end while
```

Можно улучшить временную границу алгоритма 5, если реализовать вычисление увеличивающего пути p в строке **while** как поиск в ширину, т.е. если в качестве увеличивающего

пути выбрать кратчайший путь из s в t в остаточной сети, где каждое ребро имеет единичную длину (вес). Такая реализация метода Форда-Фалкерсона называется *алгоритмом Эдмондса-Карпа* (Edmonds-Karp algorithm)[3].

Сложность алгоритма: $\mathcal{O}(p \cdot q^2)$.

1.10 Вычисление потока минимальной стоимости

Рассмотрим задачу определения потока заданной величины θ от s к t в сети $G(V, E, c, d)$, в которой каждая дуга $(u, v) \in E$ характеризуется не только пропускной способностью $c(u, v)$, но и неотрицательной стоимостью $d(u, v)$ пересылки единичного потока из i в j вдоль дуги (u, v). $d : E \to \mathbb{R}_+$.

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

Математическая модель задачи — минимизировать целевую функцию выбором потока f:

$$\sum_{(u,v)\in E} d(u, v) \cdot f(u, v) \to \min_f,$$

где $d(u, v)$ — вес дуги (u, v), $f(u, v)$ — величина потока вдоль дуги (u, v)[5].

1.11 Задача определения максимального потока в сети с несколькими истоками и несколькими стоками

В задаче о максимальном потоке может быть несколько истоков и стоков. Например, m истоков $\{s_1, s_2, \ldots, s_m\}$ и n стоков $\{t_1, t_2, \ldots, t_n\}$.

Задача определения максимальном потока в сети с несколькими истоками и несколькими стоками сводится к обычной задаче о максимальном потоке. Для этого в сеть добавляется *фиктивный исток* (supersource) s и ориентированные ребра (s, s_i) с пропускной способностью

$$c(s, s_i) = \infty, \qquad i = 1, \ldots, m.$$

Точно так же создается *фиктивный сток* (supresink) t и добавляются ориентированные ребра (t_i, t), обладающие

$$c(t_i, t) = \infty, \qquad i = 1, \ldots, n.$$

Единственный исток s обеспечивает поток любой требуемой величины к истокам s_i, а единственный сток t аналогичным образом потребляет поток любой желаемой величины от множественных стоков t_i[3].

1.12 Минимальный остов

Остовным деревом или *остовом* графа $G(V, E)$ называется связный подграф без циклов, содержащий все вершины исходного графа. Любое остовное дерево в графе с $|V| = n$ вершинами содержит ровно $n - 1$ ребро.

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

Задача о минимальном остове: во взвешенном связном графе найти остов минимального веса, то есть остов, суммарный вес ребер которого является минимальным[4].

1.13 Алгоритм Прима

Алгоритм Прима — алгоритм построения минимального остовного дерева взвешенного связного неориентированного графа.

В данном алгоритме кратчайший остов порождается в процессе разрастания одного дерева, к которому присоединяются ближайшие одиночные вершины. При этом для каждой вершины v, кроме начальной, используются две пометки: $\alpha[v]$ — это ближайшая к v вершина, уже включённая в остов, а $\beta[v]$ — это длина ребра, соединяющего v с остовом. Если вершину v ещё нельзя соединить с остовом одним ребром, то $\alpha[v] := 0$, $\beta[v] := \infty$.

На выходе мы получаем множество ребер, которые лежат в минимальном остовном дереве[4].

Алгоритм 6 Прима, нахождения остовного дерева минимальной стоимости

Вход: граф $G(V, E)$, заданный матрицей длин рёбер C.
Выход: множество T рёбер кратчайшего остова.

```
select u ∈ V                                  ▷ выбираем произвольную вершину
S := {u}                          ▷ S — множество вершин, включённых в кратчайший остов
T := ∅                              ▷ T — множество рёбер, включённых в кратчайший остов
for v ∈ V − u do
    if v ∈ Γ(u) then
        α[v] := u                                  ▷ u — ближайшая вершина остова
        β[v] := C[u, v]                ▷ C[u, v] — длина соответствующего ребра
    else
        α[v] := 0                          ▷ ближайшая вершина остова неизвестна
        β[v] := ∞                                  ▷ и расстояние также неизвестно
    end if
end for
for i := 1, p − 1 do
    x := ∞                          ▷ начальное значение для поиска ближайшей вершины
    for v ∈ V \ S do
        if β[v] < x then
            w := v                                 ▷ нашли более близкую вершину
            x := β[v]                                  ▷ и расстояние до неё
        end if
    end for
    S := S + w                                 ▷ добавляем найденную вершину в остов
    T := T + (α[w], w)                         ▷ добавляем найденное ребро в остов
    for v ∈ Γ(w) do
        if v ∉ S ∧ β[v] > C[v, w] then
            α[v] := w                          ▷ изменяем ближайшую вершину остова
            β[v] := C[v, w]                        ▷ и длину ведущего к ней ребра
        end if
    end for
end for
```

Сложность алгоритма: $\mathcal{O}(p^2)$.

1.14 Алгоритм Краскала

Алгоритм Краскала — жадный алгоритм построения минимального остовного дерева взвешенного связного неориентированного графа.

Алгоритм 7 состоит из двух фаз.

1. На подготовительной фазе все дуги удаляются из дерева и упорядочиваются по возрастанию их весов. В графе остаются только вершины, каждая из которых образует отдельную компоненту связности.
2. Во второй фазе дуги перебираются в порядке возрастания веса. Если начало и конец очередной дуги принадлежат одной и той же компоненте связности, дуга игнорируется. Если же они лежат в разных компонентах связности. дуга добавляется к графу, а эти две компоненты связности объединяются в одну. Если число компонент связности дойдет до 1, цикл прерывается.

Сложность алгоритма: $\mathcal{O}(q \log p)$, где p — количество вершин, q — количество ребер графа.

Алгоритм 7 Краскала, нахождения остовного дерева минимальной стоимости

Вход: список E рёбер графа G с длинами, упорядоченный в порядке возрастания длин.
Выход: множество T рёбер кратчайшего остова.

```
T := ∅
k := 1                                  ▷ номер рассматриваемого ребра
for i := 1, p − 1 do
    while z(T + E[k]) > 0 do
        k := k + 1                      ▷ пропустить это ребро
    end while
    T := T + E[k]                       ▷ добавить это ребро в SST
    k := k + 1                          ▷ и исключить его из рассмотрения
end for
```

1.15 Матричная теорема Кирхгофа

Матрица Кирхгофа — матрица $B(G) = (\beta_{ij})_{p\times p} = \operatorname{diag}\{\deg v_i\} - A(G)$, где p — количество вершин графа. $A(G) = (\alpha_{ij})_{p\times p} = (v_i \in \Gamma(v_j))$ — матрица смежности.

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

1.15.1 Свойства матрицы Кирхгофа

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

Матричная теорема Кирхгофа

Число остовных деревьев в связном графе G порядка $p \geqslant 2$ равно алгебраическому дополнению A_{ij} любого элемента матрицы Кирхгофа $B(G)$[5].

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

где $|M_{ij}|$ — минор, полученный из матрицы B вычеркиванием i-й строки и j-го столбца.

1.16 Код Прюфера

Код Прюфера — это способ взаимно однозначного кодирования помеченных деревьев с p вершинами с помощью последовательности $p-2$ целых чисел в отрезке $[1, p]$. То есть, код Прюфера — это биекция между всеми остовными деревьями полного графа и числовыми последовательностями.

Алгоритм 8 Построение кода Прюфера свободного дерева

Вход: Дерево $T(V, E)$ в любом представлении, вершины дерева пронумерованы числами $1..p$ произвольным образом.
Выход: Массив A: **array**$[1..p-1]$ **of** $1..p$ — код Прюфера дерева T.

```
for i := 1, p − 1 do
    v := min{k ∈ V | d(k) = 1}          ▷ выбираем висячую вершину v
    A[i] := Γ(v)          ▷ заносим в код номер единственной вершины, смежной с v
    V := V − v                          ▷ удаляем вершину v из дерева
end for
```

Сложность алгоритма 8: $\mathcal{O}(p \log p)$, если поиск минимума осуществляется за $\mathcal{O}(\log p)$.

Восстанавление закодированного по коду Прюфера дерева — алгоритм 9.

Алгоритм 9 Распаковка кода Прюфера свободного дерева

Вход: Массив A: **array**$[1..p-1]$ **of** $1..p$ — код Прюфера дерева T.
Выход: Дерево $T(V, E)$, заданное множеством рёбер E, вершины пронумерованы числами $1..p$.

```
E := ∅                                  ▷ вначале множество рёбер пусто
B := 1..p                       ▷ множество неиспользованных номеров вершин
for i := 1, p − 1 do
                    ▷ выбираем вершину v — неиспользованную вершину с наименьшим
                    ▷ номером, который не встречается в остатке кода Прюфера
    v := min {k ∈ B | ∀j ⩾ i(k ≠ A[j])}
    E := E + (v, A[i])                  ▷ добавляем ребро (v, A[i])
    B := B − v                  ▷ удаляем вершину v из списка неиспользованных
end for
```

Сложность алгоритма 9: $\mathcal{O}(p \log p)$, если поиск минимума осуществляется за $\mathcal{O}(\log p)$.

1.17 Эйлеров граф

Если связный граф имеет цикл (не обязательно простой), содержащий все рёбра графа (каждое — по одному разу), то такой цикл называется *эйлеровым* циклом, а граф называется *эйлеровым* графом. Эйлеров цикл содержит все вершины графа.

Теорема Если граф G связен и нетривиален, то G — эйлеров граф $\iff$ каждая вершина G имеет чётную степень[4].

Сложность алгоритма 10: $\mathcal{O}(q)$.

Алгоритм 10 построения эйлерова цикла

Вход: эйлеров граф $G(V, E)$, заданный списками смежности ($\Gamma[v]$ — список вершин, смежных с вершиной v).

Выход: последовательность вершин эйлерова цикла.

```
S := ∅                                          ▷ стек для хранения вершин
select v ∈ V                                    ▷ произвольная вершина
v → S                                           ▷ положить v в стек S
while S ≠ ∅ do
    v := top S                                  ▷ v — верхний элемент стека
    if Γ[v] = ∅ then
        v ← S; yield v                          ▷ очередная вершина эйлерова цикла
    else
        select u ∈ Γ[v]                         ▷ взять первую вершину из списка смежности
        u → S                                   ▷ положить u в стек
        Γ[v] := Γ[v] − u; Γ[u] := Γ[u] − v      ▷ удалить ребро (v, u)
    end if
end while
```

1.18 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 может быть только связный граф.

Достаточное условие гамильтоновости графа (условие Дирака):

Если $p(G) \geqslant 3$ и $\delta(G) \geqslant \frac{p}{2}$, то граф G является гамильтоновым[4].

1.19 Задача коммивояжёра

Задача коммивояжёра — задача отыскания кратчайшего гамильтонова цикла в нагруженном полном графе.[4]

Гамильтонов цикл — с комбинаторной точки зрения — перестановка вершин графа. При этом в качестве начальной вершины цикла можно выбрать любую, так что можно рассматривать перестановки с фиксированным первым элементом.

Пусть дана матрица стоимостей $C = (c_{ij})$, где c_{ij} — стоимость перехода из вершины i в j $(i, j = 1, \dots, p)$. Обозначим $1, v_1, v_2, \dots, v_{n-1}, 1$ — номера вершин, записанные в порядке их обхода. То есть v_k — номер вершины, посещаемой на k-м шаге, $k = 0, \dots, p$, $v_0 = v_p = 1$. Тогда пройденное расстояние можно представить в виде *целевой функции*:

$$\sum_{k=0}^{n-1} w(v_k, v_{k+1}) \to \min_{(v_1,\dots,v_{n-1})},$$

где $(v_1, \dots, v_{n-1})$ — перестановка чисел $2..p$. Таким образом задача коммивояжера состоит в поиске перестановки целых чисел от 2 до p, при которой целевая функция минимальна.

Простая схема поиска гамильтонова цикла:

1. последовательно рассмотреть все эти перестановки,

2. проверить для каждой из них, представляет ли она цикл в данном графе.

Такой способ действий уже при не очень большом числе вершин становится практически неосуществимым ввиду быстрого роста числа перестановок — имеется $(p-1)!$ перестановок из p элементов с фиксированным первым элементом.[1]

1.19.1 Алгоритм поиска гамильтоновых циклов

Более рациональный подход (алгоритм 11) состоит в рассмотрении всевозможных простых путей, начинающихся в произвольно выбранной стартовой вершине a, до тех пор, пока не будет обнаружен гамильтонов цикл или все возможные пути не будут исследованы. Так же производится перебор перестановок, но значительно сокращенный — если, например, вершина b не смежна с вершиной a, то все $(p-2)!$ перестановок, у которых на первом месте стоит a, а на втором b, не рассматриваются[1].

Алгоритм 11 поиска гамильтоновых циклов

Вход: граф $G(V, E)$ заданный списками смежности: $\Gamma[x]$ — список вершин смежных с x.
Выход: последовательность гамильтоновых циклов S

```
select a ∈ V                                   ▷ выбрать произвольно вершину a
S = {a}
N[a] := Γ[a]
while S ≠ ∅ do
    x := top S
    if N[x] ≠ ∅ then
        select y ∈ N[x]
        N[x] := N[x] − y
        if вершина y не находится в S then
            y ⇒ S                              ▷ поместить y в S
            N[y] := Γ[y]
            if S содержит все вершины ∧ y смежна с a then
                yield S
            end if
        end if
    else удалить вершину x из S
    end if
end while
```

2 Особенности реализации

2.1 Структуры данных и константы

Ориентированный взвешенный граф представлен в программе матрицей весов (дуг):

```
template<typename T=int>
using adjacency_matrix = std::vector<std::vector<T>>;
```

$$M[i][j] = \begin{cases} 0, & \text{если вершины } i, j \text{ не смежны,} \\ \omega, & \text{если вес ребра } ij = \omega. \quad \omega \in \mathbb{Z} \setminus \{0\} \end{cases}$$

Таким образом, нулевой вес в матрице однозначно говорит об отсутствии дуги.

Вершина представлена своим номером — беззнаковое целое:

```
using Vertex = size_t;
```

Взвешенное ребро (дуга) представлено в программе структурой `edge_t`:

```
struct edge_t {
    Vertex from;
    Vertex to;
    int weight;

    // сравнение ребер по весу
    bool operator<(const edge_t& rhs) const {
        return std::tie(weight, from, to) < std::tie(rhs.weight, rhs.from, rhs.to);
    }
    bool operator==(const edge_t& rhs) const {
        return std::tie(from, to) == std::tie(rhs.from, rhs.to);
    }

    edge_t(Vertex from, Vertex to, int weight) : from(from), to(to), weight(weight) {}
};
```

Пути в графе представлены массивом вершин:

```
using path_t = std::vector<Vertex>;
```

Путь вместе со своим суммарным весом представлен структурой `costed_path_t`:

```
struct costed_path_t {
    path_t path;
    // суммарный вес пути
    size_t cost;
    // непустой ли путь
    bool exists() const {
        return !path.empty();
    }
    // сравнение путей по их весу
    bool operator<(const costed_path_t& rhs) const {
        return cost < rhs.cost;
    }
};
```

Поток в сети представлен структурой `flow_graph_t`:

```
struct flow_graph_t {
    // матрица пропускных способностей
    adjacency_matrix<> capacity;
    // матрица стоимостей
```

```
    // 0 значит ∞ стоимость
    adjacency_matrix<> cost;
    // вершина источник
    Vertex source;
    // вершина сток
    Vertex sink;
};
```

Код Прюфера для дерева из p вершин представлен в программе парой, первый ее элемент — массив из $p-2$ номеров вершин, второй — массив весов соответствующих ребер ($p-1$ число).

```
std::pair<std::vector<Vertex>, std::vector<int>>
```

В программе определены следующие константы:

```
// бесконечный вес ребра
constexpr auto INF = INT32_MAX;
// номер несуществующей вершины
constexpr Vertex NO_VERTEX = -1u;
```

Значение `NO_VERTEX` используется в матрице (и векторе) предшествования для обозначения, что соответствующий путь не существует. См.~подробнее Алгоритм Дейкстры, Алгоритм Флойда-Уоршелла.

2.2 Генерация графа. Построение маршрута

2.2.1 Генерация случайной величины с распределением Пойа 1

Функция `template <class IntType = int> std::discrete_distribution<IntType> polya_1(IntType black_n, IntType red_n, IntType c, IntType size);`

Вход функции

- `black_n` — параметр распределения b
- `red_n` — параметр распределения r
- `c` — параметр распределения c
- `size` — параметр распределения n — количество различных значений

Выход функции представитель шаблона `std::discrete_distribution`, хранящий в себе соответствующую таблицу распределения вероятностей.

Описание работы функции Алгоритм 1 реализован следующим образом:

1. Функция `polya_1` заполняет таблицу вероятностей для значений из диапазона $0 \leqslant x < n$,
2. Функция `std::discrete_distribution::operator()`[1] выдает случайное число с заданным распределением вероятностей.

Чтобы убедиться в том, что с помощью функции `polya_1` генерируются случайные числа с необходимым распределением, были построены гистограммы для 4 различных наборов параметров. Ниже представлен код, выдающий исходные данные для построения гистограмм. Сами гистограммы изображены на рис. 2.

[1] https://en.cppreference.com/w/cpp/numeric/random/discrete_distribution/operator()

```
void polya_1_histograms() {
    std::mt19937 gen{std::random_device{}()};
    constexpr auto N = 10000; // итераций
    constexpr auto n = 5;
    for (auto b : {3, 140, 12, 20}) {
        auto distribution = polya_1(b, 20, 1, n);
        std::vector<size_t> data(n + 1);
        std::cout << "b = " << b << '\n';
        for (int i = 0; i < N; ++i) {
            data[distribution(gen)]++;
        }
        std::cout << 'x' << '\t' << "p(x)" << '\n';
        for (int i = 0; i <= n; ++i) {
            std::cout << i << '\t' << static_cast<double>(data[i]) / N << '\n';
        }
    }
}
```

Исходный код

```
template <class IntType = int>
std::discrete_distribution<IntType>
polya_1(IntType black_n, IntType red_n, IntType c, IntType size) {
    std::vector<double> probabilities;
    probabilities.reserve(size + 1);
    double p0 = 1;
    for (int i = 0; i < size; i++) {
        p0 *= (red_n + i * c);
        p0 /= (black_n + red_n + i * c);
    }
    double p = p0;
    probabilities.push_back(p);
    for (int x = 1; x <= size; ++x) {
        p *= (size - (x - 1)) * (black_n + (x - 1) * c);
        p /= x * (red_n + (size - x) * c);
        probabilities.push_back(p);
    }
    return { probabilities.begin(), probabilities.end() };
}
```

Пример использования:

```
auto dis = polya_1<>(20, 20, 1, 5);
std::random_device rd;
std::mt19937 gen(rd());
std::cout << "random - " << dis(gen);
```

2.2.2 Генерация графа

Отсутствие циклов гарантируется тем, что заполняются только элементы матрицы выше главной диагонали, т. о. получается верхнетреугольная матрица.

Связность гарантируется тем, что генерируемые степени вершин $\deg^+ v > 0$, кроме одной вершины.

Веса ребер также подчиняются тому же распределению, но с другими коэффициентами.

Функция `adjacency_matrix<> generate(size_t nVertices, std::mt19937& gen);`

Вход функции Количество вершин `nVertices`, генератор случайных чисел `gen`.

Выход функции Матрица весов, определяющая граф.

Описание работы функции

1. Функция `out_degrees` формирует массив степеней (выхода) вершин орграфа, подчиняющихся заданному распределению. Причем они отсортированы в порядке убывания.
2. Функция `from_degrees` заполняет матрицу весовую графа. Количество ненулевых элементов в строке i равно `vertex_degrees[i]`. Матрица по построению верхнетреугольная (нули на главной диагонали и под ней).

Исходный код

```
adjacency_matrix<> graphs::generate(size_t nVertices, std::mt19937 &gen) {
    return from_degrees(out_degrees(nVertices, gen), gen);
}

std::vector<size_t> graphs::out_degrees(size_t nVertices, std::mt19937 &gen) {
    if (nVertices == 0) {
        return std::vector<size_t>{};
    }
    if (nVertices == 1) {
        return std::vector<size_t>{0};
    }
    // степень вершины может быть в диапазоне [1 .. n-1]
    // распределение выдает числа в диапазоне [0 .. n-2]
    auto dis = polya_1<int>(2, 20, 1, static_cast<int>(nVertices) - 3);
    std::vector<size_t> result(nVertices);
    for (size_t &x: result) {
        x = dis(gen) + 1;
    }
    std::sort(result.rbegin(), result.rend());

    // чтобы обеспечить отсутствие циклов
    for (size_t i = 0; i < nVertices; ++i) {
        result[i] = std::min(nVertices - i - 1, result[i]);
    }
    return result;
}

adjacency_matrix<> graphs::from_degrees(std::vector<size_t> vertex_degrees, std::mt19937 &gen)
 ↪  {
    auto nVertices = vertex_degrees.size();
    adjacency_matrix<> result(nVertices, std::vector<int>(nVertices));
    auto dis = polya_1<size_t>(4, 8, 3, 50 - 1);
    for (int i = 0; i < nVertices; ++i) {
        for (int j = i + 1; j < vertex_degrees[i] + i + 1; ++j) {
            result[i][j] = dis(gen) + 1;
            assert(result[i][j] > 0);
        }
        if (vertex_degrees[i] == nVertices - i - 1) {
            continue;
        }
        // degrees[i] of ones, other are zeros.
        // nVertices - i - 1 total
        std::shuffle(result[i].begin() + i + 1, result[i].end(), gen);
    }
```

```
    return result;
}
```

2.2.3 Метод Шимбелла

Функция `adjacency_matrix<> min_path_lengths(const adjacency_matrix<>& that, size_t path_length);`

`adjacency_matrix<> max_path_lengths(const adjacency_matrix<>& that, size_t path_length);`

Вход функции На вход обеим функциям подается матрица весов `that` и длина пути — количество ребер `path_length`.

Выход функции На выходе — матрица минимальных (максимальных) расстояний между вершинами графа.

Описание работы функции В основном соответствует описанию п. 1.2.

Операции с матрицами в данной реализации параметризованы бинарной операцией `extrem`, играющей роль скалярного сложения в методе Шимбелла.

Возведение матрицы в степень выполняется последовательным умножением. Умножение матриц производится тривиальным алгоритмом (по определению).

Исходный код

```
adjacency_matrix<> graphs::min_path_lengths(const adjacency_matrix<> &that, size_t path_length)
 ↪  {
    if (that.empty()) {
        return {};
    }
    const size_t &(*fn)(const unsigned long &, const unsigned long &) = &std::min<size_t>;
    return matrix_power_shimbell(that, path_length, fn);
}

adjacency_matrix<> graphs::max_path_lengths(const adjacency_matrix<> &that, size_t path_length)
 ↪  {
    if (that.empty()) {
        return {};
    }
    const size_t &(*fn)(const unsigned long &, const unsigned long &) = &std::max<size_t>;
    return matrix_power_shimbell(that, path_length, fn);
}

template<typename Func>
adjacency_matrix<>
matrix_power_shimbell(const adjacency_matrix<> &that, size_t power, Func extrem) {
    assert(power > 0);
    adjacency_matrix<> result = that;
    for (int i = 1; i < power; ++i) {
        result = matrix_multiply<>(result, that, extrem);
    }
    return result;
}

template<typename Func>
adjacency_matrix<>
```

```
matrix_multiply(const adjacency_matrix<> &lhs, const adjacency_matrix<> &rhs, Func extrem) {
    assert(lhs.size() == lhs[0].size()); // квадратная
    assert(rhs.size() == rhs[0].size()); // квадратная
    assert(lhs.size() == rhs.size());
    size_t n = lhs.size();
    adjacency_matrix<> result = lhs;
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            auto &res = result[i][j];
            res = 0;
            for (int k = 0; k < n; ++k) {
                if (lhs[i][k] == 0 || rhs[k][j] == 0) {
                    continue;
                }
                res = res ? extrem(res, lhs[i][k] + rhs[k][j]) : lhs[i][k] + rhs[k][j];
            }
        }
    }
    return result;
}
```

2.2.4 Поиск количества маршрутов между вершинами

Для данной задачи определена структура (фунуциональный объект) `count_paths`:

```
struct count_paths {
    count_paths(const adjacency_matrix<>& graph, Vertex start)
        : graph{graph}
        , start{start}
        , visited(graph.size())
        , d(graph.size())
    {
        visited[start] = true;
        d[start] = 1;
    }
    size_t operator()(Vertex v);
private:
    adjacency_matrix<> graph;
    Vertex start; // стартовая вершина
    std::vector<bool> visited; // visited[v] = true, если ответ для вершины v уже посчитан
    std::vector<size_t> d; // d[v] – число путей из вершины `start` до вершины v
};
```

Интерес представляет в первую очередь оператор `()` этой структуры.

Функция `size_t count_paths::operator()(Vertex v);`

Вход функции Матрица смежности `graph`, начальная вершина `start` (передаются в конструктор), конечная вершина `v`.

Выход функции Количество различных путей из начальной вершины в конечную.

Описание работы функции Искомое количество находится методом динамического программирования.

Пусть s — стартовая вершина (`start`), а t — конечная, для нее и посчитаем ответ. Будем поддерживать массив d, где $d[v]$ — число путей из вершины s до вершины v и массив w

(visited), где $w[v] = true$, если ответ для вершины v уже посчитан, и $w[v] = false$ в противном случае. Изначально $w[i] = false$ для всех вершин i, кроме s, а $d[s] = 1$.

Данная функция — рекурсивная функция с запоминанием. Так, значения массива d будут вычисляться по мере необходимости и не будут считаться лишний раз.

Сложность алгоритма: $\mathcal{O}(p + q)$.

Исходный код

```
size_t graphs::count_paths::operator()(size_t v) {
    if (visited[v]) {
        return d[v];
    }
    size_t result = 0;
    visited[v] = true;
    for (int i = 0; i < graph.size(); ++i) {
        if (!graph[i][v]) {
            continue;
        }
        // ∃ ребро из i в v (рекурсивный вызов)
        result += operator()(i);
    }
    return d[v] = result;
}
```

2.3 Нахождение кратчайшего пути

2.3.1 Алгоритм Дейкстры

Функция `dijkstra_result_t min_path_distances_dijkstra(const adjacency_matrix<>& g, Vertex start_vertex);`

Вход функции Матрица весов `g`, начальная вершина `start_vertex`.

Выход функции Структура `dijkstra_result_t`.

```
struct dijkstra_result_t {
    // массив длин путей от данной вершины ко всем остальным
    std::vector<int> distances;
    // массив вершин, precedents[i] непосредственно перед вершиной i в кратчайшем пути от
     ↪  начальной вершины до i
    // NO_VERTEX если такого пути нет.
    std::vector<Vertex> precedents;
    // количество итераций
    size_t iterations;
};
```

Описание работы функции Функция реализует алгоритм Дейкстры 2. Находит кратчайшие пути во взвешенном графе от одной вершины до всех остальных, если длины дуг неотрицательны.

1. Начальная вершина помечается посещенной и помещается в очередь.
2. Пока очередь не пуста,
    1. Из нее извлекается непосещенная вершина `nearest_vertex`, текущее расстояние до которой наименьшее.

2. Она помечается посещенной.
3. Рассматриваются все ребра, исходящие из вершины `nearest_vertex`, и если текущее расстояние до конца этой вершины `distances[v]` можно улучшить пройдя через рассматриваемое ребро, то обновляется значение `distances[v]` и вершина добавляется в очередь.

Найденные значения `distances[v]` и есть искомые длины кратчайших путей из начальной вершины в `v`.

В качестве структуры, позволяющей вынимать (`pop`) минимальный элемент за $\mathcal{O}(\log p)$, используется `std::priority_queue`[2], реализующая бинарную кучу.

Сложность алгоритма: $\mathcal{O}\left((p+q)\log p\right)$.

Исходный код

```
dijkstra_result_t
graphs::min_path_distances_dijkstra(const adjacency_matrix<> &g, Vertex start_vertex) {
    std::vector<Vertex> precedents(g.size(), NO_VERTEX);
    std::vector<bool> visited(g.size(), false);
    size_t iterations{};

    std::vector<int> distances(g.size(), INF);
    auto cmp = [&](Vertex i, Vertex j) {
        return distances[i] > distances[j];
    };
    std::priority_queue<Vertex, std::vector<Vertex>, decltype(cmp)> qu(cmp);
    distances[start_vertex] = 0;
    qu.push(start_vertex);
    while (!qu.empty()) {
        auto nearest_vertex = qu.top();
        qu.pop();
        if (visited[nearest_vertex]) {
            continue;
        }
        visited[nearest_vertex] = true;
        for (Vertex v{}; v < g.size(); ++v) {
            if (g[nearest_vertex][v] == 0) {
                continue;
            }
            auto distance_via_nearest = distances[nearest_vertex] + g[nearest_vertex][v];
            if (distance_via_nearest < distances[v]) {
                distances[v] = distance_via_nearest;
                qu.push(v);
                visited[v] = false;
                precedents[v] = nearest_vertex;
            }
            iterations++;
        }
    }
    return {
        .distances = distances,
        .precedents = precedents,
        .iterations = iterations
    };
}
```

[2] https://en.cppreference.com/w/cpp/container/priority_queue

2.3.2 Алгоритм Беллмана-Форда

Функция `dijkstra_result_t min_path_distances_bellman_ford(const adjacency_matrix<>& g, Vertex start_vertex);`

Вход функции Матрица весов `g`, начальная вершина `start_vertex`.

Выход функции Структура `dijkstra_result_t`.

Описание работы функции Функция находит кратчайшие пути во взвешенном графе, реализуя алгоритм Беллмана-Форда 3.

1. На первом шаге расстояния от исходной вершины до всех остальных вершин (массив `distances`) принимаются равными бесконечности, а расстояние до начальной вершины — нулю: `distances[u] = 0`.
2. Вторым шагом вычисляются самые короткие расстояния. Следующие шаги нужно выполнять $p - 1$ раз, где p —– число вершин в графе.

   Производится следующее действие для каждого ребра uv:

   Если `distances`$[v] >$ `distances`$[u] +$ вес ребра uv, то обновляется `distances`$[v] =$ `distances`$[u] +$ вес ребра uv
3. На третьем шаге сообщается, присутствует ли в графе цикл отрицательного веса. Для каждого ребра uv необходимо выполнить следующее:

   Если `distances`$[v] >$ `distances`$[u] +$ вес ребра uv, то в графе присутствует цикл отрицательного веса.

Идея шага 3 заключается в том, что шаг 2 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, это будет сигналом присутствия цикла отрицательного веса.

Релаксация каждого ребра занимает константное количество действий. Всего ребер — q, релаксация всех ребер производится p раз. Таким образом, временная сложность алгоритма — $\mathcal{O}(p \cdot q)$.

Исходный код

```
dijkstra_result_t
graphs::min_path_distances_bellman_ford(const adjacency_matrix<> &g, Vertex start_vertex) {
    std::vector<Vertex> precedents(g.size(), NO_VERTEX);
    size_t iterations{};

    std::vector<int> distances(g.size(), INF);
    distances[start_vertex] = 0;

    for (int _ = 1; _ < g.size(); ++_) {
        for (Vertex u{}; u < g.size(); ++u) {
            for (Vertex v{}; v < g.size(); ++v) {
                if (g[u][v] == 0) {
                    continue;
                }
                auto candidate = distances[u] + g[u][v];
                if (distances[u] != INF && candidate < distances[v]) {
```

```
                    // новый путь короче
                    distances[v] = candidate;
                    precedents[v] = u;
                }
                iterations++;
            }
        }
    }
    // проверка на отрицательные контуры
    for (Vertex u{}; u < g.size(); ++u) {
        for (Vertex v{}; v < g.size(); ++v) {
            if (g[u][v] == 0) {
                continue;
            }
            // найден контур отрицательного веса
            if (distances[u] != INF && distances[v] > distances[u] + g[u][v]) {
                return {
                    .iterations = iterations
                };
            }
        }
    }

    return {
        .distances = distances,
        .precedents = precedents,
        .iterations = iterations
    };
}
```

2.3.3 Алгоритм Флойда-Уоршелла

Поиск кратчайших путей от каждой вершины к каждой.

Функция `floyd_warshall_result_t min_path_distances_floyd_warshall(const adjacency_matrix<>& g);`

Вход функции Матрица весов `g`.

Выход функции Структура `floyd_warshall_result_t`.

```
struct floyd_warshall_result_t {
    // матрица длин путей между всеми парами вершин
    // `distances[i][j]` - длина кратчайшего пути от `i` к `j`
    std::vector<std::vector<int>> distances;
    // матрица вершин,
    // precedents[i][j] непосредственно предшествует `j` в кратчайшем пути от `i` к `j`;
    //                  =NO_VERTEX если пути нет
    std::vector<std::vector<Vertex>> precedents;
    // количество итераций
    size_t iterations;
};
```

Описание работы функции Функция следует алгоритму 4.

Исходная матрица расстояний — матрица весов, где все нули, кроме главной диагонали заменены символом ∞.

В циклах по i, j 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, новый путь записывается как наименьший.

Время работы алгоритма — три вложенных цикла от 1 до p — $\mathcal{O}\left(p^3\right)$.

Исходный код

```
floyd_warshall_result_t
graphs::min_path_distances_floyd_warshall(const adjacency_matrix<> &g) {
    std::vector<std::vector<Vertex>> precedents(g.size(), std::vector<Vertex>(g.size(),
    ↪  NO_VERTEX));
    size_t iterations{};
    std::vector<std::vector<int>> distances(g.size(), std::vector<int>(g.size(), INF));

    for (Vertex i{}; i < g.size(); ++i) {
        for (Vertex j{}; j < g.size(); ++j) {
            if (i == j) {
                distances[i][j] = 0;
            }
            if (g[i][j] == 0) {
                continue;
            }
            distances[i][j] = g[i][j];
            precedents[i][j] = i;
        }
    }

    for (Vertex i{}; i < g.size(); ++i) {
        for (Vertex j{}; j < g.size(); ++j) {
            if (j == i || distances[j][i] == INF) {
                continue;
            }
            for (Vertex k{}; k < g.size(); ++k) {
                iterations++;
                if (k == i || distances[i][k] == INF) {
                    continue;
                }
                auto candidate = distances[j][i] + distances[i][k];
                if (distances[j][k] == INF || distances[j][k] > candidate) {
                    distances[j][k] = candidate;
                    precedents[j][k] = i;
                }
            }
        }
        for (Vertex j{}; j < g.size(); ++j) {
            if (distances[j][j] < 0) {
                // Узел `j` входит в отрицательный контур
                return {
                    .iterations = iterations
                };
            }
        }
    }

    return {
        .distances = distances,
        .precedents = precedents,
        .iterations = iterations
    };
```

```
}
```

2.3.4 Восстановление пути по вектору предшествования

Функция `std::vector<Vertex> reconstruct_path(const std::vector<Vertex>& precedents, Vertex from, Vertex to);`

Вход функции Вектор предшествования `precedents`, начальная вершина `from`, конечная — `to`.

Выход функции Массив `result` вершин в порядке из обхода по маршруту, представленного вектором предшествования.

Описание работы функции Производится обход вектора предшествования: последовательно, начиная с последней вершины `to` извлекается вершина, достигаемая кратчайшим путем из начальной вершины в текущую. Полученная последовательность обращается. См. Вспомогательные процедуры.

Исходный код

```
path_t
graphs::reconstruct_path(const std::vector<Vertex> &precedents, Vertex from, Vertex to) {
    std::vector<Vertex> result;
    result.reserve(precedents.size());
    result.push_back(to);
    while (to != from && precedents[to] != NO_VERTEX) {
        to = precedents[to];
        result.push_back(to);
    }
    if (to != from) {
        return {};
    }
    std::reverse(result.begin(), result.end());
    return result;
}
```

2.4 Нахожение максимального потоков и потока минимальной стоимости

2.4.1 Определение истока и стока

Функция `flow_graph_t add_supersource_supersink(const adjacency_matrix<>& capacity, const adjacency_matrix<>& cost);`

Вход функции Матрица `capacity` пропускных способностей, матрица `cost` стоимостей.

Выход функции Структура `flow_graph_t`, представляющая поток в графе, содержащем один исток и один сток.

Описание работы функции Производится поиск вершин, удовлетворяющих условию истока (строка i матрицы смежности нулевая) и условию стока (столбец j матрицы смежности нулевой).

Если стоков (истоков) несколько, то добавляется фиктивный сток (исток), соединенный дугой со всеми стоками (истоками).

Если сток (исток) единственный, то он и возвращается в поле `sink` (`source`) структуры `flow_graph_t`. Стоимости добавленных дуг принимаются нулевыми, а пропускные способности — бесконечно большими.

Исходный код

```
flow_graph_t graphs::add_supersource_supersink(const adjacency_matrix<> &capacity, const
 ↪  adjacency_matrix<> &cost) {
    // identify sources and sinks
    std::vector<bool> is_source(capacity.size(), true);
    std::vector<bool> is_sink(capacity.size(), true);
    for (Vertex i{}; i < capacity.size(); ++i) {
        for (Vertex j{}; j < capacity.size(); ++j) {
            if (capacity[i][j] != 0) {
                is_source[j] = false;
                is_sink[i] = false;
            }
        }
    }
    std::vector<Vertex> sources;
    std::vector<Vertex> sinks;
    for (Vertex i{}; i < capacity.size(); ++i) {
        if (is_sink[i]) {
            sinks.push_back(i);
            continue;
        }
        if (is_source[i]) {
            sources.push_back(i);
        }
    }
    // fix matrices if needed
    flow_graph_t result{
        .capacity = capacity,
        .cost = cost,
    };

    assert(!sources.empty());
    if (sources.size() == 1) {
        result.source = sources.front();
    } else {
        result.source = capacity.size();
        for (auto &row: result.cost) {
            row.emplace_back();
        }
        for (auto &row: result.capacity) {
            row.emplace_back();
        }
        result.cost.emplace_back(result.source + 1, 0u);
        result.capacity.emplace_back(result.source + 1, 0u);
        for (auto v: sources) {
            result.capacity[result.source][v] = INF;
        }
    }
```

```
    assert(!sinks.empty());
    if (sinks.size() == 1) {
        result.sink = sinks.front();
    } else {
        result.sink = capacity.size() + (sources.size() > 1);
        for (auto &row: result.cost) {
            row.emplace_back();
        }
        for (auto &row: result.capacity) {
            row.emplace_back();
        }
        result.cost.emplace_back(result.sink + 1, 0u);
        result.capacity.emplace_back(result.sink + 1, 0u);
        for (auto v: sinks) {
            result.capacity[v][result.sink] = INF;
        }
    }
    return result;
}
```

2.4.2 Алгоритм Форда-Фалкерсона

Функция `flow_result_t max_flow_ford_fulkerson(const flow_graph_t& g);`

Вход функции Структура `flow_graph_t`.

Выход функции Структура `flow_result_t` — матрица потока и суммарная его величина.

```
struct flow_result_t {
    int max_flow; // величина потока
    adjacency_matrix<> flow; // матрица потока
};
```

Описание работы функции Находит максимальный поток по данной матрице пропускных способностей (игнорируя стоимости).

Исходный код

```
flow_result_t graphs::max_flow_ford_fulkerson(const flow_graph_t &g) {
    const auto s = g.source;
    const auto t = g.sink;

    auto capacity = g.capacity;
    const auto n = g.capacity.size();
    auto parent = std::vector<Vertex>(g.capacity.size(), -1u);

    // Returns true if there is a path from
    // source `s` to sink `t` in residual graph.
    // Also fills parent[] to store the path.
    const auto bfs = [n, s, t, &parent, &capacity = std::as_const(capacity)]() -> bool {
        // Mark all the vertices as not visited
        std::vector<bool> visited(n, false);
        // Create a queue for BFS
        std::deque<Vertex> queue{s};
        // Mark the source node as visited and enqueue it
        visited[s] = true;
        parent[s] = s;
```

```
        // Standard BFS loop
        while (!(queue.empty())) {
            Vertex u = queue.front();
            queue.pop_front();
            // Get all adjacent vertices of the dequeued vertex u
            // If an adjacent has not been visited, then mark it
            // visited and enqueue it
            for (Vertex v{}; v < n; v++) {
                if (visited[v] || capacity[u][v] <= 0) {
                    continue;
                }
                queue.push_back(v);
                parent[v] = u;
                visited[v] = true;
            }
        }
        // If we reached sink in BFS starting from source, then return
        // true, else false
        return visited[t];
    };

    flow_result_t result = {
        .max_flow = 0,
        .flow = adjacency_matrix<>(g.capacity.size(), std::vector<int>(g.capacity.size()))
    };
    while (bfs()) {
        int path_flow = INF;
        for (Vertex v = t; v != s; v = parent[v]) {
            path_flow = std::min(path_flow, capacity[parent[v]][v]);
        }
        for (Vertex v = t; v != s; v = parent[v]) {
            Vertex u = parent[v];
            capacity[u][v] -= path_flow;
            capacity[v][u] += path_flow;
            result.flow[u][v] += path_flow;
        }
        result.max_flow += path_flow;
    }
    return result;
}
```

2.4.3 Поиск заданного потока минимальной стоимости

Функция `min_cost_flow_result_t min_cost_flow(const flow_graph_t& g, int desired_flow);`

Вход функции Структура `flow_graph_t`, желаемая величина потока `desired_flow`. Она должна быть меньше или равна максимальной величины $f_{\max}$. В программе передается величина $\frac{2}{3} f_{\max}$.

Выход функции Структура `min_cost_flow_result_t`.

```
struct min_cost_flow_result_t {
    int flow_sum; // величина потока
    int cost; // стоимость
    adjacency_matrix<> flow; // матрица потока
};
```

Описание работы функции Находит поток заданной величины с минимальной стоимостью в графе (алгоритм 5).

1. Ищется минимальный по стоимости путь из истока в сток с помощью алгоритма Дейкстры (все дуги имеют неотрицательную стоимость).
2. Вычисляется максимальный поток по найденному пути.
3. Полученное значение прибавляется к значению потока каждой дуги найденного увеличивающего пути и отнимается от заданного значения потока.
4. Повторять, пока заданное значение потока не станет равным нулю.

Исходный код

```
min_cost_flow_result_t graphs::min_cost_flow(const flow_graph_t &g, int desired_flow) {

    constexpr auto custom_dijkstra = [](const adjacency_matrix<> &capacity, const
     ↪  adjacency_matrix<> &cost,
                                        Vertex start_vertex) -> dijkstra_result_t {
        std::vector<Vertex> precedents(capacity.size(), NO_VERTEX);
        std::vector<bool> visited(capacity.size(), false);
        size_t iterations{};

        std::vector<int> distances(capacity.size(), INF);
        auto cmp = [&](Vertex i, Vertex j) {
            return distances[i] > distances[j];
        };
        std::priority_queue<Vertex, std::vector<Vertex>, decltype(cmp)> qu(cmp);
        distances[start_vertex] = 0;
        qu.push(start_vertex);
        while (!qu.empty()) {
            auto nearest_vertex = qu.top();
            qu.pop();
            if (visited[nearest_vertex]) {
                continue;
            }
            visited[nearest_vertex] = true;
            for (Vertex v{}; v < capacity.size(); ++v) {
                if (capacity[nearest_vertex][v] == 0) {
                    continue;
                }
               auto distance_via_nearest = distances[nearest_vertex] + cost[nearest_vertex][v];
                if (distance_via_nearest < distances[v]) {
                    distances[v] = distance_via_nearest;
                    qu.push(v);
                    visited[v] = false;
                    precedents[v] = nearest_vertex;
                }
                iterations++;
            }
        }
        return {
            .distances = distances,
            .precedents = precedents,
            .iterations = iterations
        };
    };

    auto my_cost = g.cost;
    for (Vertex i{}; i < g.capacity.size(); ++i) {
        for (Vertex j{}; j < g.capacity.size(); ++j) {
```

```
            if (g.capacity[i][j] == 0) {
                continue;
            }
            my_cost[j][i] = -my_cost[i][j];
        }
    }
    auto my_capacity = g.capacity;

    int flow = 0;
    int cost = 0;
    adjacency_matrix<> flows(g.capacity.size(), std::vector<int>(g.capacity.size()));
    while (flow < desired_flow) {
        auto [cost_to_reach, precedent, _] = custom_dijkstra(my_capacity, my_cost, g.source);
        if (cost_to_reach[g.sink] == INF) {
            break;
        }
        // find max flow on that path
        int f = desired_flow - flow;
        for (auto cur = g.sink; cur != g.source; cur = precedent[cur]) {
            f = std::min(f, my_capacity[precedent[cur]][cur]);
        }

        // apply flow
        flow += f;
        cost += f * cost_to_reach[g.sink];

        for (auto cur = g.sink; cur != g.source; cur = precedent[cur]) {
            my_capacity[precedent[cur]][cur] -= f;
            my_capacity[cur][precedent[cur]] += f;
            flows[precedent[cur]][cur] += f;
        }
    }
    return {
        .flow_sum = flow,
        .cost = flow < desired_flow ? -1 : cost,
        .flow = flows
    };
}
```

2.5 Поиск минимального остова

2.5.1 Алгоритм Прима

Функция `min_st_result_t kruskal_mst(const adjacency_matrix<>& g);`

Вход функции Матрица смежности g.

Выход функции Структура `min_st_result_t`. Остов минимальной стоимости представлен множеством дуг.

```
struct min_st_result_t {
    std::set<edge_t> spanning_tree;
    // sum of costs of edges
    size_t cost;
    // iterations count
    size_t iterations;
};
```

Описание работы функции Функция следует алгоритму 6.

1. Берётся произвольная вершина и находится ребро, инцидентное данной вершине и обладающее наименьшей стоимостью. Найденное ребро и соединяемые им две вершины образуют дерево.
2. Рассматриваются рёбра графа, один конец которых — уже принадлежащая дереву вершина, а другой — ещё не принадлежащая. Из этих рёбер выбирается ребро наименьшей стоимости.
3. Выбираемое на каждом шаге ребро присоединяется к дереву. Рост дерева происходит до тех пор, пока не будут исчерпаны все вершины исходного графа.

Сложность алгоритма: $\mathcal{O}(p^2)$.

Исходный код

```
min_st_result_t
graphs::kruskal_mst(const adjacency_matrix<> &g) {
    std::set<edge_t> result;
    auto edges = edges_of(g);
    std::sort(edges.begin(), edges.end());
    size_t iterations = 0;

    // disjoint set union
    struct dsu_t {
        // сжатый массив предков, т.е. для каждой вершины там может храниться
        // не непосредственный предок, а предок предка, предок предка предка, и т.д.
        std::vector<Vertex> parent;
        std::vector<size_t> rank;
        size_t& m_iterations;

        explicit dsu_t(size_t nVertices, size_t& iterations)
            : parent(nVertices), rank(nVertices), m_iterations{iterations}{
            for (int i = 0; i < nVertices; ++i) {
                parent[i] = i;
            }
        }

        Vertex operator()(Vertex v) {
            return v == parent[v] ? v : (++m_iterations, parent[v] = operator()(parent[v]));
        }

        void unite(Vertex a, Vertex b) {
            a = operator()(a);
            b = operator()(b);
            if (a == b) {
                return;
            }
            if (rank[a] < rank[b]) {
                std::swap(a, b);
            }
            parent[b] = a;
            ++m_iterations;
            if (rank[a] == rank[b]) {
                ++rank[a];
            }
        }
    };

    dsu_t dsu{g.size(), iterations};
```

```
    auto it = edges.begin();
    size_t cost = 0;
    for (int _ = 1; _ < g.size(); ++_) {
        while (dsu(it->from) == dsu(it->to)) {
            iterations++;
            it++;
        }
        dsu.unite(it->from, it->to);
        result.insert(*it);
        cost += it->weight;
        it++;
    }
    return {
        .spanning_tree = result,
        .cost = cost,
        .iterations = iterations
    };
}
```

2.5.2 Алгоритм Краскала

Функция `min_st_result_t prim_mst(const adjacency_matrix<>& g);`

Вход функции Матрица смежности `g`.

Выход функции Структура `min_st_result_t`.

Описание работы функции Функция следует алгоритму 7.

Каждая вершина помещается в свое множество.

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

Если нет, то добавляем данное ребро в 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 рёбер, является его остовным деревом минимального веса.

Подмножества множества вершин представлены структурой данных «Система непересекающихся множеств» (disjoint-set) вместе с эвристикой сжатия путей и эвристикой Union-By-Height (ранг дерева — его высота). Ее использование позволяет эффективно определить операции объединения множеств и поиска компоненты связности.

Сложность алгоритма: $\mathcal{O}(q \log p)$.

Исходный код

```
min_st_result_t
graphs::prim_mst(const adjacency_matrix<> &g) {
    size_t iterations = 0;
```

```
    size_t cost = 0;
    std::set<Vertex> vertexes; // V \ S
    for (Vertex i = 0; i < g.size(); ++i) {
        vertexes.insert(i);
    }
    std::vector<int> min_e(g.size(), INF); // вес наименьшего допустимого ребра из вершины β
    std::vector<Vertex> sel_e(g.size(), NO_VERTEX); // конец этого наименьшего ребра α
    min_e[0] = 0;
    auto spanning_tree = std::set<edge_t>{};
    for (Vertex i = 0; i < g.size(); ++i)
    {
        Vertex v = NO_VERTEX;
        for (Vertex j : vertexes) {
            if (v == NO_VERTEX || min_e[j] < min_e[v]) {
                v = j;
            }
        }
        if (min_e[v] == INF) {
            // No minimum spanning tree
            return {{}, INF, iterations};
        }
        vertexes.erase(v); // означает, что вершина включена в остов

        for (Vertex to{}; to < g.size(); ++to) {
            if (g[v][to] != 0 && g[v][to] < min_e[to]) {
                min_e[to] = g[v][to];
                sel_e[to] = v;
            } else if (g[to][v] != 0 && g[to][v] < min_e[to]) {
                min_e[to] = g[to][v];
                sel_e[to] = v;
            }
            iterations++;
        }

        if (sel_e[v] == NO_VERTEX) {
            continue;
        }
        spanning_tree.insert(edge_t{v, sel_e[v], min_e[v]});
        cost += min_e[v];
    }
    return {
        .spanning_tree = spanning_tree,
        .cost = cost,
        .iterations = iterations,
    };
}
```

2.5.3 Матричная теорема Кирхгофа

Подсчет количества остовных деревьев согласно теореме Кирхгофа.

Функция `size_t spanning_trees_count(const adjacency_matrix<>& g);`

Вход функции Матрица смежности g.

Выход функции Количество остовных деревьев.

Описание работы функции Матрица Кирхгофа строится по определению, ее минор находится рекурсивно, по формуле разложения определителя по строке.

Исходный код

```
size_t graphs::spanning_trees_count(const adjacency_matrix<> &g) {
    auto b_matrix = kirchhoff_matrixify(g);
    // minor of i = 0, j = g.size() - 1
    adjacency_matrix<> m(++b_matrix.begin(), b_matrix.end());
    for (auto& row: m) {
        row.pop_back();
    }
    // (-1)^{i+j}
    return (b_matrix.size() % 2 == 0 ? -1 : 1) * determinant(m);
}

adjacency_matrix<> kirchhoff_matrixify(adjacency_matrix<> g) {
    for (Vertex i{}; i < g.size(); ++i) {
        int deg = 0;
        for (Vertex j{}; j < g.size(); ++j) {
            if (i == j) {
                continue;
            }
            g[j][i] = g[i][j] = g[i][j] != 0 ? (++deg, -1) : 0;
        }
        g[i][i] = deg;
    }
    return g;
}

size_t determinant(std::vector<std::vector<int>> a) {
    size_t n = a.size();
    // error condition, should never get here
    if (n < 1) {
        return 0;
    }
    // should not get here
    if (n == 1) {
        return a[0][0];
    }
    if (n == 2) {
        // basic 2x2 sub-matrix determinate
        // definition. When n==2, this ends the
        // the recursion series
        return a[0][0] * a[1][1] - a[1][0] * a[0][1];
    }

    // recursion continues, solve next sub-matrix
    // solve the next minor by building a
    // sub matrix
    // initialize determinant of sub-matrix
    size_t det = 0;
    size_t j2 = 0;

    // for each column in sub-matrix
    for (size_t j1 = 0; j1 < n; j1++) {
        // get space for the pointer list
        auto m = std::vector<std::vector<int>>(n - 1, std::vector<int>(n - 1));
        // build sub-matrix with minor elements excluded
        for (size_t i = 1; i < n; i++) {
```

```
            // start at first sum-matrix column position
            j2 = 0;
            // loop to copy source matrix less one column
            for (size_t j = 0; j < n; j++) {
                // don't copy the minor column element
                if (j == j1) continue;

                // copy source element into new sub-matrix
                m[i - 1][j2] = a[i][j];
                // i-1 because new sub-matrix is one row
                // (and column) smaller with excluded minors
                j2++; // move to next sub-matrix column position
            }
        }
        det += (j1 % 2 == 0 ? 1 : -1) * a[0][j1] * determinant(m);
        // sum x raised to y power
        // recursively get determinant of next
        // sub-matrix which is now one
        // row & column smaller
    }
    return det;
}
```

2.6 Код Прюфера

2.6.1 Кодирование Прюфера

Функция `std::pair<std::vector<Vertex>, std::vector<int>> encode(const adjacency_matrix<>& g );`

Вход функции Матрица весов g (ориентированного графа).

Выход функции Код Прюфера минимального остова графа.

Описание работы функции Функция реализует алгоритм 8.

В цикле выбирается лист с наименьшим номером, в код Прюфера добавляется номер вершины, связанной с этим листом. Процедура повторяется $p - 1$ раз. Параллельно запоминаются ребра, связанные с текущим листом.

Исходный код

```
std::pair<std::vector<Vertex>, std::vector<int>>
graphs::pruefer::encode(const adjacency_matrix<> &g) {
    auto adj = list_from(kruskal_mst(g).spanning_tree, g.size());
    size_t n = adj.size();
    std::set<Vertex> leafs;
    std::vector<size_t> degree(n);
    std::vector<bool> killed(n, false);
    for (int i = 0; i < n; i++) {
        degree[i] = adj[i].size();
        if (degree[i] == 1)
            leafs.insert(i);
    }
    std::vector<Vertex> code(n - 2);
    std::vector<int> weights(n - 1);
    for (int i = 0; i < n - 1; i++) {
```

```
        Vertex leaf = *leafs.begin();
        leafs.erase(leafs.begin());
        killed[leaf] = true;
        Vertex v;
        int weight;
        bool check_me = false;
        for (auto [u, u_weight] : adj[leaf]) {
            if (killed[u]) {
                continue;
            }
            v = u;
            weight = u_weight;
            check_me = true;
            break;
        }
        assert(check_me);
        if (i != n - 2) {
            code[i] = v;
        }
        weights[i] = weight;
        if (--degree[v] == 1) {
            leafs.insert(v);
        }
    }
    return {code, weights};
}
```

2.6.2 Декодирование Прюфера

Функция `adjacency_matrix<> decode(const std::vector<Vertex>& code, const std::vector<int>& weights);`

Вход функции Код Прюфера: вектор номеров вершин и весов ребер.

Выход функции Матрица весов дерева, соответствующего коду. По построению матрица верхнетреугольная (орграф).

Описание работы функции Функция реализует алгоритм 9.

По построенному коду можно восстанавливается исходное дерево:

Берётся первый элемент кода Прюфера, и по всем вершинам дерева производится поиск вершины с наименьшим номером, не содержащейся в коде.

Найденная вершина и текущий элемент составляют ребро дерева. Вес ребра извлекается из массива весов.

Исходный код

```
adjacency_matrix<>
graphs::pruefer::decode(const std::vector<Vertex>& code, const std::vector<int>& weights) {
    size_t nVertices = weights.size() + 1;
    assert(code.size() == nVertices - 2);
    adjacency_matrix<> g(nVertices, std::vector<int>(nVertices));
    std::vector<size_t> degree(nVertices, 1);
    for (Vertex i : code) {
        degree[i]++;
```

```
    }
    std::set<Vertex> leaves;
    for (int i = 0; i < nVertices; i++) {
        if (degree[i] == 1)
            leaves.insert(i);
    }
    for (size_t i = 0; i < code.size(); ++i) {
        Vertex v = code[i];
        Vertex leaf = *leaves.begin();
        leaves.erase(leaves.begin());
        if (--degree[v] == 1)
            leaves.insert(v);
        if (leaf > v) {
            std::swap(leaf, v);
        }
        g[leaf][v] = weights[i];
    }
    auto [a, b] = std::minmax(*leaves.begin(), nVertices - 1);
    g[a][b] = weights.back();
    return g;
}
```

2.7 Эйлеровы и Гамильтоновы графы

2.7.1 Проверка графа на эйлеровость

Функция `bool is_eulerian(const adjacency_matrix<>& g);`

Вход функции Матрица смежности (ориентированного) графа g.

Выход функции `true` если неориентированный граф, полученный из исходного забыванием направления дуг, — эйлеров. Иначе `false`.

Описание работы функции Непосредственно проверяется критерий эйлеровости.

Исходный код

```
bool graphs::is_eulerian(const adjacency_matrix<>& g) {
    size_t nVertices = g.size();
    assert(nVertices > 2);
    auto degree = degrees_of(unoriented(g));
    return std::all_of(degree.begin(), degree.end(),
                       [](int d) { return d % 2 == 0;});
}
```

2.7.2 Поиск Эйлерова цикла в эйлеровом графе

Функция `path_t euler_cycle(adjacency_matrix<> g);`

Вход функции Матрица смежности (ориентированного) графа g.

Выход функции Эйлеров цикл неориентированного графа, полученного из исходного забыванием направления дуг. Если такого цикла нет, то — пустой массив.

Описание работы функции Функция реализует алгоритм 10.

Начав с вершины v (с наименьшей степенью), строим путь, удаляя рёбра и запоминая вершины в стеке, до тех пор пока множество смежности очередной вершины не окажется пустым, что означает, что путь удлинить нельзя.

Исходный код

```
path_t graphs::euler_cycle(adjacency_matrix<> g) {
    size_t nVertices = g.size();
    g = unoriented(g);
    std::vector<Vertex> result;
    result.reserve(nVertices);
    std::stack<Vertex> s;
    auto degree = degrees_of(g);
    s.push(std::min_element(degree.begin(), degree.end()) - degree.begin());
    while (!s.empty()) {
        Vertex i = s.top();
        if (degree[i] == 0) {
            s.pop();
            result.push_back(i);
            continue;
        }
        for (int j = 0; j < nVertices; j++) {
            if (g[i][j] == 0) continue;
            g[i][j] = 0;
            g[j][i] = 0;
            degree[i] -= 1;
            degree[j] -= 1;
            s.push(j);
            break;
        }
    }
    return result;
}
```

2.7.3 Проверка наличия гамильтонова цикла в неориентированном графе

Функция `bool is_hamiltonian(adjacency_matrix<> g);`

Вход функции Матрица смежности (ориентированного) графа `g`.

Выход функции `true` если неориентированный граф, полученный из исходного забыванием направления дуг, — гамильтонов. Иначе `false`.

Описание работы функции Сначала выполняются проверки (достаточного) условия теоремы Дирака и необходимого условия. Если обе проверки несостоятельны, то производится поиск гамильтонова цикла.

Исходный код

```
bool graphs::is_hamiltonian(adjacency_matrix<> g) {
    if (g.size() <= 2) {
        return false;
    }
    g = unoriented(g);
    auto ds = degrees_of(g);
```

```
    // Dirac's Theorem
    if (g.size() > 3 && std::all_of(ds.begin(), ds.end(),
                                    [&](auto d) { return 2 * d >= g.size(); })) {
        return true;
    }
    // necessary condition (easy)
    if (std::any_of(ds.begin(), ds.end(),
                    [&](auto d) { return d <= 1; })) {
        return false;
    }
    hamilton_cycles hc{g};
    return hc.begin() != hc.end();
}
```

2.7.4 Задача коммивояжера

Для нахождения всех гамильтоновых циклов в графе определена структура `hamilton_cycles`, которая реализует интерфейсы, необходимые для использования ее в конструкции range-based for[3] языка C++. Внутренняя структура `hamilton_cycles::iterator` удовлетворяет требованиям LegacyInputIterator[4], что, в частности, позволяет использовать ее в качестве аргумента некоторых функций стандартной библиотеки `<algorithm>`, например, `std::for_each`. Ключевое значение имеет функция `hamilton_cycles::iterator::find_next`, о ней см. ниже.

```
struct hamilton_cycles {
    // инициализация поиска гамильтоновых циклов
    explicit hamilton_cycles(const adjacency_matrix<> &g) : g{unoriented(g)} {
    }

    struct iterator {
        using iterator_category = std::input_iterator_tag;
        using value_type = costed_path_t;
        using difference_type = std::ptrdiff_t;
        using pointer = costed_path_t*;
        using reference = costed_path_t&;

        std::vector<std::deque<Vertex>> N;
        hamilton_cycles* container;
        path_t candidate;
        costed_path_t result;

        explicit iterator(hamilton_cycles* container) : container{container} { }
        bool operator==(const iterator& rhs) const { return candidate == rhs.candidate; }
        bool operator!=(const iterator& rhs) const { return !(*this == rhs); }
        costed_path_t operator*() const { return result; }
        iterator& operator++() {
            result = find_next();
            return *this;
        }

    private:
        // стоимость пути
        static size_t cost(adjacency_matrix<> &g, decltype(candidate) &cand);
        // найти следующий гамильтонов цикл
        costed_path_t find_next();
    };
```

[3] https://en.cppreference.com/w/cpp/language/range-for
[4] https://en.cppreference.com/w/cpp/named_req/InputIterator

```
    iterator begin();
    iterator end() { return iterator{this}; }

private:
    // начальная вершина
    static const Vertex start;
    // матрица смежности
    adjacency_matrix<> g;
    // номера вершин смежных с `x`
    std::deque<Vertex> adj(Vertex x) const;
};
```

Функция `costed_path_t hamilton_cycles::iterator::find_next`

Вход функции Матрица смежности g неориентированного графа, начальная вершина цикла `start`, текущий отрезок пути `candidate`.

Выход функции Следующий гамильтонов путь или пустой массив, если таковых больше нет.

Описание работы функции Функция следует алгоритму 11.

На каждом шаге алгоритма имеется уже построенный отрезок пути, он хранится в стеке `candidate`. Для каждой вершины x, входящей в `candidate`, хранится множество $N[x]$ всех вершин, смежных с x, которые еще не рассматривались в качестве возможных продолжений пути из вершины x. Когда вершина x добавляется к пути, множество $N(x)$ полагается равным его множеству смежности. В дальнейшем рассмотренные вершины удаляются из этого множества. Очередной шаг состоит в исследовании окрестности последней вершины x пути `candidate`. Если $N(x)$ не пусто и в $N(x)$ имеются вершины, не принадлежащие пути, то одна из таких вершин y добавляется к пути. В противном случае вершина x исключается из стека.

Когда после добавления к пути очередной вершины оказывается, что путь содержит все вершины графа, остается проверить, смежны ли первая и последняя вершины пути. Если это так, то возвращается очередной гамильтонов цикл[1].

Исходный код

```
costed_path_t hamilton_cycles::iterator::find_next() {
    while (!candidate.empty()) {
        auto x = candidate.back();
        if (N[x].empty()) {
            candidate.pop_back();
            continue;
        }
        auto y = N[x].back();
        N[x].pop_back();
        if (std::find(candidate.begin(), candidate.end(), y) != candidate.end()) {
            continue;
        }
        candidate.push_back(y);
        N[y] = container->adj(y);
        if (candidate.size() != container->g.size()) {
            continue;
        }
```

```
        if (std::find(N[y].begin(), N[y].end(), start) == N[y].end()) {
            candidate.pop_back();
            continue;
        }
        auto res = candidate;
        candidate.pop_back();
        res.push_back(start);
        return {res, cost(container->g, res)};
    }
    return {};
}
```

2.7.5 Модификация графа до эйлерова

Функция `graph_change_t eulerize(const adjacency_matrix<>& g_original);`

Вход функции Матрица смежности (ориентированного) графа `g_original`.

Выход функции Структура `graph_change_t`, в которой `changed` — матрица смежности графа, полученного ориентированием ребер эйлерова графа, который получен из исходного добавлением или удалением некоторых ребер.

```
struct graph_change_t {
    adjacency_matrix<> changed; // матрица смежности модифицированного графа
    std::vector<edge_t> added; // добавленные ребра
    std::vector<edge_t> removed; // удаленные ребра
    bool has_changed() const { return !added.empty() || !removed.empty(); }
};
```

Описание работы функции Перед началом работы происходит забывание направлений дуг исходного графа. Получаем неориентированный граф.

Идея алгоритма следующая:

1. Найти вершины с нечётной степенью.
2. Выбрать паросочетание из них, желательно чтобы такого ребра ещё не было (т. е. минимальное паросочетание для двоичной матрицы смежности)
3. Для каждой пары вершин из паросочетания, добавить ребро если его не было, удалить если было
4. Если граф стал несвязным, то
5. Если компонент связности 3 или больше, соединить компоненты в цепь, выбрав в каждой из них по одной вершине, через которую она проходит.
6. Если компонент связности 2, то соединить 2 пары вершин из обеих.

Ориентирование ребер полученного графа происходит занулением весов дуг ниже главной диагонали матрицы смежности.

Исходный код

```
graph_change_t graphs::eulerize(const adjacency_matrix<>& g_original) {
    static const auto odd_only = [](const std::vector<int>& degree){
        std::vector<Vertex> result;
        result.reserve(degree.size() / 2);
        for (int i = 0; i < degree.size(); ++i) {
            if (degree[i] % 2 != 0) {
                result.push_back(i);
```

```
            }
        }
        assert(result.size() % 2 == 0 && "Number of vertices with odd degree should be even");
        return result;
    };
    size_t nVertices = g_original.size();
    if (nVertices <= 2) {
        // There are 1/2 vertices in the graph! Euler cycle is impossible!
        return {};
    }
    auto g = unoriented(g_original);
    std::vector<edge_t> added, removed;
    std::vector<int> degree;
    std::vector<Vertex> odd_vertices = odd_only(degrees_of(g));

    constexpr auto add = [](decltype(added)& added, decltype(g)& g, Vertex from, Vertex to, int
     ↪  weight = 1) {
        std::tie(from, to) = std::minmax({from, to});
        assert(!g[from][to]);
        assert(from != to);
        g[from][to] = 1;
        g[to][from] = 1;
        added.emplace_back(from, to, weight);
    };
    constexpr auto remove = [](decltype(removed)& removed, decltype(g)& g, Vertex from, Vertex
     ↪  to) {
        std::tie(from, to) = std::minmax({from, to});
        removed.emplace_back(from, to, g[from][to]);
        g[from][to] = 0;
        g[to][from] = 0;
    };

    // minimal cost matching by cost from given boolean adj matrix

    // step 1. add edges we can add with greed.
    auto used = std::vector<bool>(odd_vertices.size());
    for (size_t i = 0; i < odd_vertices.size(); ++i) {
        if (used[i]) { continue;}
        for (size_t j = i + 1; j < odd_vertices.size(); ++j) {
            if (used[j]) { continue;}
            auto from = odd_vertices[i], to = odd_vertices[j];
            if (g[from][to] == 0) {
                used[i] = used[j] = true;
                add(added, g, from, to);
                break;
            }
        }
    }
    // if used all odd vertexes ⇒ nothing else needed
    if (std::all_of(used.begin(), used.end(), [](auto x){return x;})) {
        return {
            .changed = oriented(g),
            .added = added,
            .removed = removed
        };
    }
    // step 2.1. remove matched vertexes
    for (int i = odd_vertices.size() - 1; i >= 0; --i) {
        if (used[i]) {
            odd_vertices.erase(odd_vertices.begin() +
↪   static_cast<decltype(odd_vertices)::difference_type>(i));
```

```
        }
    }
    // step 2.2. remove edges
    used = std::vector<bool>(odd_vertices.size());
    std::vector<edge_t> cut_edges;
    for (size_t i = 0; i < odd_vertices.size(); ++i) {
        if (used[i]) { continue;}
        for (size_t j = i + 1; j < odd_vertices.size(); ++j) {
            if (used[j]) { continue;}
            auto from = odd_vertices[i], to = odd_vertices[j];
            if (g[from][to] != 0) {
                used[i] = used[j] = true;
                remove(removed, g, from, to);
                // check if there still exists path between `from` and `to`.
                auto dij = min_path_distances_dijkstra(g, from);
                // If not, we have cut the bridge and should fix that!
                if (dij.distances[to] == INF) {
                    cut_edges.push_back(removed.back());
                }
                break; // go to next i iteration
            }
        }
    }
    if (cut_edges.size() > 1) {
        // we have >=3 convex components HOORAY!
        // step 3. choose 1 vertex from each component and join them into cycle.
        assert(("we found several cut edges and didn't manage to connect them greedily on step
         ↪  1. Awful.", false)); // need to check
    }
    else if (cut_edges.size() == 1) {
        auto [from, to, _] = cut_edges.front();
        auto dij = min_path_distances_dijkstra(g, from);

        // recover that edge (undo removing)
        add(added, g, from, to);
        added.pop_back();
        removed.pop_back();

        // find vertex i in first or second component, and use it to recover connection
        Vertex i;
        for (i = 0; i < g.size(); ++i) {
            if (g[from][i] != 0 && i != to) {
                remove(removed, g, from, i);
                add(added, g, i, to);
                break;
            }
            if (g[to][i] != 0 && i != from) {
                remove(removed, g, to, i);
                add(added, g, i, from);
                break;
            }
        }
        assert(("we did't found such vertex. Awfuuuuul!", i != g.size()));
    }
    assert(is_eulerian(g));
    return {
        .changed = oriented(g),
        .added = added,
        .removed = removed
    };
```

```
}
```

2.7.6 Дополнение графа до гамильтонова

Функция `graph_change_t hamiltonize(const adjacency_matrix<>& g);`

Вход функции Матрица смежности (ориентированного) графа `g`.

Выход функции Структура `graph_change_t`, в которой `changed` — матрица смежности графа, полученного ориентированием ребер гамильтонова графа, который получен из исходного добавлением некоторых ребер.

Описание работы функции Сперва все висячие вершины соединяются путем. Так не остается ни одной висячей вершины и выполняется очевидное необходимое условие гамильтоновости.

Если получившийся граф гамильтонов, функция завершается.

Затем производится попытка добавления сначала одной, затем двух, а потом трех ребер. Если после добавления граф становится гамильтоновым, функция завершается, если нет, то ребро удаляется обратно и поиск продолжается. Если же не нашлось тройки ребер, после добавление которых граф становится гамильтоновым, то функция завершает свою работу аварийно.

Попытка добавления всех возможных `level` ребер и проверка на гамильтоновость производится рекурсивной лямбда-функцией `iterate`, объявленной внутри функции `hamiltonize`. Рекурсия в этом случае возможна благодаря передаче ссылки на функциональный объект при вызове. Глубина рекурсии ограничивается передаваемым параметром `level`.

Ориентирование ребер графа производится из условия, что получившаяся матрица верхнетреугольная.

Исходный код

```
graph_change_t graphs::hamiltonize(const adjacency_matrix<> &g_orig) {
    auto g = unoriented(g_orig);
    auto added = std::vector<edge_t>{};
    {
        auto ds = degrees_of(g);
        auto leafs = std::deque<Vertex>{};
        // connect all leafs with a path
        for (Vertex i{}; i < g.size(); ++i) {
            if (ds[i] <= 1) {
                leafs.push_back(i);
            }
            if (leafs.size() > 1) {
                assert(g[leafs[0]][leafs[1]] == 0);
                g[leafs[1]][leafs[0]] = g[leafs[0]][leafs[1]] = 1;
                added.emplace_back(leafs[0], leafs[1], 1);
                leafs.pop_front();
            }
        }
    }
    auto result = [&]() -> graph_change_t {
        return {
            .changed = oriented(g),
```

```
            .added = added,
            .removed = {}
        };
    };
    if (is_hamiltonian(g)) {
        return result();
    }

    // Create list of no-edges
    auto no_edges = std::vector<edge_t>{};
    no_edges.reserve(g.size() * g.size() / 4);
    for (Vertex i = 0; i < g.size(); ++i) {
        for (Vertex j = i + 1; j < g.size(); ++j) {
            if (g[i][j] == 0) {
                no_edges.emplace_back(i, j, 1);
            }
        }
    }

    int zero = (int)added.size();

    auto iterate = [&](auto& that, int level) {
        if (level == 0) {
            return is_hamiltonian(g);
        }
        for (auto e : no_edges) {
            if (std::find(added.begin() + zero, added.end(), e) != added.end()) {
                continue;
            }
            // Try with edge e
            g[e.from][e.to] = g[e.to][e.from] = e.weight;
            added.push_back(e);
            if (that(that, level - 1)) {
                return true;
            }
            // If it still doesn't work, remove edge e
            g[e.from][e.to] = g[e.to][e.from] = 0;
            added.pop_back();
        }
        return false;
    };
    for (int k = 1; k <= 3; ++k) {
        if (iterate(iterate, k)) {
            return result();
        }
    }
//    assert(is_hamiltonian(g));
    return result();
}
```

3 Результаты работы программы

Генерация случайного графа из 10 вершин продемонстрирована на рис. 3.

Таблица 1: Матрица весов сгенерированного графа (ациклического, связного, ориентированного)

	a	*b*	*c*	*d*	*e*	*f*	*g*	*h*	*i*	*j*
a	—	—	27	—	37	—	—	—	13	—
b	—	—	—	—	—	—	—	10	—	29
c	—	—	—	—	—	—	30	28	—	—
d	—	—	—	—	—	—	34	—	—	37
e	—	—	—	—	—	—	—	21	—	—
f	—	—	—	—	—	—	10	—	—	—
g	—	—	—	—	—	—	—	—	—	25
h	—	—	—	—	—	—	—	—	—	8
i	—	—	—	—	—	—	—	—	—	10
j	—	—	—	—	—	—	—	—	—	—

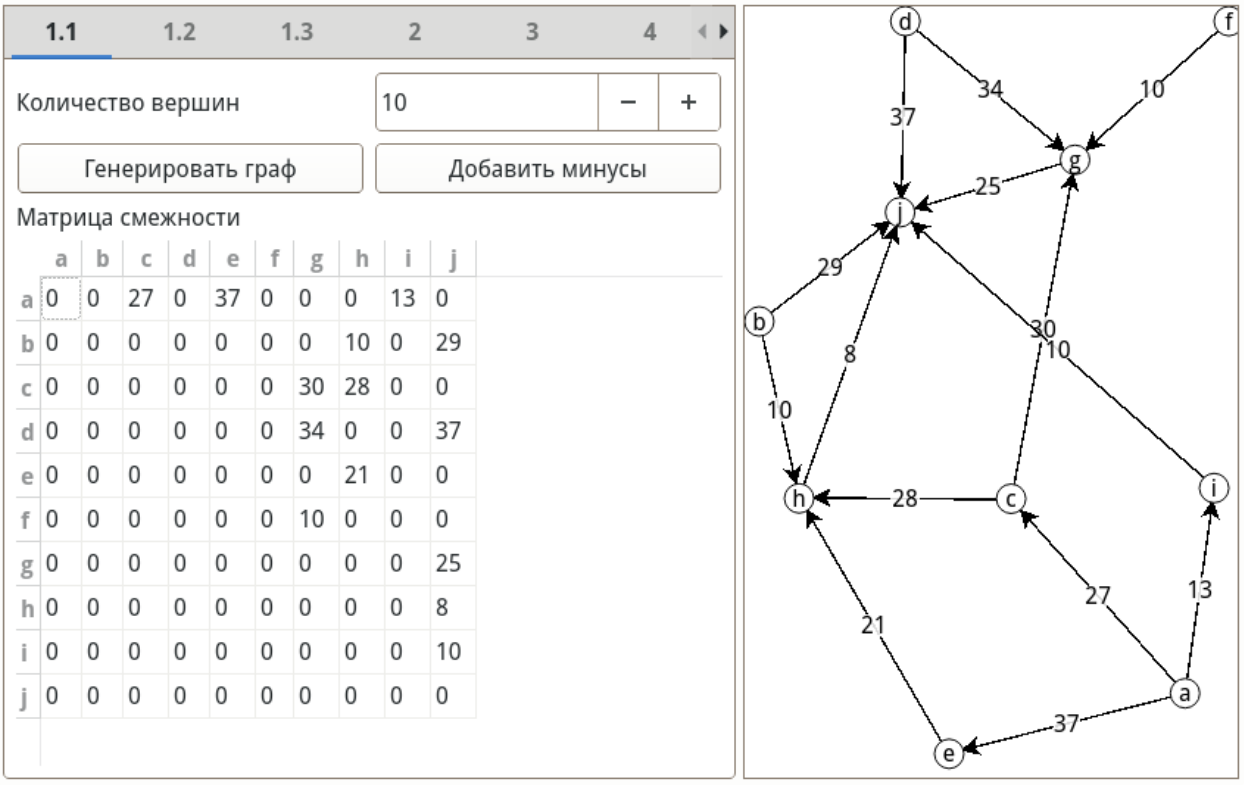

Рис. 3: Генерация графа, $p = 10$

На рис. 4 представлена матрица Шимбелла минимальных путей для двух рёбер, на рис. 5 — максимальных.

На рис. 6 показана проверка существования пути из вершины *a* в вершину *h*.

Следующие три рисунка (рис. 7-9) показывают работу алгоритмов по поиску кратчайших путей в графе.

Рис. 10-13 показывают работу программы по поиску максимального потока по алгоритму Форда-Фалкерсона, а также поиск потока минимальной стоимости. В том числе и генерацию

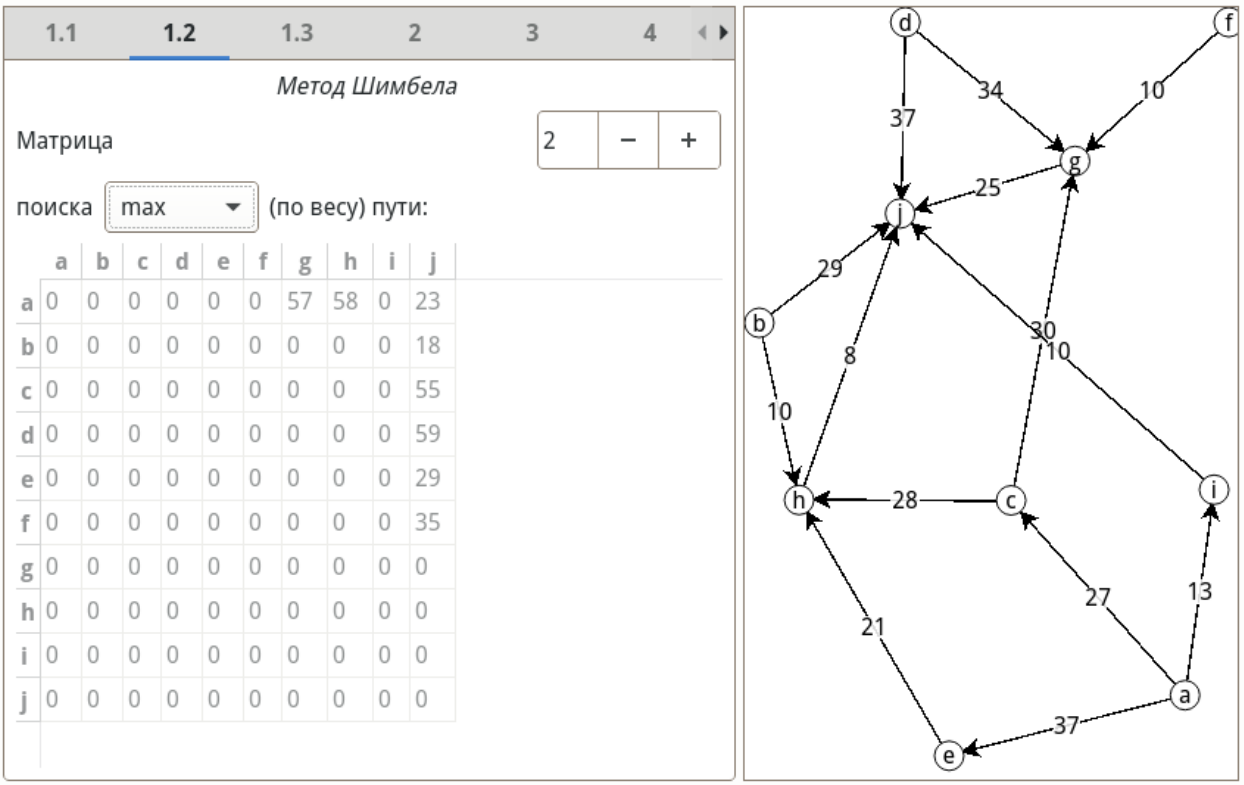

Рис. 4: Метод Шимбелла. Поиск минимального пути, матрица с положительными весами

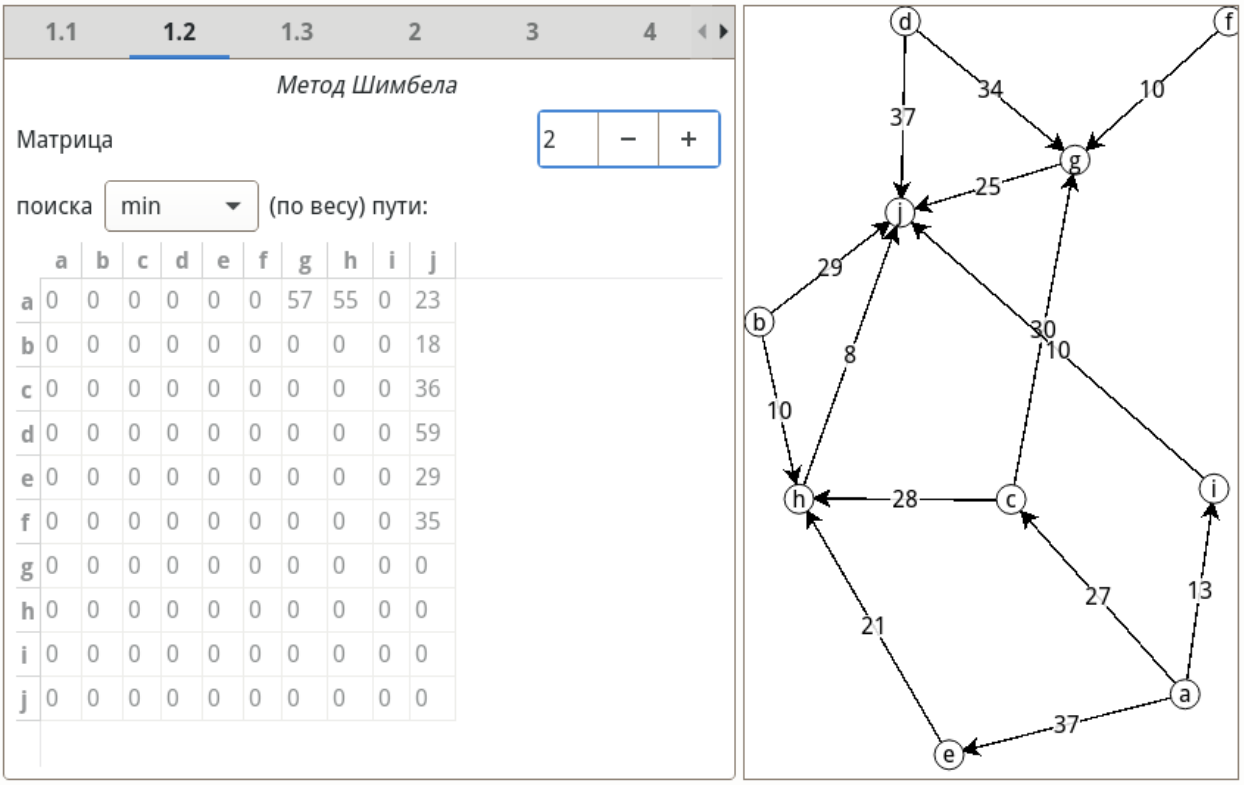

Рис. 5: Метод Шимбелла. Поиск максимального пути, матрица с положительными весами

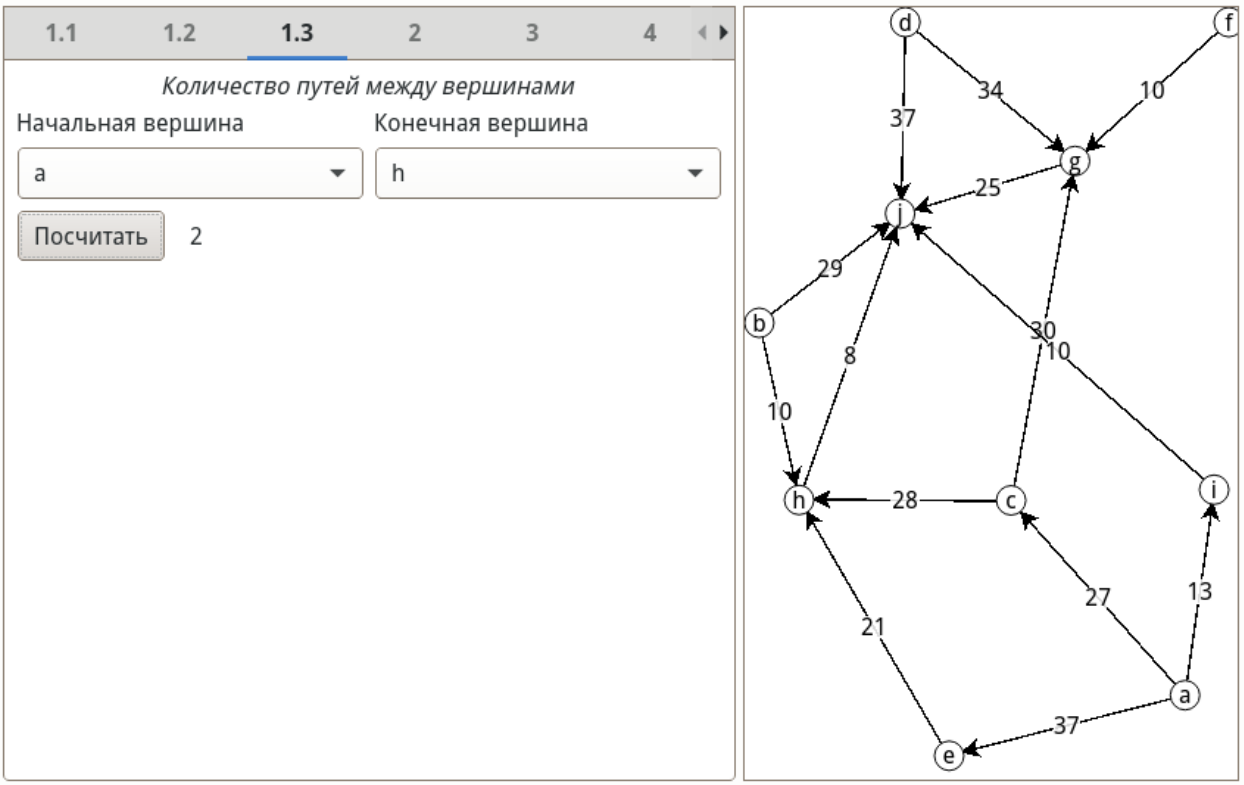

Рис. 6: Существование и количество путей между вершинами *a*, *h*

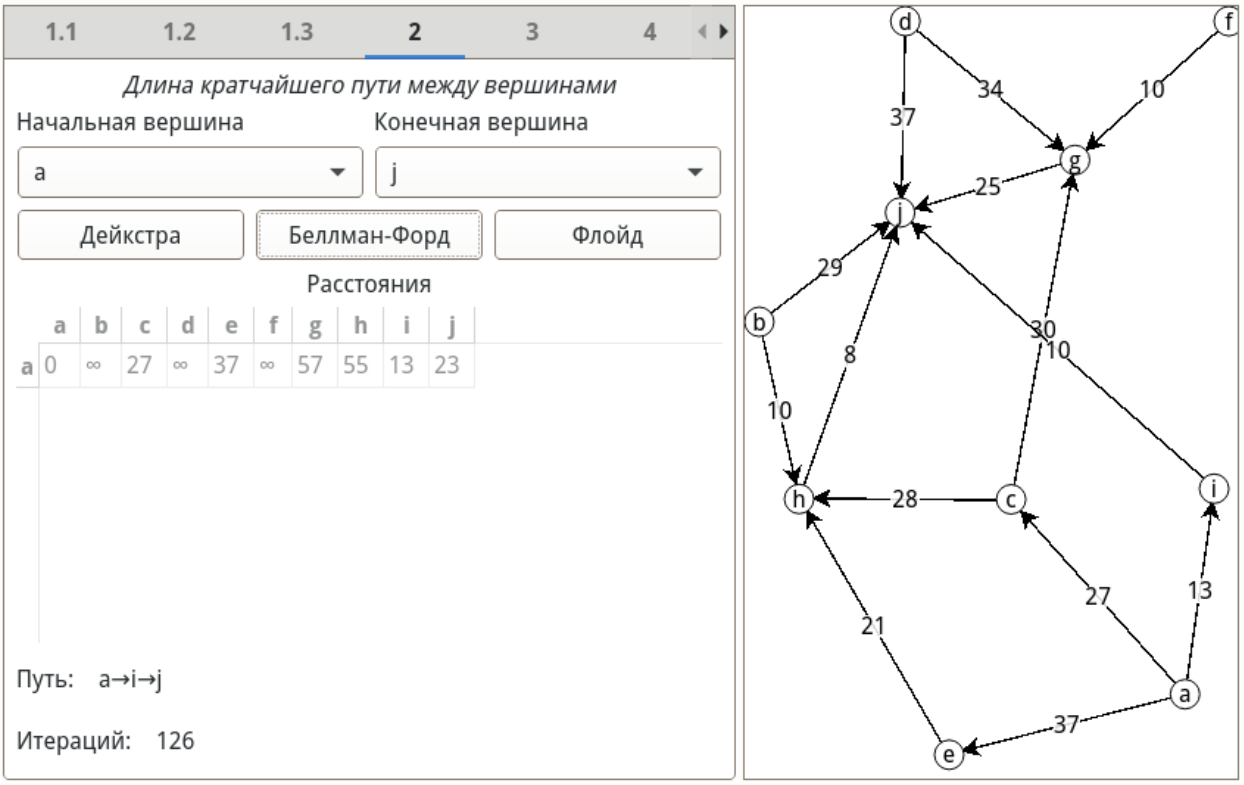

Рис. 7: Алгоритм Дейкстры

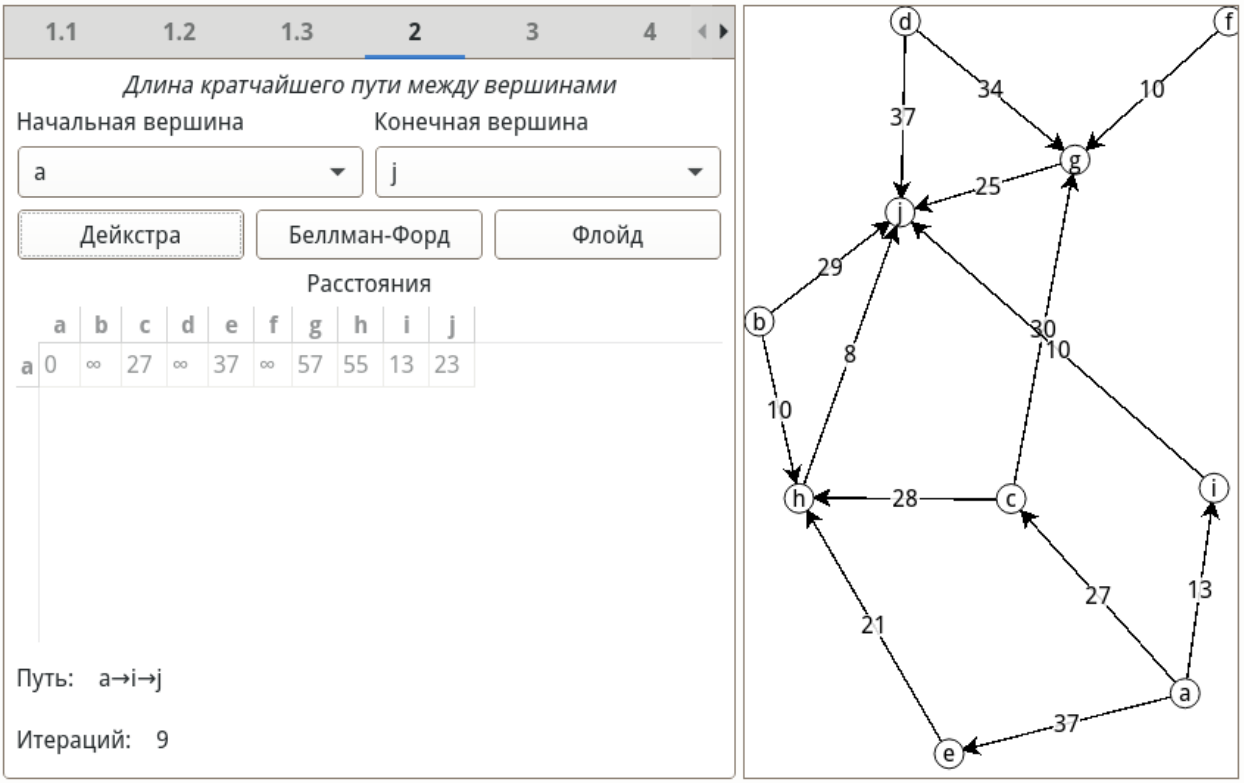

Рис. 8: Алгоритм Беллмана-Форда

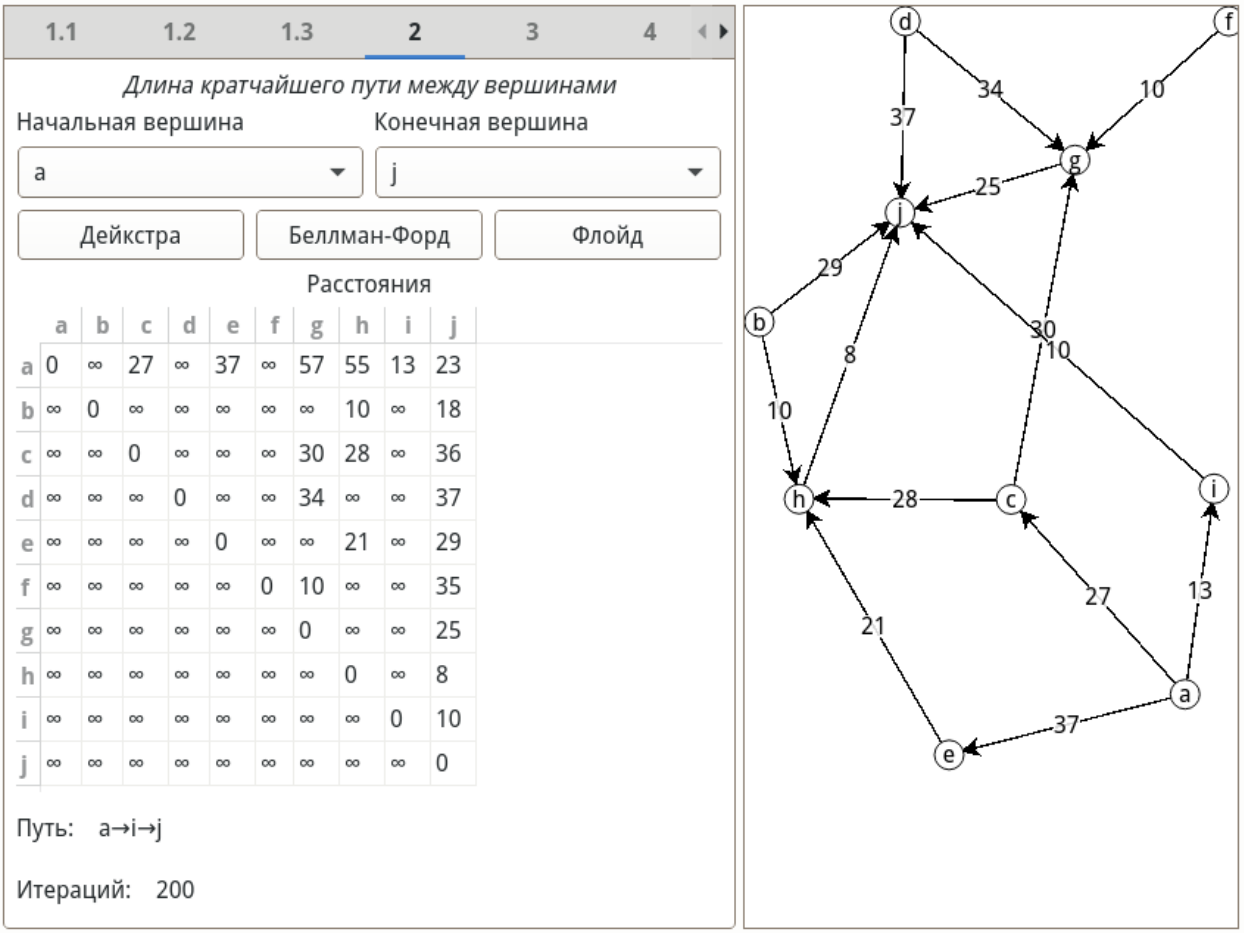

Рис. 9: Алгоритм Флойда-Уоршелла

матрицы стоимостей.

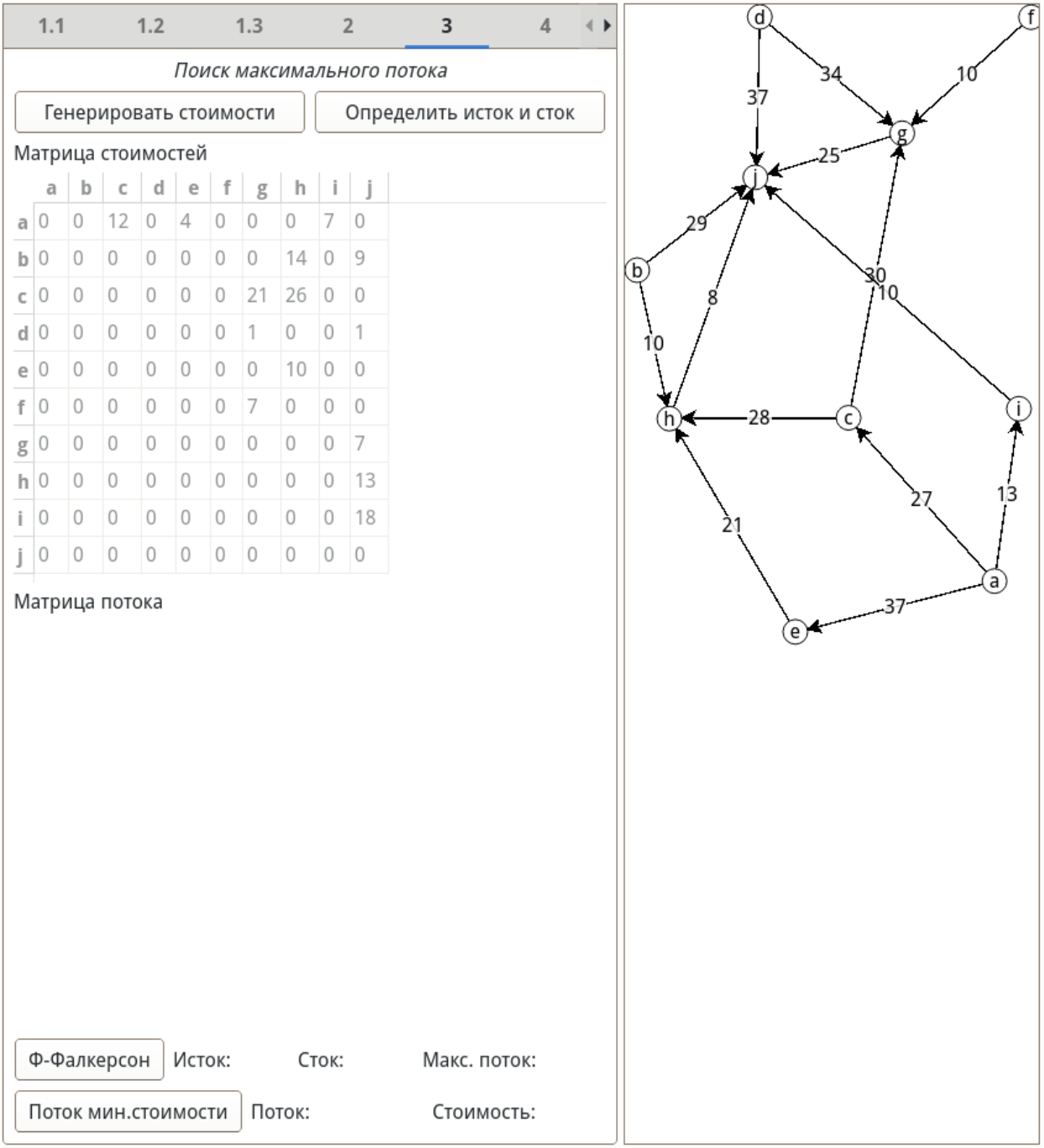

Рис. 10: Генерация матрицы стоимостей

Поиск количества остовных деревьев по матричной теореме Кирхгофа показан на рис. 14.

Следующие два рисунка (рис. 15-16) показывают работу алгоритмов по поиску минимального остова в (неориентированном) графе. Остов выделен цветом.

На рис. 17 представлено кодирование полученного минимального остова кодом Прюфера.

На рис. 18 представлено декодирование кода Прюфера.

На рис. 19 показан результат работы алгоритма, который проверяет является ли граф эйлеровым. На рис. 20 — гамильтоновым.

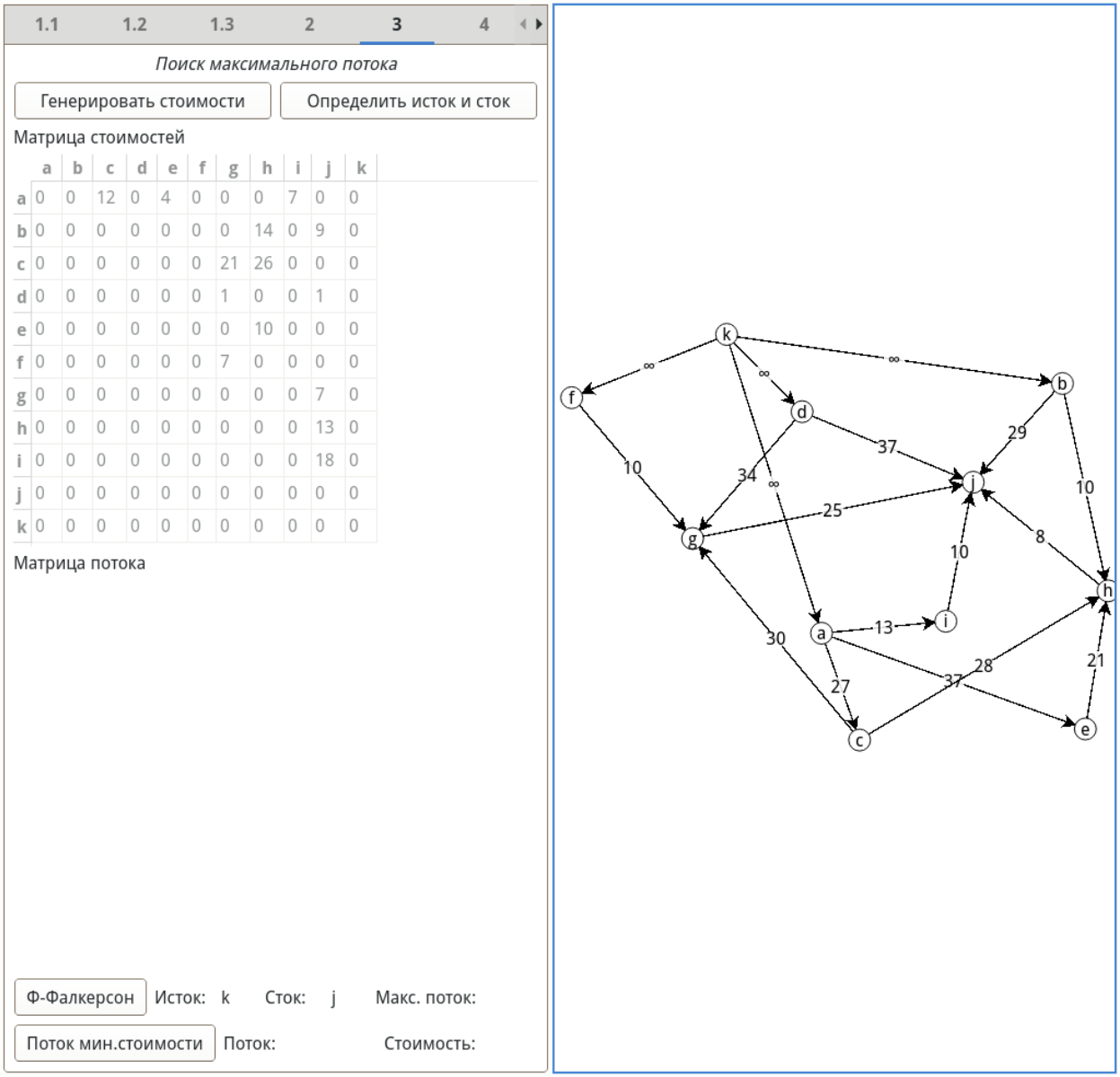

Рис. 11: Поиск и добавление (фиктивных) стока и истока

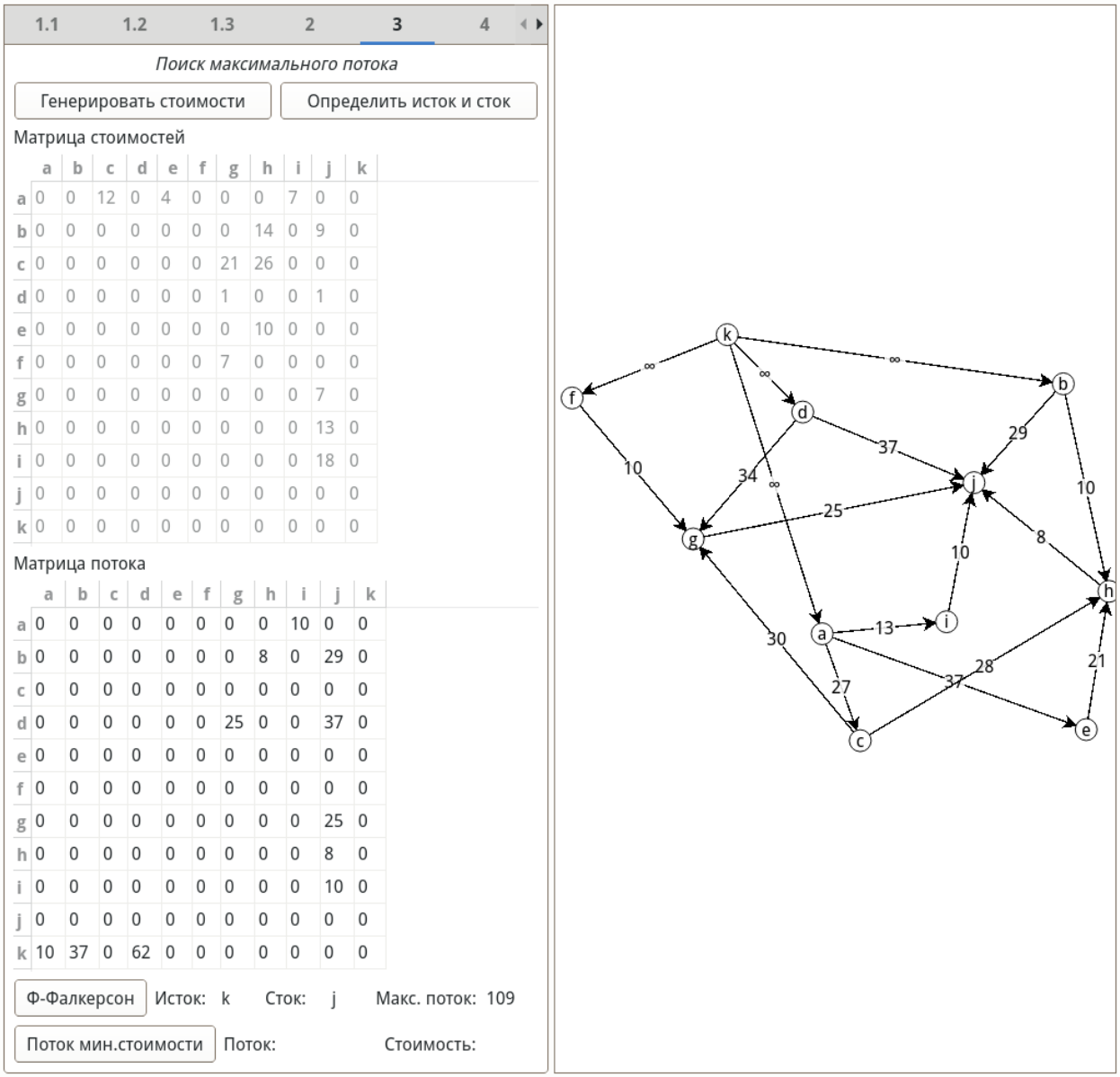

Рис. 12: Алгоритм Форда-Фалкерсона

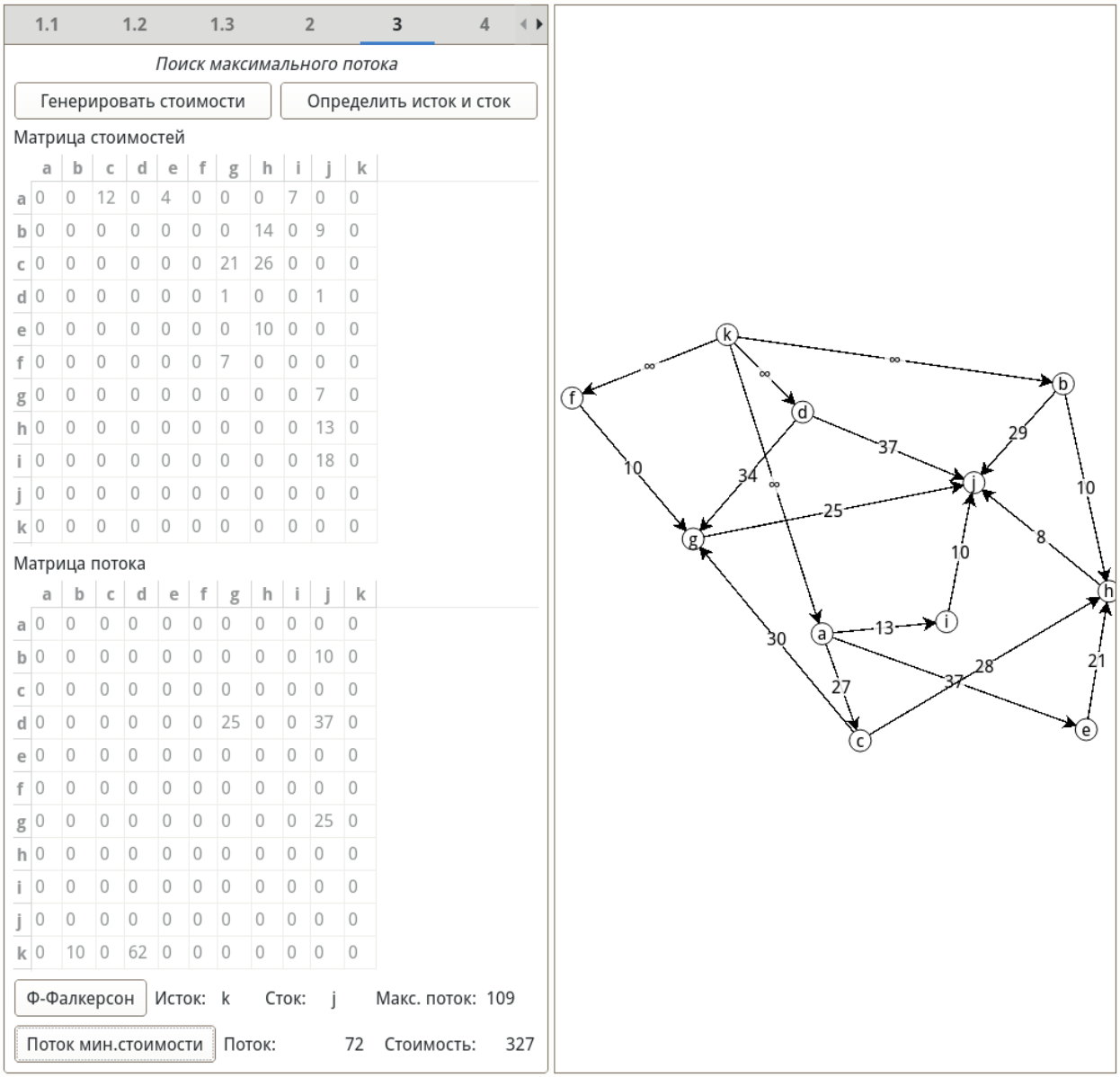

Рис. 13: Поиск потока минимальной стоимости заданной величины

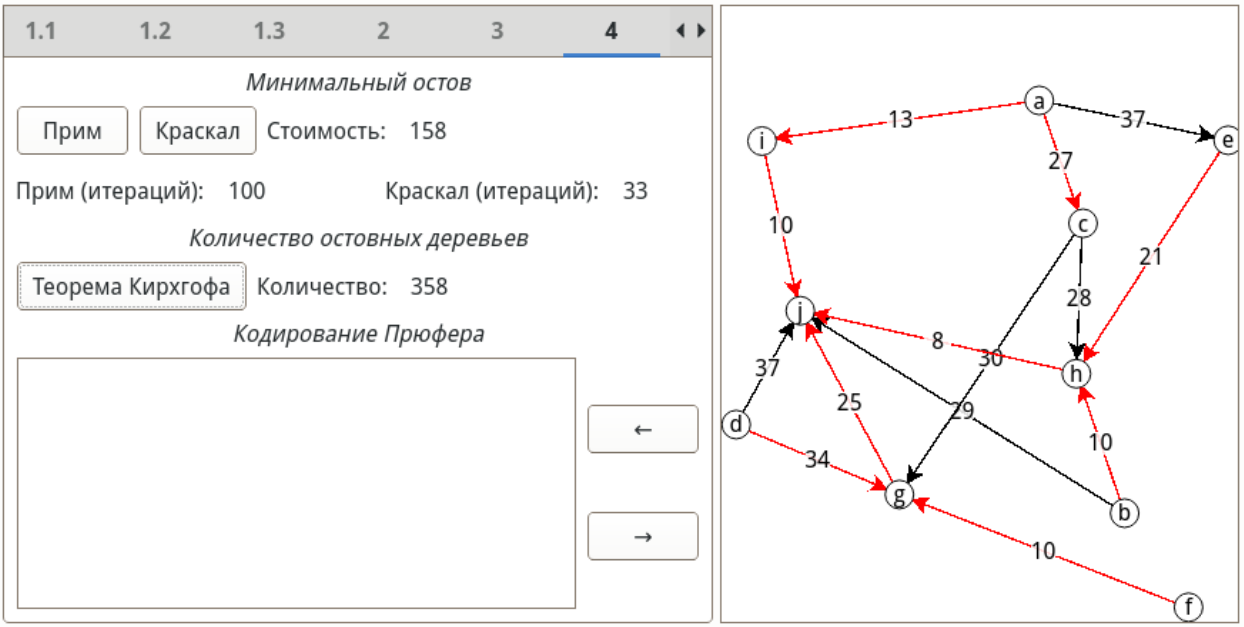

Рис. 14: Подсчет числа остовных деревьев по матричной теореме Кирхгофа

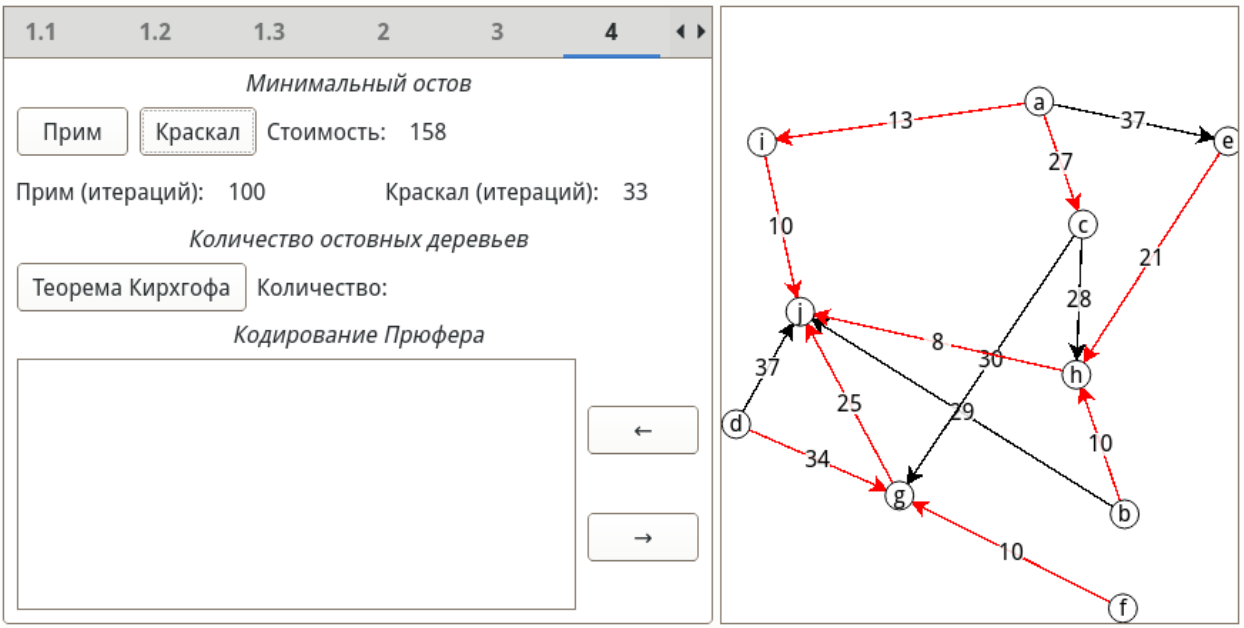

Рис. 15: Алгоритм Краскала

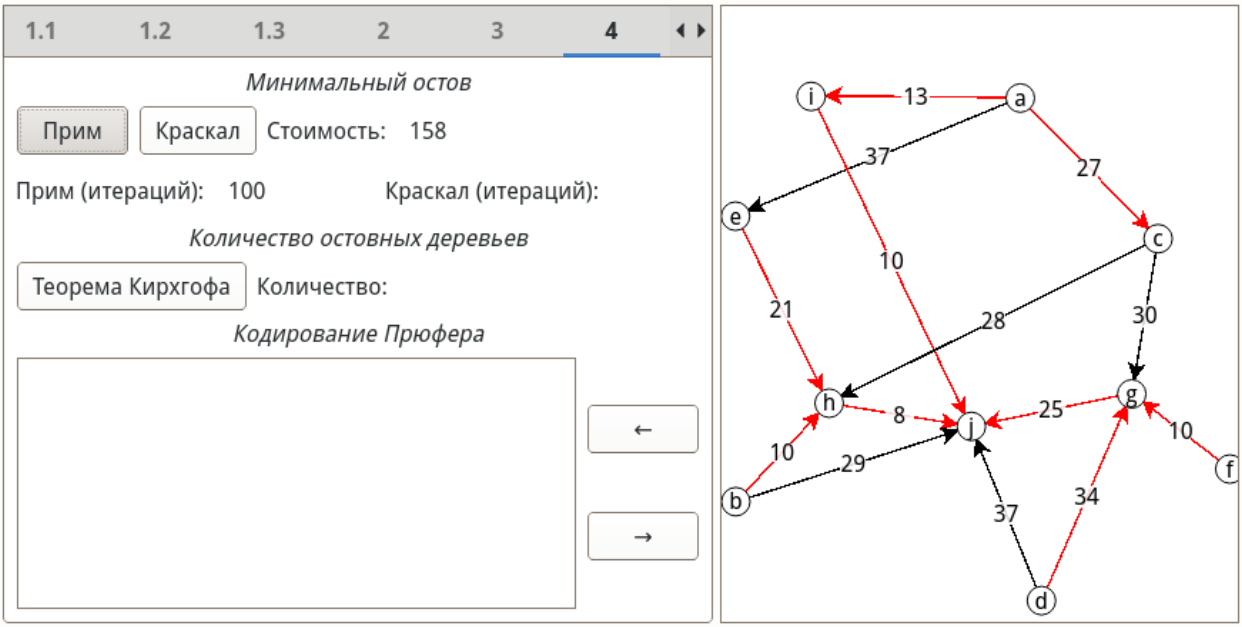

Рис. 16: Алгоритм Прима

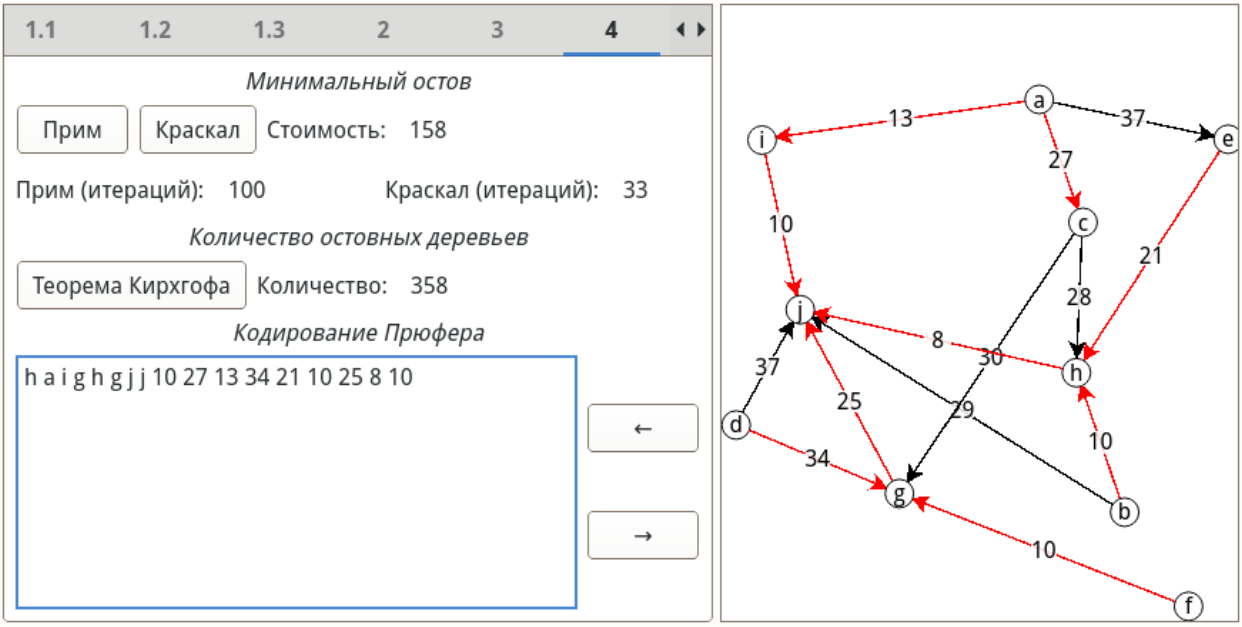

Рис. 17: Кодирование Прюфера

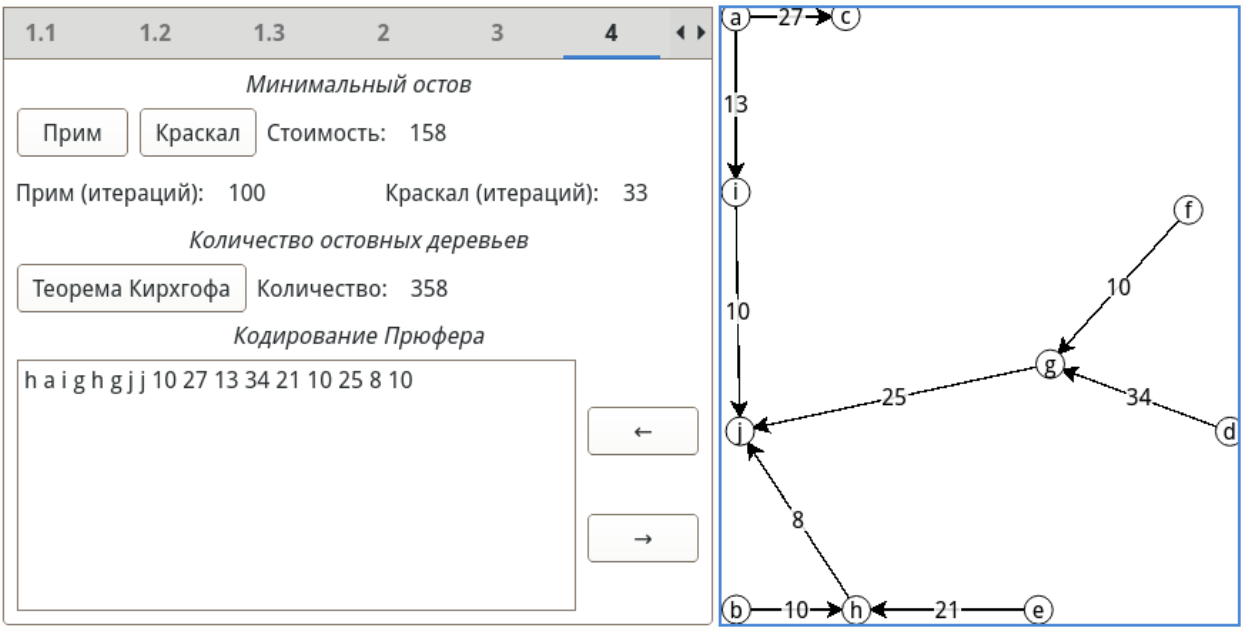

Рис. 18: Декодирование кода Прюфера

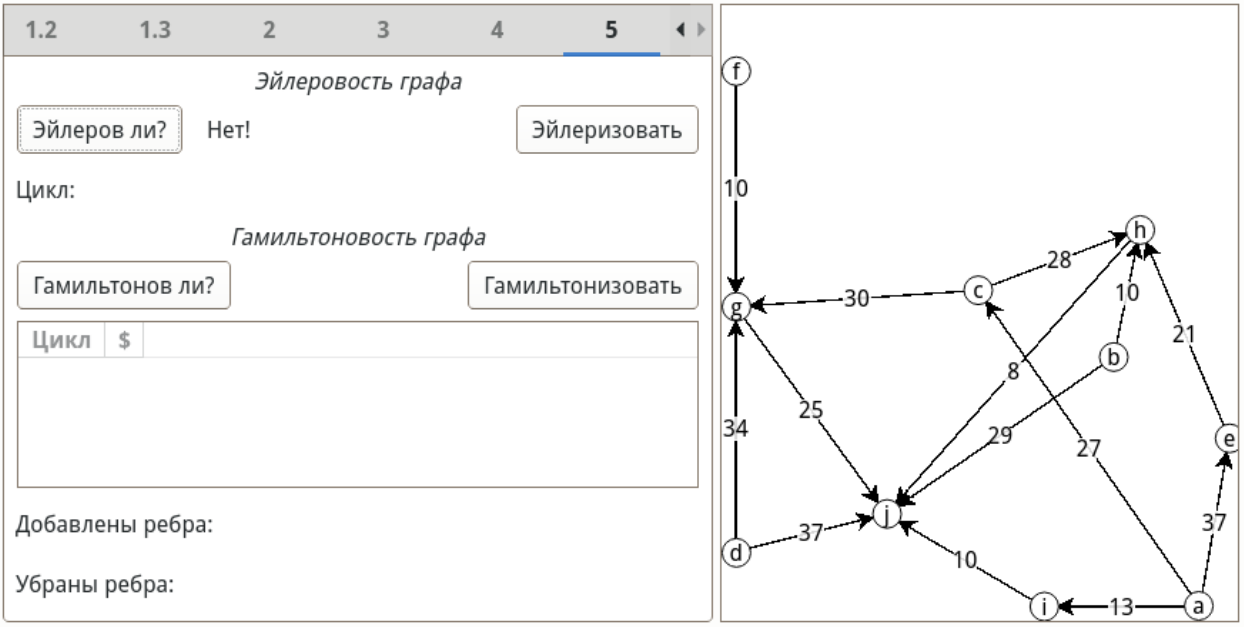

Рис. 19: Является ли эйлеровым

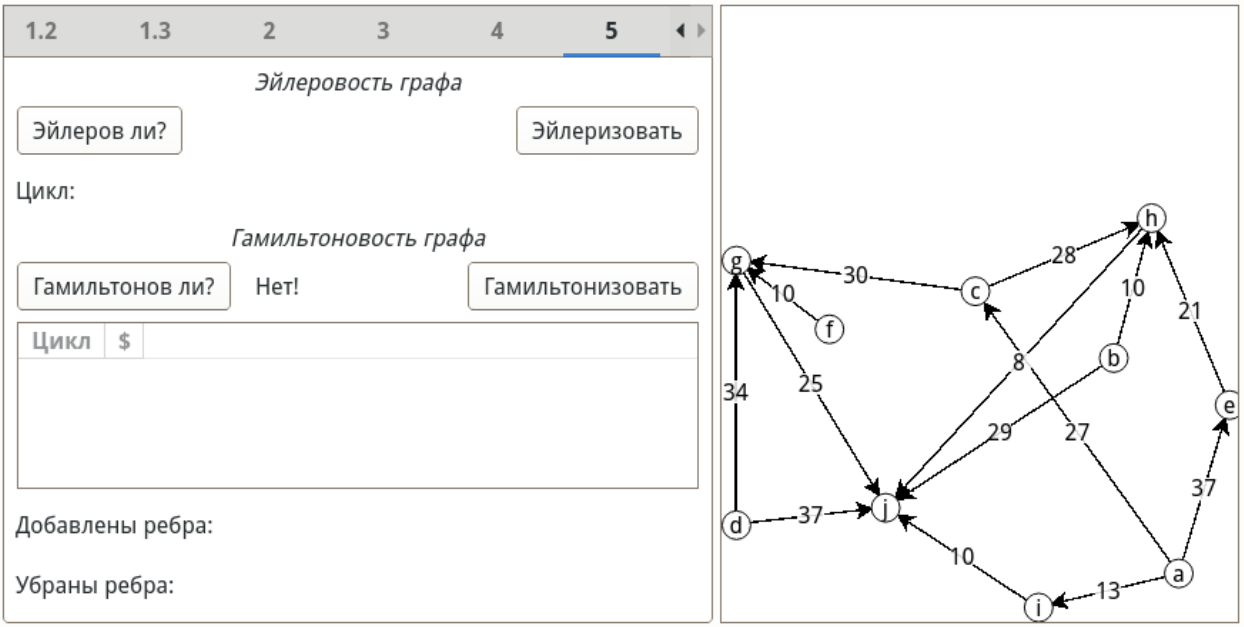

Рис. 20: Является ли гамильтоновым

На рис. 21 представлен результат модификации графа до эйлерова и найденный эйлеров цикл.

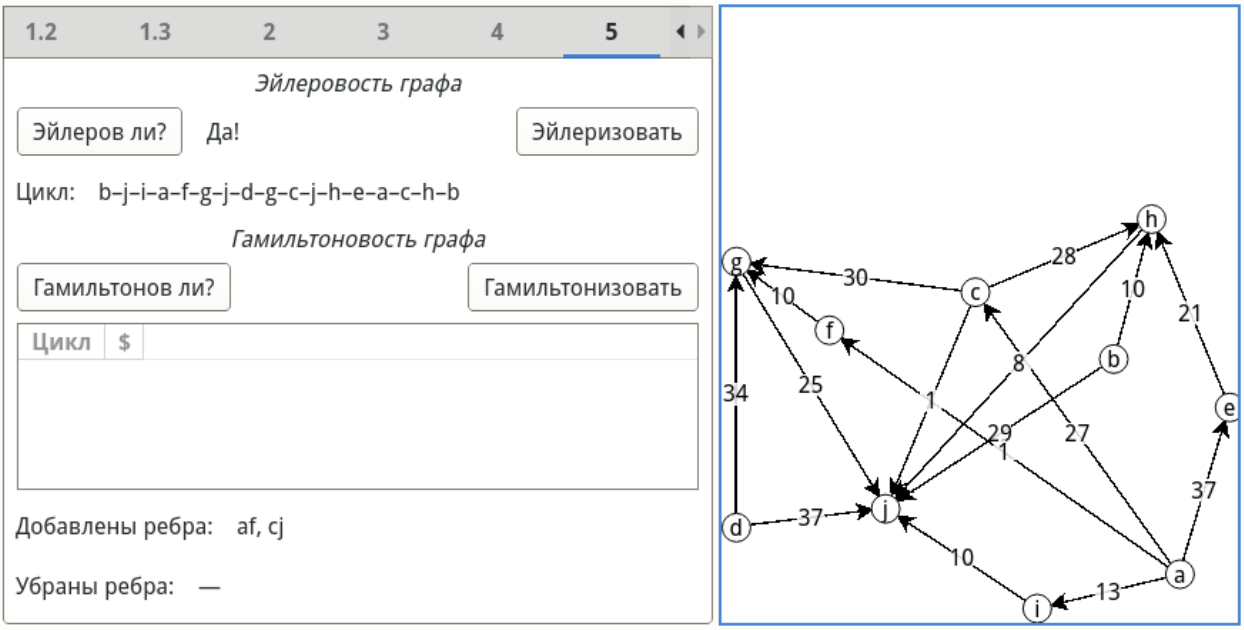

Рис. 21: Модификация графа до эйлерова

На рис. 22 представлен результат модификации графа до гамильтонова и найденные гамильтоновы циклы.

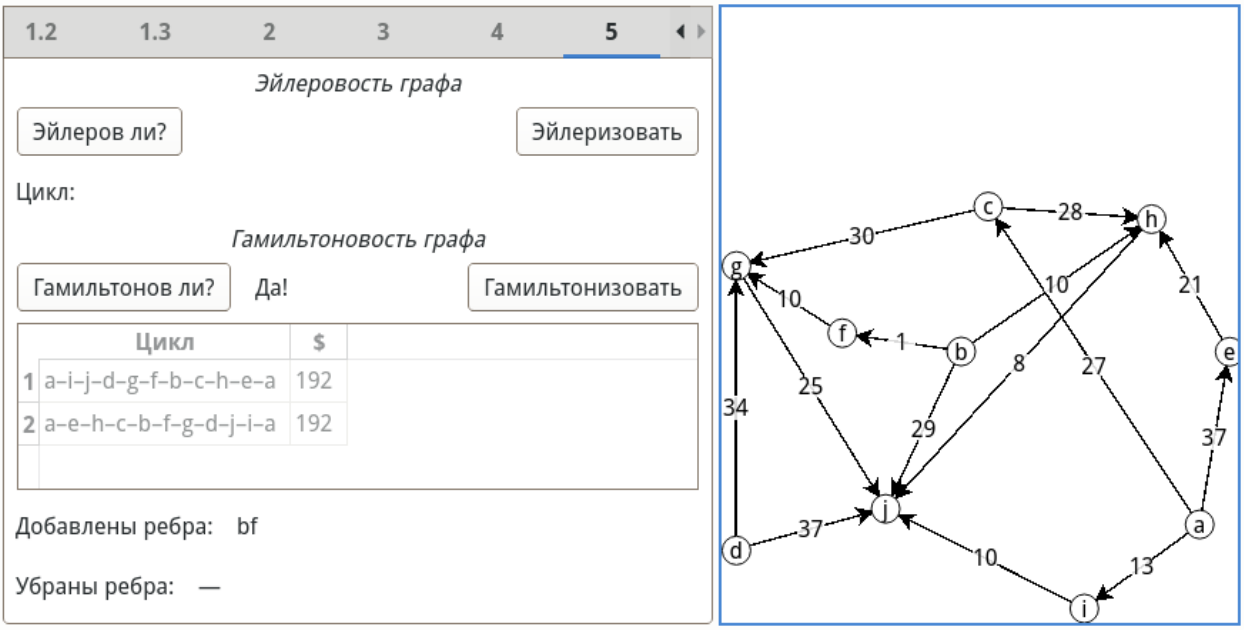

Рис. 22: Модификация графа до гамильтонова

Заключение

Результатом проделанной работы стала программа с графическим интерфейсом и статическая библиотека функций, выполняющие 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; поиск максимального потока с помощью алгоритма Форда-Фалкерсона; поиск потока минимальной стоимости; построение минимального по весу остова с помощью алгоритмов Прима и Краскала; поиск числа остовных деревьев, основываясь на матричной теореме Кирхгофа; (де-)кодирование деревьев с помощью кода Прюфера; построение Эйлеровых и Гамильтоновых циклов, а также решение задачи коммивояжера.

В статической библиотеке 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 $\mathcal{O}\left((p+q)\log p\right)$), но не применяется при наличии отрицательных весов дуг.
  - Алгоритм *Беллмана-Форда* медленнее, чем алгоритм Дейкстры ($\mathcal{O}(p \cdot q)$), так как дуг в графе обычно больше, чем вершин. Преимущество данного алгоритма заключается в том, что он может работать с отрицательными дугами, при условии, что в графе отсутствуют циклы отрицательного веса.
  - Алгоритм *Флойда-Уоршелла* имеет самую высокую временную сложность среди всех перечисленных ($\mathcal{O}(p^3)$). Преимуществом данного алгоритма является то, что он может работать с ребрами как положительного, так и отрицательного веса. кроме того, этот алгоритм находит кратчайшие пути между всеми парами вершин.
- Построение минимального по весу остова (алгоритмы Прима и Краскала). Стоит отметить, что в результате работы этих алгоритмов могут получиться разные остовы, но их суммарный вес будет одинаков (и он будет минимальным).
  - Алгоритм *Прима*. Недостаток: необходимость просматривать все оставшиеся ребра для поиска минимального. ($\mathcal{O}(p^2)$) Сложнее в реализации, чем алгоритм Краскала.
  - Алгоритм *Краскала*. Недостаток: необходимость в реализации хранения и сортировки списка ребер. ($\mathcal{O}(q \log p)$).

Достоинства программы:

- в программе реализованы методы обхода в глубину и ширину и методы нахождения кратчайших путей, которые могут использоваться в других алгоритмах;
- программа использует эффективные версии алгоритмов Дейкстры (бинарная куча вместо массива), Краскала (система непересекающихся множеств вместо списков);
- кодирование и декодирование Прюфера осуществляется вместе с весами ребер;
- представлены оригинальные методы приведения графа к эйлерову и гамильтонову;
- генерация гамильтоновых циклов выражена таким образом, чтобы позволить использовать ее вместе с конструкцией range-based loop языка C++;
- в решении задачи коммивояжёра полный перебор сокращен: рассматриваются лишь пути, начинающиеся с фиксированной вершины; Таким образом, в полном графе рассматриваются не $p!$ перестановок, а $(p-1)!$;
- программа обладает графическим интерфейсом, в то же время функции статической

библиотеки доступны в отрыве от фреймворка Qt. Такое свойство достигается благодаря решению разделить программу на две отдельные части: статическая библиотека и зависящий от нее модуль представления (графическое Qt-приложение).

Недостатки программы:

- в текущей реализации веса ребер должны быть ненулевые. Так, например, не получится добавить ребро нулевого веса.
- задача коммивояжера решена полным перебором;

Количество вершин графа ограничен сверху $p \leqslant 26$ в графическом интерфейсе, ввиду отображения вершин буквами латиницы. В статической библиотеке количество вершин ограничено лишь, с одной стороны, значением 2^{64} (используется целое 64-битное беззнаковое представление номера вершины), с другой размером оперативной памяти, т.к. необходимо хранить матрицу смежности соответствующего размера. Задача коммивояжера решается за разумное время только для $p \leqslant 11$. (При $p = 11$ файл, в который выводятся циклы достигает 177 MB).

Масштабирование программы:

- задачу коммивояжера можно решить более эффективными методами (метод ветвей и границ, муравьиный алгоритм, метод имитации отжига) Так же можно было сократить перебор вдвое, не рассматривая пути, различающиеся лишь направлением обхода: $abcda = adcba$;
- можно дополнить реализацию другими алгоритмами решения поставленных задач.

Список использованных источников

1. Алексеев, В.Е. Графы и алгоритмы. Структуры данных. Модели вычислений. – М.: БИНОМ. Лаборатория знаний, 2012. – С. 320.
2. Вадзинский, Р.Н. Справочник по вероятностным распределениям. – СПб.: Наука, 2001. – С. 295.
3. Кормен, Т.Х. Алгоритмы: построение и анализ. – М.: ООО «И. Д. Вильямс», 2013. – С. 1328.
4. Новиков, Ф.А. Дискретная математика для программистов. – СПб.: Питер, 2008. – С. 384.
5. Шапорев, С.Д. Дискретная математика. Курс лекций и практических занятий. – СПб.: БХВ-Петербург, 2006. – С. 400.