САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ им. ПЕТРА ВЕЛИКОГО

Институт Компьютерных Наук и Кибербезопасности

Высшая школа технологий искусственного интеллекта

Направление 02.03.01 Математика и компьютерные науки

Отчет по лабораторным работам №1-5 по дисциплине «Теория графов»

Алгоритмы на графах

Группа - 5130201/20002

Студент: _____ Салимли Айзек

Преподаватель: _____ Востров Алексей Владимирович

«___» _____20__г.

Санкт-Петербург — 2024г.

Содержание:

Введение

В отчете содержится описание лабораторной работы по дисциплине «Теория графов». Лабораторная работа включает реализацию разных алгоритмов таких как - алгоритм Шимбелла, алгоритм нахождения путей, Поиск в ширину, Поиск в глубину, алгоритм Беллмана-Форда, Флойда-Уоршелла, алгоритм Дейкстры, алгоритм Прима и Краскала с последующем кодированием и декодированием методом Прюфера, алгоритм, Форда-Фалкерсона и Эдмондса-Карпа, нахождения потока минимальной стоимости, алгоритм для нахождения минимального гамильтоново цикла, алгоритм нахождения эйлерова цикла, муравьиный алгоритм методом MMAS, алгоритм A*, алгоритм Магу с последующей раскраской графа , применяемых к связному ациклическому графу.
Граф сформирован при помощи двойного экспоненциального распределения Лапласа.
Так же присутствует реализация двух видов матриц представления графа - положительная и смешанная (с отрицательными весами ребер графа).

Работа была выполнена в интегрированной среде разработки Xcode, на языке программирования C++.

1. Постановка задачи

Основная задача - построить случайный связный ациклический граф в соответствии с заданным распределнием (Лапласа). На построенном графе требуется реализовать следующие алгоритмы:

1. Поиск экстремальных путей, методом Шимбелла.
2. Определение возможности построения маршрута двумя вершинами (нахождение количества таковых маршрутов).
3. Нахождение кратчайшего расстояния алгоритмам Дейкстры.
4. Нахождение минимального остовного графа с помощью алгоритма Прима и Краскала. С последующей кодировкой и декодом методом Прюфера.
5. Нахождение максимального потока алгоритмом Форда-Фалкерсона с поиском потока минимальной стоимости.
6. Нахождение минимального Гамильтонова цикла.
7. Нахождение минимального Эйлерова цикла.

По собственному желанию выполнены:

1. Нахождение минимального Гамильтонова цикла с эвристикой ACO методом MMAS.
2. Нахождение кратчайшего расстояния алгоритмом A* с эвристикой «Манхэттенское расстояние».
3. Нахождение МНП, и раскраска графа алгоритмом Магу.
4. Нахождение кратчайшего расстояния алгоритмом Беллмана-Форда
5. Нахождение кратчайшего расстояния алгоритмом Флойда-Уоршалла
6. Нахождение кратчайшего расстояния алгоритмом BFS

2. Математическое описание

2.1 Определение графа

Граф G(V,E) - совокупность двух множеств - непустого множества V (множества вершин) и множества E неупорядоченных пар различных элементов мно-жества V (E - множество ребер).

$$G(V,E) =< V,E >, V \neq \varnothing, E \subset V \times V, E = E^{-1}$$

p - кол-во вершин, а q - число ребер

Связный граф - граф, содержащий ровно одну компоненту связности. Это означает, что между любой парой вершин этого графа существует как минимум один путь.

Ацикличный граф - граф который не содержит циклов (конечная вершина не входит в исходную)

2.2 Двойное экспоненциальное распределение Лапласа

Распределение вероятностей случайной величины X , заданное плотностью вероятности:

$$p(x,\alpha,\beta) = \frac{1}{2}\alpha e^{-\alpha|x-\beta|}, \quad -\infty < t < +\infty,$$

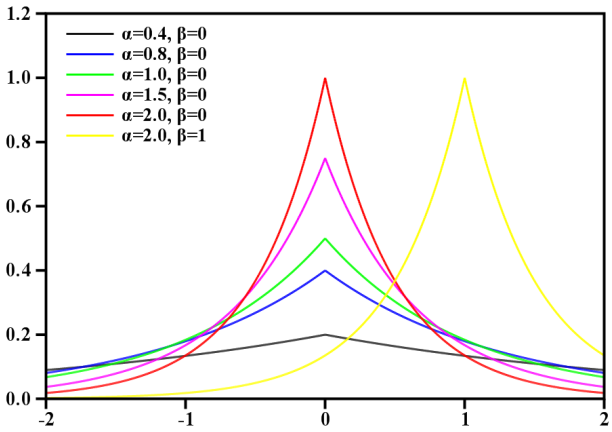
α и β, $\alpha > 0$, $-\infty < \beta < \infty$, – параметры

Распределение Лапласа совпадает с распределением случайной величины,

$$\beta + X_1 - X_2, \text{ где } X_1 \text{ и } X_2$$

где X1 и X2 – независимые случайные величины, имеющие одинаковое показательное распределение с плотностью, равной 0 при x<=0 и равной

$$\alpha e^{-\alpha x} \text{ при } x > 0.$$



В нашей задачи распределение вероятности Лапласа применяется в коде программы формулой.

```
int LaplaceD() {
    random_device rd;
    mt19937 mersenne(rd());

    const double a = 10.0; // Параметр a для распределения Лапласа
    const double b = 6.0; // Параметр b для распределения Лапласа

    std::uniform_real_distribution<double> dist(0, 1);
    double u = dist(mersenne); // Генерируем u из равномерного распределения на [0, 1]
    //Если u равно 0, тогда 1 - u равно 1, и log(2 * (1 - u)) равно 0. Таким образом, x будет равно b, и это соответствует максимальной вероятности для значения x в      распределении Лапласа.
    //Если u равно 1, тогда 1 - u равно 0, и log(2 * (1 - u)) стремится к минус бесконечности. Таким образом, x будет стремиться к бесконечности отрицательной стороны,   что также соответствует распределению Лапласа.
    //Целая часть x: После того, как мы вычислили значение x, мы возвращаем его целую часть, так как в исходной функции указано, что результат должен быть целым числом.
    // то есть u определяет, какое конкретное значение из Лапласа будет выбрано для возврата
    // Используем обратное преобразование для генерации случайного числа
    double x;
    if (u < 0.5)
        x = b + a * log(2 * u);
    else
        x = b - a * log(2 * (1 - u));

    return static_cast<int>(x); // Возвращаем целую часть
}
```

2.3 Метод Шимбелла

Метод используется для определения минимальных или максимальных расстояний между парами вершин и учитывает число ребер, входящих в соответствующие простые цепи.

В методе заданы свойства операции умножения и сложения, а так же правило сложения:

Свойства сложения -

$$a * b = b * a \Rightarrow a + b = b + a \text{ И } a * 0 = 0 * a = 0 \Rightarrow a + 0 = 0 + a = 0$$

Сама операция -

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

С помощью этих операций длины кратчайших (или максимальных) путей определяются возведением в степень x весовой матрицы Ω , где x – количество рёбер в пути. Элемент $\Omega^{x}_{[i][j]}$ покажет длину экстремального пути длиной в x рёбер из вершины i в вершину j.

2.4 Алгоритм Дейкстры

Пусть дан ориентированный или неориентированный граф G = (V , E) с набором вершин V и набором рёбер E. Каждое ребро (u , v) имеет неотрицательный вес w(u,v), который обозначает длину пути от вершины u до вершины v. Пусть также дана начальная вершина s, от которой ищется кратчайший путь до всех остальных вершин графа.
Инициализация: Создаём множество S, в котором будут храниться вершины, для которых уже найден кратчайший путь от вершины s. Инициализируем множество расстояний d для каждой вершины графа. Начально d(s)=0, а для всех остальных вершин $v \in V$, $d(v) = \infty$.
Шаг алгоритма: Находим вершину u из множества вершин $V \setminus S$ с минимальным значением d(u). Добавляем вершину u в множество S. Для каждой вершины v из множества $V \setminus S$,

смежной с вершиной u : Если $d(u) + w(u, v) < d(v)$, обновляем $d(v) = d(u) + w(u, v)$, то есть уменьшаем длину пути от s до v.
Повторяем шаг 2, пока все вершины не будут добавлены в множество S. Вывод результата: После завершения алгоритма, для каждой вершины v в множестве V, d(v) содержит длину кратчайшего пути от начальной вершины s до вершины v.

2.5 Алгоритм DFS

Один из методов обхода графа. Стратегия поиска в глубину, как и следует из названия, состоит в том, чтобы идти «вглубь» графа, насколько это возможно. Алгоритм поиска описывается рекурсивно: перебираем все исходящие из рассматриваемой вершины рёбра. Если ребро ведёт в вершину, которая не была рассмотрена ранее, то запускаем алгоритм от этой нерассмотренной вершины, а после возвращаемся и продолжаем перебирать рёбра. Возврат происходит в том случае, если в рассматриваемой вершине не осталось рёбер, которые ведут в нерассмотренную вершину. Если после завершения алгоритма не все вершины были рассмотрены, то необходимо запустить алгоритм от одной из нерассмотренных вершин.

1.Выбираем любую вершину из еще не пройденных, обозначим ее как u.
2.Запускаем процедуру dfs(u) Помечаем вершину u как пройденную.
3.Для каждой не пройденной смежной с u вершиной (назовем ее v) запускаем dfs(v).
4.Повторяем шаги 1 и 2, пока все вершины не окажутся пройденными.

Сложность алгоритма: O(V)

2.6 Алгоритм BFS

Алгоритм основан на обходе вершин графа "по слоям". На каждом шаге есть множество "передовых" вершин, для смежных к которым производится проверка, относятся ли они к еще не посещенным. Все еще не посещенные вершины добавляются в новое множество "передовых" вершин, обрабатываемых на следующем шаге. Изначально в множество "передовых" вершин входит только вершина-источник, от которой и начинается обход.

G=(V,E) с выделенной вершиной-источником u. Путем $P(u,v)$ между вершинами u и v называется множество ребер $(u,v1),(v1,v2),...(vn-1,v)$. Длиной пути $d(u,v)$ обозначим число ребер в данном пути между вершинами u и v. Поиск в ширину находит кратчайшие пути $d(u,v)$ от вершины u до всех остальных вершин графа описанным далее образом. В начале работы алгоритма расстояние до вершины-источника $d(u)$=0, до остальных вершин $d(v)=\infty, \forall v \neq u$. Также в начале работы алгоритма инициализируется множество $F=\{u\}$. Далее на каждом шаге алгоритма строится множество вершин $P=w$, таких, что для $\forall v \in F \exists (v,w) \in E$| $d(w)=\infty$, при этом обновляются расстояния $d(w)=d(v)+1$ для $\forall w \in P$. Затем производится переход на следующий шаг до тех пор, пока $P \neq \varnothing$; при этом в начале каждого шага множество F заменяется на P.

Сложность алгоритма: O(|V|+|E|), где V - количество вершин, а E - количество ребер.

2.7 Алгоритм Беллмана-Форда

Пусть дан ориентированный граф G = (V , E) с набором вершин V и набором рёбер E. Каждое ребро (u,v) имеет вес w (u , v), который обозначает длину пути от вершины u до вершины v. Пусть также дана начальная вершина s, от которой ищется кратчайший путь до всех остальных вершин графа.

Инициализация: Инициализируем массив расстояний d для каждой вершины графа. Начально d (s) = 0, а для всех остальных вершин v∈V, d(v)=∞.

Процесс обновления расстояний: Повторяем следующие действия | V | − 1 раз, где |V| — количество вершин в графе:

Проходим по всем рёбрам графа (u , v) и для каждого ребра: Если d (u) + w (u , v) < d (v), обновляем +w(u,v), то есть уменьшаем длину пути от s до v.

Проверка наличия отрицательных циклов: Повторяем проход по всем рёбрам графа и проверяем, если имеется ребро, для которого выполнено d (u) + w (u , v) < d (v), то граф содержит отрицательный цикл.

Вывод результата: После завершения алгоритма, для каждой вершины v массив d содержит длину кратчайшего пути от начальной вершины s s до вершины v, или значение $-\infty$, если путь от s до v не существует из-за наличия отрицательного цикла.

Сложность алгоритма: O(u*v), где v — кол-во вершин, а u — кол-во ребер.

2.8 Алгоритм Флойда-Уоршалла

Алгоритм Флойда-Уоршелла находит кратчайшие пути между всеми парами вершин (узлов) в 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 впоследствии понадобятся для решения исходной задачи, просчитываются заранее и затем используются.

Идея алгоритма — разбиение процесса поиска кратчайших путей на фазы. Перед k-ой фазой величина $d_{[i][j]}$ равна длине кратчайшего пути из вершины i в вершину j, если этому пути разрешается заходить только в вершины с номерами, меньшими k. На k-ой фазе мы попытаемся улучшить путь $i \to j$, пройдя через вершину k: $d_{[i][j]} = \min(d_{[i][j]}, d_{[i][k]} + d_{[k][j]})$. Матрица d и является искомой матрицей расстояний.

Сложность алгоритма: $O(n^3)$.

2.9 Минимальный остовный граф

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

Минимальное остовное дерево - 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.

2.10 Алгоритм Прима и Алгоритм Каркала + код Прюфера

Алгоритм Прима также используется для поиска минимального остовного дерева в связном взвешенном графе. Он начинает с одной из вершин и постепенно добавляет к остовному дереву новые рёбра с наименьшими весами, при этом убеждаясь, что новое ребро присоединяет вершину к уже существующему остовному дереву. Вот математическое описание алгоритма:
Пусть G=(V,E) — связный граф с вершинами V и рёбрами E, и w:E→R — функция весов на рёбрах.
Тогда минимальное остовное дерево T для G можно найти следующим образом с использованием алгоритма Прима: Инициализируем T как пустой граф и множество S как пустое множество вершин. Выбираем начальную вершину v из V и добавляем её в S. Пока S не содержит все вершины: Найдем ребро (u , v), где u∈S, а v ∉ S такое, что вес этого ребра минимален. Добавляем вершину v к S и ребро (u , v) к T. Возвращаем T.
Алгоритм обеспечивает, что в конце мы получим остовное дерево с минимальной суммой весов рёбер.

Сложность алгоритма: O(V)+O(V*log(V))+O(E*log(V))

Краскалов алгоритм это алгоритм на графах для поиска минимального остовного дерева. Он выбирает рёбра графа по возрастанию их весов, при этом так, чтобы добавление каждого нового ребра не создавало циклов в остовном дереве. Давайте подробнее: Математическое описание:
Пусть G = (V , E) G=(V,E) — неориентированный связный граф с вершинами V и рёбрами E, и w:E→R — функция весов на рёбрах.
Тогда минимальное остовное дерево T для G можно найти следующим образом:
Инициализируем T как пустой граф. Сортируем все рёбра графа E по возрастанию весов.

Поочередно добавляем к T рёбра из E с наименьшими весами, при условии, что добавление этого ребра не создаст цикл в T.

Сложность алгоритма: O(E*log(E)) - где E ребра

Код Прюфера - представление помеченного дерева с n вершинами в виде последовательности из $n-2$ чисел. Код Прюфера представляет собой способ кодирования структуры дерева с помощью последовательности, которая легко восстанавливается обратно в дерево. Пусть T=(V,E) — помеченное дерево с n вершинами, где V={1,2,...,n}. Тогда код Прюфера представляет собой последовательность из n−2 чисел, каждое из которых принимает значения от 1 до n. Математическое описание алгоритма формирования кода Прюфера:
Инициализация: Начинаем с произвольного помеченного дерева T с n вершинами.
Получение кода Прюфера: На каждом шаге выбираем самую листовую вершину v с наименьшим номером. Добавляем в код Прюфера номер соседа u, соединённого с v ребром. Удаляем вершину v из дерева и соответствующее ребро.
Критерий останова: Продолжаем процесс, пока в дереве не останется две вершины.
Вывод результата: Последовательность чисел, полученная на шаге 2, является кодом Прюфера для заданного помеченного дерева T.
Математически, код Прюфера для дерева с n вершинами — это последовательность $n-2$ чисел $p_1, p_2, \ldots, p_{n-2}$, где каждое p_i принимает значения от 1 до n, представляющие номера вершин дерева. А для восстановления дерева из кода Прюфера используется обратный процесс, который выполняется за линейное время от размера кода Прюфера.

2.11 Максимальный поток в сети

Сеть - направленный слабосвязный орграф с одним истоком и одним стоком. Пусть G(V,E) - сеть, s и t - соответственно, источник и сток сети. Дуги сети нагружены неотрицательными вещественными числами c: $E \to R^+$. Если u и v - узлы сети, то число c(u,v) - называется пропускной способностью ребра (u,v)
Дивергенцией функции f в узле v называется число div(f, v), которое определяется следующим образом:

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

Функция f : E → R называется потоком в сети G, если:

1) Для любых (u, v) ∈ E(0 ≤ f (u, v) ≤ c(u, v)), то есть поток через дугу неотрицателен и не превосходит пропускной способности дуги.
2) Для любого u ∈ V s, t(div(f, u) = 0), то есть дивергенция потока равна нулю во всех узлах, кроме источника и стока.

2.12 Алгоритм Форда-Фалкерсона

Теорема Форда - Фалкерсона :

Максимальный поток в сети равен минимальной пропускной способности разреза, то есть существует поток **f*** такой, что:

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

Таким образом реализуется алгоритм Форда - Фалкерсона для определения максимального потока в сети, заданной матрицей пропускных способностей дуг. Алгоритм решает проблему нахождения максимального потока в транспортной сети. Если использовать алгоритм Беллмана-Форда и задать величину искомого потока и матрицу стоимостей, то получится алгоритм нахождения потока минимальной стоимости. Для ребра [i][j] она определяет стоимость пересылки единичного потока из вершины с номером i в вершину с номером j.

Сложность алгоритма: O(q*f), где q - ребра, а f - величина максимального потока.

2.13 Поиск потока минимальной стоимости

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

Сложность алгоритма: $O(q^3)$.

2.14 Гамильтоновы и Эйлеровые циклы

Гамильтонов цикл в неориентированном графе — это такой цикл, который проходит через каждую вершину графа ровно один раз и возвращается в начальную вершину. Давайте опишем его формально: Пусть $G = (V , E)$ — неориентированный граф с n вершинами ($|V|=n$) и m m рёбрами ($| E | = m$). Тогда Гамильтонов цикл в G — это такая последовательность вершин (v1 ,v2 ,…,v n ,v1), что:

Каждая вершина $v i$ встречается в последовательности ровно один раз. Для каждого i от 1 до $n-1$, в графе существует ребро $(v i , v i + 1)$. Существует ребро между последней вершиной $v n$ $v n$ и начальной вершиной $v 1$, то есть $(v n , v 1)$.

Гамильтонов цикл проходит через каждую вершину графа ровно один раз и возвращается в начальную вершину, образуя замкнутый цикл. Если такой цикл существует в графе, говорят, что граф содержит Гамильтонов цикл. Если Гамильтонов цикл проходит через каждое ребро графа ровно один раз, то такой граф называется гамильтоновым.

Эйлеров цикл - это такой цикл, который проходит через каждое ребро графа ровно один раз. Пусть $G = (V , E)$ — неориентированный граф с n вершинами ($| V | = n$) и m рёбрами ($|E|=m$). Тогда Эйлеров цикл в G — это такая последовательность рёбер $(e 1 , e 2 , . . . , e m)$, что:

Каждое ребро $e i$ встречается в последовательности ровно один раз. Конец каждого ребра $e i$ совпадает с началом следующего ребра $e i + 1$ в последовательности, за исключением, возможно, начального ребра и конечного ребра. Начальное ребро $e 1$ совпадает с конечным ребром $e m$ в последовательности.

Эйлеров цикл проходит через каждое ребро графа ровно один раз, образуя замкнутый цикл, кроме возможно начального и конечного ребра, которые могут совпадать или различаться в зависимости от того, начинается ли цикл и заканчивается ли он в одной и той же вершине.

Если такой цикл существует в графе, говорят, что граф содержит Эйлеров цикл.

Если каждая вершина графа принадлежит Эйлерову циклу, то граф называется Эйлеровым.

2.15 Эвристическая функция - Муравьиный алгоритм MMAS

Муравьиный алгоритм (Ant Colony Optimization, ACO) — это метаэвристический алгоритм оптимизации, вдохновленный поведением муравьёв при поиске пути к источнику питания.
Алгоритм с минимальными априорными знаниями (MMAS) для нахождения гамильтонова цикла в графе:
Пусть G = (V , E) — граф с n вершинами (|V|=n) и m рёбрами (|E|=m). Будем рассматривать задачу поиска гамильтонова цикла в графе.
Инициализация: Создадим муравьев и разместим их в случайных вершинах графа. Каждый муравей начинает построение пути из своей начальной вершины. Выбор следующей вершины: На каждом шаге каждый муравей выбирает следующую вершину для посещения, основываясь на значениях феромона на рёбрах и эвристических информациях о доступных вершинах. Вероятность выбора вершины j из вершины i определяется по формуле:

$$P_{ij} = \frac{\tau_{ij}^{\alpha}\,\eta_{ij}^{\beta}}{\sum\limits_{m \in \text{доступные вершины}} \tau_{im}^{\alpha}\,\eta_{im}^{\beta}}$$

Где, pij — вероятность перехода из вершины i в вершину j.
τ ij — количество феромона на ребре между вершинами i и j.
η ij — эвристическая информация, обычно обратная длине ребра между вершинами i и j.
α и β — параметры, регулирующие влияние феромона и эвристики на выбор муравьём следующей вершины.
J — множество вершин, доступных для выбора.
Построение пути: Каждый муравей поочерёдно выбирает следующие вершины на основе вероятностей, определённых выше, и добавляет их к своему пути, при условии, что вершина ещё не была посещена.
Обновление феромона: После того, как все муравьи завершили построение пути, феромон на каждом ребре обновляется на основе длины пути, найденного муравьём. Феромон на каждом ребре испаряется с течением времени. Обновление феромона происходит по формуле:

$$\Delta\tau_{ij,k(t)} = \begin{cases} \dfrac{Q}{L_{k(t)}}, & \text{если } (i,j) \in T_{k(t)}\,, \\ 0, & \text{если } (i,j) \notin T_{k(t)}\,, \end{cases}$$

$$\tau_{ij\,(t+1)} = (1-p)\cdot\tau_{ij(t)} + \sum_{k=1}^{m} \Delta\tau_{ij,k(t)}\,,$$

Где, ρ — коэффициент испарения феромона.

$\Delta \tau i j (k)$ — изменение феромона на ребре между вершинами i и j, основанное на длине пути k-ого муравья.

Критерий останова: Повторяем шаги 2-4, пока не будет выполнен критерий останова (достижение максимального числа итераций или улучшение текущего лучшего результата).

2.16 Алгоритм Магу и Раскраска графа

Пусть $G = (V , E)$ — граф с n вершинами ($|V|=n$) и m рёбрами ($E|=m$).

Задача состоит в поиске ядра графа, то есть максимального по размеру подмножества вершин $K \subseteq V$, которое образует индуцированный подграф, в котором каждая вершина имеет степень не менее k, где k — пороговое значение степени.

1) Инициализация: Начинаем с пустого множества вершин K.

2) Поиск ядра графа: Перебираем все вершины графа V и добавляем их в множество K, если степень вершины не менее заданного порогового значения k.

Таким образом, для каждой вершины $v i \in V$, мы проверяем, что степень вершины vi не меньше k.

3) Критерий останова: Продолжаем добавлять вершины в K до тех пор, пока не будет найдено максимальное по размеру ядро графа.

4) Вывод результата: После завершения алгоритма множество K будет содержать максимальное по размеру ядро графа а также нахождения максимально независимых подмножеств вершин графа.

В нашем случае, алгоритм Магу находит максимально независимые пожножества графа, после чего дает им определенный цвет.

2.17 Алгоритм A-star и эвристическая функция - «Манхэттенское расстояние»

Алгоритм поиска кратчайшего пути в графе от начальной вершины к целевой вершине. Он использует эвристическую функцию оценки, чтобы выбирать следующий узел для раскрытия, что позволяет ему делать более информированные выборы и избегать раскрытия бесперспективных путей. Манхэттенское расстояние может использоваться как эвристическая функция для оценки расстояния между текущим узлом и целевым узлом в матрице.

Пусть: G = (V , E) — граф с вершинами V и рёбрами E. s — начальная вершина. t — целевая вершина. h(v) — манхэттенское расстояние между вершиной v и целевой вершиной t.
Инициализация:
Инициализируем два списка: Открытый список O, содержащий начальную вершину s. Закрытый список C, изначально пустой.
Цикл: Пока открытый список не пуст: Выбираем вершину v из открытого списка O с минимальной оценкой f (v) = g (v) + h (v), где: g(v) — длина пути от начальной вершины s до вершины v. h (v)— манхэттенское расстояние между вершиной v и целевой вершиной t. Если v=t, то путь найден, завершаем алгоритм. Перемещаем вершину v из открытого списка O в закрытый список C. Для каждой соседней вершины u вершины v, не находящейся в списке C: Если вершина u не в открытом списке O, добавляем её туда.
Если вершина u уже находится в открытом списке O, обновляем информацию о ней, если новый путь короче, чем ранее найденный.
Вывод пути: Восстанавливаем путь от t до s из информации о родителях для каждой вершины.
=> алгоритм A* с использованием манхэттенского расстояния работает, минимизируя сумму длины найденного пути и оценки оставшегося расстояния до цели, основанной на манхэттенском расстоянии. => Это позволяет ему находить кратчайший путь к цели, используя более быстрый подход, чем простой поиск в ширину или алгоритм Дейкстры.

3. Реализации графа и алгоритмов и вспомогательные методы

3.1 Класс MyGraph

Класс MyGraph является основной структурой данных в этой работе. Класс содержит всю необходимую информацию о графе, методы позволяющие создавать и пересоздавать граф, методы реализующие алгоритмы.

```
#pragma once
#include "globals.h"
#include <set>
using namespace std;

enum class ShimbellMode {
    Short = 0,
    Long = 1
};

struct Node {
    int vertex;
    int g_cost; // стоимость пути от начальной вершины до текущей
    int h_cost; // эвристическая оценка стоимости от текущей вершины до конечной
    Node(int v, int g, int h) : vertex(v), g_cost(g), h_cost(h) {}
};

struct CompareNodes {
```

```
    bool operator()(const Node& n1, const Node& n2) {
        // сравниваем суммарную стоимость f = g + h
        return n1.g_cost + n1.h_cost > n2.g_cost + n2.h_cost;
    }
};

struct edge {
    int a, b, cost;
    edge(int a2, int b3, int cost4) {
        a = a2;
        b = b3;
        cost = cost4;
    }
};

class MyGraph {

private:
    int vertexCnt; // переменная, которая хранит количество вершин в графе.
    vector<vector<int> > MatrixSmejn; // двумерный вектор, который представляет матрицу смежности графа.
    vector<vector<int> > posWeightsMatrix; // двумерный вектор, который представляет матрицу весов ребер графа.
    vector<vector<int> > mixedWeightsMatrix; // двумерный вектор, который представляет смешанную матрицу весов ребер графа.
    vector<vector<int> > modPosWeightsMx; // двумерный вектор, который представляет модифицированную матрицу весов ребер графа.
    vector<pair<int, int>> previous; // A*
    vector<vector<double>> pheromoneMatrix;
    vector<vector<int> > modMixedWeightsMx; //двумерный вектор, который представляет модифицированную смкшанную матрицу весов ребер
графа.
    vector<vector<int> > MatrixReach;  // двумерный вектор, который представляет матрицу достижимости графа.
    vector<int> VertPrepare() const; //функция, которая подготавливает вершины графа для матрицы смежности.
    void AssignWeights(); // функция, которая присваивает веса ребрам графа.
    void MixWeights(); // смешанные веса
    void ModifyWeights(vector<vector<int> >& WeightsMatrix, vector<vector<int> >& modified); // функция, которая модифицирует веса
ребер в матрице.
    vector<vector<int>> MultByShimbell(const vector<vector<int> > fMx, // функция, которая умножает две матрицы с использованием
алгоритма Шимбелла
                                       const vector<vector<int> > sMx, ShimbellMode mode) const;
    vector<vector<int>> throughPutMx; // двумерный вектор, который представляет матрицу пропускной способности ребер графа.
    //vector<vector<int>> torrent;// матрица стоимости
    vector<vector<int>> torrentMatForFord;
    vector<vector<int>> MatrixFakeSource; //двумерный вектор, который представляет матрицу фиктивного источника графа. потом если не
нужен то станет копией смежности
    vector<vector<int> > UnOrMxSmejn; //двумерный вектор, который представляет матрицу смежности графа без отрицательных весов ребер.
    vector<vector<int> > UnOrPosWeightMx; // веса >0
    vector<vector<int> >minTree; // потом
    vector<pair<int, int> > PruferTree; // потом ???
public:

    MyGraph(int n); //конструктор класса MyGraph, который инициализирует переменные класса.
    vector<vector<int>> torrent; // матрица стоимости
    int GetVertexCount() const; //функция, которая возвращает количество вершин в графе.
    vector<vector<int> > GetAdjMatrix() const; //возвращает матрицу смежности графа.
    vector<vector<int> > GetWeightsMatrix(int type) const; //весов
    vector<vector<int> > GetReachMatrix() const; // достижимости
    vector<vector<int> > GetTorrentMatrix() const; // матрица стоимости
    vector<vector<int>> CalcShimbell(int edgeCnt, ShimbellMode mode) const;
    void MakeUndirected(); // сделать неориентированным
    void CreateUndirictedMatrix(const vector<vector<int>>& weightsMatrix);
    void DFSUtil(int v, int endVertex, vector<bool>& visited, vector<int>& visitedNodes, vector<int>& parent, int& totalDistance);
    void printPath(int startVertex, int endVertex, const vector<int>& parent, const vector<vector<int>>& weights) const;
    void printPathDFS(int startVertex, int endVertex, const vector<int>& parent, const vector<vector<int>>& weights) const;
    void DFS(int startVertex, int endVertex); // обход в глубину для графа. выводы и т.д.
    void DFS1(int currentVertex, int endVertex, vector<int>& path, vector<bool>& visited, vector<int>& shortestPath, int&
shortestWeight);
    int DFS2(int currentVertex, int endVertex, vector<int>& path, vector<bool>& visited, vector<int>& shortestPath, int&
shortestWeight);
    void findShortestPathDFS1(int startVertex, int endVertex);
    void BFS(int startVertex, int endVertex); //обход в ширину для графа, начиная с указанной вершины
    int BFS1(int startVertex,int endVertex);
    void ReachMatrixGenerator(); // матрица достижимости графа.
    void BuildFakeSource();
    vector<int> Dijkstra(int inpVert, int& counter) const; //алгоритм Дейкстры для нахождения кратчайшего пути в графе
    vector<vector<int> > RestorePaths(int inpVert, const vector<int>& distances, const vector<vector<int> > weightMx) const;//
восстанавливает пути в графе. потом
    vector<int> BellmanFord(int inpVert, int& counter, vector<vector<int> > mx) const; //алгоритм Беллмана-Форда для нахождения
кратчайшего пути в графе.
    vector<int> BellmanFord(int inpVert, vector<vector<int> > mx) const; //алгоритм Беллмана-Форда для нахождения кратчайшего пути в
графе но для весовой.
    vector<vector<int> > FloydWarshall(int& counter, vector<vector<int> > mx) const; //алгоритм Флойда-Уоршала с INF нахождения
кратчайшего пути в графе.
    vector<int> encodePruefer(const vector<vector<pair<int, int>>>& tree);
    vector<vector<pair<int, int>>> PrimMST(); // алгоритм Прима для минимального остовного графа
    bool isAchievable(int vertexOne, int vertexTwo, vector<vector<int>>& graph, int lastVertex) const;
    void Kraskal(vector<vector<int>> weightMx, int &iterationCounter);
    void Prim(vector<vector<int>> weightMx, int& iterationCounter);
    void PruferCode();
    void PruferDecode();
    void AssignTorrent();
    bool bfs_FordFulkerson(vector<vector<int>> residualG, int source, int sink, vector<int>&path) const;
    int fordFulkerson(int source,int sink) const;
    int calcMinCostFlow(int s, int t) const;
    void AssignTorrent1();
    void MakeUndirected1();
    bool bfs_FordFulkerson1(vector<vector<int>> matrix, int source, int sink, vector<int>& path) const;
    void addEdge(int u, int v, int capacity);
    void fordFulkerson1(int source, int sink) const;
    int minCut(vector<vector<int>>& torrent2); ////////////
    int countCrossingEdges(vector<vector<int>>& matrix, vector<bool>& cutSet);
    void mergeVertices(vector<vector<int>>& matrix,vector<bool>& cutSet, int u,int v);
    int DijkstraAS(int startVertex, int endVertex) const;
    int heuristic(int start, int end, const vector<vector<int>>& graph);
    vector<int> aStarSearch(const vector<vector<int>>& graph, int start, int end);
    void Hamilton(vector<vector<int>> weightMx, int weightMode) const;
    void findHamiltonCycle(ofstream& fout, vector<vector<int>>& graph, vector<int>& path, int length, vector<int>& minimumPath, int&
minimumLength) const;
```

```
    void AntColonyOptimization(vector<vector<int>> weightMx, int weightMode) const;
    void findHamiltonCycleACD(ofstream& fout, vector<vector<int>>& graph, vector<int>& path, int length, vector<int>& minimumPath,
int& minimumLength) const;
    void Euler(vector<vector<int>> weightMx, int weightMode) const;
    void EulerCycles(vector<vector<int>> weightMx, vector<int> degrees) const;
    vector<vector<int>> FindMaximalIndependentSets();
    void BronKerboschIterative(set<int>& r, set<int>& p, set<int>& x, vector<vector<int>>& maximalIndependentSets,
vector<vector<int>>& tempMatrix);
    void NewGraph();
};
```

3.1.1 Поля

- **int vertexCnt - количество вершин в графе.**
- **vector<vector<int>> MatrixSmejn - не взвешенная матрица смежности вер- шин графа.**
- **vector<vector<int>> posWeightsMatrix - взвешенная положительная матрица смежности.**
- **vector<vector<int>> mixedWeightsMatrix - взвешенная смешанная матрица смежности.**
- **vector<vector<int>> modPosWeightsMx - модифицированная взвешенная положительная матрица смежности. Используется в алгоритме Дейкстры.**
- **vector<vector<int>> modMixedWeightsMx - модифицированная взвешенная смешанная матрица смежности. Используется в алгоритме поиска потока минимальной стоимости.**
- **vector<vector<int>> MatrixReach - матрица достижимости.**
- **vector<vector<int>> torrent - матрица пропускной способности графа.**
- **vector<vector<int>> UnOrMxSmejn - матрица смежности для графа без ориентации.**

3.1.2 Методы

MyGraph(int n) -

Конструктор с параметром, принимает количество вершин. В конструкторе формируется матрица смежности, а так же весовые матрицы, матрица достижимости, матрица пропускных способностей, матрица смежности графа без ориентации.

```
MyGraph::MyGraph(int n) : vertexCnt(n), MatrixSmejn(n, vector<int>(n, 0)),
posWeightsMatrix(n, vector<int>(n, 0)), MatrixReach(),torrent(n , vector<int>(n, 0))//,Инициализирует переменные vertexCnt, MatrixSmejn, posWeightsMatrix и
MatrixReach.

{
  int tmp;
```

```
random_device rd;
mt19937 mersenne(rd()); // случайные числа
vector<int> vertDegrees = VertPrepare(); // вектор степеней графа вызов функции VertPrepare для инициализации вектора vertDegrees
Shuffler<int> shuffler; //инициализация объекта shuffler типа Shuffler<int>.
for (int i = 0; i < vertexCnt - 2; i++) { // цикл, который проходит по всем вершинам, кроме двух последних.
    shuffler.Clear(); //очистка объекта shuffler. для дальнейшего
    for (int j = i + 1; j < vertexCnt; j++) { //цикл, который проходит по всем вершинам, начиная с вершины i + 1
        shuffler.Push(j);//добавление вершины j в объект shuffler
    }
    shuffler.Shuffle(); //перемешивание вершин в объекте shuffler
    // поочерёдно в стоки ставим единички в случайные места с контролем того чтобы матрица смежность была верхняя треугольная
    for (int j = 0; j < vertDegrees[i]; j++) { //цикл, который проходит по всем вершинам, смежным с вершиной i.
        tmp = shuffler.Pop(); //получение вершины из объекта shuffler.
        MatrixSmejn[i][tmp] = 1; //установка значения в матрице смежности между вершинами i и tmp равным 1.
    }
}
// достиг до сети? :
bool flag = true;   //избавляемся от тупиковых вершин
for (int i = 0; i < vertexCnt-1; i++){
    flag = true;
    for (int j = 0; j < vertexCnt; j++) {
        if (MatrixSmejn[i][j] != 0) { //проверка, есть ли путь между вершинами i и j.
            flag = false; //роверка, является ли вершина i тупиковой.
            break;
        }
    }
    if (flag == true) {
        MatrixSmejn[i][i+1] = 1; //установка значения в матрице смежности между вершинами i и i + 1 равным 1.
    }
}
vector<int> sources; //инициализация вектора sources
//избавление от дополнительных истоков
flag = true;
for (int i = 1; i < vertexCnt - 1; i++) { //цикл, который проходит по всем вершинам, кроме первой и последней
    flag = true;
    if (MatrixSmejn[0][i] == 0) { //проверка, есть ли путь между вершинами 0 и i.
        for (int j = 1; j < vertexCnt; j++) {
            if (MatrixSmejn[j][i] != 0) { //проверка, есть ли путь между вершинами j и i.
                flag = false;
                break;
            }
        }
        if (flag) { // тупикова вкршина?
            sources.push_back(i); //добавление вершины i в вектор sources
        }
    }
}
if (sources.size() != 0) { //проверка, является ли вектор sources пустым.
    for (int i = 0; i < sources.size(); i++) { //цикл, который проходит по всем вершинам в векторе sources.
        MatrixSmejn[0][sources[i]] = 1; //установка значения в матрице
    }
}
//MatrixSmejn[0][(vertexCnt / 2) + 1] = 0;
//if(vertexCnt != 2)
//    MatrixSmejn[0][vertexCnt-1] = 0;
```

```
AssignWeights(); //весы
MixWeights(); //смкшанные весы
ModifyWeights(posWeightsMatrix, modPosWeightsMx); //вызов функции ModifyWeights для модификации весов ребер в матрице posWeightsMatrix.
ModifyWeights(mixedWeightsMatrix, modMixedWeightsMx); //вызов функции ModifyWeights для модификации весов ребер в матрицк
ReachMatrixGenerator(); //достежимости
```

```
    AssignTorrent(); // Стоимость пропуск
}
```

DFSUtil( ) -

Рекурсивный метод, который используется для обхода графа в глубину. Он выполняет обход в глубину, начиная с вершины v, и посещает все смежные вершины, которые еще не были посещены. Метод не возвращает никакого значения, но изменяет состояние массива visited и вектора visitedNodes. Он помечает вершины как посещенные и добавляет их в список посещенных вершин. Метод также выводит информацию о посещенных вершинах.

```
void MyGraph::DFSUtil(int v, vector<bool>& visited, vector<int>& visitedNodes) {
    // Помечаем текущую вершину как посещенную
    visited[v] = true;
    // Добавляем вершину в список посещенных вершин
    visitedNodes.push_back(v);

    // Выводим текущую вершину
    cout << "Посещаем вершину " << v+1;
    if (!visitedNodes.empty() && visitedNodes.size() > 1) {
        cout << " -> ";
    }

    // Рекурсивно вызываем DFS для всех смежных вершин, если они еще не посещены
    for (int i = 0; i < vertexCnt; ++i) {
        if (MatrixSmejn[v][i] && !visited[i]) {
            cout << endl; // Перенос строки между вершинами
            DFSUtil(i, visited, visitedNodes);
        }
    }
}
```

DFS( ) -

Алгоритм обхода в глубину

метод, который запускает обход в глубину (DFS) в графе. Он инициализирует массив visited для отслеживания посещенных вершин и вектор visitedNodes для хранения списка посещенных вершин. Затем он вызывает рекурсивную вспомогательную функцию DFSUtil() для обхода. в глубину, начиная с вершины startVertex. После завершения обхода в глубину, метод выводит сообщение о завершении обхода. Метод DFS() не возвращает никакого значения, но изменяет состояние массива visited и вектора visitedNodes.

```
void MyGraph::DFS(int startVertex) {
    cout << "Поиск в Глубину (DFS)" << endl;
    // Массив для отслеживания посещенных вершин
    vector<bool> visited(vertexCnt, false);
    // Список посещенных вершин
    vector<int> visitedNodes;

    // Вызываем рекурсивную вспомогательную функцию для обхода в глубину
    cout << "Начинаем обход в глубину с вершины " << startVertex << endl;
    DFSUtil(startVertex, visited, visitedNodes);

    // Выводим завершение обхода в глубину
    cout << endl << "Обход в глубину завершен." << endl;
}
```

BFS( ) -

Алгоритм обхода в ширину

метод, который запускает обход в ширину (BFS) в графе. Он инициализирует массив visited для отслеживания посещенных вершин и очередь q для хранения вершин, которые еще не были посещены. Затем он помечает начальную вершину как посещенную и добавляет ее в очередь. Метод продолжает обход, извлекая вершину из очереди, посещая ее, и добавляя все еще не посещенные смежные вершины в очередь. После завершения обхода в ширину, метод выводит сообщение о завершении обхода. Метод BFS( ) не возвращает никакого значения, но изменяет состояние массива visited.

```
void MyGraph::BFS(int startVertex1) {
    vector<bool> visited(vertexCnt, false); // Массив для отслеживания посещенных вершин
    queue<int> q; // Очередь для BFS

    // Начальная вершина помечается как посещенная и добавляется в очередь
    visited[startVertex1] = true;
    q.push(startVertex1);

    cout << "Начинаем обход в ширину с вершины " << startVertex1 << endl;

    while (!q.empty()) {
        int currentVertex = q.front();
        q.pop();
```

```
        cout << "Посещаем вершину " << currentVertex + 1 << " -> " << endl;

        // Проходим по всем смежным вершинам текущей вершины
        for (int i = 0; i < vertexCnt; ++i) {
            // Если есть ребро между текущей и следующей вершиной, и следующая вершина еще не посещалась
            if (MatrixSmejn[currentVertex][i] && !visited[i]) {
                // Помечаем вершину как посещенную и добавляем ее в очередь
                visited[i] = true;
                q.push(i);
            }
        }
    }

    cout << endl << "Обход в ширину завершен." << endl;
}
```

VertPrepare( ) -

Метод, используемый в конструкторе при формировании матрицы смежности.

Возвращает вектор чисел, сгенерированных в соответствии с распределением. Распределение реализовано при помощи функции LaplaceD( ). // Распределение Лапласа

```
vector<int> MyGraph::VertPrepare() const {
    vector<int> vertexDegrees;
    int tmpVertDeg;
    for (int i = 0; i < vertexCnt - 2; i++) {
        // цикл -2 потому что последняя вершина это сток и связь предпоследнего с последним в ручную в конструкторе
        do {
            tmpVertDeg = LaplaceD();
        } while (tmpVertDeg >= vertexCnt);
        vertexDegrees.push_back(tmpVertDeg);
    }
    std::sort(vertexDegrees.begin(), vertexDegrees.end(), std::greater<int>());
    return vertexDegrees;
}
```

CalcShimbell( ) -

Метод, принимающий количество вершин графа и объект класса enum (0 - кратчайшие пути, 1 - длиннейшие). Возвращает матрицу Шимбела. В методе нужное количество раз вызывается операция MultByShimbell( );

```
vector<vector<int> > MyGraph::CalcShimbell(int edgeCnt, ShimbellMode mode) const {
    vector<vector<int> > resM = posWeightsMatrix;
    for (int i = 0; i < edgeCnt - 1; i++) {
        resM = MultByShimbell(resM, posWeightsMatrix, mode);
    }
    return resM;
}
```

MultByShimbell( ) -

Метод, реализующий операцию умножения матриц в соответствии с математическим описанием метода Шимбелла. Принимает две матрицы и переменную, показывающую, какие требуются пути : кратчайшие или длиннейшие. Возвращает матрицу Шимбелла.

```
vector<vector<int> > MyGraph::MultByShimbell(const vector<vector<int> > firstM, const vector<vector<int> > secM, ShimbellMode mode) const {
    vector<vector<int> > resM(vertexCnt, vector<int>(vertexCnt, 0));
    vector<int> tmp;
    bool isNotZero;
    for (int i = 0; i < vertexCnt; i++) {
        for (int j = 0; j < vertexCnt; j++) {
            tmp.clear();
            isNotZero = false;
            for (int k = 0; k < vertexCnt; k++) {
                // элемент строки + элемент столбца как при сложение эл матриц
                if ((firstM[i][k] != 0) && (secM[k][j] != 0)) {
                    tmp.push_back(firstM[i][k] + secM[k][j]);
                    isNotZero = true;
                }
            }
            if (isNotZero) {
                if (mode == ShimbellMode::Short) {
                    resM[i][j] = *std::min_element(tmp.begin(), tmp.end());
                }
                else {    //mode == ShimbellMode::Long
                    resM[i][j] = *std::max_element(tmp.begin(), tmp.end());
                }
            }
            else {
                resM[i][j] = 0;
            }
        }
    }
    return resM;
}
```

Dijkstra( ) -

Метод, строящий кратчайший марщрут из точки по алгоритму Дейкстры. Принимает номер вершины-начала. Возвращает вектор, содержащий расстояния найденного кратчайшего пути.

```
vector<int> MyGraph::Dijkstra(int vertNum, int& counter) const { //O(V3) до O(V*E*log(V))
    vector<int> distances(modPosWeightsMx[vertNum]); //инициализация массива расстояний.
    distances[vertNum] = 0; // начальное = 0

    vector<bool> isVisitedVector(vertexCnt, false); // инициализация массива посещенных вершин.
    isVisitedVector[vertNum] = true; //зашли в вершину

    int min, curVert = vertexCnt - 1; //нициализация переменной min и переменной curVert.

    for (int i = 0; i < vertexCnt; i++) { //цикл, который проходит по всем вершинам.
        min = INF; //инициализация переменной min значением INF
        // Поиск минимальной вершины
        for (int j = 0; j < vertexCnt; j++, counter++) {
            if (!isVisitedVector[j] && (distances[j] < min)) { //проверка, является ли вершина j не посещенной и имеет ли она меньшее расстояние, чем переменная min.
                min = distances[j]; //обновление переменной min на значение расстояния до вершины j.
                curVert = j; //обновление переменной curVert на значение индекса вершины j.
            }
        }
        // Соседи этой вершины
        isVisitedVector[curVert] = true; // установка флага посещенной для вершины curVert.
        for (int j = 0; j < vertexCnt; j++, counter++) {
            if (!isVisitedVector[j] && (modPosWeightsMx[curVert][j] != INF)//проверка, является ли вершина j не посещенной, есть ли путь между вершинами curVert и j, не является ли текущее расстояние до вершины curVert бесконечным и является ли сумма расстояния до вершины curVert и веса ребра между ними меньше текущего расстояния до вершины j.
                && (distances[curVert] != INF) && ((distances[curVert] + modPosWeightsMx[curVert][j]) < distances[j]))
            {
                distances[j] = distances[curVert] + modPosWeightsMx[curVert][j];//обновление расстояния до вершины j на сумму расстояния до вершины curVert и веса ребра между ними.
            }
        }
    }
    return distances; //возврат массива расстояний.
}
```

BellmanFord( ) -

Метод , строящий кратчайший марщрут из точки по алгоритму Беллмана-Форда. Принимает номер вершины-начала и матрицу, в которой требуется найти кратчайший путь. Возвращает вектор, содержащий расстояния найденного кратчайшего пути.

```
vector<int> MyGraph::BellmanFord(int inpVert, int& counter, vector<vector<int> > mx) const {// для смежности сложность O(|V||E|) против O(|E| + |V|\ln(|V|))
    counter = 0;
    vector<int> distances(vertexCnt, INF); //инициализация vec расстояний.
    distances[inpVert] = 0; // установка начального расстояния до вершины inpVert равным 0.

    int curVert, newDistance;

    deque<int> dq; //инициализация очереди
    dq.push_back(inpVert); //добавление вершины inpVert в очередь

    while (!dq.empty()) { //цикл, который продолжается до тех пор, пока очередь не станет пустой.
        curVert = dq.front();
        dq.pop_front(); //получение вершины из начала очереди и удаление ее из очереди.
        for (int i = curVert + 1; i < vertexCnt; i++, counter++) {//цикл, который проходит по всем вершинам, смежным с текущей вершиной.
            if (mx[curVert][i] != INF) { //проверка, есть ли путь между текущей и следующей вершиной.
                newDistance = distances[curVert] + mx[curVert][i]; //вычисление нового расстояния.
                if (newDistance < distances[i]) { //проверка, является ли новое расстояние меньше текущего расстояния до следующей вершины.
                    distances[i] = newDistance; //обновление расстояния до следующей вершины.
                    if (std::find(dq.begin(), dq.end(), i) == dq.end()) {        //вершины в очереди нет
                        dq.push_back(i);                                    //добавили в конец очереди
                    }
                    else {                                                  //вершина есть в очереди
                        std::remove(dq.begin(), dq.end(), i);                   //удалили из очереди
                        dq.push_front(i);                                   //добавили в начало очереди
                    }
                }
            }
        }
    }
    return distances;
}
```

RestorePaths( ) -

метод восстановления путей. В качестве параметров принима- ет стартовую вершину, вектор расстояний, полученный из алгоритмов Дейкст- ры или Беллмана-Форда, и весовую матрицу. Возвращает матрицу, содержащую кратчайший путь.

```
vector<vector<int> > MyGraph::RestorePaths(int inpVert, const vector<int>& distances, const vector<vector<int> > weightMx) const {
    vector<vector<int> > paths(vertexCnt, vector<int>());
    int tmp, curVert;
    for (int i = 0; i < vertexCnt; i++) {
        if (distances[i] != 0) {        //== 0 -- исходная вершина
            if (distances[i] != INF) {    //Если есть путь
                curVert = i;
                paths[i].push_back(curVert);
                while (curVert != inpVert) {    //Пока не дошли до исходной вершины
                    for (int j = 0; j < vertexCnt; j++) {    //Ищем такую смежную вершину с нынешней что бы
                                                //значения ее метки было = разности значений
                                                //метки нынешней и веса ребра между ними
                        if (weightMx[j][curVert] != INF) { // если есть ребра
                            if ((distances[curVert] - weightMx[j][curVert]) == distances[j]) { // вычисляем
                                curVert = j;
                                paths[i].push_back(j);
                                break;
                            }
                        }
                    }
                }
            }
            else {
                paths[i].push_back(INF);
            }
        }
        else {
            paths[i].push_back(INF);
        }
    }
    return paths;
}
```

LaplaceD( ) -

Эта функция реализует заданное распределение. Возвращает сгенерированное число.

```
int LaplaceD() {
    random_device rd;
    mt19937 mersenne(rd());

    const double a = 10.0; // Параметр a для распределения Лапласа
    const double b = 6.0; // Параметр b для распределения Лапласа

    std::uniform_real_distribution<double> dist(0, 1);
    double u = dist(mersenne); // Генерируем u из равномерного распределения на [0, 1]
    //Если u равно 0, тогда 1 - u равно 1, и log(2 * (1 - u)) равно 0. Таким образом, x будет равно b, и это соответствует максимальной вероятности для значения x в    распределении Лапласа.
    //Если u равно 1, тогда 1 - u равно 0, и log(2 * (1 - u)) стремится к минус бесконечности. Таким образом, x будет стремиться к бесконечности отрицательной стороны,   что также соответствует распределению Лапласа.
```

```
    //Целая часть x: После того, как мы вычислили значение x, мы возвращаем его целую часть, так как в исходной функции указано, что результат должен быть целым числом.
    // то есть u определяет, какое конкретное значение из Лапласа будет выбрано для возврата
    // Используем обратное преобразование для генерации случайного числа
    double x;
    if (u < 0.5)
        x = b + a * log(2 * u);
    else
        x = b - a * log(2 * (1 - u));

    return static_cast<int>(x); // Возвращаем целую часть
}
```

PrintMatrix( ) -

Метод, выводящий в консоль матрицу, принятую в качестве параметра.

```
void PrintMatrix(const vector<vector<int> > mx) {
    int n = mx.size();
    int m = mx[0].size();
    cout << "   |";
    for (int i = 0; i < n; i++) {
        cout << setw(3) << i + 1 << "|";
    }
    cout << "\n";
    for (int i = 0; i < n; i++) {
        cout << setw(3) << i+1 << "|";
        for (int j = 0; j < m; j++) {
            if(mx[i][j] != INF)
                cout << setw(3) << mx[i][j] << " ";
            else
                cout << setw(3) << "inf" << " ";
        }
        cout << '\n';
    }
}
```

AddMatrix( ) -

Метод, реализующий сложение двух матриц. Принимает две матрицы, возвращает матрицу результата сложения.

```
vector<vector<int> > AddMatrixs(const vector<vector<int> > fMx, const vector<vector<int> > sMx) {
    vector<vector<int> > res(fMx.size(), vector<int>(fMx.size(), 0));
    for (int i = 0; i < fMx.size(); i++) {
```

```
        for (int j = 0; j < fMx.size(); j++) {
            res[i][j] = fMx[i][j] + sMx[i][j];
        }
    }
    return res;
}
```

SetMultMatrix( ) -

Метод умножени матриц с записью в ячейки матрицы максимального результата. Принимает две матрицы. Возвращает матрицу - результат умножения. Исполь- зуется для возведения матрицы смежности в степень при построении матрицы достижимости

```
vector<vector<int> > SetMultMatrixs(const vector<vector<int> > fMx, const vector<vector<int> > sMx) {
    vector<vector<int> > res(fMx.size(), vector<int>(fMx.size(), 0));
    vector<int> buf;
    for (int i = 0; i < fMx.size(); i++) {
        for (int j = 0; j < fMx.size(); j++) {
            buf.clear();
            for (int k = 0; k < fMx.size(); k++) {
                buf.push_back(fMx[i][k] && sMx[k][j]);
            }
            res[i][j] = *std::max_element(buf.begin(), buf.end());
        }
    }
    return res;
}
```

ShimbellMethod( ) -

Функция, вызываюшаяся в main. Принимает ссыылку на объект класса граф. Запрашивает у пользователя количество вершин, желаемую матрицу (кратчай- ших или длиннейших маршрутов). Выводит на экран матрицу весов и вызывает метод CalcShimbell( ) у графа, переданного в качестве параметра.

```
void ShimbellMethod(const MyGraph& graph) {
    int edgesCnt = InputEdgeCount(graph);
    int ShimMode = InputShimbelMode();

    cout << "Матрица весов:\n";
    PrintMatrix(graph.GetWeightsMatrix(0));
    cout << '\n';
    cout << "Матрица Шиммбелла:\n";
```

```
    PrintMatrix(graph.CalcShimbell(edgesCnt, static_cast<ShimbellMode>(ShimMode)));
    cout << '\n';
}
```

ReachabilityOperation( ) -

Функция, вызывающаяся в main. Принимает ссыылку на объект класса граф. За- прашивает у пользователя номера двух вершин. Затем выводит на экран матрицу достижимости, сформированную еще в конструкторе графа, а так же количество маршрутов из матрицы достижимости. Если маршрутов между вершинами нет, собщает об этом.

```
void ReachabilityOperation(const MyGraph& graph) {
    // Нумерация с 1
    cout << "Вершина 1: ";
    int vert1 = InputNumberOfVert(graph);

    cout << "Вершина 2: ";
    int vert2 = InputNumberOfVert(graph);
    vector<vector<int> > MatrixReach = graph.GetReachMatrix();
    vert1--;
    vert2--;

    if (vert1 != vert2) {
        cout << "Матрица достижимости:\n";
        PrintMatrix(MatrixReach);
        cout << '\n';
        if (MatrixReach[vert1][vert2] != 0) {
            cout << "Есть " << MatrixReach[vert1][vert2] << " путей между этими вершинами!\n";
        }
        else {
            cout << "У этих вершин путей не существует!\n";
        }
    }
    else
        cout << "Попытка построить путь из " << vert1 + 1 << " в " << vert2 + 1 << "не удалась("<< endl;
    cout << '\n';
}
```

DejkstraAlgorithm( ) -

Функция, вызывающаяся в main. Принимает ссылку на объект класса граф. Спрашивает у пользователя, с какой весовой матрицей требуется операция. Если со смешанной, выводится сообщение об ошибке. У пользователя запрашивается стартовая вершина и вызываются

методы Dijkstra() и RestorePaths(). После этого на экран выводятся найденные кратчайшие пути и количество итераций алгоритма Дейкстры

```
void DijkstraAlgorithm(const MyGraph& graph) {
    int decision = InputAlgorithmMode();
    if (decision == 0) {
        int inpVert = InputStartVert(graph), counter = 0;
        cout << '\n';
        inpVert--;

        cout << "Матрица весов:\n";
        PrintMatrix(graph.GetWeightsMatrix(0));
        cout << '\n';

        vector<int> distances = graph.Dijkstra(inpVert, counter);
        vector<vector<int> > paths = graph.RestorePaths(inpVert, distances, graph.GetWeightsMatrix(2));

        for (int i = 0; i < graph.GetVertexCount(); i++) {
            if (i != inpVert) {
                if (paths[i][0] != INF) {
                    for (int j = paths[i].size() - 1; j > 0; j--) {
                        cout << std::setw(2) << paths[i][j] + 1 << "--> ";
                    }
                    cout << std::setw(2) << paths[i][0] + 1;
                    cout << " это минимальный путь до вершины " << std::setw(2) << i + 1 << " длинною в " << std::setw(2) << distances[i];
                    cout << endl;
                }
                else {
                    cout << "До вершины " << std::setw(2) << i + 1 << " путя нет!\n";
                }
            }
        }
        cout << '\n';
        cout << " Кол-во итераций: " << counter << "\n\n";
    }
    else
        cout << "Алгоритм Дейкстры только с матрицей с положительными значениями!\n\n";
}
```

FloydWarshall( ) -

метод поиска кратчайших марщрутов по алгоритму Флойда - Уоршелла. Принимает матрицу, в которой нужно найти кратчайшие пути. Воз- вращает матрицу расстояний, в которой, если маршрута нет, стоит бесконечность (в программной реализации - макрос «inf», равный 999999999).

```
vector<vector<int> > MyGraph::FloydWarshall(int& counter, vector<vector<int> > mx) const { //O(n^3) time O(n^2) mem берем только ребра => матрицу расстояний и перебераем затем присваеваем вершины k  а ребра i - j
    vector<vector<int> > distancesMx = mx; //инициализация матрицы расстояний.
```

```
    for (int i = 0; i < vertexCnt; i++) { //цикл, который проходит по всем вершинам.
        for (int j = 0; j < vertexCnt; j++) {
            for (int k = 0; k < vertexCnt; k++, counter++) {
                if (distancesMx[i][k] != INF && distancesMx[k][j] != INF) { //проверка, есть ли путь между вершинами i, k и k, j.
                    distancesMx[i][j] = min(distancesMx[i][j], (distancesMx[i][k] + distancesMx[k][j]));  //обновление расстояния между
вершинами
                }
            }
        }
    }

    for (int i = 0; i < vertexCnt; i++) {
        distancesMx[i][i] = 0; //становка расстояния от вершины до самой себя равным 0. (так как граф без узлов)
    }

    return distancesMx;
}
```

bfs_FordFulkerson( ) -

метод, основанный на алгоритме поиска в глубину, используемый для поиска путей в реализации алгоритма Форда-Фалкерсона. Принимает матрицу пропускныых способностей, номер стартовой вершины, номер конечной вершины, ссылку на вектор для хранения пути. Возвращает true если между вершинами путь есть, и false если пути нет. (Сам алгоритм называется алгоритмом Эдмондса-Карпа!)

```
bool MyGraph::bfs_FordFulkerson(vector<vector<int>> matrix, int source, int sink, vector<int>& path) const {
    vector<bool> isVisitedArr(matrix.size(), false);
    queue<int> q;

    int curVert;

    q.push(source);
    isVisitedArr[source] = true;
    path[source] = -1;

    while (!q.empty()) {
        curVert = q.front();
        q.pop();

        for (int i = 0; i < matrix.size(); i++) {
            if (!isVisitedArr[i] && matrix[curVert][i] > 0) {
                path[i] = curVert;
                if (i == sink) {
                    return true; // Найден путь до стока
                }
                q.push(i);
```

```
                isVisitedArr[i] = true;
            }
        }
    }
    return false; // Путь до стока не найден
```

fordFulkerson( ) -

метод, реализующий алгоритм Форда-Фалкерсона. Принимает номер стартовой вершины и номер конечной вершины. Возвращает значение максимального потока, который можно пропустить через граф.

```
int MyGraph::fordFulkerson(int source, int sink) const {
    int tmpSink = sink;
    vector<vector<int>> residualMatrix = torrent; // Используем исходную матрицу для остаточной матрицы
    vector<int> path(residualMatrix.size(), 0); // Путь для BFS

    int maxFlow = 0;
    int curFlow;

    while (bfs_FordFulkerson(residualMatrix, source, tmpSink, path)) {
        curFlow = INF; // INF

        // Находим минимальную пропускную способность на пути от источника к стоку
        for (int i = tmpSink; i != source; i = path[i]) {
            curFlow = std::min(curFlow, residualMatrix[path[i]][i]);
        }

        // Увеличиваем поток вдоль найденного пути
        for (int i = tmpSink; i != source; i = path[i]) {
            residualMatrix[path[i]][i] -= curFlow;
            residualMatrix[i][path[i]] += curFlow;
        }

        // Увеличиваем общий поток
        maxFlow += curFlow;
    }

    return maxFlow;
}
```

BellmanFordAlgorithm( ) -

Функция, вызывающаяся в main. Принимает ссыылку на объект класса граф. Спрашивает у поьзователя, с какой весовой матрицей требуется операция. У поль- зователя запрашивается стартовая вершина и вызываются методы BellmanFord() и RestorePaths(). После этого на экран выводятся найденные кратчайшие пути и количество итераций алгоритма Беллмана-Форда.

```
void BellmanFordAlgorithm(const MyGraph& graph) {
    int decision = InputAlgorithmMode();
    if (decision == 1) {
        int inpVert = InputStartVert(graph), counter = 0;

        cout << '\n';
        inpVert--;

        cout << "Матрица весов:\n";
        PrintMatrix(graph.GetWeightsMatrix(1));
        cout << '\n';

        vector<int> distances = graph.BellmanFord(inpVert, counter, graph.GetWeightsMatrix(3));
        vector<vector<int>> paths = graph.RestorePaths(inpVert, distances, graph.GetWeightsMatrix(3));

        for (int i = 0; i < graph.GetVertexCount(); i++) {
            if (i != inpVert) {
                if (paths[i][0] != INF) {
                    for (int j = paths[i].size() - 1; j > 0; j--) {
                        cout << std::setw(2) << paths[i][j] + 1 << "--> ";
                    }
                    cout << std::setw(2) << paths[i][0] + 1;
                    cout << " Минимальный путь до вершины " << std::setw(2) << i + 1 << " длиною в " << std::setw(2) << distances[i];
                    cout << endl;
                }
                else {
                    cout << "До вершины" << std::setw(2) << i + 1 << " путя нет!\n";
                }
            }
        }
        cout << '\n';
        cout << " Количество итераций: " << counter << '\n';
        cout << '\n';
    }
    else {
        int inpVert = InputStartVert(graph), counter = 0;

        cout << '\n';
        inpVert--;

        cout << "Матрица весов:\n";
        PrintMatrix(graph.GetWeightsMatrix(0));
        cout << '\n';

        vector<int> distances = graph.BellmanFord(inpVert, counter, graph.GetWeightsMatrix(2));
        vector<vector<int>> paths = graph.RestorePaths(inpVert, distances, graph.GetWeightsMatrix(2));

        for (int i = 0; i < graph.GetVertexCount(); i++) {
            if (i != inpVert) {
                if (paths[i][0] != INF) {
                    for (int j = paths[i].size() - 1; j > 0; j--) {
                        cout << std::setw(2) << paths[i][j] + 1 << "--> ";
                    }
                    cout << std::setw(2) << paths[i][0] + 1;
                    cout << " это минимальный путь до вершины " << std::setw(2) << i + 1 << " длиною в " << std::setw(2) << distances[i];
                    cout << endl;
                }
                else {
                    cout << "До вершины " << std::setw(2) << i + 1 << " путя нет !\n";
                }
            }
        }
```

```
    }
    cout << '\n';
    cout << " Количество итераций: " << counter << '\n';
    cout << '\n';
  }
}
```

FloydWarshallAlgorithm( ) -

Функция, вызывающаяся в main. Принимает ссыылку на объект класса граф. Спрашивает у поьзователя, с какой весовой матрицей требуется операция. У поль- зователя запрашивается стартовая вершина и вызывается метод FloydWarshall(). После этого на экран выводится матрица, полученная в результате работы алго- ритма Флойда-Уоршела и количество итераций

```
void FloydWarshallAlgorithm(const MyGraph& graph) {
  int decision = InputAlgorithmMode();
  if (decision == 1) {
    int counter = 0;

    cout << "Матрица весов:\n";
    PrintMatrix(graph.GetWeightsMatrix(1));
    cout << '\n';

    vector<vector<int> > distancesMx = graph.FloydWarshall(counter, graph.GetWeightsMatrix(3));

    cout << "Матрица расстояний:\n";
    cout << "   |";
    for (int i = 0; i < graph.GetVertexCount(); i++) {
      cout << setw(3) << i + 1 << "|";
    }
    cout << endl;
    for (int i = 0; i < graph.GetVertexCount(); i++) {
      cout << setw(3) << i + 1 << "|";
      for (int j = 0; j < graph.GetVertexCount(); j++) {

        if (distancesMx[i][j] != INF) {
          cout << std::setw(3) << distancesMx[i][j] << " ";
        }
        else {
          cout << std::setw(3) << "inf" << " ";
        }
      }
      cout << '\n';
    }
    cout << '\n';

    cout << " Кол-во итераций: " << counter << '\n';
    cout << '\n';
  }
```

```
    else {
        int counter = 0;

        cout << "Матрица весов:\n";
        PrintMatrix(graph.GetWeightsMatrix(0));
        cout << '\n';

        vector<vector<int> > distancesMx = graph.FloydWarshall(counter, graph.GetWeightsMatrix(2));

        cout << "Матрица расстояний:\n";
        cout << "   |";
        for (int i = 0; i < graph.GetVertexCount(); i++) {
            cout << setw(3) << i + 1 << "|";
        }
        cout << endl;
        for (int i = 0; i < graph.GetVertexCount(); i++) {
            cout << setw(3) << i + 1 << "|";
            for (int j = 0; j < graph.GetVertexCount(); j++) {

                if (distancesMx[i][j] != INF) {
                    cout << std::setw(3) << distancesMx[i][j] << " ";
                }
                else {
                    cout << std::setw(3) << "inf" << " ";
                }
            }
            cout << '\n';
        }
        cout << '\n';

        cout << " Кол-во итераций: " << counter << "\n\n";
    }
}
```

MinCostFlow( ) -

Функция, вызывающаяся в main. Принимает ссыылку на объект класса граф. Спрашиваету пользователя номера двух вершин, между которыми будет прохо- дить поток. Вызывается метод calcMinCostFlow( ) и на экран выводится получен- ный поток минимальной стоимости.

```
void MinCostFlow(const MyGraph& graph) {
    bool flag = false;
    int firstV, lastV;
    do {
        cout << "Введите номер первой вершины : " << endl;
        firstV = InputVertex(1, graph.GetVertexCount());
        cout << "Введите номер второй вершины : " << endl;
        lastV = InputVertex(firstV+1, graph.GetVertexCount());
        if (firstV == lastV)
            cout << "Нельзя искать поток минимальной стоимости из вершины в нее саму !!!" << endl;
```

```
    } while (firstV == lastV);
        firstV--; lastV--;
        int schetMin = graph.calcMinCostFlow(firstV, lastV);
        cout << "Поток минимальной стоимости : " << schetMin << endl << endl;
}
```

CalcMinCostFlow( ) -

метод вычисления потока минимальной стоимости. При- нимает начальную и конечную вершны. В методе вычисляется максимальный по- ток при помощи алгоритма Форда-Фалкерсона. В процессе вычислений на жкран выводятся пути, по которым пускается поток, и стоимость за единицу потока. Возвращает число - найденный поток минимальной стоимости.

```
int MyGraph::calcMinCostFlow(int s, int t) const {
int flow = fordFulkerson(s, t); // max potok
int minCostFlow = 0;

    cout << "Максимальный поток : " << flow << ". ";
    flow = flow * 2 / 3;
    cout << "Используемый поток : " << flow << "\n";

    vector<vector<int>> tmpmodMixedWeightsMx = modMixedWeightsMx;
    vector<vector<int>> tmpTorrent = torrent; // Матрица пропуск. способностей

cout << "Матрица стоимости :" << endl;
    PrintMatrix(modMixedWeightsMx);
cout << "Матрица пропускных способностей :" << endl;
    PrintMatrix(torrent);

while (flow != 0) {
    vector<int> route = RestorePaths(s, BellmanFord(s, tmpmodMixedWeightsMx), tmpmodMixedWeightsMx)[t];
    sort(route.begin(), route.end());

int bottleNeck = INF;
int cost = 0;
vector<pair<int, int>> edgesToDelete;

for (vector<int>::iterator it = route.begin(); it != route.end() - 1; it++) {
    if (tmpTorrent[*it][*(it + 1)] < bottleNeck)
        bottleNeck = tmpTorrent[*it][*(it + 1)];
    cout << *it + 1 << "->";
}

bottleNeck = min(bottleNeck, flow);
    cout << t + 1 << " Пускаем поток " << bottleNeck;

for (vector<int>::iterator it = route.begin(); it != route.end() - 1; it++) {
    tmpTorrent[*it][*(it + 1)] -= bottleNeck;
    cost += tmpmodMixedWeightsMx[*it][*(it + 1)];
        if (!tmpTorrent[*it][*(it + 1)])
            edgesToDelete.push_back(pair<int, int>(*it, *(it + 1)));
    }

cout << " со стоимостью за единицу потока " << cost;
```

```
    minCostFlow = minCostFlow + (cost * bottleNeck);

cout << ". Итоговая стоимость " << cost * bottleNeck << "\n";

for (vector<pair<int,int>>::iterator it = edgesToDelete.begin(); it != edgesToDelete.end(); it++)
    tmpmodMixedWeightsMx[it->first][it->second] = INF;

    edgesToDelete.clear();

    flow -= bottleNeck;

}
cout << endl;

return minCostFlow;
}
```

PrimMST( ) -

метод, который используется для нахождения минимального остовного дерева в неориентированном графе. Он определяет минимальное остовное дерево как дерево, которое соединяет все вершины графа и имеет минимальную сумму весов ребер. Метод возвращает список рёбер этого минимального остовного дерева.

```
vector<tuple<int, int, int>> MyGraph::PrimMST() {
  int V = GetVertexCount();

  // Массив для хранения веса ключа для каждой вершины и индекса ребра
  vector<pair<int, pair<int, int>>> key(V + 1, {INT_MAX, {-1, -1}}); // Увеличиваем размер на 1 для вершин с 1 по V
  // Массив для отслеживания включенных вершин в MST
  vector<bool> inMST(V + 1, false); // Увеличиваем размер на 1 для вершин с 1 по V

  // Приоритетная очередь для хранения пар (вес, вершина)
  priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int, int>>> pq;

  // Начать с первой вершины (нумерация с 1)
  int src = 1; // Используем первую вершину
  key[src] = {0, {0, src}}; // Используем индексацию с 1
  pq.push({0, src});

  // Пока приоритетная очередь не пуста
  while (!pq.empty()) {
    int u = pq.top().second;
    pq.pop();

    // Включаем текущую вершину в MST
    inMST[u] = true;

    // Пройтись по смежным вершинам и обновить ключи
    for (int v = 1; v <= V; ++v) {
      if (MatrixSmejn[u][v] && !inMST[v] && posWeightsMatrix[u][v] < key[v].first) {
        key[v] = {posWeightsMatrix[u][v], {u, v}};
        pq.push({key[v].first, v});
      }
    }
  }
```

```
    // Создать и вернуть список рёбер MST
    vector<tuple<int, int, int>> result;
    for (int i = 1; i <= V; ++i) {
        if (key[i].second.first != -1) {
            // Добавляем ребро в список рёбер MST, но исключаем вершину 0
            if (key[i].second.first != 0) {
                // Получаем индексы вершин из списка ключей
                int u = key[i].second.first;
                int v = key[i].second.second;
                // Используем вершины - 1 в качестве индексов в матрице весов
                result.push_back({u, v, posWeightsMatrix[u - 1][v - 1]});
            }
        }
    }
```

```
    // Вывод рёбер минимального остовного дерева
    cout << "Рёбра миним.остовного дерева:" << endl;
    for (const auto& edge : result) {
        int u = get<0>(edge);
        int v = get<1>(edge);
        int weight = get<2>(edge);
        cout << u << " -> " << v << " (Вес пути: " << weight << ")" << endl;
    }
```

```
    return result;
}
```

Kraskal( ) -

Алгоритм Краскала для поиска минимального остовного дерева в графе начинает сортировку всех рёбер по их весу. Затем, по очереди, каждое ребро добавляется в остовное дерево, если его добавление не вызовет образование цикла. Для этого используется проверка на наличие пути между вершинами, соединяемыми текущим ребром. Если путь существует, то ребро не добавляется. Построенное остовное дерево выводится в виде матрицы, а также вычисляется и выводится общий вес остова.

```
void MyGraph::Kraskal(vector<vector<int>> weightMx, int &iterationCounter){
    minTree.clear();
    minTree = vector<vector<int>>(vertexCnt, vector<int>(vertexCnt, 0));
    list<std::pair<int, int>> sortedEdges; // для сортировки
```

```
    //Заполнить список вершин в порядке возрастания
    for (int i = 0; i < vertexCnt; i++) {
        for (int j = 0; j < vertexCnt; j++) {
            if (weightMx[i][j] != 0) {
                bool isEmplaced = false;
                for (auto iter = sortedEdges.begin(); iter != sortedEdges.end(); iter++) {
                    iterationCounter++;
                    if (weightMx[(*iter).first][(*iter).second] >= weightMx[i][j]) {
                        sortedEdges.emplace(iter, std::make_pair(i, j));
                        isEmplaced = true;
                        break;
```

```
                    }
                }
                if (!isEmplaced)
                    sortedEdges.push_back(std::make_pair(i, j));
            }
        }
    }

    // Заполняем кратчайший остов
    while (!sortedEdges.empty()) {
        auto curEdge = *(sortedEdges.begin());
        sortedEdges.pop_front();
        if (!isAchievable(curEdge.first, curEdge.second, minTree, -1)) {
            minTree[curEdge.first][curEdge.second] = weightMx[curEdge.first][curEdge.second];
            minTree[curEdge.second][curEdge.first] = weightMx[curEdge.first][curEdge.second];
        }
        if (isAchievable(curEdge.first, curEdge.first, minTree, -1)) {
            minTree[curEdge.first][curEdge.second] = 0;
            minTree[curEdge.second][curEdge.first] = 0;
        }
    }
    int resultSum = 0;
    for (int i = 0; i < vertexCnt; i++)
        for (int j = i + 1; j < vertexCnt; j++)
            if (minTree[i][j] != 0)
                resultSum += minTree[i][j];

    cout << "Кратчайший остов : " << endl;
    PrintMatrix(minTree);
    cout << "\nВес кратчайшего остова : " << resultSum << '\n';
}
```

PruferCode()-

Алгоритм для построения кода Прюфера по заданному минимальному остовному дереву. Алгоритм начинает с копирования остовного дерева, чтобы не изменять исходное. Затем для каждой вершины в графе проверяется, является ли она листом. Если вершина является листом и ещё не была добавлена в код Прюфера, то добавляется ребро, соединяющее эту вершину с ближайшей к ней вершиной в остовном дереве. После добавления ребра текущая вершина удаляется из копии остовного дерева, и процесс начинается заново. В результате получается код Прюфера, который является компактным представлением минимального остовного дерева, сохраняя информацию о связях между вершинами и весах рёбер.

```
void MyGraph::PruferCode(){
    PruferTree.clear();
    if (minTree.size() == 0) {
        cout << "Остов не был сгенерирован !!!\n";
        return;
    }

    auto copySpanTree = minTree;
```

```
    for (int i = 0; i < vertexCnt; i++) {
        bool isLeaf = true;
        int count = 0;

        for (int j = 0; j < vertexCnt; j++) {
            if (copySpanTree[i][j] != 0) {
                count++;
                if (count > 1) {
                    isLeaf = false;
                    break;
                }
            }
        }

        if (isLeaf && count != 0) {
            bool isAppripriate = true;
            if (!PruferTree.empty())
                for (auto iter = PruferTree.begin(); iter != PruferTree.begin(); iter++)
                    if ((*iter).first == i) { isAppripriate = false; break; }

            int nearVertex = 0;
            for (int k = 0; k < vertexCnt; k++)
                if (copySpanTree[i][k] != 0)
                    nearVertex = k;

            if (isAppripriate) {
                PruferTree.push_back(make_pair(nearVertex, copySpanTree[i][nearVertex]));
                copySpanTree[i][nearVertex] = 0;
                copySpanTree[nearVertex][i] = 0;
                i = 0;
            }
        }
    }

    cout << "Код Прюфера с весом: ";
    for (int i = 0; i < PruferTree.size(); i++)
        cout << PruferTree[i].first + 1 << "(" << PruferTree[i].second << ")" << '\t';

    cout << "\n\n";
}
```

PruferDecode()-

Функция PruferDecode восстанавливает минимальное остовное дерево по его коду Прюфера. В начале создаются вспомогательные структуры данных: вектор isUsed, который отслеживает использование вершин, и матрица decodeTree, представляющая восстановленное остовное дерево. Затем для каждой пары вершин в коде Прюфера ищется вершина, которая не входит в код Прюфера и ранее не была использована. Эта вершина добавляется в остовное дерево, связываясь с одной из вершин из кода Прюфера. После завершения восстановления дерева оно выводится на экран, а также сравнивается с

исходным минимальным остовным графом. Если они совпадают, выводится сообщение об успешном восстановлении.

```
void MyGraph::PruferDecode(){
    vector<bool> isUsed(vertexCnt, false);
    vector<vector<int>> decodeTree(vertexCnt, vector<int>(vertexCnt, 0));

    for (auto iter = PruferTree.begin(); iter != PruferTree.end(); iter++) {
        vector<bool> isNotInCode(vertexCnt, true);
        for (auto jter = iter; jter != PruferTree.end(); jter++) {
            isNotInCode[(*jter).first] = false;
        }
        for (int j = 0; j < vertexCnt; j++) {
            if (isNotInCode[j] && !(isUsed[j])) {
                decodeTree[j][(*iter).first] = (*iter).second;
                decodeTree[(*iter).first][j] = (*iter).second;
                isUsed[j] = true;
                break;
            }
        }
    }

    cout << "Результат декодирования :" << endl;
    PrintMatrix(decodeTree);
    cout << "Сам Мин.Ост.Граф :" << endl;
    PrintMatrix(minTree);

    if(minTree == decodeTree){
        cout << "Успешно!" << endl;
    }
    else cout << "Ошибка(" << endl;

    cout << endl;
}
```

Hamilton()-

В начале проверяется число вершин в графе. Если граф состоит из двух вершин, выводится сообщение о невозможности построения гамильтонова цикла, и процесс завершается. Преобразование матрицы весов: Матрица весов преобразуется таким образом, чтобы она стала симметричной. Проверка гамильтоновости: Если граф содержит три вершины, проверяется наличие рёбер между всеми парами вершин. Если в графе отсутствуют рёбра между некоторыми парами вершин, они добавляются случайным образом. Построение гамильтонова цикла: Если граф не является гамильтоновым, добавляются рёбра до тех пор, пока граф не станет гамильтоновым. Для этого используется теорема Дирака, согласно которой граф, у которого степени всех вершин не меньше половины числа вершин, является гамильтоновым. Если граф не удовлетворяет этому условию, добавляются рёбра до тех пор, пока все вершины не будут иметь степень не меньше половины числа вершин. Поиск всех гамильтоновых циклов: После этого начинается поиск всех гамильтоновых циклов с использованием рекурсивной функции findHamiltonCycle. Она проверяет все возможные

комбинации вершин для построения цикла и сохраняет минимальный найденный цикл.

Вывод результатов: После выполнения алгоритма выводятся все найденные гамильтоновы циклы, а также находится минимальный цикл и выводится его вес.

```
void MyGraph::Hamilton(vector<vector<int>> weightMx, int weightMode) const {
    ofstream fout("ListOfHamiltonianCycles.txt");
    if (vertexCnt == 2) {
     cout << "Нельзя составить цикл из 2-ух вершин(\n\n";
     fout.close();
     return;
    }

    auto modWeightMatrix = weightMx;
    for (int i = 0; i < vertexCnt; i++) {
     for (int j = 0; j < i; j++) {
      modWeightMatrix[i][j] = weightMx[j][i];
     }
    }

    // (Дирак при V>3 & степени не больше чем V/2) (если степени > Vert/2 => Гамельтонов
    // Оре?? V>2 & deg(x) + deg(y) >= V :: x,y пренадлежит несмежные вершины пары
    //по сути Бонди-Хватала но нет Дирак+Оре при этом sort(deg) с импликацией dk≤k<n/2⇒dn-k≥n-k,(*)
    if (vertexCnt == 3) {
     bool isHamiltonian = true;
     random_device rd;
     mt19937 mersenne(rd());
     for (int i = 0; i < vertexCnt; i++) {
      for (int j = 0; j < vertexCnt; j++) {
       if (i != j && modWeightMatrix[i][j] == 0) {
        isHamiltonian = false;
        modWeightMatrix[i][j] = (mersenne() % 40) + 1;
        if (weightMode == 1)
         modWeightMatrix[i][j] *= std::pow(-1, mersenne() % 2);
        modWeightMatrix[j][i] = modWeightMatrix[i][j];
       }
      }
     }
     if (!isHamiltonian) {
      PrintMatrix(GetWeightsMatrix(0));
      cout << "Граф не гамильтонов. Гамильтонова модифицированная матрица: \n";
      PrintMatrix(modWeightMatrix);
      cout << '\n';
     }
     else {
      cout << "Граф гамильтонов.\n";
      PrintMatrix(modWeightMatrix);
     }
    }
    else {
     bool isHamiltonian = true;
     vector<int> degrees(vertexCnt, 0);
     random_device rd;
     mt19937 mersenne(rd());

        // степени ++ для вершин если не 0
     for (int i = 0; i < vertexCnt; i++) {
      for (int j = 0; j < vertexCnt; j++) {
       if (modWeightMatrix[i][j] != 0) degrees[i]++;
      }
```

```
      if (degrees[i] < (vertexCnt / 2)) {
       isHamiltonian = false;
      }
     }

     // Если не гамельтонов то добавим ребро

     if (isHamiltonian) cout << "Граф гамильтонов\n";
     else {
      cout << "Граф не гамильтонов\n";

      while (!isHamiltonian) {
       bool isChanged = false;

          //с учетом теоремы Дирака так как должно быть N>V/2
       for (int i = 0; i < vertexCnt; i++) {
        if (degrees[i] < (vertexCnt / 2)) {
         // Добавляем ребра для deg > V/2
         int appropriateVertex = -1;

         for (int j = 0; j < vertexCnt; j++) {
          if (modWeightMatrix[i][j] == 0 && i != j) {
           if (appropriateVertex == -1) appropriateVertex = j;

           if (degrees[j] < (vertexCnt / 2)) {
            isChanged = true;
            degrees[i]++;
            degrees[j]++;
            modWeightMatrix[i][j] = (mersenne() % 100) + 1;
            if (weightMode == 1)
             modWeightMatrix[i][j] *= std::pow(-1, mersenne() % 2);
            modWeightMatrix[j][i] = modWeightMatrix[i][j];
            break;
           }
          }
         }

         if (!isChanged && appropriateVertex != -1) {
          isChanged = true;
          degrees[i]++;
          degrees[appropriateVertex]++;
          modWeightMatrix[i][appropriateVertex] = (mersenne() % 100) + 1;
          if (weightMode == 1)
           modWeightMatrix[i][appropriateVertex] *= std::pow(-1, mersenne() % 2);
          modWeightMatrix[appropriateVertex][i] = modWeightMatrix[i][appropriateVertex];
         }
        }
       }

       if (!isChanged) isHamiltonian = true;
      }

      cout << "Гамильтонова модифицированная матрица: \n";
      PrintMatrix(modWeightMatrix);
      std::cout << '\n';
     }
    }

    // Находим все гамильтоновы циклы
    vector<int> path;
    path.push_back(0);
```

```
    vector<int> minimumPath;
    int length = 0;
    int minimumLength = INT_MAX;

    findHamiltonCycle(fout, modWeightMatrix, path, length, minimumPath, minimumLength);

    cout << "\nМинимальный цикл: ";
    int i = 0;
    for (i; i < minimumPath.size()-1; i++)
     cout << minimumPath[i]+1 << "->";
    cout << minimumPath[i]+1 << endl;
    cout << "Вес: " << minimumLength << "\n\n";
    PrintMatrix(modWeightMatrix);
    fout.close();
}
```

+findHamiltonCycle() -

```
void MyGraph::findHamiltonCycle(ofstream& fout, vector<vector<int>>& graph, vector<int>& path, int length,
vector<int>& minimumPath, int& minimumLength) const
{
    if (path.size() == vertexCnt)
     {
      if (graph[path[path.size() - 1]][0] != 0)
      {
       length += graph[path[path.size() - 1]][0];
       path.push_back(0);

       if (minimumLength > length)
       {
        minimumPath = path;
        minimumLength = length;
       }

       fout << "Цикл: ";
       int i = 0;
       for (i ; i < path.size()-1; i++) {
        fout << path[i]+1 << "->";
       }
       fout << path[i]+1 << endl;
       fout << "Вес:" << length << '\n';

       path.pop_back();
      }
      return;
     }
     else {
      for (int i = 0; i < vertexCnt; i++) {
       if (graph[path[path.size() - 1]][i] != 0){
        bool isInPath = false;

        for (int j = 0; j < path.size(); j++) {
         if (path[j] == i) {
          isInPath = true;
          break;
         }
        }

        if (!isInPath)
        {
         int newLength = length + graph[path[path.size() - 1]][i];
```

```
            path.push_back(i);
            findHamiltonCycle(fout, graph, path, newLength, minimumPath, minimumLength);
            path.pop_back();
          }
        }
      }

      return;
    }
}
```

Euler() -

Элейровый цикл - цикл который проходит по ребрам (по одному разу) затем возвращается в начало - реализация начинается с проверки на количество вершин в графе. Если граф состоит из двух вершин, выводится сообщение о невозможности построения эйлерова цикла. Затем матрица весов преобразуется для обеспечения симметричности. После этого происходит проверка наличия эйлерова цикла в графе. Если граф уже является эйлеровым, выводится соответствующее сообщение. В противном случае добавляются рёбра до тех пор, пока все вершины графа не будут иметь чётные степени. Если граф остаётся неэйлеровым, происходит попытка сделать его эйлеровым путём добавления рёбер. Если после добавления рёбер граф остаётся неэйлеровым, происходит удаление одного ребра, чтобы избежать попыток бесконечного добавления рёбер. Затем вызывается функция EulerCycles, которая находит и выводит все эйлеровы циклы в графе.

```
void MyGraph::Euler(vector<vector<int>> weightMx, int weightMode) const {  // 0 если положительную, 1 если
смешанную
 if (vertexCnt == 2) {
  std::cout << "\nГраф из двух вершин не может иметь эйлеров цикл !!!\n\n";
  return;
 }

 auto modWeightMatrix = weightMx;
 for (int i = 0; i < modWeightMatrix[0].size(); i++) {
  for (int j = 0; j < i; j++) {
   modWeightMatrix[i][j] = weightMx[j][i];
  }
 }
 //Проверка : является ли граф эйлеровым ?
 bool isEulerian = true;
 bool modByEdges = true;
 vector<int> degrees(modWeightMatrix[0].size(), 0);
 random_device rd;
 mt19937 mersenne(rd());

 for (int i = 0; i < modWeightMatrix[0].size(); i++) {
  for (int j = 0; j < modWeightMatrix[0].size(); j++) {
   if (modWeightMatrix[i][j] != 0)
    degrees[i]++;
  }
  if (degrees[i] % 2 == 1) {
   if (degrees[i] == modWeightMatrix[0].size() - 1 && modByEdges) {
```

```
    cout << "Граф не является эйлеровым! Его нельзя сделать эйлеровым при помощи добавления ребер !!\n";
    modByEdges = false;
   }
   isEulerian = false;
  }
 }
auto tmpDeg = degrees;
if (modByEdges) {
 // Если граф не эйлеров, то добавить ребра чтобы сделать эйлеровым
 if (isEulerian) cout << "Граф является эйлеровым\n";
 else {
  //cout << "\nGraph is not Eulerian\n";
  while (!isEulerian) {
   bool isChanged = false;
   for (int i = 0; i < modWeightMatrix[0].size(); i++) {
    if (degrees[i] % 2 == 1) {
     // Добавляем ребра чтобы сделать степень четной
     int appropriateVertex = -1;
     for (int j = 0; j < modWeightMatrix[0].size(); j++) {
      if (modWeightMatrix[i][j] == 0 && i != j) {
       if (appropriateVertex == -1) {
        if (modWeightMatrix[0].size() % 2 == 0) {
         if (degrees[j] != modWeightMatrix[0].size() - 1)
          appropriateVertex = j;
        }
        else
         appropriateVertex = j;
       }
       if (degrees[j] % 2 == 1) {
        isChanged = true;
        degrees[i]++;
        degrees[j]++;
        modWeightMatrix[i][j] = (mersenne() % 40) + 1;
        if (weightMode == 1)
         modWeightMatrix[i][j] *= std::pow(-1, mersenne() % 2);
        modWeightMatrix[j][i] = modWeightMatrix[i][j];
        break;
       }
      }
     }

     if (!isChanged && appropriateVertex != -1) {
      isChanged = true;
      degrees[i]++;
      degrees[appropriateVertex]++;
      modWeightMatrix[i][appropriateVertex] = (mersenne() % 100) + 1;
      if (weightMode == 1)
       modWeightMatrix[i][appropriateVertex] *= std::pow(-1, mersenne() % 2);
      modWeightMatrix[appropriateVertex][i] = modWeightMatrix[i][appropriateVertex];
     }

     if (appropriateVertex == -1) {
      cout << "Граф не является эйлеровым!\n";
      modByEdges = false;
      break;
      //return;
     }
    }
   }

   if (!isChanged) isEulerian = true;
```

```
    if (!modByEdges) break;
   }
   if (modByEdges) {
    cout << "\nВесовая матрица эйлерова графа: \n";
    PrintMatrix(modWeightMatrix);
    cout << '\n';
   }
  }
 }
 // Если произошел амогус, то удалить ребро
 if (!modByEdges) {
  int k = 0;
  for (k; k < tmpDeg.size(); k++) {
   if (tmpDeg[k] % 2 == 1) {
    break;
   }
  }
  //cout << "ДО\n";
  //PrintMatrix(modWeightMatrix);
  for (int i = k; i < tmpDeg.size(); i++) {
   if (tmpDeg[i] % 2 == 1 && modWeightMatrix[k][i] != 0) {
    modWeightMatrix[k][i] = 0;
    modWeightMatrix[i][k] = 0;
    tmpDeg[k]--;
    tmpDeg[i]--;
    cout << "Удалено ребро\n";
    break;
   }
  }
 }
 EulerCycles(modWeightMatrix, tmpDeg);}
```

AntColonyOptimization()-

Так же происходит проверка на количество вершин в графе. Если граф состоит из двух вершин, выводится сообщение о невозможности составления цикла. Затем матрица весов преобразуется для обеспечения симметричности. Если граф состоит из трёх вершин, проверяется, является ли он гамильтоновым. В случае необходимости генерируются случайные веса для рёбер графа. Если граф состоит из более чем трёх вершин, проверяется, является ли он гамильтоновым. Если да, выводится соответствующее сообщение. В противном случае алгоритм завершается. Затем происходит инициализация параметров муравьиной оптимизации: количество муравьёв, количество итераций, вероятность испарения феромонов, а также коэффициенты притяжения феромона и геометрии пространства. Для каждой итерации муравьиной оптимизации создаётся путь каждым муравьём, применяется правило обновления феромонов, и находится минимальный гамильтонов цикл. По завершении поиска выводится минимальный цикл и его длина.

Функция findHamiltonCycleACD реализует выбор следующей вершины для муравья на основе вероятностей, которые зависят от количества феромона на рёбрах и их весов.

```
void MyGraph::AntColonyOptimization(vector<vector<int>> weightMx, int weightMode) const {
    ofstream fout("ListOfHamiltonianCycles.txt");
    if (vertexCnt == 2) {
        cout << "Нельзя составить цикл из 2-ух вершин(\n\n";
        fout.close();
        return;
    }


    auto modWeightMatrix = weightMx;
    for (int i = 0; i < vertexCnt; i++) {
        for (int j = 0; j < i; j++) {
            modWeightMatrix[i][j] = weightMx[j][i];
        }
    }

    if (vertexCnt == 3) {
        bool isHamiltonian = true;
        random_device rd;
        mt19937 mersenne(rd());
        for (int i = 0; i < vertexCnt; i++) {
            for (int j = 0; j < vertexCnt; j++) {
                if (i != j && modWeightMatrix[i][j] == 0) {
                    isHamiltonian = false;
                    modWeightMatrix[i][j] = (mersenne() % 40) + 1;
                    modWeightMatrix[j][i] = modWeightMatrix[i][j];
                }
            }
        }
        if (!isHamiltonian) {
            cout << "Граф не гамильтонов. Гамильтонова модифицированная матрица: \n";
            PrintMatrix(modWeightMatrix);
        }
        else {
            cout << "Граф гамильтонов.\n";
            PrintMatrix(modWeightMatrix);
        }
    }
    else {
        bool isHamiltonian = true;
        vector<int> degrees(vertexCnt, 0);
        random_device rd;
        mt19937 mersenne(rd());

        for (int i = 0; i < vertexCnt; i++) {
            for (int j = 0; j < vertexCnt; j++) {
                if (modWeightMatrix[i][j] != 0) degrees[i]++;
            }
            if (degrees[i] < (vertexCnt / 2)) {
                isHamiltonian = false;
            }
        }

        if (isHamiltonian) {
            cout << "Граф гамильтонов.\n";
            PrintMatrix(modWeightMatrix);
        } else {
```

```
            cout << "Граф не гамильтонов\n";
            return;
        }

        vector<vector<double>> pheromoneMatrix(vertexCnt, vector<double>(vertexCnt, 1.0));

        int n = 10; // количество муравьев
        int t = 100; // количество итераций
        double rho = 0.5; // вероятность испарения феромона
        double alpha = 10.0; // коэффициент притяжения феромона
        double beta = 5.0; // коэффициент притяжения геометрии пространства

        vector<int> minimumPath;
        int minimumLength = INT_MAX;

        for (int iter = 0; iter < t; ++iter) {
            vector<int> path;
            path.push_back(0);
            int length = 0;

            findHamiltonCycleACD(fout, modWeightMatrix, path, length, minimumPath, minimumLength);

            // Обновление феромонов
            for (int i = 0; i < vertexCnt; ++i) {
                for (int j = 0; j < vertexCnt; ++j) {
                    if (i != j && modWeightMatrix[i][j] != 0) {
                        pheromoneMatrix[i][j] *= (1.0 - rho);
                    }
                }
            }
        }

        // После завершения поиска, вывод минимального цикла и его длины
        cout << "\nМинимальный цикл: ";
        for (int i = 0; i < minimumPath.size() - 1; i++) {
            cout << minimumPath[i] + 1 << "->";
        }
        cout << minimumPath.back() + 1 << endl;
        cout << "Вес: " << minimumLength << "\n\n";

        fout.close();
    }
}
```

+ findHamiltonCycleACO()-

```
void MyGraph::findHamiltonCycleACD(ofstream& fout, vector<vector<int>>& graph, vector<int>& path,
int length, vector<int>& minimumPath, int& minimumLength) const {
    float ALPHA = 10.0;
    float BETA = 5.0;
    // Инициализация феромонов
    static vector<vector<double>> pheromoneMatrix(vertexCnt, vector<double>(vertexCnt, 1.0));
    if (path.size() == vertexCnt) {
        // Проверка завершения цикла
        if (graph[path.back()][path.front()] != 0) {
            length += graph[path.back()][path.front()];  // Добавляем вес последнего ребра
            path.push_back(path.front());  // Добавляем начальную вершину для замыкания цикла
```

```
            // Проверка на минимальный цикл
            if (length < minimumLength) {
                minimumLength = length;
                minimumPath = path;
            }

            // Вывод найденного цикла и его длины в файл
            fout << "Найден цикл: ";
            for (int i = 0; i < path.size() - 1; i++) {
                fout << path[i] + 1 << " -> ";
            }
            fout << path.back() + 1 << endl;  // Вывод последней вершины
            fout << "Длина цикла: " << length << endl;

            // Обратно отменяем изменения для продолжения поиска
            path.pop_back();
            length -= graph[path.back()][path.front()];
        }
        return;
    }

    // Расчет вероятностей и выбор следующей вершины
    int currentVertex = path.back();
    vector<double> probabilities(vertexCnt, 0.0);
    double sumProbabilities = 0.0;

    for (int i = 0; i < vertexCnt; i++) {
        if (graph[currentVertex][i] != 0 && std::find(path.begin(), path.end(), i) == path.end()) {
            probabilities[i] = pow(pheromoneMatrix[currentVertex][i], ALPHA) * pow(1.0 /
graph[currentVertex][i], BETA);
            sumProbabilities += probabilities[i];
        }
    }

    // Нормализация вероятностей
    for (int i = 0; i < vertexCnt; i++) {
        if (probabilities[i] > 0) {
            probabilities[i] /= sumProbabilities;
        }
    }

    // Выбор следующей вершины
    random_device rd;
    mt19937 mersenne(rd());
    uniform_real_distribution<double> dist(0.0, 1.0);
    double randValue = dist(mersenne);
    double cumulativeProbability = 0.0;

    for (int i = 0; i < vertexCnt; i++) {
        cumulativeProbability += probabilities[i];
        if (randValue <= cumulativeProbability) {
            path.push_back(i);
```

```
            findHamiltonCycleACD(fout, graph, path, length + graph[currentVertex][i], minimumPath,
minimumLength);
            path.pop_back();
            break;
        }
    }
}
```

FMIS()- // Алгоритм Магу

```
vector<vector<int>> MyGraph::FindMaximalIndependentSets() {
    vector<vector<int>> maximalIndependentSets;
    std::vector<std::vector<int>> Square = {
        {0, 1, 1, 0},
        {0, 0, 0, 1},
        {0, 0, 0, 1},
        {0, 0, 0, 0}
    };
    vector<vector<int>> tempMatrix = CreateUndirectedMatrix(Square);

    int vertexCount = vertexCnt;
    //  (от 0 до 2^v - 1)
    for (int mask = 1; mask < (1 << vertexCount); mask++) {
        vector<int> independentSet;

        bool isIndependent = true;
        //подмножество независимое ?
        for (int i = 0; i < vertexCount && isIndependent; i++) {
            if (mask & (1 << i)) {
                // вершина i включена в подмножество? проверить смежность с другими включенными
                for (int j = 0; j < vertexCount; j++) {
                    if (i != j && (mask & (1 << j)) && tempMatrix[i][j] != 0) {
                        isIndependent = false;
                        break;
                    }
                }
                // если независимое + вершину i в независимое множество
                if (isIndependent) {
                    independentSet.push_back(i + 1);
                }
            }
        }
        if (isIndependent) {
            bool isMaximal = true;
            // Проверяем, не является ли текущее независимое множество подмножеством другого уже
найденного максимального независимого множества
            for (const auto& maxSet : maximalIndependentSets) {
```

```
                if (includes(maxSet.begin(), maxSet.end(), independentSet.begin(),
independentSet.end())) {
                    isMaximal = false;
                    break;
                }
            }

            if (isMaximal) {
                maximalIndependentSets.push_back(independentSet);
            }
        }
    }

    return maximalIndependentSets;
}
```

ColorAndPrintIS() - //Раскраска графа

Создаётся матрица Square, представляющая собой квадратный граф, где единицы обозначают наличие ребра между соответствующими вершинами, а нули - отсутствие ребра. Далее вызывается функция CreateUndirectedMatrix, которая создаёт матрицу смежности для переданной матрицы квадратного графа. Затем запускается основной цикл, в котором перебираются все возможные подмножества вершин. Для каждого подмножества проверяется, является ли оно независимым множеством, т.е. не содержит пару вершин, соединённых ребром. Если подмножество является независимым, то проверяется, является ли оно максимальным среди уже найденных максимальных независимых множеств. Если да, то оно добавляется в список максимальных независимых множеств. По завершении работы алгоритма возвращается список всех найденных максимальных независимых множеств. Затем функция ColorAndPrintIndependentSets() дает цвета этим множествам, отличные от простых вершин.

```
void ColorAndPrintIndependentSets(MyGraph& graph) {
    // Найдем максимальные независимые множества.
    vector<vector<int>> independentSets = graph.FindMaximalIndependentSets();

    // Словарь для отображения цвета к каждой вершине.
    unordered_map<int, string> vertexColorMap;

    // Цвета для раскраски множеств.
    vector<string> colors = {"Red", "Green", "Blue", "Yellow", "Orange", "Purple",
"Pink"};

    // Переменная для отслеживания текущего цвета.
    int currentColorIndex = 0;
```

```
    // Итерация по каждому независимому множеству.
    for (const auto& set : independentSets) {
        // Назначаем цвет текущему независимому множеству.
        string currentColor = colors[currentColorIndex % colors.size()];
        // Раскрашиваем каждую вершину в этом множестве текущим цветом.
        cout << "{";
        for (size_t i = 0; i < set.size(); i++) {
            vertexColorMap[set[i]] = currentColor;
            cout << set[i];
            if (i < set.size() - 1) {
                cout << ", ";
            }
        }
        cout << "} - " << currentColor << endl;
        currentColorIndex++;
    }
}
```

aStarSearch()-

Начиная с заданной вершины, он ищет путь к целевой вершине в графе, минимизируя общую стоимость (сумму расстояния от начальной вершины до текущей и эвристической оценки от текущей до целевой вершины). Он использует очередь с приоритетом для поиска пути, обновляя стоимости вершин при нахождении более короткого пути. Когда целевая вершина достигнута, алгоритм восстанавливает путь от нее до начальной вершины и возвращает его.

```
vector<int> MyGraph::aStarSearch(const vector<vector<int>>& graph, int start, int end) {
    int n = graph.size();
           if (n == 0 || n != graph[0].size()) return {};
           vector<pair<int, int>> dist(n, {numeric_limits<int>::max(),
numeric_limits<int>::max()});
           previous.assign(n, {-1, -1});
           dist[start] = {0, heuristic(start, end, graph)};
           priority_queue<pair<int, int>, vector<pair<int, int>>, greater<pair<int,
int>>> pq;
           pq.push({0, start});
           while (!pq.empty()) {
               auto [cost, u] = pq.top();
               pq.pop();
               if (u == end) {
                   vector<int> path;
                   while (u != -1) {
                       path.push_back(u);
                       u = previous[u].first;
                   }
                   reverse(path.begin(), path.end());
```

```
                    return path;
                }
                for (int v = 1; v < n; ++v) {
                    if (graph[u][v] != 0) {
                        int new_cost = dist[u].first + graph[u][v];
                        if (new_cost < dist[v].first) {
                            dist[v].first = new_cost;
                            dist[v].second = new_cost + heuristic(v, end, graph); //
                            previous[v] = {u, graph[u][v]};
                            pq.push({dist[v].second, v});
                        }
                    }
                }
            }
            return {};
```

_____Функции выводов и вспомогательные функции проверок_____

```
#include "Operations.h"
int InputEdgeCount(const MyGraph& graph) {
    string strInp;
    bool flagInp;
    int res;
    do {
        cout << "Введите кол-во ребер графа- \n";
        getline(cin, strInp);
        if (regex_match(strInp, kRxNumber)) {
            res = stoi(strInp);
            if (res >= 1 && res <= graph.GetVertexCount() - 1) {
                flagInp = true;
            }
            else {
                cout << "Ошибочка( введите правильно!" << endl;
                flagInp = false;
            }
        }
        else {
            cout << "Ошибочка( введите правильно!!!" << endl;
            flagInp = false;
        }
    } while (!flagInp);
    cout << '\n';
    return res;
};
int InputShimbelMode() {
    string strInp;
    bool flagInp;
    int res;
```

```
    do {
        cout << "Желаемая матрица: матрица кратчайших путей [0], матрица длиннейших
путей [1]\n";
        getline(cin, strInp);
        if (regex_match(strInp, kRxNumber)) {
            res = stoi(strInp);
            if (res >= 0 && res <= 1) {
                flagInp = true;
            }
            else {
                cout << "Ошибочка!" << endl;
                flagInp = false;
            }
        }
        else {
            cout << "Ошибочка!" << endl;
            flagInp = false;
        }
    } while (!flagInp);
    cout << '\n';
    return res;
};
int InputNumberOfVert(const MyGraph& graph){
    string strInp;
    bool flagInp;
    int res;
    do {
        cout << "Введите номер вершины:\n";
        getline(cin, strInp);
        if (regex_match(strInp, kRxNumber)) {
            res = stoi(strInp);
            if (res >= 1 && res <= graph.GetVertexCount()) {
                flagInp = true;
            }
            else {
                cout << "Ошибочка!" << endl;
                flagInp = false;
            }
        }
        else {
            cout << "Ошибочка!" << endl;
            flagInp = false;
        }
    } while (!flagInp);
    cout << '\n';
    return res;
}
int InputStartVert(const MyGraph& graph) {
```

```
    string strInp;
    bool flagInp;
    int res;
    do {
        cout << "Введите исходную вершину:\n";
        getline(cin, strInp);
        if (regex_match(strInp, kRxNumber)) {
            res = stoi(strInp);
            if (res >= 1 && res <= graph.GetVertexCount()) {
                flagInp = true;
            }
            else {
                cout << "Ошибочка!" << endl;
                flagInp = false;
            }
        }
        else {
            cout << "Ошибочка!" << endl;
            flagInp = false;
        }
    } while (!flagInp);
    cout << '\n';
    return res;
}

int InputAlgorithmMode1() {
    string strInp;
    bool flagInp;
    int res;
    do {
        cout << "Какой алгоритм используем? [0] - Прима. [1] - Краскал.\n";
        getline(cin, strInp);
        if (regex_match(strInp, kRxNumber)) {
            res = stoi(strInp);
            if (res >= 0 && res <= 1) {
                flagInp = true;
            }
            else {
                cout << "Ошибочка!" << endl;
                flagInp = false;
            }
        }
        else {
            cout << "Ошибочка!" << endl;
            flagInp = false;
        }
    } while (!flagInp);
    cout << '\n';
```

```
    return res;
};

int InputAlgorithmMode() {
    string strInp;
    bool flagInp;
    int res;
    do {
        cout << "С какой матрицей работать (весов) ? [0] - с положительной. [1] - со
смешанной.\n";
        getline(cin, strInp);
        if (regex_match(strInp, kRxNumber)) {
            res = stoi(strInp);
            if (res >= 0 && res <= 1) {
                flagInp = true;
            }
            else {
                cout << "Ошибочка!" << endl;
                flagInp = false;
            }
        }
        else {
            cout << "Ошибочка!" << endl;
            flagInp = false;
        }
    } while (!flagInp);
    cout << '\n';
    return res;
};
int InputAlgorithmModeACD() {
    string strInp;
    bool flagInp;
    int res;
    do {
        cout << "Какой алгоритм хотите ? [0] - NP-полный алгоритм. [1] - Муравьинный
алгоритм.\n";
        getline(cin, strInp);
        if (regex_match(strInp, kRxNumber)) {
            res = stoi(strInp);
            if (res >= 0 && res <= 1) {
                flagInp = true;
            }
            else {
                cout << "Ошибочка!" << endl;
                flagInp = false;
            }
        }
        else {
```

```
            cout << "Ошибочка!" << endl;
            flagInp = false;
        }
    } while (!flagInp);
    cout << '\n';
    return res;
};
int InputVertex(int f, int s) {
    string strInp;
    bool flagInp;
    int res;
    do {
        cout << "Введите число от " << f << " до " << s << ":\n";
        getline(cin, strInp);
        if (regex_match(strInp, kRxNumber)) {
            res = stoi(strInp);
            if (res >= f && res <= s) {
                flagInp = true;
            }
            else {
                cout << "Ошибочка" << endl;
                flagInp = false;
            }
        }
        else {
            cout << "Ошибочка!" << endl;
            flagInp = false;
        }
    } while (!flagInp);
    cout << '\n';
    return res;
}

int InputVertex1(int f, int s) {
    string strInp;
    bool flagInp;
    int res;
    do {
        cout << "Введите число от " << f+1 << " до " << s << ":\n";
        getline(cin, strInp);
        if (regex_match(strInp, kRxNumber)) {
            res = stoi(strInp);
            if (res >= f && res <= s) {
                flagInp = true;
            }
            else {
                cout << "Ошибочка" << endl;
                flagInp = false;
```

```
            }
        }
        else {
            cout << "Ошибочка!" << endl;
            flagInp = false;
        }
    } while (!flagInp);
    cout << '\n';
    return res;
}

void ShimbellMethod(const MyGraph& graph) {
    unsigned int start_time = clock();
    int edgesCnt = InputEdgeCount(graph);
    int ShimMode = InputShimbelMode();

    cout << "Матрица весов:\n";
    PrintMatrix(graph.GetWeightsMatrix(0));
    cout << '\n';
    cout << "Матрица Шиммбелла:\n";
    PrintMatrix(graph.CalcShimbell(edgesCnt, static_cast<ShimbellMode>(ShimMode)));
    cout << '\n';
    cout << "Время выполнения: " << endl;
    unsigned int start_end = clock();
    cout<< start_end - start_time << " " << "мс";
    cout << " " << endl;
}

void ReachabilityOperation(const MyGraph& graph) {
    unsigned int start_time = clock();
    // Нумерация с 1
    cout << "Вершина 1: ";
    int vert1 = InputNumberOfVert(graph);

    cout << "Вершина 2: ";
    int vert2 = InputNumberOfVert(graph);
    vector<vector<int> > MatrixReach = graph.GetReachMatrix();
    vert1--; // спасибо C++
    vert2--;

    if (vert1 != vert2) {
        cout << "Матрица достижимости:\n";
        PrintMatrix(MatrixReach);
        cout << '\n';
        if (MatrixReach[vert1][vert2] != 0) {
            cout << "Есть " << MatrixReach[vert1][vert2] << " путей между этими
вершинами!\n";
        }
```

```
        else {
            cout << "У этих вершин путей не существуеn!\n";
        }
    }
    else
        cout << "Мы уже находимся в вершине " << vert1 + 1 << endl;
    cout << '\n';
    cout << "Время выполнения: " << endl;
    unsigned int start_end = clock();
    cout<< start_end - start_time << " " << "мс";
    cout << " " << endl;
}


void DijkstraAlgorithm(const MyGraph& graph) {
    unsigned int start_time = clock();
    int decision = InputAlgorithmMode();
    if (decision == 0) {
        int inpVert = InputStartVert(graph), counter = 0;
        cout << '\n';
        inpVert--;

        cout << "Матрица весов:\n";
        PrintMatrix(graph.GetWeightsMatrix(0));
        cout << '\n';

        vector<int> distances = graph.Dijkstra(inpVert, counter); //вызов функции
        vector<vector<int> > paths = graph.RestorePaths(inpVert, distances,
graph.GetWeightsMatrix(2));
//цикл, который проходит по всем вершинам графа, кроме начальной.
        for (int i = 0; i < graph.GetVertexCount(); i++) {
            if (i != inpVert) { //проверка, что текущая вершина не является начальной.
                if (paths[i][0] != INF) { //проверка, что путь до текущей вершины
существует.
                    //цикл, который проходит по элементам пути от текущей вершины до
начальной.
                    for (int j = paths[i].size() - 1; j > 0; j--) {
                        cout << std::setw(2) << paths[i][j] + 1 << "--> "; //вывод
элемента пути
                    }
                    // setw для прикола что б читалось лучше
                    cout << std::setw(2) << paths[i][0] + 1; //вывод начальной вершины
пути.
                    cout << " это минимальный путь до вершины " << std::setw(2) << i + 1
<< " длинною в " << std::setw(2) << distances[i];
                    cout << endl;
                }
                else {
```

```
                    cout << "До вершины " << std::setw(2) << i + 1 << " путя нет!\n";
                }
            }
        }
        cout << '\n';
        cout << " Кол-во итераций: " << counter << "\n\n";
        cout << "Время выполнения: " << endl;
        unsigned int start_end = clock();
        cout<< start_end - start_time << " " << "мс";
        cout << " " << endl;
    }
    else
        cout << "Алгоритм Дейкстры только с матрицей с положительными значениями!\n\n";
}

void BellmanFordAlgorithm(const MyGraph& graph) {
    unsigned int start_time = clock();
    int decision = InputAlgorithmMode();
    if (decision == 1) {
        int inpVert = InputStartVert(graph), counter = 0;

        cout << '\n';
        inpVert--;

        cout << "Матрица весов:\n";
        PrintMatrix(graph.GetWeightsMatrix(1));
        cout << '\n';

        vector<int> distances = graph.BellmanFord(inpVert, counter,
graph.GetWeightsMatrix(3));
        vector<vector<int> > paths = graph.RestorePaths(inpVert, distances,
graph.GetWeightsMatrix(3));

        for (int i = 0; i < graph.GetVertexCount(); i++) {
            if (i != inpVert) {
                if (paths[i][0] != INF) {
                    for (int j = paths[i].size() - 1; j > 0; j--) {
                        cout << std::setw(2) << paths[i][j] + 1 << "--> ";
                    }
                    cout << std::setw(2) << paths[i][0] + 1;
                    cout << " Минимальный путь до вершины " << std::setw(2) << i + 1 <<
" длиною в " << std::setw(2) << distances[i];
                    cout << endl;
                }
                else {
                    cout << "До вершины" << std::setw(2) << i + 1 << " путя нет!\n";
                }
            }
```

```
        }
        cout << '\n';
        cout << " Количество итераций: " << counter << '\n';
        cout << '\n';
        cout << "Время выполнения: " << endl;
        unsigned int start_end = clock();
        cout<< start_end - start_time << " " << "мс";
        cout << " " << endl;
    }
    else {
        int inpVert = InputStartVert(graph), counter = 0;

        cout << '\n';
        inpVert--;

        cout << "Матрица весов:\n";
        PrintMatrix(graph.GetWeightsMatrix(0));
        cout << '\n';

        vector<int> distances = graph.BellmanFord(inpVert, counter,
graph.GetWeightsMatrix(2));
        vector<vector<int> > paths = graph.RestorePaths(inpVert, distances,
graph.GetWeightsMatrix(2));

        for (int i = 0; i < graph.GetVertexCount(); i++) {
            if (i != inpVert) {
                if (paths[i][0] != INF) {
                    for (int j = paths[i].size() - 1; j > 0; j--) {
                        cout << std::setw(2) << paths[i][j] + 1 << "--> ";
                    }
                    cout << std::setw(2) << paths[i][0] + 1;
                    cout << " это минимальный путь до вершины " << std::setw(2) << i + 1
<< " длиною в " << std::setw(2) << distances[i];
                    cout << endl;
                }
                else {
                    cout << "До вершины " << std::setw(2) << i + 1 << " путя нет !\n";
                }
            }
        }
        cout << '\n';
        cout << " Количество итераций: " << counter << '\n';
        cout << '\n';
        cout << "Время выполнения: " << endl;
        unsigned int start_end = clock();
        cout<< start_end - start_time << " " << "мс";
        cout << " " << endl;
    }
```

```
}

//что бы восстонавить пути Флойда-Уоршалла улучшить алгоритм restore
void FloydWarshallAlgorithm(const MyGraph& graph) {
    unsigned int start_time = clock();
    int decision = InputAlgorithmMode();
    if (decision == 1) {
        int counter = 0;

        cout << "Матрица весов:\n";
        PrintMatrix(graph.GetWeightsMatrix(1));
        cout << '\n';

        vector<vector<int> > distancesMx = graph.FloydWarshall(counter,
graph.GetWeightsMatrix(3));

        cout << "Матрица расстояний:\n";
        cout << "   |";
        for (int i = 0; i < graph.GetVertexCount(); i++) {
            cout << setw(3) << i + 1 << "|";
        }
        cout << endl;
        for (int i = 0; i < graph.GetVertexCount(); i++) {
            cout << setw(3) << i + 1 << "|";
            for (int j = 0; j < graph.GetVertexCount(); j++) {

                if (distancesMx[i][j] != INF) {
                    cout << std::setw(3) << distancesMx[i][j] << " ";
                }
                else {
                    cout << std::setw(3) << "inf" << " ";
                }
            }
            cout << '\n';
        }
        cout << '\n';

        cout << " Кол-во итераций: " << counter << '\n';
        cout << '\n';
        cout << "Время выполнения: " << endl;
        unsigned int start_end = clock();
        cout<< start_end - start_time << " " << "мс";
        cout << " " << endl;
    }
    else {
        int counter = 0;

        cout << "Матрица весов:\n";
```

```
        PrintMatrix(graph.GetWeightsMatrix(0));
        cout << '\n';

        vector<vector<int> > distancesMx = graph.FloydWarshall(counter,
graph.GetWeightsMatrix(2));

        cout << "Матрица расстояний:\n";
        cout << "   |";
        for (int i = 0; i < graph.GetVertexCount(); i++) {
            cout << setw(3) << i + 1 << "|";
        }
        cout << endl;
        for (int i = 0; i < graph.GetVertexCount(); i++) {
            cout << setw(3) << i + 1 << "|";
            for (int j = 0; j < graph.GetVertexCount(); j++) {

                if (distancesMx[i][j] != INF) {
                    cout << std::setw(3) << distancesMx[i][j] << " ";
                }
                else {
                    cout << std::setw(3) << "inf" << " ";
                }
            }
            cout << '\n';
        }
        cout << '\n';

        cout << " Кол-во итераций: " << counter << "\n\n";
        cout << "Время выполнения: " << endl;
        unsigned int start_end = clock();
        cout<< start_end - start_time << " " << "мс";
        cout << " " << endl;
    }
}

void MinCostFlow(const MyGraph& graph) {
    unsigned int start_time = clock();
    bool flag = false;
    int firstV, lastV;
    do {
        cout << "Введите номер первой вершины : " << endl;
        firstV = InputVertex(1, graph.GetVertexCount()-1);
        cout << "Введите номер второй вершины : " << endl;
        lastV = InputVertex(firstV+1, graph.GetVertexCount());
        if (firstV == lastV)
            cout << "Нельзя искать поток минимальной стоимости из вершины в нее
саму !!!" << endl;
    } while (firstV == lastV);
```

```
        firstV--; lastV--;
        int schetMin = graph.calcMinCostFlow(firstV, lastV);
        cout << "Поток минимальной стоимости : " << schetMin << endl << endl;
    cout << "Время выполнения: " << endl;
    unsigned int start_end = clock();
    cout<< start_end - start_time << " " << "мс";
    cout << " " << endl;
}

void Hamilton(const MyGraph& graph) {
    unsigned int start_time = clock();
    int mode = InputAlgorithmModeACD();
    // Если работаем с положительной:
    if (mode == 0) {
        graph.Hamilton(graph.GetWeightsMatrix(0), mode); }
    //Эвристика :
    else {
        graph.AntColonyOptimization(graph.GetWeightsMatrix(0), mode);
    }
    cout << "Время выполнения: " << endl;
    unsigned int start_end = clock();
    cout<< start_end - start_time << " " << "мс";
    cout << " " << endl;
}

void Euler(const MyGraph& graph) { int mode = InputAlgorithmMode();
    unsigned int start_time = clock();
 // Если работаем с положительной:
    if (mode == 0) {
        graph.Euler(graph.GetWeightsMatrix(0), mode);
    }
 //Если с отрицательной :
    else {
        graph.Euler(graph.GetWeightsMatrix(1), mode);
    }
    cout << "Время выполнения: " << endl;
    unsigned int start_end = clock();
    cout<< start_end - start_time << " " << "мс";
    cout << " " << endl;}
```

4.Результаты программы:

Создание графа с 8 вершинами и меню с рекомендациями

```
Введите число вершин графа от 2 до 60:
8

1803
Матрица смежности вершин:
   |  1|  2|  3|  4|  5|  6|  7|  8|
  1|  0   1   1   1   1   1   1   1
  2|  0   0   1   1   1   1   0   1
  3|  0   0   0   1   1   1   1   0
  4|  0   0   0   0   1   1   1   0
  5|  0   0   0   0   0   1   1   1
  6|  0   0   0   0   0   0   1   1
  7|  0   0   0   0   0   0   0   1
  8|  0   0   0   0   0   0   0   0

Весовая положительная матрица:
   |  1|  2|  3|  4|  5|  6|  7|  8|
  1|  0   3   9  10   9  12   6   2
  2|  0   0   7   1  11  10   0   5
  3|  0   0   0   7   5  14  14   0
  4|  0   0   0   0  13   2   4   0
  5|  0   0   0   0   0   7  14  12
  6|  0   0   0   0   0   0   4  10
  7|  0   0   0   0   0   0   0   3
  8|  0   0   0   0   0   0   0   0

Весовая смешанная матрица:
   |  1|  2|  3|  4|  5|  6|  7|  8|
  1|  0   3   9 -10   9 -12   6  -2
  2|  0   0  -7   1 -11  10   0  -5
  3|  0   0   0  -7   5 -14  14   0
  4|  0   0   0   0  13  -2  -4   0
  5|  0   0   0   0   0  -7 -14  12
  6|  0   0   0   0   0   0  -4  10
  7|  0   0   0   0   0   0   0   3
  8|  0   0   0   0   0   0   0   0


Количество вершин: 8
Количество ребер: 25
```

```
Количество вершин: 8
Количество ребер: 25

Рекомендации:

Для быстрого поиска кратчайших путей можете выбрать:
Алгоритм Дейкстры, Алгоритмы DFS и BFS, Алгоритм Флойда-Уоршелла
Для минимального остовного дерева можно выбрать алгоритм по желанию


Метод Шимбелла _ _   _   _   _   _   _   _  [0]
Построить маршрут с n до m вершины_   _   _ [1]
Алгоритм Дейкстры_   _   _   _   _   _   _  [2]
Алгоритм Беллмана-Форда _   _   _   _   _   [3]
Алгоритм Флойда-Уоршалла _   _   _   _   _  [4]
Алгоритм поиска в глубину (DFS)_   _   _  _ [5]
Алгоритм поиска в ширину (BFS)_   _   _   _ [6]
Алгоритм Краскала либо Примы_  _  _  _  _ _ [7]
Код Прюфера (Заранее создайте Мин.Ост.Гр!!) [8]
Алгоритм Мин.стоим. через Эдмондса-Карпа_ _ [9]
Алгоритм минимального разреза графа_   _  _ [10]
Алгормтм А*_   _   _   _   _   _   _   _  _ [11]
Создать новый граф?_   _   _   _   _   _  _ [12]
Алгоритм Луна_   _   _   _   _   _   _    _ [13]
Гамельтонов цикл. мин_   _   _   _   _    _ [14]
Эйлеровый цикл_  _   _   _   _   _   _    _ [15]
Алгоритм Магу + Раскраска_   _   _   _    _ [16]
Выход_   _   _   _   _   _   _   _   _    _ [17]

Выберите пункт меню нажав нужную цифру:
```

Метод Шимбелла

```
Выберите пункт меню нажав нужную цифру:
0

Введите кол-во ребер графа-
2

Желаемая матрица: матрица кратчайших путей [0], матрица длиннейших путей [1]
0

Матрица весов:
   |  1|  2|  3|  4|  5|  6|  7|  8|
  1|  0   3   9  10   9  12   6   2
  2|  0   0   7   1  11  10   0   5
  3|  0   0   0   7   5  14  14   0
  4|  0   0   0   0  13   2   4   0
  5|  0   0   0   0   0   7  14  12
  6|  0   0   0   0   0   0   4  10
  7|  0   0   0   0   0   0   0   3
  8|  0   0   0   0   0   0   0   0

Матрица Шиммбелла:
   |  1|  2|  3|  4|  5|  6|  7|  8|
  1|  0   0  10   4  14  12  14   8
  2|  0   0   0  14  12   3   5  20
  3|  0   0   0   0  20   9  11  17
  4|  0   0   0   0   0  20   6   7
  5|  0   0   0   0   0   0  11  17
  6|  0   0   0   0   0   0   0   7
  7|  0   0   0   0   0   0   0   0
  8|  0   0   0   0   0   0   0   0

Время выполнения:
1314 мс
Продолжаем? (Y/N): |
```

Построение маршрута от n до m вершины

```
Выберите пункт меню нажав нужную цифру:
1

Вершина 1: Введите номер вершины:
2

Вершина 2: Введите номер вершины:
7

Матрица достижимости:
   |  1|  2|  3|  4|  5|  6|  7|  8|
  1|  1   0   1   2   3   4   5   6
  2|  0   1   0   1   2   3   4   5
  3|  0   0   1   0   1   2   3   4
  4|  0   0   0   1   0   1   2   3
  5|  0   0   0   0   1   0   1   2
  6|  0   0   0   0   0   1   0   1
  7|  0   0   0   0   0   0   1   0
  8|  0   0   0   0   0   0   0   1

Есть 4 путей между этими вершинами!

Время выполнения:
890 мс
Продолжаем? (Y/N):
```

Алгоритм Дейкстры

```
Выберите пункт меню нажав нужную цифру:
2

С какой матрицей работать (весов) ? [0] – с положительной. [1] – со смешанной.
0

Введите исходную вершину:
1


Матрица весов:
   |  1|  2|  3|  4|  5|  6|  7|  8|
  1|  0   3   9  10   9  12   6   2
  2|  0   0   7   1  11  10   0   5
  3|  0   0   0   7   5  14  14   0
  4|  0   0   0   0  13   2   4   0
  5|  0   0   0   0   0   7  14  12
  6|  0   0   0   0   0   0   4  10
  7|  0   0   0   0   0   0   0   3
  8|  0   0   0   0   0   0   0   0

 1-->  2 это минимальный путь до вершины  2 длинною в  3
 1-->  3 это минимальный путь до вершины  3 длинною в  9
 1-->  2-->  4 это минимальный путь до вершины  4 длинною в  4
 1-->  5 это минимальный путь до вершины  5 длинною в  9
 1-->  2-->  4-->  6 это минимальный путь до вершины  6 длинною в  6
 1-->  7 это минимальный путь до вершины  7 длинною в  6
 1-->  8 это минимальный путь до вершины  8 длинною в  2

 Кол-во итераций: 128

Время выполнения:
1314 мс
Продолжаем? (Y/N):
```

Алгоритм поиска кратчайшего расстояния с помощью DFS

```
Выберите пункт меню нажав нужную цифру:
5

Введите начальную вершину для поиска в глубину: 1
Введите конечную вершину для поиска в глубину: 5
Кратчайший путь из 1 в 5: 1 -> 5 (суммарный вес пути: 9)
Время выполнения:
96 мс
Продолжаем? (Y/N):
```

Алгоритм Примы (справа) и Краскала (слева)

```
С какой матрицей работать (весов) ? [0] - с положительной. [1] - со смешанной.
0

Кратчайший остов :
РАБОТА С НЕОРИЕНТИРОВАННЫЙ ГРАФ!
   |  1|  2|  3|  4|  5|  6|  7|  8|
  1|  0   3   0   0   0   0   0   2
  2|  3   0   7   1   0   0   0   0
  3|  0   7   0   0   5   0   0   0
  4|  0   1   0   0   0   2   0   0
  5|  0   0   5   0   0   0   0   0
  6|  0   0   0   2   0   0   0   0
  7|  0   0   0   0   0   0   0   3
  8|  2   0   0   0   0   0   3   0
Вес кратчайшего остова : 23
Количество итераций : 224

Время выполнения:
314 мс
```

```
С какой матрицей работать (весов) ? [0] - с положительной. [1] - со смешанной.
1

Кратчайший остов :
   |  1|  2|  3|  4|  5|  6|  7|  8|
  1|  0   3   0   0   0   0   0   2
  2|  3   0   0   1   0   0   0   0
  3|  0   0   0   0   5   0   0   0
  4|  0   1   0   0   0   2   0   0
  5|  0   0   5   0   0   7   0   0
  6|  0   0   0   2   7   0   0   0
  7|  0   0   0   0   0   0   0   3
  8|  2   0   0   0   0   0   3   0

Вес кратчайшего остова : 23
Количество итераций : 168

Время выполнения:
843 мс
```

Алгоритм нахождения минимального Гамильтонова цикла (NP-полный слева, ACO-справа)

```
Какой алгоритм хотите ? [0] - NP-полный алгоритм. [1] - Муравьинный алгоритм.
0

Граф гамильтонов

Минимальный цикл: 1->2->3->6->7->4->8->5->1
Вес: 24

   |  1|  2|  3|  4|  5|  6|  7|  8|
  1|  0   1  11   2   3  12  11   6
  2|  1   0   1   2  14  14  10   8
  3| 11   1   0   9  12  12  12  14
  4|  2   2   9   0   6   9   2   1
  5|  3  14  12   6   0  14   0   1
  6| 12  14  12   9  14   0   3   0
  7| 11  10  12   2   0   3   0  10
  8|  6   8  14   1   1   0  10   0
Время выполнения:
22057 мс
Продолжаем? (Y/N): y
```

```
Какой алгоритм хотите ? [0] - NP-полный алгоритм. [1] - Муравьинный алгоритм.
1

Граф гамильтонов.
   |  1|  2|  3|  4|  5|  6|  7|  8|
  1|  0   1  11   2   3  12  11   6
  2|  1   0   1   2  14  14  10   8
  3| 11   1   0   9  12  12  12  14
  4|  2   2   9   0   6   9   2   1
  5|  3  14  12   6   0  14   0   1
  6| 12  14  12   9  14   0   3   0
  7| 11  10  12   2   0   3   0  10
  8|  6   8  14   1   1   0  10   0

Минимальный цикл: 1->2->3->6->7->4->8->5->1
Вес: 24

Время выполнения:
13159 мс
```

Алгоритм нахождения минимального Эйлерова цикла

```
Выберите пункт меню нажав нужную цифру:
15

С какой матрицей работать (весов) ? [0] - с положительной. [1] - со смешанной.
0

Граф не является эйлеровым! Его нельзя сделать эйлеровым при помощи добавления ребер !!
Удалено ребро
Матрица эйлерова графа
   |  1|  2|  3|  4|  5|  6|  7|  8|
  1|  0   0  11   2   3  12  11   6
  2|  0   0   1   2  14  14  10   8
  3| 11   1   0   9  12  12  12  14
  4|  2   2   9   0   6   9   2   1
  5|  3  14  12   6   0  14   0   1
  6| 12  14  12   9  14   0   3   0
  7| 11  10  12   2   0   3   0  10
  8|  6   8  14   1   1   0  10   0

Вершины эйлерового цикла  графе: 1->3->4->7->8->5->6->7->4->6->3->5->4->3->8->2->7->1->6->2->5->1->4->2->3->1

Время выполнения:
1056 мс
```

Примеры защиты от ввода

```
Продолжаем? (Y/N): н
Ошибочка! Вводите y/Y либо n/N!
Продолжаем? (Y/N): y


Метод Шимбелла _ _   _   _   _   _   _   _  [0]
Построить маршрут с n до m вершины_   _   _ [1]
Алгоритм Дейкстры_   _   _   _   _   _   _  [2]
Алгоритм Беллмана-Форда _   _   _   _   _   [3]
Алгоритм Флойда-Уоршалла _   _   _   _   _  [4]
Алгоритм поиска в глубину (DFS)_   _   _  _ [5]
Алгоритм поиска в ширину (BFS)_   _   _   _ [6]
Алгоритм Краскала либо Примы_  _  _  _  _ _ [7]
Код Прюфера (Заранее создайте Мин.Ост.Гр!!) [8]
Алгоритм Мин.стоим. через Эдмондса-Карпа_ _ [9]
Алгоритм минимального разреза графа_   _  _ [10]
Алгормтм A*_   _   _   _   _   _   _   _  _ [11]
Создать новый граф?_   _   _   _   _   _  _ [12]
Алгоритм Луна_   _   _   _   _   _    _   _ [13]
Гамельтонов цикл. мин_   _   _   _    _   _ [14]
Эйлеровый цикл_  _   _   _   _   _   _    _ [15]
Алгоритм Магу + Раскраска_   _   _   _    _ [16]
Выход_   _   _   _   _   _   _   _   _    _ [17]

Выберите пункт меню нажав нужную цифру:
58
Ошибочка! Введите цифру из того что есть в меню
Выберите пункт меню нажав нужную цифру:
```

```
Выберите пункт меню нажав нужную цифру:
1

Вершина 1: Введите номер вершины:
2

Вершина 2: Введите номер вершины:
2

Мы уже находимся в вершине 2
```

```
С какой матрицей работать (весов) ? [0] - с положительной. [1] - со смешанной.
0

Введите исходную вершину:
19
Ошибочка!
Введите исходную вершину:
38
Ошибочка!
Введите исходную вершину:
```

Алгоритм нахождения мин.стоимости с помощью алгоритма Эдмонда-Карпа.

```
Выберите пункт меню нажав нужную цифру:
9

Введите номер первой вершины :
Введите число от 1 до 7:
1

Введите номер второй вершины :
Введите число от 2 до 8:
7

Максимальный поток : 28. Используемый поток : 18
Матрица стоимости :
   |  1|  2|  3|  4|  5|  6|  7|  8|
  1|inf   1  11   2   3  12  11   6
  2|inf inf   1   2  14  14  10   8
  3|inf inf inf   9  12  12  12  14
  4|inf inf inf inf   6   9   2   1
  5|inf inf inf inf inf  14 inf   1
  6|inf inf inf inf inf inf   3 inf
  7|inf inf inf inf inf inf inf  10
  8|inf inf inf inf inf inf inf inf
Матрица пропускных способностей :
   |  1|  2|  3|  4|  5|  6|  7|  8|
  1|  0  10   3   1   4  13   9   2
  2|  0   0  12  13   7   7   7  14
  3|  0   0   0   3   5  14  12   5
  4|  0   0   0   0  13   5   1  11
  5|  0   0   0   0   0   9   0   3
  6|  0   0   0   0   0   0   5   0
  7|  0   0   0   0   0   0   0  13
  8|  0   0   0   0   0   0   0   0
1->4->7 Пускаем поток 1 со стоимостью за единицу потока 4. Итоговая стоимость 4
1->7 Пускаем поток 9 со стоимостью за единицу потока 11. Итоговая стоимость 99
1->2->7 Пускаем поток 7 со стоимостью за единицу потока 11. Итоговая стоимость 77
1->2->3->7 Пускаем поток 1 со стоимостью за единицу потока 14. Итоговая стоимость 14

Поток минимальной стоимости : 194

Время выполнения:
2243 мс
Продолжаем? (Y/N):
```

Кодировка и Декод методом Прюфера

```
Выберите пункт меню нажав нужную цифру:
8

Код Прюфера с весом: 2(1)   1(1)    8(1)    7(3)    4(2)    4(1)    1(2)

Результат декодирования :
    |  1|  2|  3|  4|  5|  6|  7|  8|
  1|  0   1   0   2   0   0   0   0
  2|  1   0   1   0   0   0   0   0
  3|  0   1   0   0   0   0   0   0
  4|  2   0   0   0   0   0   2   1
  5|  0   0   0   0   0   0   0   1
  6|  0   0   0   0   0   0   3   0
  7|  0   0   0   2   0   3   0   0
  8|  0   0   0   1   1   0   0   0
Сам Мин.Ост.Граф :
    |  1|  2|  3|  4|  5|  6|  7|  8|
  1|  0   1   0   2   0   0   0   0
  2|  1   0   1   0   0   0   0   0
  3|  0   1   0   0   0   0   0   0
  4|  2   0   0   0   0   0   2   1
  5|  0   0   0   0   0   0   0   1
  6|  0   0   0   0   0   0   3   0
  7|  0   0   0   2   0   3   0   0
  8|  0   0   0   1   1   0   0   0
Успешно!
```

5. Заключение

В результате работы на языке программирования C++ были реализованы:
алгоритм создания связного ациклического графа, с распределением ребер с помощью распределения Лапласа. Реализованы алгоритмы - Поиска минимальних и максимальных путей методом Шимбелла, построение маршрута от выбранных вершин, алгоритм Дейкстры, алгоритм Флойда-Уоршалла, алгоритм Беллмана-Форда, алгоритм кратчайших путей с использованием обхода DFS, а так же BFS, алгоритм A* с эвристической функцией Манхэтеннское расстояние, алгоритм Эдмондса-Карпа, алгоритм нахождения минимальной стоимости с помощью алгоритма Эдмондса-Карпа, алгоритм, алгоритм нахождения минимального остовного графа - Прима и Краскала, после чего остовы можно было закодировать методом Прюфера и так же декодирование кода Прюфера, алгоритм нахождения минимального гамильтонова цикла, с записью всех остальных циклов в файл (np-полная задача), алгоритм нахождения гамильтонова цикла с помощью оптимизации ACO (Ant Colony System) Муравьиный Алгоритм, для него был выбран способ MMAS (MinMax Ant System), алгоритм нахождения эйлерова цикла, с помощью степеней вершин, алгоритм Магу с последующей раскраской МНП графа.
Результаты сравнений (самые быстрые алгоритмы и самые медленные в нашей реализации)
Для кратчайших путей быстрее всех - A*, медленнее - BFS
Для минимального остовного графа быстрее всех - Краскала, медленнее - Прима
Для нахождения Гамильтонова цикла быстрее всех - ACO, медленнее NP-перебор
Для нахождения максимального потока - Форд-Фалкерсон, медленнее Эдмондс-Карп (в нашей реализации)
Достоинства - 1) Все алгоритмы можно использовать под другие графы
2) Почти все алгоритмы оптимизированы и/ или имеют оптимизированные аналоги.
Недостатки - 1) Не найдена оптимизация для алгоритмов кратчайших путей BFS и DFS.
2) Специальные алгоритмы достроения включаются в общую сложность алгоритмов, что может повлиять на скорость выполнения.

6. Список литературы

1. Ф.А.Новиков. Дискретная математика для программистов. СПб: Питер Пресс, 2009г. 364с.
2. Томас Кормен. Алгоритмы и Структуры Данных
3. Стивен Скиена - Алгоритмы
4. Бхаргава Адитья - Грокаем Алгоритмы
5. Вадзинский Р.Н. Справочник по вероятностным распределениям. 94с.
6. КОМПЬЮТЕРНАЯ МАТЕМАТИКА. Д.Кук, Г.Бейз. М.: Наука. Гл. ред. физ.-мат. лит.,1990, 304 с.