САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ им. ПЕТРА ВЕЛИКОГО
Институт Компьютерных наук и технологий
Высшая школа искусственного интеллекта
Направление 02.03.01 Математика и компьютерные науки

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

Реализация алгоритмов на графах

Группа: 3530201/00002

Студент: _____ Перекрестов Глеб Владимирович

Преподаватель: _____ Востров Алексей Владимирович

«____»_____ 20__г.

Санкт-Петербург – 2022

Содержание

Введение

В отчете содержится описание лабораторных работ №1-5 по дисциплине «Теория графов». Лабораторные работы включают в себя реализацию алгоритмов, применяемых к связному ациклическому графу. Граф сформирован при помощи отрицательного гипергеометрического распределения 2.

Работа была выполнена в среде Visual Studio 2019 на языке программирования C++.

1 Постановка задачи

Задача состоит в построении случайного связного ациклического графа в соответствии с заданным распределнием. Необходимо реализовать алгоритмы для выполнения следующих задач:

- Поиск самого длинного и самого короткого путей при помощи метода Шимбелла;
- Определение возможности построения маршрута между вершинами и количества таких маршрутов;
- Нахождение кратчайших путей при помощи алгоритмов Дейкстры, Беллмана-Форда и Флойда-Уоршелла;
- Поиск максимального потока при помощи алгоритма Форда-Фалкерсона и поиск потока минимальной стоимости;
- Построение минимального по весу остова при помощи алгоритмов Прима и Краскала;
- Нахождение количества остовных деревьев по матричной теореме Кирхгофа;
- Кодирование и декодирование минимального остова с помощью кода Прюфера;
- Модификация графа до эйлерова и до гамильтонова;
- Поиск эйлеровых циклов;
- Нахождение минимального по весу гамильтонового цикла.

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 множества V (E - множество ребер).

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

p - число вершин графа, 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.

Ациклический граф - граф, который не содержит циклов.

2.2 Отрицательное гипергеометрическое распределение 2

Случайная величина Y(N,m,M), имеющая отрицательное гипергеометрическое распределение 2, связана со случайной величиной X(N,m,M), имеющей отрицательное гипергеометрическое распределение 1, очевидным соотношением Y(N,m,M) = m + X(N,m,M), где m, M и N – целые неотрицательные числа, удовлетворяющие условию $m \leq M \leq N$.

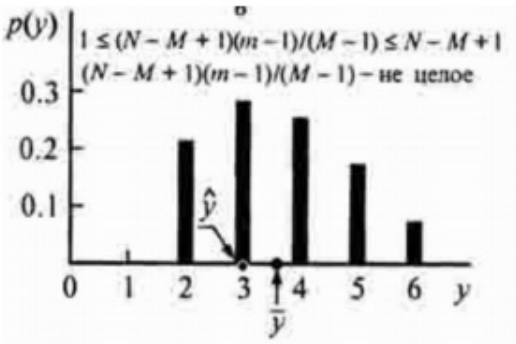

Рис. 1: График функции вероятности с параметрами N=8, M=4, m=2

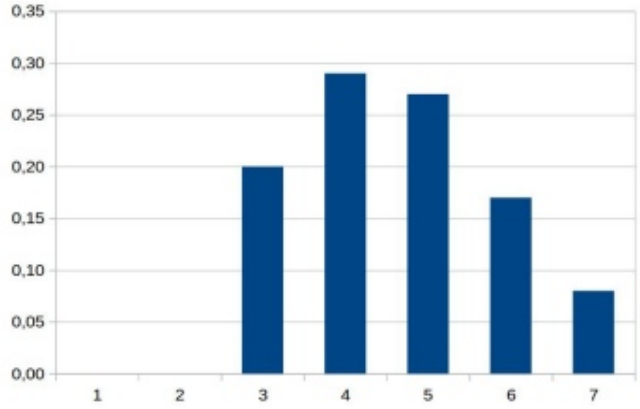

Рис. 2: График функции вероятности построенный на основе полученных чисел

Генерирование случайных чисел в соответствии с этим распределением происходит по алгоритму, представленному в блок-схеме на рис. 2:

2.3 Метод Шимбелла

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

В методе заданы свойства операций умножения и сложения:

1. Умножение:

   $a * b = b * a \Rightarrow a + b = b + a$

   $a * 0 = 0 * a = 0 \Rightarrow a + 0 = 0 + a = 0$

2. Сложение:

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

С помощью этих операций длины кратчайших и максимальных путей определяются возведением в степень x весовой матрицы Ω , где x – количество рёбер в пути. Элемент Ω^x [i][j] покажет длину экстремального пути длиной в x рёбер из вершины i в вершину j.

2.4 Алгоритм Дейкстры

Алгоритм Дейкстры является жадным алгоритмом. Алгоритм находит кратчайшие пути во взвешенном графе от одной вершины до всех остальных. Не работает, если есть ребра с отрицательными весами.

Описание работы алгоритма Дейкстры:

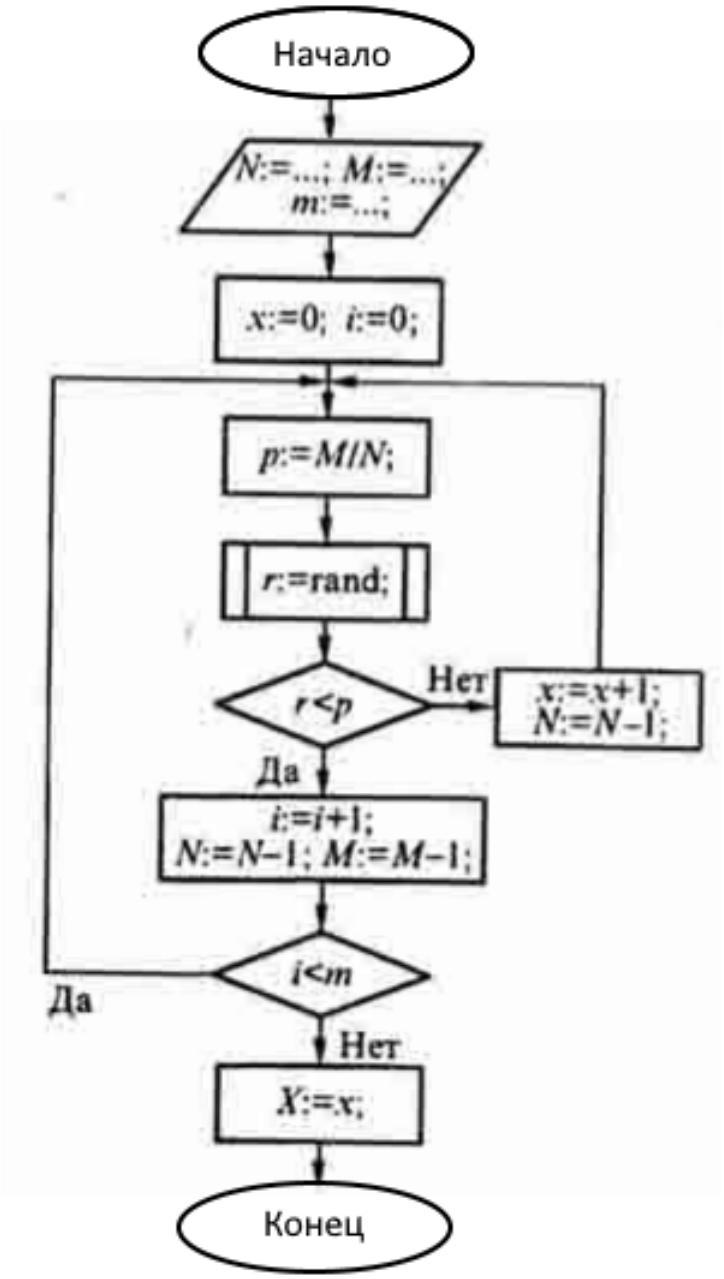

Рис. 3: Блок-схема алгоритма генерации случайных чисел

На первой итерации выбрана будет стартовая вершина s. Выбранная таким образом вершина v отмечается помеченной. Далее, на текущей итерации, из вершины v рассматриваются все рёбра (v,to) исходящие из вершины v. Для каждой такой вершины to алгоритм пытается улучшить значение d[to].

Пусть длина текущего ребра равна len, тогда d[to] = min(d[to], d[v] + len). На этом текущая итерация заканчивается, алгоритм переходит к следующей итерации (снова выбирается вершина с наименьшей величиной d и т.д.). При этом, после n итераций, все вершины графа станут помеченными, и алгоритм свою работу завершает. Найденные значения d[v] и есть искомые длины кратчайших путей из s в p.

Сложность алгоритма: O(n^2).

2.5 Алгоритм Беллмана - Форда

Алгоритм Беллмана-Форда находит кратчайшие пути во взвешенном графе, не содержащем циклы с отрицательным суммарным весом, от одной вершины до всех остальных. При этом алгоритм Беллмана-Форда позволяет определить наличие циклов отрицательного веса, достижимых из начальной вершины.

Сложность алгоритма: O(p*q), где p - количество вершин, q - количество ребер.

2.6 Алгоритм Флойда-Уоршелла

Алгоритм Флойда-Уоршелла находит кратчайшие пути между всеми парами вершин 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 $\rightarrow$ 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).

2.7 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* дуги (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 $f : E \to R$ называется *потоком* в сети G, если:

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

2.8 Алгоритм Форда-Фалкерсона

Теорема Форда - Фалкерсона:

Максимальный поток в сети равен минимальной пропускной способности разреза, то есть существует поток f^*, такой, что $w(f^*) = \max\limits_f w(f) = \min\limits_P C(P)$.

На основе данной теоремы реализуется алгоритм Форда - Фалкерсона для определения максимального потока в сети, заданной матрицей пропускных способностей дуг. Алгоритм 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 Форда-Фалекрсона: $O(q * f)$, где q - число ребер, а f - величина максимального потока.

2.9 Поиск потока минимальной стоимости

Пусть дана сеть G(V,E), S,T∈V - источник и сток. Ребра (u,v)∈E имееют пропускную способность c(u,v), поток f(u,v) и цену за единицу потока a(u,v). Тогда задачу поиска потока минимальной стоимости можно представить в виде *целевой функции:*

$$p(u,v) = \sum_{u,v\in V, f(u,v)>0} a(u,v) * f(v,u)$$

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(p^3)

2.10 Поиск минимального остова

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.

2.11 Алгоритм Краскала

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

Работа алгоритма:

1. Все ребра графа сортируются по весу (в порядке неубывания).
2. Начинается перебор всех ребер в порядке сортировки. Если ребро не создает цикл и его добавление имеет смысл (вершины, образующие ребро недостижимы друг из друга), то оно добавляется в формируемый подграф. В результате перебора полученный подграф будет минимальным остовом.

Сложность: O($q * log(q)$), где q - количество ребер.

2.12 Алгоритм Прима

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

В данном алгоритме кратчайший остов порождается в процессе разрастания одного дерева,к которому присоединяются ближайшие одиночные вершины. Каждая одиночная вершина является деревом. На выходе мы получаем множество ребер, которые лежат в минимальном остовном дереве.

Работа алгоритма:

1. Изначально остов полагается состоящим из единственной вершины (вершина выбирается случайно).

2. Затем выбирается ребро минимального веса, исходящее из этой вершины, и добавляется в минимальный остов.

3. Теперь в цикле каждый раз ищется минимальное по весу ребро, один конец которого — уже взятая в остов вершина, а другой конец — ещё не взятая, это ребро добавляется в остов (если таких ребер несколько, можно взять любое). Этот процесс повторяется до тех пор, пока остов не станет содержать все вершины. В итоге будет построен минимальный остов.

Сложность алгоритма: O(q), где q - количество ребер.

2.13 Теорема Кирхгофа

Матрица Кирхгофа - матрица размера $n \times n$, где n - количество вершин графа.

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

Свойства матрицы Кирхгофа:

1. Суммы элементов в каждой строке и каждом столбце матрице равны 0.

2. Алгебраические дополнения всех элементов матрицы равны между собой.

3. Определитель матрицы Кирхгофа равен нулю.

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

2.14 Код Прюфера

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

Алгоритма построения кода Прюфера:
Вход: дерево с n вершинами.
Выход: код Прюфера длины n – 2.
Повторить n – 2 раза:

1. Выбрать вершину v – лист дерева с наименьшим номером.

2. Добавить номер смежной вершины v в последовательность кода Прюфера.

3. Удалить вершину v из дерева.

В результате циклического выполнения этих шагов будет получен код Прюфера.

2.15 Эйлеров граф и алгоритм Флёри

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

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

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

1. G — эйлеров граф.
2. Каждая вершина G имеет чётную степень.
3. Множество рёбер G можно разбить на простые циклы.

Для проверки наличия эйлерова цикла в графе достаточно убедиться, что степени всех вершин чётны.

Работа алгоритма по нажождению эйлерова цикла:

Создается стек и туда кладется начальная вершина (любая по выбору). Далее запускается цикл, пока стек не стенет пустым:

1. Берем верхний элемент стека - v.
2. Если список смежности для этой вершины пуст, удаляем ее из стека, добавляем в эйлеров цикл
3. Иначе берем u - первую вершину из списка смежности и кладем ее в стек. Удаляем ребро (v, u).

Полученный в итоге цикл и будет эйлеровым.
Сложность: O(p), где p - количество вершин.

2.16 Гамильтонов граф и задача коммивояжера

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

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

Теорема: Если $\delta(G) \geq p/2$, то граф G является гамильтоновым.

Задача коммивояжера:
Нужно найти гамильтонов цикл минимального веса.

Решением является перебор всех гамильтоновых циклов с целью нахождения минимального по весу. Вес цикла складывается из суммы весов ребер в цикле, которые задаются целевой функцией. Она определяет для ребра [i][j] значение стоимости 'прохода' из вершины с номером i в вершину с номером j.

Задачу можно представить в виде *целевой функции:*
Пусть $1, p_1, p_2, ..., p_{n-1}, 1$ - номера городов, записанные в порядке их обхода. Иными словами p_k - номер города, посещаемого на k-ом шаге, $k = 0, 1, ..., n$. Тогда пройденное расстояние можно представить в виде целевой функции:

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

Среди чисел $p_1, p_2, ..., p_{n-1}$ по одному разу встречается каждое число из интервала $2..n$. Задача коммивояжера заключается в поиске перестановки целых чисел от 2 до n, при кототорой целевая функция минимальная.
Сложность: O($p!$), где p - количество вершин.

3 Особенности реализации

3.1 класс Graph

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

```
class Graph {
   int vertCnt;
   vector<vector<int>> adjMx;
   vector<vector<int>> posWeightsMx;
   vector<vector<int>> mixWeightsMx;
   vector<vector<int>> mPosWeightsMx;
   vector<vector<int>> mMixWeightsMx;
   vector<vector<int>> reachMx;
   vector<vector<int>> maxFlowMx;
   void GenReachMx();
   void GenMaxFlowMx();
   vector<int> PrepVertices();
   void AssignWeights();
   void MixWeights();
   void ModifyWeights(vector<vector<int>>& WeightsMx, vector<vector<int>>& modified);
   vector<vector<int>> ShimbellMult(vector<vector<int>> mxA, vector<vector<int>> mxB,
       Mode mode);
   vector<vector<int>> AddFictVert();
   bool bfs(vector<vector<int>> graphMx, int v1, int v2, vector<int>* path = nullptr,
       int* counter = nullptr);
   priority_queue<Edge, vector<Edge>,
       greater<Edge>> SortEdges(vector<vector<int>> weightsMx);
   vector<vector<int>> Unorient(vector<vector<int>> weightsMx);
   vector<int> getDegrees(vector<vector<int>> weightsMx);
public:
   Graph(int n);
   int GetVertexCount();
   vector<vector<int>> GetAdjMx();
   vector<vector<int>> GetWeightsMx(Weights type);
   vector<vector<int>> GetReachMx();
   vector<vector<int>> GetMaxFlowMx();
   vector<vector<int>> CalcShimbell(int eCnt, Mode mode);
   vector<int> Dijkstra(int vert, int& counter);
   vector<vector<int>> RestorePaths(int vert, vector<int>& dists,
       vector<vector<int>> weightMx);
   vector<int> BellmanFord(int vert, vector<vector<int>> wieghtsMx, int& counter);
   vector<vector<int>> FloydWarshall(vector<vector<int>> weightMx, int& counter);
   int fordFulkerson(int s, int t);
   int MCF(int s, int t, int flow, vector<vector<int>>& costMx, vector<vector<int>>&
       flowMx);
   vector<vector<int>> Kruskal(vector<vector<int>> weightsMx, int* counter = nullptr,
       int* sum = nullptr);
   vector<vector<int>> Prim(vector<vector<int>> weightsMx, int* counter = nullptr,
       int* sum = nullptr);
```

```
    vector<vector<int>> GenKirchhoffMx();
    int kirchhoffMinorDet();
    void PruferEncode(vector<vector<int>>& weightsMx, vector<int>& pruferCode,
        vector<int>& pruferWeights);
    vector<vector<int>> PruferDecode(vector<int>& pruferCode, vector<int>& pruferWeights);
    Type IsEuler(vector<vector<int>> weightsMx);
    vector<int> findEuler(vector<vector<int>> weightsMx, vector<vector<int>>& modWeightsMx,
        Type& eul);
    vector<int> Hamilton(vector<vector<int>> weightsMx, vector<vector<int>>& modWeightsMx,
        Type& isHamRes, int& mlen);
    void findHamiltons(ofstream& ofs, vector<vector<int>>& weightsMx,
        vector<int>& path, vector<int>& mpath, int& len, int& mlen);
};
```

3.1.1 Поля

- int vertCnt - количество вершин в графе
- vector<vector<int≫ adjMx - не взвешенная матрица смежности вершин графа.
- vector<vector<int≫ posWeightsMx - взвешенная положительная матрица смежности.
- vector<vector<int≫ mixWeightsMx - взвешенная смешанная матрица смежности.
- vector<vector<int≫ mPosWeightsMx - модифицированная взвешенная положительная матрица смежности.
- vector<vector<int≫ mMixWeightsMx - модифицированная взвешенная смешанная матрица смежности.
- vector<vector<int≫ reachMx - матрица достижимости.
- vector<vector<int≫ maxFlowMx - матрица пропускных способностей.

3.1.2 Методы

Graph(int n) - конструктор с параметром, который принимающий количество вершин. В конструкторе формируется матрица смежности вершин, весовые матрицы, матрица достижимости и матрица пропускных способностей.

```
Graph::Graph(int n) : vertCnt(n), adjMx(n, vector<int>(n, 0)),
    posWeightsMx(n, vector<int>(n, 0)),
    mixWeightsMx(), mPosWeightsMx(), mMixWeightsMx(), reachMx(), maxFlowMx() {
  if ((vertCnt >= 2) && (vertCnt <= 25)) {
    random_device rd;
    mt19937 mersenne(rd());
```

```
      vector<int> vertDegs = PrepVertices();
      vector<int> forShuffle;
      for (int i = 0; i < vertCnt - 2; i++) {
         forShuffle.clear();
         for (int j = i + 1; j < vertCnt; j++) {
            forShuffle.push_back(j);
         }
         shuffle(forShuffle.begin(), forShuffle.end(), mersenne);
         for (int j = 0; j < vertDegs[i]; j++) {
            adjMx[i][forShuffle.back()] = 1;
            forShuffle.pop_back();
         }
      }
      adjMx[vertCnt - 2][vertCnt - 1] = 1;

      AssignWeights();
      MixWeights();
      ModifyWeights(posWeightsMx, mPosWeightsMx);
      ModifyWeights(mixWeightsMx, mMixWeightsMx);
      GenReachMx();
      GenMaxFlowMx();
   }
   else {
      throw exception();
   }
}
```

PrepVertices() - метод, используемый в конструкторе при формировании матрицы смежности. возвращающий вектор чисел, сгенерированных в соответствии с распределением. Распределение реализовано при помощи функции **NHD2()**.

```
vector<int> Graph::PrepVertices() {
   vector<int> vertDegs;
   int tmpVertDeg;
   for (int i = 0; i < vertCnt - 2; i++) {
      do {
         tmpVertDeg = NHD2();
      } while (tmpVertDeg >= vertCnt);
      vertDegs.push_back(tmpVertDeg);
   }
   sort(vertDegs.begin(), vertDegs.end(), greater<int>());
   for (int i = 0; i < vertCnt - 2; i++) {
      if (vertDegs[i] > (vertCnt - 1 - i)) {
         vertDegs[i] = vertCnt - 1 - i;
      }
   }
   return vertDegs;
}
```

CalcShimbell() - метод, принимающий количество вершин графа и объект класса enum (0 - кратчайшие пути, 1 - длиннейшие), возвращающий матрицу Шимбелла. Является оберткой для вызова метода **ShimbellMult()** число раз,

равное количеству вершин.

```
   vector<vector<int>> Graph::CalcShimbell(int eCnt, Mode mode) {
   vector<vector<int>> resMx = posWeightsMx;
   for (int i = 0; i < eCnt - 1; i++) {
      resMx = ShimbellMult(resMx, posWeightsMx, mode);
   }
   return resMx;
}
```

ShimbellMult() - метод, реализующий операцию умножения матриц по свойствам, определенным методом Шимбелла, принимающий две матрицы и пользовательскую переменную, определяющую режим работы алгоритма, а именно то, кратчайшие или длиннейшие пути требуется найти, и возвращающий матрицу.

```
vector<vector<int>> Graph::ShimbellMult(vector<vector<int>> mxA, vector<vector<int>> mxB,
    Mode mode) {
   vector<vector<int>> resMx(vertCnt, vector<int>(vertCnt, 0));
   vector<int> buf;
   bool isNotZero;
   for (int i = 0; i < vertCnt; i++) {
      for (int j = 0; j < vertCnt; j++) {
         buf.clear();
         isNotZero = false;
         for (int k = 0; k < vertCnt; k++) {
            if ((mxA[i][k] != 0) && (mxB[k][j] != 0)) {
               buf.push_back(mxA[i][k] + mxB[k][j]);
               isNotZero = true;
            }
         }
         if (isNotZero) {
            if (mode == Mode::Short) {
               resMx[i][j] = *min_element(buf.begin(), buf.end());
            }
            else {
               resMx[i][j] = *max_element(buf.begin(), buf.end());
            }
         }
         else {
            resMx[i][j] = 0;
         }
      }
   }
   return resMx;
}
```

Dijkstra() - метод поиска кратчайшего маршрут из точки по алгоритму Дейкстры, принимающий номер вершины-начала и счетчик итераций, возвращающий вектор, содержащий расстояния найденного кратчайшего пути.

```
vector<int> MyGraph::Dijkstra(int vertNum, int& counter) {
   vector<int> dists(mPosWeightsMx[vertNum]);
```

```
   dists[vertNum] = 0;
   vector<bool> isVisitedVector(vertCnt, false);
   isVisitedVector[vertNum] = true;
   int min, curVert = vertCnt - 1;
   for (int i = 0; i < vertCnt; i++) {
      min = INF;
      for (int j = 0; j < vertCnt; j++, counter++) {
         if (!isVisitedVector[j] && (dists[j] < min)) {
            min = dists[j];
            curVert = j;
         }
      }
      isVisitedVector[curVert] = true;
      for (int j = 0; j < vertCnt; j++, counter++) {
         if (!isVisitedVector[j] && (mPosWeightsMx[curVert][j] != INF)
            && (dists[curVert] != INF) &&
            ((dists[curVert] + mPosWeightsMx[curVert][j]) < dists[j]))
         {
            dists[j] = dists[curVert] + mPosWeightsMx[curVert][j];
         }
      }
   }
   return dists;
}
```

BellmanFord() - метод поиска кратчайшего маршрута из точки по алгоритму Беллмана-Форда, принимающий номер вершины-начала, выбранную матрицу весов, и счетчик итераций, возвращающий вектор, содержащий расстояния найденного кратчайшего пути.

```
vector<int> Graph::BellmanFord(int vert, vector<vector<int>> weightMx, int& counter) {
   counter = 0;
   vector<int> dists(vertCnt, INF);
   dists[vert] = 0;
   int curVert, newDist;
   deque<int> dq;
   dq.push_back(vert);
   while (!dq.empty()) {
      curVert = dq.front();
      dq.pop_front();
      for (int i = curVert + 1; i < vertCnt; i++, counter++) {
         if (weightMx[curVert][i] != INF && (weightMx[curVert][i] != 0)) {
            newDist = dists[curVert] + weightMx[curVert][i];
            if (newDist < dists[i]) {
               dists[i] = newDist;
               if (find(dq.begin(), dq.end(), i) == dq.end()) dq.push_back(i);
               else {
                  remove(dq.begin(), dq.end(), i);
                  dq.push_front(i);
               }
            }
         }
```

```
      }
   }
   return dists;
}
```

RestorePaths() - метод восстановления путей, принимающий стартовую вершину, вектор расстояний, полученный при выполнении алгоритма Дейкстры/ Беллмана-Форда, и матрицу весов, возвращающий матрицу, содержащую кратчайшие пути.

```
vector<vector<int>> Graph::RestorePaths(int vert, vector<int>& dists, vector<vector<int>>
    weightMx) {
   vector<vector<int>> paths(vertCnt, vector<int>());
   int tmp, curVert;
   for (int i = 0; i < vertCnt; i++) {
      if (dists[i]) {
         if (dists[i] != INF) {
            curVert = i;
            paths[i].push_back(curVert);
            while (curVert != vert) {
               for (int j = 0; j < vertCnt; j++) {
                  if (weightMx[j][curVert] != INF) {
                     if ((dists[curVert] - weightMx[j][curVert]) == dists[j]) {
                        curVert = j;
                        paths[i].push_back(j);
                        break;
                     }
                  }
               }
            }
         }
         else paths[i].push_back(INF);
      }
      else paths[i].push_back(INF);
   }
   return paths;
}
```

FloydWarshall() - метод поиска кратчайших маршрутов по алгоритму Флойда-Уоршелла, принимающий выбранную матрицу весов и счетчик итераций, возвращающий матрицу расстояний между парами вершин.

```
vector<vector<int>> Graph::FloydWarshall(vector<vector<int>> weightMx, int& counter) {
   vector<vector<int>> distsMx = weightMx;
   for (int i = 0; i < vertCnt; i++) {
      for (int j = 0; j < vertCnt; j++) {
         for (int k = 0; k < vertCnt; k++, counter++) {
            if (distsMx[i][k] != INF && distsMx[k][j] != INF) {
               distsMx[i][j] = min(distsMx[i][j], (distsMx[i][k] + distsMx[k][j]));
            }
         }
      }
```

```
    }
    for (int i = 0; i < vertCnt; i++) distsMx[i][i] = 0;
    return distsMx;
}
```

fordFulkerson() - метод, реализующий алгоритм Форда-Фалкерсона, принимающий номера начальной и конечной вершин, возвращающий значение максимального потока.

```
int Graph::fordFulkerson(int s, int t) {
    int temp = t;
    vector<vector<int>> rg = AddFictVert();
    if (rg.size() != vertCnt) {
        temp++;
       s++;
    }
    vector<int> path(rg.size(), 0);
    int maxFlow = 0;
    int curFlow;
    while (bfs(rg, s, temp, path)) {
       curFlow = INF;
       for (int i = temp; i != s; i = path[i]) {
          curFlow = min(curFlow, rg[path[i]][i]);
       }
       for (int i = temp; i != s; i = path[i]) {
           rg[i][path[i]] += curFlow;
          rg[path[i]][i] -= curFlow;
       }
       maxFlow += curFlow;
    }
    return maxFlow;
}
```

MCF() - метод вычисления потока минимальной стоимости, принимающий начальную и конечную вершины, значение используемого потока (2/3 от максимального), ссылки на матрицы стоимости и потока, используемый для хранения значений, вычесленных в описываемом методе: пути, по которым проходит поток, величины потоков, их стоимости за единицу, матрица потоков и матрица стоимостей за единицу потока. Возвращает поток минимальной стоимости.

```
int Graph::MinCostFlow(int s, int t, int flow, vector<vector<int>>& costMx,
    vector<vector<int>>& flowMx) {
    int cost;
    int mcf = 0;
    int neck;
    vector<int> path;
    vector<pair<int, int>> rEdges;
    int counter;
    while (flow) {
       cost = 0;
       neck = INF;
```

```
        rEdges.clear();
        path = RestorePaths(s, BellmanFord(s, mcfVals.costMx, counter), mcfVals.costMx)[t];
        mcfVals.paths.push_back(path);
        for (int i = path.size() - 1; i > 0; i--) neck =
            min(neck, mcfVals.flowMx[path[i]][path[i - 1]]);
        neck = min(flow, neck);
        mcfVals.flows.push_back(neck);
        for (int i = path.size() - 1; i > 0; i--) {
           mcfVals.flowMx[path[i]][path[i - 1]] -= neck;
           cost += mcfVals.costMx[path[i]][path[i - 1]];
           if (mcfVals.flowMx[path[i]][path[i - 1]] == 0) {
              rEdges.push_back(make_pair(path[i], path[i - 1]));
           }
        }
        mcfVals.costsPerPath.push_back(cost);
        mcf += neck * cost;
        for (auto it = rEdges.begin(); it != rEdges.end(); ++it)
            mcfVals.costMx[it->first][it->second] = INF;
        flow -= neck;
    }
    return mcf;
}
```

Kruskal() - метод, реализующий алгоритм Краскала, принимающий матрицу весов, счетчик итераций и переменную хранения веса кратчайшего остова, возвращающий матрицу весов кратчайшего остова.

```
vector<vector<int>> Graph::Kruskal(vector<vector<int>> weightsMx, int* counter, int* sum)
    {
    vector<vector<int>> spanTree = vector<vector<int>>(vertCnt, vector<int>(vertCnt, 0));
    priority_queue<Edge, vector<Edge>, greater<Edge>> pq = SortEdges(weightsMx);
    Edge e;
    if (counter) {
       (*counter) = 0;
    }
    while (!pq.empty()) {
        if (counter) (*counter)++;
       e = pq.top();
       pq.pop();
       if (!bfs(spanTree, e.vert1, e.vert2, nullptr, nullptr)) {
          spanTree[e.vert1][e.vert2] = spanTree[e.vert2][e.vert1] =
              weightsMx[e.vert1][e.vert2];
       }
    }
    if (sum) {
       (*sum) = 0;
       for (int i = 0; i < vertCnt; i++) {
          for (int j = i + 1; j < vertCnt; j++) {
             if (spanTree[i][j] != 0) {
                (*sum) += spanTree[i][j];
             }
          }
```

```
      }
   }
   return spanTree;
}
```

Prim() - метод, реализующий алгоритм Прима, принимающий матрицу весов, счетчик итераций и переменную хранения веса кратчайшего остова, возвращающий матрицу весов кратчайшего остова.

```
vector<vector<int>> Graph::Prim(vector<vector<int>> weightsMx, int* counter, int* sum) {
   vector<vector<int>> spanTree = vector<vector<int>>(vertCnt, vector<int>(vertCnt, 0));
   vector<int> path(vertCnt, 0);
   path[0] = -1;
   vector<int> keys(vertCnt, INF);
   keys[0] = 0;
   vector<bool> flags(vertCnt, false);
   int k, ind;
   if (counter) {
      (*counter) = 0;
   }
   for (int i = 0; i < vertCnt; i++) {
      for (int j = 0; j < i; j++) {
         weightsMx[i][j] = weightsMx[j][i];
      }
   }
   for (int i = 0; i < vertCnt - 1; i++) {
      k = INF;
      for (int j = 0; j < vertCnt; j++) {
         if ((flags[j] == false) && (keys[j] < k)) {
            ind = j;
            k = keys[j];
         }
         if (counter) {
            (*counter)++;
         }
      }
      flags[ind] = true;
      for (int j = 0; j < vertCnt; j++) {
         if ((weightsMx[ind][j] != 0) && (flags[j] == false) && (weightsMx[ind][j] <
             keys[j])) {
            keys[j] = weightsMx[ind][j];
            path[j] = ind;
         }
         if (counter) {
            (*counter)++;
         }
      }
   }
   for (int i = 1; i < vertCnt; i++) {
      spanTree[i][path[i]] = spanTree[path[i]][i] = weightsMx[i][path[i]];
   }
   if (sum) {
```

```
      (*sum) = 0;
      for (int i = 0; i < vertCnt; i++) {
         for (int j = i + 1; j < vertCnt; j++) {
            if (spanTree[i][j] != 0) (*sum) += spanTree[i][j];
         }
      }
   }
   return spanTree;
}
```

GenKirchhoffMx() - метод генерации матрицы Кирхгофа, принимающий матрицу смежности вершин, возвращающий матрицу Кирхгофа.

```
vector<vector<int>> Graph::GenKirchhoffMx() {
   vector<vector<int>> kirchhoffMx = vector<vector<int>>(vertCnt, vector<int>(vertCnt,
       0));
   int vertDeg;
   vector<vector<int>> modAdjMx = adjMx;
   for (int i = 0; i < vertCnt; i++) {
      for (int j = 0; j < i; j++) {
         modAdjMx[i][j] = adjMx[j][i];
      }
   }
   for (int i = 0; i < vertCnt; i++) {
      vertDeg = 0;
      for (int j = 0; j < vertCnt; j++) {
         if (modAdjMx[i][j]) {
            kirchhoffMx[i][j] = -1;
            vertDeg++;
         }
      }
      kirchhoffMx[i][i] = vertDeg;
   }
   return kirchhoffMx;
}
```

kirchhoffMinorDet() - метод поиска числа остовных подграфов согласно теореме Кирхгофа, ничего не принимающий и возвращающий Алгоритм генерирует матрицу Кирхгофа, находит минор и вычисляет его определитель методом **mxDet()**, что является числом остовных деревьев в графе.

```
int Graph::kirchhoffMinorDet() {
   vector<vector<int>> kirchhoff = GenKirchhoffMx();
   vector<vector<int>> kirchhoffMinor = vector<vector<int>>(vertCnt - 1,
       vector<int>(vertCnt - 1, 0));
   for (int i = 0; i < vertCnt - 1; i++) {
      for (int j = 0; j < vertCnt - 1; j++) {
         kirchhoffMinor[i][j] = kirchhoff[i][j];
      }
   }
   return mxDet(kirchhoffMinor);
}
```

PruferEncode() - метод, генирирующий код Прюфера для графа, принимающий матрицу весов кратчайшего остова, переменные для хранения кода Прюфера и веса ребер, возвращающий код Прюфера. Сначала происходит проверка, сгенерирован ли минимальный остов. Является рекурсивным.

```
void Graph::PruferEncode(vector<vector<int>>& weightsMx, vector<int>& pruferCode,
    vector<int>& pruferWeights) {
   if (pruferCode.size() == weightsMx.size() - 2) {
      for (int i = 0; i < weightsMx.size(); i++) {
         for (int j = 0; j < weightsMx.size(); j++) {
            if (weightsMx[i][j] != 0) {
               pruferCode.push_back(i + 1);
               pruferWeights.push_back(weightsMx[i][j]);
               return;
            }
         }
      }
   }
   Edge e;
   vector<vector<int>> tmpWeightsMx = weightsMx;
   int cnt;
   for (int i = 0; i < weightsMx.size(); i++) {
      cnt = 0;
      for (int j = 0; j < weightsMx.size(); j++) {
         if (weightsMx[i][j] != 0) {
            if (cnt > 1) {
               break;
            }
            else {
               e.vert2 = j;
               cnt++;
            }
         }
      }
      if (cnt == 1) {
         e.vert1 = i;
         e.weight = weightsMx[e.vert1][e.vert2];
         break;
      }
   }
   pruferCode.push_back(e.vert2 + 1);
   pruferWeights.push_back(e.weight);
   tmpWeightsMx[e.vert1][e.vert2] = tmpWeightsMx[e.vert2][e.vert1] = 0;
   PruferEncode(tmpWeightsMx, pruferCode, pruferWeights);
}
```

PruferDecode() - метод декодирования кода Прюфера, принимающий код Прюфера и веса ребер и возвращающий декодированную матрицу весов кратчайшего остова.

```
vector<vector<int>> Graph::PruferDecode(vector<int>& pruferCode, vector<int>&
    pruferWeights) {
```

```
    vector<vector<int>> weightMx = vector<vector<int>>(pruferCode.size() + 1,
        vector<int>(pruferCode.size() + 1, 0));
    vector<bool> isUsed(weightMx.size(), false);
    queue<int> qPruferCode, qPruferWeights;
    for (int i = 0; i < pruferCode.size(); i++) {
        qPruferCode.push(pruferCode[i] - 1);
        qPruferWeights.push(pruferWeights[i]);
    }
    int tmpVert, tmpWeight;
    while (!qPruferCode.empty()) {
        for (int i = 0; i < weightMx.size(); i++) {
            if (qPruferCode._Get_container().end() ==
                find(qPruferCode._Get_container().begin(),
                qPruferCode._Get_container().end(), i)) {
                if (isUsed[i] == false) {
                    isUsed[i] = true;
                    tmpVert = qPruferCode.front(); tmpWeight = qPruferWeights.front();
                    weightMx[i][tmpVert] = weightMx[tmpVert][i] = tmpWeight;
                    qPruferCode.pop(); qPruferWeights.pop();
                    break;
                }
            }
        }
    }
    return weightMx;
}
```

findEuler() - метод поиска эйлерова цикла в графе, принимающий матрицу эйлерова графа и переменные для хранения результирующей матрицы и пользовательского типа Type, возвращающий эйлеров цикл. По необходимости преобразует граф в эйлеров.

```
vector<int> Graph::findEuler(vector<vector<int>> weightsMx, vector<vector<int>>&
    modWeightsMx, Type& eul) {
    eul = IsEuler(weightsMx);
    if (eul == Type::twoVert) {
        return vector<int>();
    }
    modWeightsMx = Unorient(weightsMx);
    vector<int> vertDeg = getDegrees(modWeightsMx);
    if (eul == Type::Mod2) {
        bool isEuler = false, isChanged = false;
        int con = -1;
        random_device rd;
        mt19937 mersenne(rd());
        while (!isEuler) {
            isChanged = false;
            for (int i = 0; i < modWeightsMx.size(); i++) {
                if ((vertDeg[i] % 2) == 1) {
                    con = -1;
                    for (int j = 0; j < modWeightsMx.size(); j++) {
                        if ((modWeightsMx[i][j] == 0) && (i != j)) {
```

```
                if (con == -1) {
                    if (modWeightsMx.size() % 2 == 0) {
                        if (vertDeg[j] != modWeightsMx.size() - 1) {
                            con = j;
                        }
                    }
                    else {
                        con = j;
                    }
                }
                if ((vertDeg[j] % 2) == 1) {
                    isChanged = true;
                    vertDeg[i]++;
                    vertDeg[j]++;
                    modWeightsMx[i][j] = modWeightsMx[j][i] = mersenne() % 15;
                    break;
                }
            }
        }
        if (!isChanged && (con != -1)) {
            isChanged = true;
            vertDeg[i]++;
            vertDeg[con]++;
            modWeightsMx[i][con] = modWeightsMx[con][i] = mersenne() % 15;
        }
        if (con == -1) {
            eul = Type::Mod1;
        }
      }
    }
    if (!isChanged) {
        isEuler = true;
    }
  }
}
if (eul == Type::Mod1) {
  int vertToDisconnect = -1;
  for (int i = 0; i < modWeightsMx.size(); i++) {
    if (vertDeg[i] % 2 == 1) {
        vertToDisconnect = i;
        break;
    }
  }
  for (int i = vertToDisconnect + 1; i < modWeightsMx.size(); i++) {
    if ((vertDeg[i] % 2 == 1) && (modWeightsMx[vertToDisconnect][i])) {
        vertDeg[vertToDisconnect]--;
        vertDeg[i]--;
        modWeightsMx[vertToDisconnect][i] = modWeightsMx[i][vertToDisconnect] = 0;
    }
  }
}
vector<int> eulerPath;
int curVert;
```

```
    stack<int> vertices;
    vertices.push(0);
    vector<vector<int>> weightsForDec = modWeightsMx;
    while (!vertices.empty()) {
        curVert = vertices.top();
        if (vertDeg[curVert] == 0) {
            vertices.pop();
            eulerPath.push_back(curVert);
        }
        else {
            for (int i = 0; i < vertCnt; i++) {
                if (weightsForDec[curVert][i] != 0) {
                    vertices.push(i);
                    vertDeg[i]--;
                    vertDeg[curVert]--;
                    weightsForDec[curVert][i] = 0;
                    weightsForDec[i][curVert] = 0;
                    break;
                }
            }
        }
    }
    return eulerPath;
}
```

---

findHamiltons() - метод поиска гамильтонова цикла в графе, принимающий ссылку на поток файлового вывода, матрицу весов, переменные для хранения гамильтонова цикла, минимального гамильтонова цикла, размеров минимального пути и минимального пути для сравнения, ничего не возвращающий. Является рекурсивным.

```
void Graph::findHamiltons(ofstream& ofs, vector<vector<int>>& weightsMx, vector<int>&
    path, vector<int>& mpath, int& len, int& mlen) {
    if (path.size() == vertCnt) {
        if (weightsMx[path[path.size() - 1]][0] != 0) {
            len += weightsMx[path[path.size() - 1]][0];
            path.push_back(0);
            if (len < mlen) {
                mlen = len;
                mpath = path;
            }
            for (int i = 0; i < path.size() - 1; i++) ofs << path[i] + 1 << " - ";
            ofs << path[path.size() - 1] + 1 << '\t';
            ofs << "Вес: " << len << '\n';
            path.pop_back();
        }
        return;
    }
    for (int i = 0; i < weightsMx.size(); i++) {
        if (weightsMx[path[path.size() - 1]][i] != 0) {
            if (path.end() == find(path.begin(), path.end(), i)) {
                int tlen = len + weightsMx[path[path.size() - 1]][i];
```

```
            path.push_back(i);
            findHamiltons(ofs, weightsMx, path, mpath, tlen, mlen);
            path.pop_back();
         }
      }
   }
}
```

Hamilton() - метод поиска гамильтоновых циклов в графе, принимающий весовую матрицу, переменные для хранения результирующей матрицы, пользовательского типа Type и размера минимального пути. Все гамильтоновы циклы записываются в файл. По необходимости преобразует граф в гамильтонов.

```
vector<int> Graph::Hamilton(vector<vector<int>> weightsMx, vector<vector<int>>&
    modWeightsMx, Type& isHamRes, int& mlen) {
   if (weightsMx.size() == 2) {
      isHamRes = Type::twoVert;
      return vector<int>();
   }
   modWeightsMx = Unorient(weightsMx);
   vector<int> vertDeg;
   bool isHamilton = true, isChanged = false;
   isHamRes = Type::True;
   int con = -1;
   random_device rd;
   mt19937 mersenne(rd());
   if (modWeightsMx.size() == 3) {
      for (int i = 0; i < weightsMx.size(); i++) {
         for (int j = 0; j < weightsMx.size(); j++) {
            if ((i != j) && (modWeightsMx[i][j] == 0)) {
               modWeightsMx[i][j] = modWeightsMx[j][i] = mersenne() % 15;
            }
         }
      }
   }
   else {
      vertDeg = getDegrees(modWeightsMx);
      for (int i = 0; i < weightsMx.size(); i++) {
         if (vertDeg[i] < (vertCnt / 2)) {
            isHamilton = false;
            isHamRes = Type::Mod2;
            break;
         }
      }
      while (!isHamilton) {
         isChanged = false;
         for (int i = 0; i < weightsMx.size(); i++) {
            con = -1;
            if (vertDeg[i] < (vertCnt / 2)) {
               for (int j = 0; j < weightsMx.size(); j++) {
                  if ((modWeightsMx[i][j] == 0) && (i != j)) {
                     con = j;
```

```
                    if (vertDeg[j] < (vertCnt / 2)) {
                        isChanged = true;
                        vertDeg[i]++;
                        vertDeg[j]++;
                        modWeightsMx[i][j] = modWeightsMx[j][i] = mersenne() % 15;
                        break;
                    }
                }
            }
            if (!isChanged && (con != -1)) {
                isChanged = true;
                vertDeg[i]++;
                modWeightsMx[i][con] = modWeightsMx[con][i] = mersenne() % 15;
            }
        }
    }
    if (!isChanged) {
        isHamilton = true;
    }
  }
 }
 ofstream ofs("hm.txt");
 vector<int> path;
 path.push_back(0);
 vector<int> mpath;
 int len = 0;
 mlen = INT_MAX;
 findHamiltons(ofs, modWeightsMx, path, mpath, len, mlen);
 ofs.close();
 return mpath;
}
```

4 Результаты работы программы

В начале работы программа запрашивает число от 2 до 25, использующееся как количество вершин графа, затем выводится сгенерированные на основе ввода матрицы смежности вершин, весов, выводится меню и предлагается выбрать пункт из него, что показано на рис. 4:

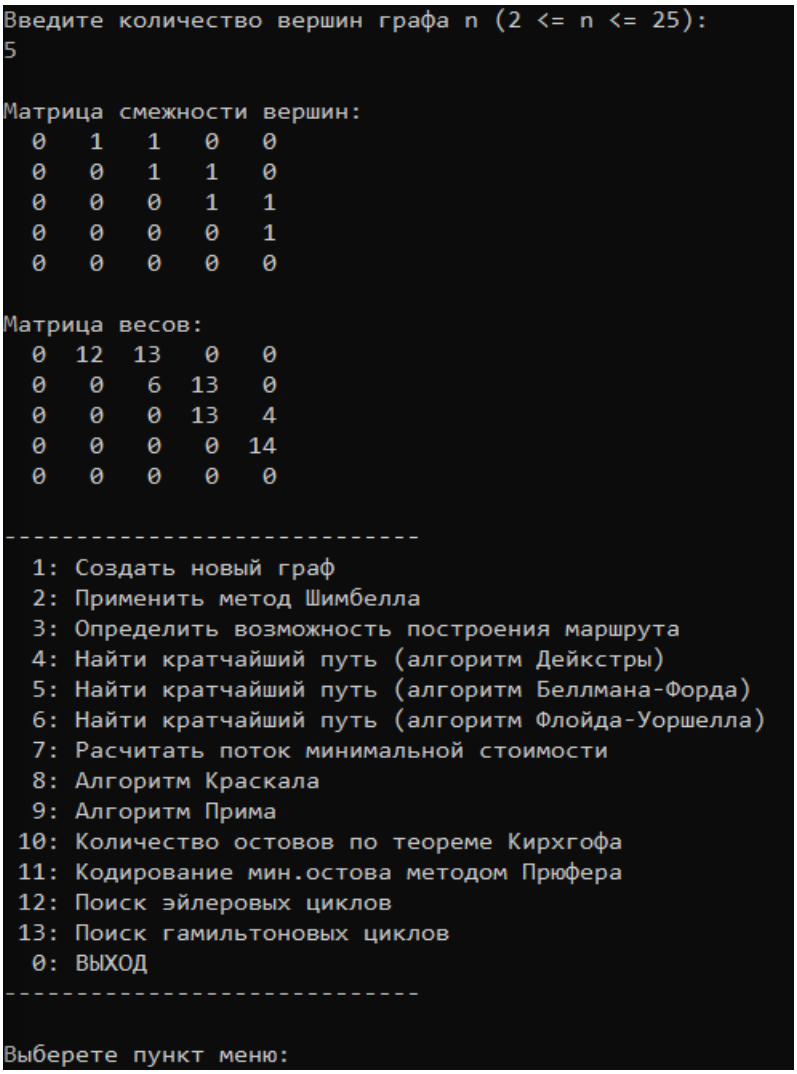

Рис. 4: Начало работы программы

В случае некорректного ввода предагается выбрать пункт повторно, как показано на рис. 5:

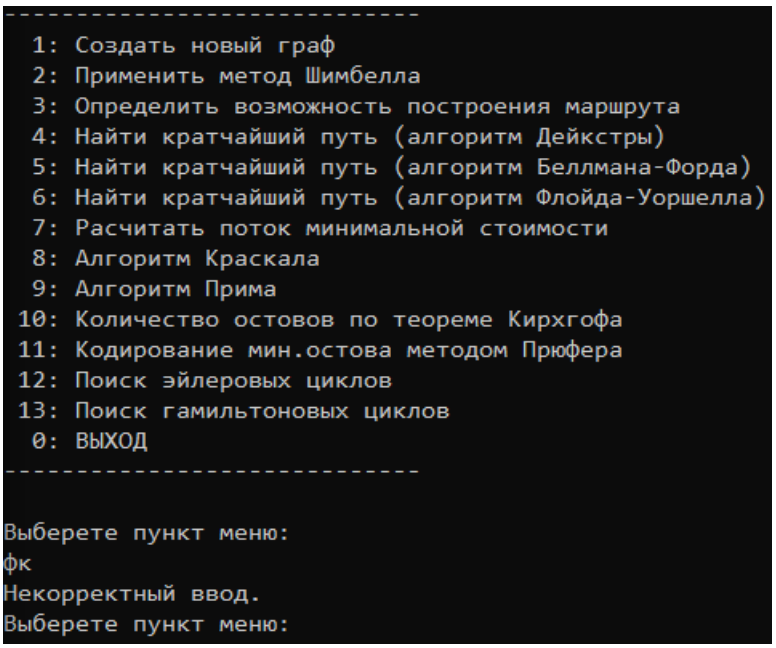

Рис. 5: Некорректный ввод

В качестве пунктов меню можно выбрать следующее:

- Создать новый граф (рис. 6).
- Применить метод Шимбелла (рис. 7).
- Определить возможность построения маршрута между двумя вершинами и найти их количество (рис. 8).
- Применить алгоритм Дейкстры (рис. 9).
- Применить алгоритм Беллмана-Форда (рис. 10).
- Применить алгоритм Флойда-Уоршелла (рис. 11).
- Расчитать поток минимальной стоимости (рис. 12).
- Применить алгоритм Краскала (рис. 13).
- Применить алгоритм Прима (рис. 14).
- Посчитать количество остовных деревьев в графе по теореме Кирхгофа (рис. 15).
- Закодировать минимальный остов методом Прюфера с последующим декодированием для проверки (рис. 16).
- Найти эйлеров цикл в графе (рис. 17).

- Найти гамильтоновы циклы в графе с последующим выводом минимального в консоль и всех в файл (рис. 18).

```
Выберете пункт меню:
1

Введите количество вершин графа n (2 <= n <= 25):
6

Матрица смежности вершин:
  0   0   1   1   0   1
  0   0   0   1   1   1
  0   0   0   0   1   1
  0   0   0   0   1   1
  0   0   0   0   0   1
  0   0   0   0   0   0

Матрица весов:
  0   0  11   5   0  12
  0   0   0   6   6  13
  0   0   0   0  11   3
  0   0   0   0   8   3
  0   0   0   0   0   2
  0   0   0   0   0   0
```

Рис. 6: Создание нового графа

```
Выберете пункт меню:
2

Введите количество ребер:
3

Выберете требование к матрице:
0: Матрица кратчайших маршрутов
1: Матрица длиннейших маршрутов
0

Матрица весов:
  0   0  11   5   0  12
  0   0   0   6   6  13
  0   0   0   0  11   3
  0   0   0   0   8   3
  0   0   0   0   0   2
  0   0   0   0   0   0

Матрица Шимбелла:
  0   0   0   0   0  15
  0   0   0   0   0  16
  0   0   0   0   0   0
  0   0   0   0   0   0
  0   0   0   0   0   0
  0   0   0   0   0   0
```

Рис. 7: Метод Шимбелла

```
Выберете пункт меню:
3

Введите номер первой вершины:
2

Введите номер второй вершины:
5

Матрица достижимости:
  1   0   0   0   1   2
  0   1   0   0   1   2
  0   0   1   0   0   1
  0   0   0   1   0   1
  0   0   0   0   1   0
  0   0   0   0   0   1

Существует 1 маршрут(-а/-ов) между ними.
```

Рис. 8: Поиск количества путей между двумя вершинами

```
Выберете пункт меню:
4

Введите исходную вершину:
2

Матрица весов:
  0   0  11   5   0  12
  0   0   0   6   6  13
  0   0   0   0  11   3
  0   0   0   0   8   3
  0   0   0   0   0   2
  0   0   0   0   0   0

До вершины  1 пути нет.
До вершины  3 пути нет.
Кратчайший путь длиной  6 до вершины  4:  2- 4
Кратчайший путь длиной  6 до вершины  5:  2- 5
Кратчайший путь длиной  8 до вершины  6:  2- 5- 6

Количество итераций: 72
```

Рис. 9: Алгоритм Дейкстры

```
Выберете пункт меню:
5

Введите исходную вершину:
3

Какую матрицу весов использовать?
1: Положительную
2: Отрицательную
2
Матрица весов:
  0   0  11   5   0  12
  0   0   0  -6  -6  13
  0   0   0   0 -11   3
  0   0   0   0  -8   3
  0   0   0   0   0   2
  0   0   0   0   0   0

До вершины  1 пути нет.
До вершины  2 пути нет.
До вершины  4 пути нет.
Кратчайший путь длиной -11 до вершины  5:  3- 5
Кратчайший путь длиной  -9 до вершины  6:  3- 5- 6

Количество итераций: 4
```

Рис. 10: Алгоритм Беллмана-Форда

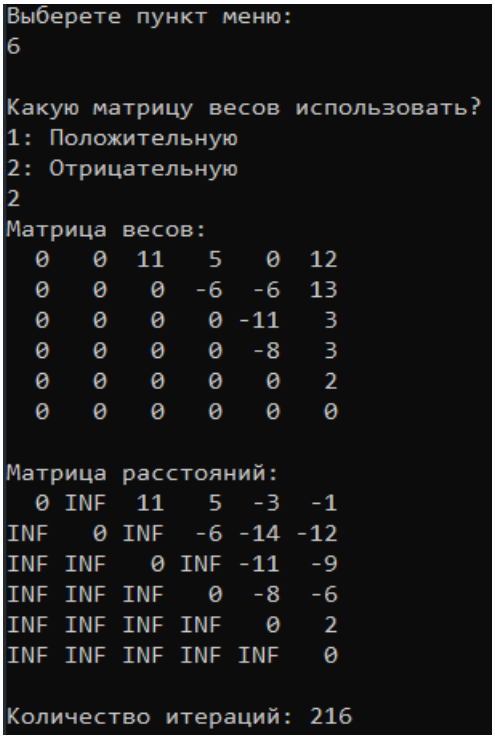

Рис. 11: Алгоритм Флойда-Уоршалла

```
Выберете пункт меню:
7

Введите номер первой вершины:
1

Введите номер второй вершины:
6

Матрица пропускных способностей:
  0   6   0   8  12  16
  0   0   8   0  19   5
  0   0   0   9   4   8
  0   0   0   0   2  18
  0   0   0   0   0   8
  0   0   0   0   0   0

Максимальный поток: 38
Используемый поток: 25

Матрица стоимостей:
999   2 999   4  10   9
999 999   6 999  10   9
999 999 999  12   7   5
999 999 999 999  14  11
999 999 999 999 999  10
999 999 999 999 999 999

Матрица потоков:
  0   0   0   5  12   0
  0   0   7   0  19   0
  0   0   0   9   4   7
  0   0   0   0   2  15
  0   0   0   0   0   8
  0   0   0   0   0   0

Матрица стоимостей за единицу потока:
999 999 999   4  10 999
999 999   6 999  10 999
999 999 999  12   7   5
999 999 999 999  14  11
999 999 999 999 999  10
999 999 999 999 999 999

Поток величины 16 со стоимостью 9 за единицу потока
по пути: 1 -> 6
Итоговая стоимость: 144

Поток величины 5 со стоимостью 11 за единицу потока
по пути: 1 -> 2 -> 6
Итоговая стоимость: 55

Поток величины 1 со стоимостью 13 за единицу потока
по пути: 1 -> 2 -> 3 -> 6
Итоговая стоимость: 13

Поток величины 3 со стоимостью 15 за единицу потока
по пути: 1 -> 4 -> 6
Итоговая стоимость: 45

Величина потока минимальной стоимости: 257
```

Рис. 12: Вычисление потока минимальной стоимости

```
Выберете пункт меню:
8

Матрица весов кратчайшего остова:
  0   0   0   5   0   0
  0   0   0   0   6   0
  0   0   0   0   0   3
  5   0   0   0   0   3
  0   6   0   0   0   2
  0   0   3   3   2   0

Вес кратчайшего остова: 19

Количество итераций: 11
```

Рис. 13: Алгоритм Краскала

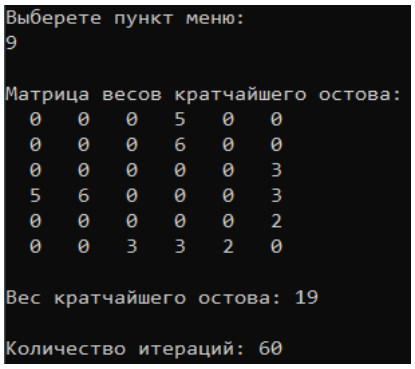

Рис. 14: Алгоритм Прима

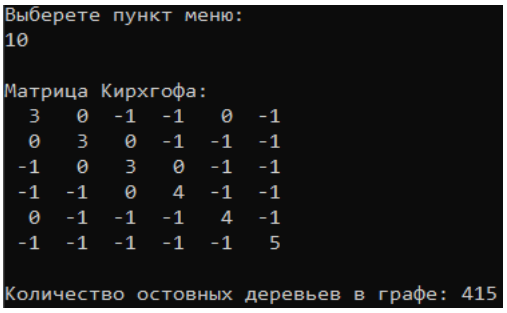

Рис. 15: Нахождение количества остовных деревьев

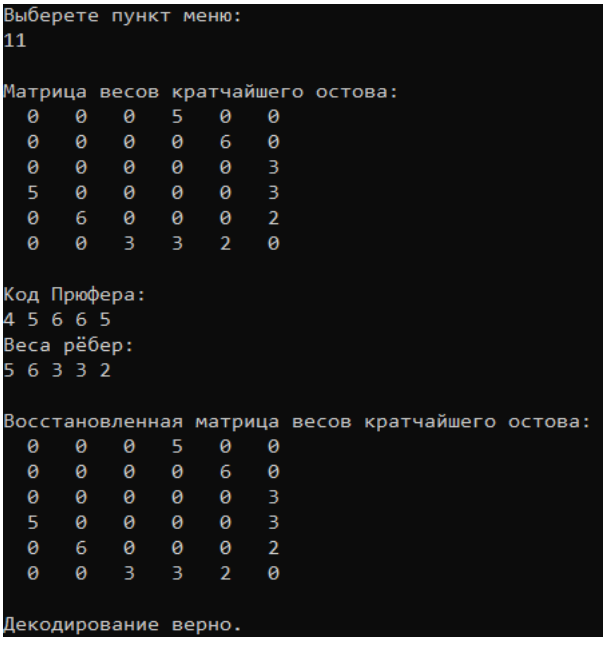

Рис. 16: Код Прюфера

```
Выберете пункт меню:
12

Граф не  является эйлеровым.
Исходный граф:
  0   2   0   4  10   9
  0   0   6   0  10   9
  0   0   0  12   7   5
  0   0   0   0  14  11
  0   0   0   0   0  10
  0   0   0   0   0   0
Весовая матрица эйлерова графа:
  0   2   0   4  10   9
  2   0   6   0  10   9
  0   6   0  12   7   5
  4   0  12   0  14  11
 10  10   7  14   0   0
  9   9   5  11   0   0

Эйлеров цикл:
1 - 6 - 4 - 5 - 3 - 6 - 2 - 5 - 1 - 4 - 3 - 2 - 1
```

Рис. 17: Поиск эйлерового цикла

```
Выберете пункт меню:
13

Граф не  является гамильтоновым.
Исходный граф:
  0   0   3   0   2   3   4   0   0
  0   0   0   0   0   6   8  11  11
  0   0   0   0   6   8   0   6   0
  0   0   0   0   0   8   0   0  14
  0   0   0   0   0   0   4   0   3
  0   0   0   0   0   0   2   7   0
  0   0   0   0   0   0   0   2   2
  0   0   0   0   0   0   0   0   1
  0   0   0   0   0   0   0   0   0
Весовая матрица гамильтонова графа:
  0   0   3   0   2   3   4   0   0
  0   0   0   0   0   6   8  11  11
  3   0   0   0   6   8   0   6   0
  0   0   0   0   0   8   9  10  14
  2   0   6   0   0   0   4   0   3
  3   6   8   8   0   0   2   7   0
  4   8   0   9   4   2   0   2   2
  0  11   6  10   0   7   2   0   1
  0  11   0  14   3   0   2   1   0

Минимальный гамильтонов цикл:
1 - 3 - 8 - 4 - 6 - 2 - 7 - 9 - 5 - 1
Вес пути: 48
```

Рис. 18: Поиск гамильтоновых циклов

Заключение

В результате работы на языке программирования C++ были реализованы: алгоритм создания связного ациклического графа, алгоритм отрицательного гипергеометрического распределения 2, метод Шимбелла, алгоритмы Дейкстры, Беллмана-Форда, Флойда-Уоршалла, Форда-Фалкерсона, расчета потока минимальной стоимости, Прима, Краскала, поиск эйлерова цикла и поиск гамильтоновых циклов.

Алгоритм Беллмана-Форда работает быстрее алгоритма Дейкстры из-за реализации последнего перебором вершин ($O(n\hat{2})$). Алгоритм Краскала работает быстрее алгоритма Прима, потому что первый проходит по каждому ребру только один раз, тогда как второй проходит по вершинам многократно.

Достоинства:

- Компактный вывод матриц и результатов работы алгоритмов в консоль.
- При достраивании графа до эйлерова, гамильтонова и до сети исходный граф сохраняется.
- Алгоритм кодирования кодом Прюфера сохраняет вес рёбер.

Недостатки:

- Реализация алгоритма Дейкстры через перебор вершин, а не через очередь.
- Задача коммивояжёра решена полным перебором, из-за чего методы поиска гамильтоновых циклов неэффективны по времени.

Масштабируемость:

- Расширение функционала работы с графом путем увеличения числа алгоритмов, например, алгоритма Борувки.
- Реализация алгоритма Дейкстры через очередь.

Список источников

1. Новиков Ф.А. Дискретная математика для программистов. 3-е издание. СПб.:Питер, 2009.–384 с.
2. Кук Д., Бейз Г. Компьютерная математика: Пер.с англ.- М.: Наука, Гл.ред. физ.-мат. лит., 1990.–384 с.
3. Вадзинский Р.Н. Справочник по вероятностным распределениям. СПб.:Наука, 2001.–295 с.