МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ

РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ

ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ

«САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ ПОЛИТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ ПЕТРА ВЕЛИКОГО»

Институт прикладной математики и механики

Высшая школа прикладной математики и вычислительной физики

Секция «Телематика»

Дисциплина: Теория графов

Отчет по лабораторным работам № 1 - 5

Реализация алгоритмов на графах

Студент:

Верещагина С. Е.
группа: 3630201/90002

Проверил:

старший преподаватель,
Востров А. В.

Санкт-Петербург – 2021

Содержание

Введение

В данном отчете представлено описание лабораторной работы по дисциплине Теория графов.

Лабораторная работа включала в себя реализацию алгоритмов, применяемых к связному ациклическому графу. Граф сформирован в соответствии с отрицательным гипергеометрическим распределением 2.

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

1 Постановка задачи

В основе выполнения работы лежала задача построения случайного связного ациклического графа в соответствии с биномиальным распределением. На полученном графе реализовывались различные алгоритмы, выполняющие следующие задачи:

1. поиск экстремальных путей методом Шимбелла;
2. определение возможности построения маршрута;
3. поиск кратчайших путей (алгоритмы Дейкстры, Беллмана-Форда и Флойда-Уоршелла);
4. построение минимального по весу остова (алгоритмы Прима и Краскала);
5. поиск числа остовных деревьев (матричная теорема Кирхгофа);
6. кодирование остова с помощью кода Прюфера,
7. поиск максимального потока (алгоритм Форда-Фалкерсона) и потока минимальной стоимости;
8. построение эйлерова цикла;
9. решение задачи коммивояжёра.

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).

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

Число вершин графа обозначается как p, а число ребер - q.

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 [1].

2.2 Биномиальное распределение

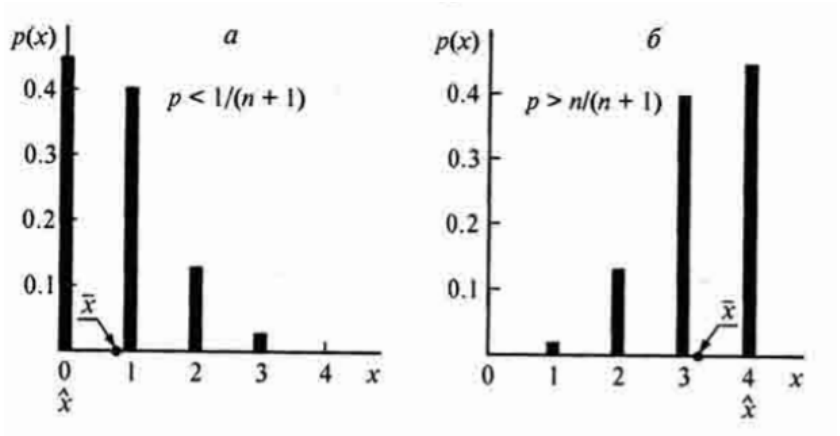
Распределение Бернулли - распределение, где X - число успехов с вероятностью успеха p и вероятностью неудачи $q = 1 - p^{-1}$.

Биномиальное распределение имеет место в тех случаях, когда последовательность испытаний Бернулли обрывается после проведения фиксированного числа n испытаний. При этом под биномиальной случайной величиной X понимается число успехов в серии из n испытаний Бернулли [3].

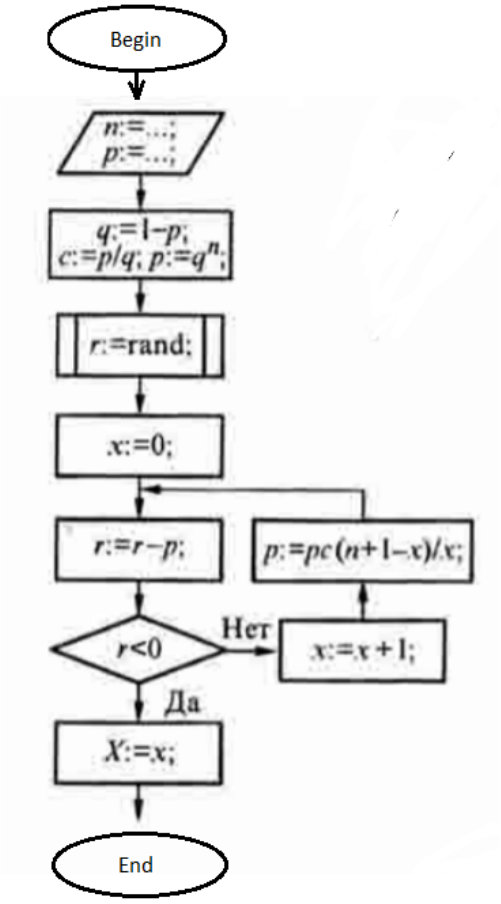
Ряд распределения выглядит следующим образом:

$$p(x) = C_n{}^x p^x q^{n-x}, x = 0, 1, ..., n, n \geqslant 1 - , 0 < p < 1, q = 1 - p$$

График функции распределения с параметрами а: $n = 4, p = 0.18$; б: $n = 4, p = 0.82$ выглядит следующим образом:



Блок-схема алгоритма генерирования биномиальных случайных чисел выглядит следующим образом:



2.3 Метод Шимбелла

Данный метод предназначен для определения минимальных (максимальных) расстояний между всеми парами вершин взвешенного графа и учитывает число ребер, входящих в соответствующие простые цепи.

Ключом к методу Шимбелла поиска экстремальных путей является специальные операции:

1. Операция умножения:

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

2. Операция сложения:

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

С помощью этих операций длины кратчайших (или максимальных) путей определяются возведением в степень x весовой матрицы Ω , где x – коли-

чество рёбер в пути. Элемент Ω^x [i][j] покажет длину экстремального пути длиной в x рёбер из вершины i в вершину j.

[1]

2.4 Алгоритм Дейкстры

Алгоритм Дейкстры является жадным алгоритмом. Алгоритм находит кратчайшие пути во взвешенном графе от одной вершины до всех остальных, если длины дуг неотрицательны. [1]

Описание работы алгоритма Дейкстры:

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 поиска кратчайшего пути во взвешенном графе. За время O(p*q) алгоритм находит кратчайшие пути от одной вершины графа до всех остальных. В отличие от алгоритма Дейкстры, алгоритм Беллмана — Форда допускает рёбра с отрицательным весом.

Заметим, что кратчайших путей может не существовать. Так, в графе, содержащем цикл с отрицательным суммарным весом, существует сколь угодно короткий путь от одной вершины этого цикла до другой (каждый обход цикла уменьшает длину пути). Цикл, сумма весов рёбер которого отрицательна, называется отрицательным циклом. [1]

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 (узлов) в (ор)графе. Веса ребер могут быть как положительными, так и отрицательными. Для нахождения кратчайших путей между всеми вершинами графа используется восходящее динамическое программирование, то есть все подзадачи, которые впоследствии понадобятся для решения исходной задачи, просчитываются заранее и затем используются. [1]

Идея алгоритма — разбиение процесса поиска кратчайших путей на фазы.

Перед k-ой фазой величина d[i][j] равна длине кратчайшего пути из вершины i в вершину j, если этому пути разрешается заходить только в вершины с номерами, меньшими k. На k-ой фазе мы попытаемся улучшить путь $i \to j$, 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: $O(n^3)$.

2.7 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.

2.8 Алгоритм Краскала

Алгоритм Краскала – жадный алгоритм построения минимального остовного дерева взвешенного связного неориентированного графа. На выходе получится множество ребер, которые лежат в минимальном остовном дереве. [1]

Описание шагов алгоритма:

1) Все дуги удаляются из дерева.

2) Все рёбра сортируются по весу (в порядке неубывания).

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

Сложность: O(n^2).

2.9 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. [1]

Описание шагов алгоритма:

1) Изначально остов полагается состоящим из единственной вершины (её можно выбрать произвольно).

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

3) После этого остов содержит уже две вершины, и теперь ищется и добавляется ребро минимального веса, имеющее один конец в одной из двух выбранных вершин, а другой — наоборот, во всех остальных, кроме этих двух. И так далее, т.е. всякий раз ищется минимальное по весу ребро, один конец которого — уже взятая в остов вершина, а другой конец — ещё не взятая, и это ребро добавляется в остов (если таких рёбер несколько, можно взять любое).

Этот процесс повторяется до тех пор, пока остов не станет содержать все вершины.

В итоге будет построен остов, являющийся минимальным. Если граф был изначально не связен, то остов найден не будет (количество выбранных рёбер останется меньше n - 1).

Сложность алгоритма: O(n^2).

2.10 Матричная теорема Кирхгофа

Матрица Кирхгофа – матрица размера $n \times n$, где n - количество вершин графа. [1]

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

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.

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

2.11 Код Прюфера

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

То есть, код Прюфера – это биекция между всеми остовными деревьями полного графа и числовыми последовательностями.

Алгоритм 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 единственного соседа v в последовательность кода Прюфера;
3. Удалить вершину v из дерева.

2.12 Максимальный поток в сети

Пусть G(V,E) - сеть, s и t - соответственно, источник и сток сети. Дуги сети нагружены неотрицательными вещественными числами, $: E \to R^+$. 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:

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

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

1. $\forall (u,v) \in E (0 \leq f(u,v) \leq c(u,v))$, 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.

Величина потока в сети G - сумма всех потоков, выходящих из истока, то есть $w(f) = div(f,s)$. [1]

2.13 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 Форда — Фалкерсона решает задачу нахождения максимального потока в транспортной сети.

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. [1]

2.14 Эйлеров граф

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

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

Теорема:

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

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

Значит для проверки наличия эйлерова цикла в графе достаточно убедиться, что степени всех вершин чётны. [1]

2.15 Гамильтонов граф

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

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

Теорема. Достаточное условие гамильтоновости графа
Если $\delta(G) \geq p/2$, то граф G является гамильтоновым. [1]

2.16 Задача коммивояжера

Задача коммивояжера – задача, в которой коммивояжер должен посетить N городов, побывав в каждом из них ровно по одному разу и завершив путешествие в том городе, с которого он начал. В какой последовательности ему нужно обходить города, чтобы общая длина его пути была наименьшей? Таким образом требуется

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

при этом каждая вершина должна быть посещена ровно один раз.

Задача коммивояжера - задача отыскания кратчайшего гамильтонова цикла в нагруженном полном графе.

3 Особенности реализации

3.1 Структура данных для хранения графа

Граф задается с помощью матрицы смежности, которая хранится в двумерном массиве целых чисел размерностью $n * n$.

```
vector<vector<int>> matrAdg;
```

3.2 Алгоритм формирование графа

Формирование графа в соответствии с распределением осуществляется в функции ArrInit() с помощью функции random.generator() и класса binomialdistribution, обьект которого принимает на вход целочисленное значение n — количество вершин графа, p - вероятность успеха.

В цикле производится заполнение массива array случайными числами, генерируемыми в соответствии с биномиальным распределением (алгоритм генерирования случайных чисел описан в п.2.2). array — вектор из степеней вершин графа. Очевидно, что размер этого массива равен n, а сумма всех элементов должна быть чётна.

Далее происходит заполнение матрицы смежности вершин M, то есть "соединение"вершин друг с другом.

Делается это по следующему принципу: две вершины u и v с максимальными степенями, если ещё не были соединены, соединяются ребром, элементам матрицы M[u][v] и M[v][u] присваивается значение 1, степени этих вершин в графе увеличиваются на единицу. Если степень вершины достигает нужного значения, хранящегося в векторе array, эта вершина не рассматривается в последующих итерациях. Описанные действия повторяются до тех пор, пока степени всех вершин не достигнут заданных значений.

```
int ArrInit(vector<vector<int>>& arr, int n) {
binomial_distribution <int> bern(n, 0.6);
random_device rd;
mt19937 generator(rd());
arr.clear();
for (int i = 0; i < n; i++)
{
arr.push_back(vector<int>(n, 0));
}
for (int i = 0; i < n; ++i)
{
for (int j = i + 1; j < n; j++)
{
/*int p = (rand() % 100) / 100.0;
if (p < 0.6)
arr.at(i).at(j) = 1;
else  arr.at(i).at(j) = 0;*/
if (j == i + 1) arr.at(i).at(j) = 1;
else arr.at(i).at(j) = bern(generator) % 2;
}
}
return 0;
}
```

3.3 Поиск экстремальных путей методом Шимбелла

Возведение матрицы arr1 в степень осуществляется перемножением матриц arr1 и arr2 по правилу умножения матриц с применением особых операций умножения и сложения (см. п. 2.3 в разделе «Математическое описание»). Выход из рекурсии производится, когда матрица оказывается возведённой в нужную степень.

Вход: Матрицы смежности вершин

Выход: Матрица Шимбелла экстремальных путей из заданной вершины.

```
int matrixMultiply(vector<vector<int>>& arr1, vector<vector<int>>& arr2, bool ExtrPath) {

vector<int> buffer;

vector<vector<int>> buffArr;
for (int i = 0; i < arr1.size(); i++)
buffArr.push_back(vector<int>(arr1.size(), 0));

for (int i = 0; i < arr1.size(); i++)
{
for (int j = 0; j < arr1.size(); j++)
{
buffer.clear();
for (int k = 0; k < arr1.size(); k++)
{
if ((arr1[i][k]) && (arr2[k][j]))
buffer.push_back(arr1[i][k] + arr2[k][j]);

else
buffer.push_back(0);

}
if (ExtrPath)
{
int min = buffer.front();
for (auto x : buffer)
{
if (((min == 0) && (x != 0)) || ((x < min) && (x != 0)))
min = x;
}
buffArr[i][j] = min;
}
else {
int max = 0;
for (auto x : buffer)
{
if (((max == 0) && (x != 0)) || ((x > max) && (x != 0)))
max = x;
}
buffArr[i][j] = max;
}

}
}
arr1 = buffArr;

return 0;
}
```

Сложность алгоритма: O(n^2).

3.4 Определение возможности построения маршрута

Для поиска пути была реализована вспомогательная функция обхода графа в губину DFS (depth-first search).

Алгоритм обхода в глубину позволяет решать множество различных задач, в том числе и для определения количества маршрутов, состоящих из k рёбер.

Данный алгоритм базируется на рекурсии.

Глубина рекурсии, в которой мы находимся, не превышает общего числа вызовов функции DFS — числа вершин. То есть, объем памяти, необходимый для работы алгоритма, равен O(V).

Вход: стартовая вершина, конечная вершина, массив, в котором хранится информация о том, была ли посещена вершина, матрица весов

Выход: значение true или false.

```
bool dfs(int u, int t, vector<bool>& visited, vector<vector<int>>& arr)
{
if (u == t)
return true;
visited.at(u) = true;
int k = 0;
for (auto v : arr.at(u))
{
if (v && !visited.at(k))
{
if (dfs(k, t, visited, arr))
return true;
}
k++;
}
return false;
}
```

3.5 Поиск кратчайших путей

3.5.1 Алгоритм Дейкстры

Алгоритм Дейкстры состоит из двух этапов:

I. Алгоритм нахождения длины кратчайшего пути:

1. Для каждой вершины v будем хранить текущую длину D[v] кратчайшего пути из s в v. Для этого создадим массив D. На первом этапе D[s]=0, для всех остальных вершин эта длина равна ∞. Каждой вершине из V сопоставляется метка – минимальное известное расстояние до данной вершины. Первый шаг алгоритма заключается в поиске вершины с наименьшим значением метки, будем считать эту вершину текущей.

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

3. Превращение меток из временных в постоянные.

Вход: граф с вершинами V, рёбрами E с весами f(e); вершина-источник u, конечная вершина c.

Выход: вектор расстояний D(v) до каждой вершины $v \in V$ от вершины u.

```
oid Dijkstra(vector<vector<int>> arr, int a, int c)
{
vector<bool> visited;
vector<int> D;
vector<int> Path(arr.size(), 0);
vector<int> p(arr.size(), -1);
int it = 0;
for (auto v : arr.at(a - 1))
{
D.push_back(v);
visited.push_back(false);
}
D.at(a - 1) = 0;
visited.at(a - 1) = true;
int index = 0, u = 0;
for (int i = 0; i < arr.size(); i++)
{
int min = INF;
for (int j = 0; j < arr.size(); j++)
{
if ((!visited[j]) && (D[j] < min))
{
min = D[j];
index = j;
}
it++;
}
u = index;
visited.at(u) = true;
for (int j = 0; j < arr.size(); j++)
{
if (!visited.at(j) && arr.at(u).at(j) != INF && D.at(u) != INF && (D.at(u) + arr.at(u).at(j) < D.
    at(j)))
{
D.at(j) = D.at(u) + arr.at(u).at(j);
Path.at(j) == u;
}
}
}
cout << endl;
cout << "Dijkstra:\t\n";
for (int i = 0; i < arr.size(); i++)
{
cout << a << " - " << "To : " << i + 1 << " = " << D[i] << endl;
}
cout << endl;
cout << endl << "Num of iterations:" << it << endl;
vector<int> ver;
int end = c;
int VER[16] = {0};
VER[0] = end;
//ver.at(0) = end + 1;
int k = 1;
int weight = D.at(end);
while (end != 0)
{
for (int i = 0; i < arr.size(); i++)
if (arr.at(i).at(end) != 0)
{
int tmp = weight - arr.at(i).at(end);
if (tmp == D.at(i))
{
weight = tmp;
end = i;
VER[k] = i + 1;
//ver.at(k) = i + 1;
k++;}}}
cout << "From " << a << " to " << c << "\n";
/*
for (int i = 0; i < ver.size(); i++)
{
cout << ver.at(i) << endl;
}
//*/

for (int i = 0; i < 16; i++)
```

```
{
cout << VER[i] << endl;
}

cout << endl;
}
```

Сложность: O(n^2).

3.5.2 Алгоритм Форда - Беллмана

1. На первом шаге инициализируются расстояния от исходной вершины до всех остальных вершин, как бесконечные, а расстояние до начальной вершины принимается равным 0. Создается массив d[ ] размера |V| со всеми значениями равными бесконечности, за исключением элемента d[u], где u — исходная вершина.

2. Вторым шагом вычисляются самые короткие расстояния. Следующие шаги нужно выполнять |V|-1 раз, где |V| — число вершин в данном графе.

   Производится следующее действие для каждого ребра u-v:

   Если d[v] > d[u] + вес ребра uv, то обновить d[v]

   d [v] = d [u] + вес ребра uv

3. На третьем шаге сообщается, присутствует ли в графе цикл отрицательного веса. Для каждого ребра u-v необходимо выполнить следующее:

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

Идея шага 3 заключается в том, что шаг 2 гарантирует кратчайшее расстояние, если граф не содержит цикла отрицательного веса. Если мы снова переберем все ребра и получим более короткий путь для любой из вершин, это будет сигналом присутствия цикла отрицательного веса.

Сложность алгоритма — O(VE).

Вход: граф с вершинами V, рёбрами E с весами f(e); вершина-источник u.
Выход: вектор расстояний d(v) до каждой вершины v ∈ V от вершины u.

```
vector<int> BellmanFord(vector<vector<int>> arr, int vershina)
{
vector<edge> e; (arr.size() * arr.size());
vector<int> d(arr.size(), INF);
vector<int> p(arr.size(), -1);
vector<int> path;
int count = 0;


for (int i = 0; i < arr.size(); i++)
for (int j = 0; j < arr.size(); j++)
{
if (arr[i][j] < INF)
{
e.push_back(edge(i, j, arr[i][j]));
count++;
}
```

```
}

d[vershina - 1] = 0;//
int it = 0;
for (int i = 0; i < arr.size(); ++i) {
for (int j = 0; j < count; ++j) {
if (d[e[j].getA()] < (INF - 100)) {
if (d[e[j].getB()] > d[e[j].getA()] + e[j].getCost()) {
d[e[j].getB()] = d[e[j].getA()] + e[j].getCost();
p[e[j].getB()] = e[j].getA();
}
}
it++;
}
}

for (int i = 0; i < d.size(); i++)
cout << "To : " << i + 1 << " - " << "Weight: " << d[i] << endl << endl;

cout << endl << "Number of iterations:" << it << endl << endl;
}
else {
for (int cur = arr.size() - 1; cur != -1; cur = p[cur])
path.push_back(cur);
reverse(path.begin(), path.end());
}
return path;

}
```

Сложность алгоритма: O(p*q)

3.5.3 Алгоритм Флойда - Уоршелла

Ключевая идея алгоритма — разбиение процесса поиска кратчайших путей на фазы.

Требуется, чтобы выполнялось d[i][i] = 0 для любых i. Если между какими-то вершинами ребра нет, то записать следует ∞.

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

Для оптимизации алгоритма вершина k не принимает значения 0 и n, где n — номер последней вершины, так как ни через ту, ни через другую, не будет проходить новый путь.

Вход: матрица весов

Выход: матрица расстояний между всеми вершинами

```
void  Warshall(vector<vector<int>>& arr)
{
int it = 0;
for (int i = 0; i < arr.size(); i++)
{
for (int j = 0; j < arr.size(); j++)
{
if (arr[i][j] == 0 && i != j)
arr[i][j] = 999;
}
}

for (int i = 0; i < arr.size(); i++) {
for (int j = 0; j < arr.size(); j++) {
for (int k = 0; k < arr.size(); k++) {
if (arr[i][k] + arr[k][j] < arr[i][j])
arr[i][j] = arr[i][k] + arr[k][j];
```

```
it++;
}
}
}
for (int i = 0; i < arr.size(); i++)
{
for (int j = 0; j < arr.size(); j++)
{
if (arr[i][j] >= 900)
cout << 0 << "\t";
else
cout << arr[i][j] << "\t";
}
cout << endl;
}
cout << endl << "Number of iterations:" << it << endl << endl;
}
```

Сложность алгоритма: $O(n^3)$.

3.6 Минимальный остов

3.6.1 Алгоритм Краскала

Алгоритм Краскала заключается в том, что мы каждую вершину помещаем в свое множество.

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

Если нет, то добавляем данное ребро в наше дерево (предварительно его создав с помощью массива ребер), после добавления мы добавляем все вершины, которые принадлежали тому же множеству, что и вторая вершина ребра, в множество первой вершины.

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

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

Вход: матрица смежности вершин, матрица весов

Выход: матрица смежности минимального остова

```
vector<set<int>>* Kruskal(vector<vector<int>>& arr, vector<vector<int>>& matrixVesov)
{

int it = 0, m = 0;
int count = 0;
for (int i = 0; i < arr.size(); i++)
for (int j = 0; j < arr.size(); j++)
{
if (arr[i][j] > 0 && arr[i][j] < (INF / 2))
m++;
}
m = m / 2;

vector < edge> g;
for (int i = 0; i < arr.size(); i++)
{
for (int j = i; j < arr.size(); j++)
if (matrixVesov[i][j] < 500)
```

```
{
g.push_back(edge(i, j, matrixVesov[i][j]));
count++;

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

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

vector<int> tree_id(arr.size());
for (int i = 0; i < arr.size(); ++i)
tree_id[i] = i;
for (int i = 0; i < m; ++i)
{
int a = g[i].getA(), b = g[i].getB(), l = g[i].getCost();
if (tree_id[a] != tree_id[b])
{
cost += l;

res.push_back(make_pair(a, b));
int old_id = tree_id[a], new_id = tree_id[b];


for (int j = 0; j < arr.size(); ++j)
{
if (tree_id[j] == old_id)
{
tree_id[j] = new_id;
}
it++;
}
}
}
vector<set<int>>* ostov = new vector<set<int>>(arr.size());
for_each(res.begin(), res.end(), [&](pair<int, int> x)
{
ostov->at(x.first).insert(x.second);
ostov->at(x.second).insert(x.first);
if (x.first > x.second)
{
cout << x.second + 1 << " --> " << x.first + 1 << endl;
}
else
{
cout << x.first + 1 << " --> " << x.second + 1 << endl;
}
});
cout << endl;
cout << "\nMin ostov cost: " << cost << endl;
cout << endl;
cout << "Number of iterations:" << it << endl;
cout << endl;

return ostov;

}
```

Сложность: $O(n^2)$.

3.6.2 Алгоритм Прима

1) Берётся произвольная вершина и находится ребро, инцидентное данной вершине и обладающее наименьшей стоимостью. Найденное ребро и соединяемые им две вершины образуют дерево.

2) Рассматриваются рёбра графа, один конец которых — уже принадлежащая дереву вершина, а другой — ещё не принадлежащая. Из этих рёбер выбирается ребро наименьшей стоимости.

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

Результатом работы алгоритма является остовное дерево минимальной стоимости.

Вход: матрица смежности вершин связного неориентированного графа, матрица весов

Выход: остовное дерево минимальной стоимости

```
vector<set<int>>* Prim(vector<vector<int>>& arr, vector<vector<int>>& matrixVesov)
{
int it = 0;

int cost = 0;
vector<bool> used(arr.size());
vector<int> min_e(arr.size(), INF), sel_e(arr.size(), -1);
min_e[0] = 0;
vector<set<int>>* ostov = new vector<set<int>>(arr.size());
for (int i = 0; i < arr.size(); ++i)
{
int v = -1;
for (int j = 0; j < arr.size(); ++j)
if (!used[j] && (v == -1 || min_e[j] < min_e[v]))
{
v = j;
}
if (min_e[v] == INF)
{
cout << endl;
cout << "No ostov exist!";
cout << endl;
exit(0);
}
used[v] = true;

for (int to = 0; to < arr.size(); ++to)
{
if (matrixVesov[v][to] < min_e[to])
{
min_e[to] = matrixVesov[v][to];
sel_e[to] = v;
}
it++;
}


if (sel_e[v] != -1)
{
ostov->at(sel_e[v]).insert(v);
ostov->at(v).insert(sel_e[v]);
if ((sel_e[v] + 1) > (v + 1))
{
cout << v + 1 << " --> " << sel_e[v] + 1 << endl;
}
else
{
cout << sel_e[v] + 1 << " --> " << v + 1 << endl;
}
cost += matrixVesov[sel_e[v]][v];

}
}
cout << endl;
cout << "\nMin ostov cost:: " << cost << endl;
cout << endl;
cout << "Num of iterations:" << it << endl;
cout << endl;

return ostov;
}
```

Сложность: $O(n^2)$.

3.6.3 Матричная теорема Кирхгофа

В матрице смежности неориентированного графа заменяем каждый элемент на противоположный, а на диагонале вместо элемента Ai,i ставим степень вершины i. Тогда, согласно матричной теореме Кирхгофа, все алгебраические дополнения этой матрицы равны между собой, и равны количеству остовных деревьев этого графа. Далее удаляем первую строку и первый столбец этой матрицы, и модуль её определителя будет равен искомому количеству.

Вход: матрица смежности неориентированного графа, матрица весов

Выход: число остовных деревьев

```
void Kirhgof(vector<vector<int>>& arr, vector<vector<int>>& matrixVesov)
{

int count = 0;
vector<vector<int>> buffer(arr.size(), vector<int>(arr.size()));
for (int i = 0; i < arr.size(); i++)
{
for (int j = 0; j < arr.size(); j++)
if (arr[i][j] == INF)
buffer[i][j] = 0;
else
{
buffer[i][j] = -arr[i][j];
count++;
}
buffer[i][i] = count;
count = 0;
}

cout << "\nKirhgof matrix: \n";
for (int i = 0; i < arr.size(); i++)
{
for (int j = 0; j < arr.size(); j++)
cout << buffer[i][j] << '\t';
cout << endl;
}

int minor = 1;

vector<vector<int>> arr2(arr.size() - 1, vector<int>(arr.size() - 1));
for (int i = 1; i < arr.size(); i++)
{
for (int j = 1; j < arr.size(); j++)
{
arr2[i - 1][j - 1] = buffer[i][j];
}
}

for (int i = 0; i < arr.size() - 1; i++)
minor += arr2[0][i] * pow(-1, (1 + i + 1)) * findMinor(arr2, 0, i);

if (arr.size() != 2)
minor -= 1;
cout << endl << "\nNum of ostov trees = " << abs(minor) << endl << endl;

}
```

3.7 Код Прюфера

3.7.1 Кодирование Прюфера

Кодирование возможно если в графе 3 и более вершин. В цикле выбирается лист с наименьшим номером, в код Прюфера добавляется номер вершины, связанной с этим листом. Процедура повторяется p-1 раз.

Вход: остовное дерево

Выход: код Прюфера

```
void PruferCoding(vector<set<int>>* ostov, vector<int>& pruferCode)
{
if (ostov == nullptr)
{
cout << endl;
cout << "\nError! Try to find ostov!\n";
cout << endl;
return;
}
if (ostov->size() == 2)
{
cout << "1" << endl;
return;
}
if (pruferCode.size() == (ostov->size() - 2))
{
cout << endl;
cout << "\nPrufer code: " << endl;
cout << endl;
for (int i = 0; i < ostov->size() - 2; i++)
cout << pruferCode.at(i) + 1 << ' ';
cout << endl;
return;
}
vector<bool> visited(ostov->size());

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

break;
}

}
break;
}
}
}
cout << endl;
cout << "\nPrufer code: " << endl;
cout << endl;
for (int i = 0; i < ostov->size() - 2; i++)
cout << pruferCode.at(i) + 1 << ' ';
cout << endl;
}
```

3.7.2 Декодирование кода Прюфера

По построенному коду можно восстановить исходное дерево по следующему алгоритму:

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

Найденная вершина и текущий элемент составляют ребро дерева.

Вход: код Прюфера
Выход: остовное дерево

```
void PruferEncoding(vector<set<int>>* ostov, vector<int>& pruferCode)
{
if (ostov == nullptr)
{
cout << endl;
cout << "\nError! Try to find ostov!\n";
cout << endl;
return;
}
if (ostov->size() == 2)
{
cout << endl;
cout << "\nDecoding Prufer:" << endl;
cout << endl;
cout << 1 << " --> " << 2 << endl;
return;
}
if (pruferCode.size() < (ostov->size() - 2))
{
cout << endl;
cout << "\nError! Try coding Prufer first!\n";
cout << endl;
return;
}
cout << endl;
cout << "\nDecoding Prufer:" << endl;
cout << endl;
deque<int> v;
bool b = false;
for (int i = 0; i < ostov->size(); i++)
v.push_back(i);
int index = 0;
while (v.size() != 2)
{
bool b = false;
for (int i = 0; i < pruferCode.size(); i++)
if (pruferCode.at(i) == v.at(index))
b = true;
if (b)
{
index++;
b = false;
}
else
{
auto iter = v.begin() + index;
cout << v.at(index) + 1 << " - " << pruferCode.at(0) + 1 << endl;
v.erase(iter);
pruferCode.erase(pruferCode.begin());
index = 0;
b = false;
}
}
cout << v.at(0) + 1 << " --> " << v.at(1) + 1 << endl;
}
```

3.8 Поток в сети

3.8.1 Алгоритм Форда - Фалкерсона

Алгоритм Форда — Фалкерсона решает задачу нахождения максимального потока в сети.

Заполняется массив потока пропускными способностями, далее идет выборка минимальной пропускной способности, вычитание из пропускной способности найденный поток и сохраненение в матрицу потока полученного значения.

Сложность алгоритма: O(qf), где q - число рёбер в графе, f - максимальный поток в графе.

Вход: источник, сток, матрица стоимости, матрица пропускных способностей.

Выход: максимальный поток.

```
int fordFulkerson(int s, int t, vector<vector<int>>& cost, vector<vector<int>>& bandwidth)
{

if ((cost.empty()) || (bandwidth.empty())) {
cout << endl;
cout << "Cost matrix and bandwidth matrixes aren't generated yet!" << endl;
cout << endl;
return 0;
}
else {
int u, v;

vector<vector<int>>potok(cost.size(), vector<int>(cost.size(), 0));

vector<vector<int>> rGraph(cost.size(), vector<int>(cost.size(), 0));
for (u = 0; u < cost.size(); u++) {
for (v = 0; v < cost.size(); v++) {
if (bandwidth[u][v] > (INF / 2)) {
rGraph[u][v] = 0;
}
else {
rGraph[u][v] = bandwidth[u][v];
}
}
}
vector<int> parent(cost.size());
int max_flow = 0;

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

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

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

max_flow += path_flow;
}
if (Print) {
cout << endl;
cout << "Flow matrix : " << endl;
outArr(potok);
cout << endl;
}

return max_flow;
}
}
```

3.8.2 Алгоритм поиска заданного потока минимальной стоимости

Заданное значение потока устанавливается как 2/3 от максимального потока в сети.

1. Ищется минимальный по весу путь из истока в сток с помощью алгоритма Беллмана-Форда (или алгоритма Дейкстры, но в этом случае в графе не должно быть дуг отрицательного веса).

2. Вычисляется максимальный поток по найденному пути.

3. Полученное значение прибавляется к значению потока каждой дуги найденного увеличивающего пути и отнимается от заданного значения потока.

4. Повторять, пока заданное значение потока не станет равным нулю.

Вход: матрица пропускных способностей, матрица стоимости

Выход: матрица потоков

```
void MinCostMaxFlow(int b, vector<vector<int>>& arr, vector<vector<int>>& costArr, vector<vector<
    int>>& bandwidthArr)
{
//flFord = false;

vector<vector<int>> matrPotok(arr.size(), vector<int>(arr.size(), 0));

if ((costArr.empty()) || (bandwidthArr.empty())) {
cout << endl;
cout << "Cost matrix and bandwidth matrixes aren't generated yet!" << endl;
return;
}
else {
int maxflow = 0;
int flow = 0;
int sigma = INF;
int result = 0;
int m = 0;
vector<vector<int>> modifiedCostsArr(costArr);
map<int, pair<vector<int>, int>> tempMap;
map<vector<int>, pair<int, int>> temporary;
set<pair<int, int>> resultVertex;
vector<vector<int>> f(arr.size(), vector<int>(arr.size(), 0));
vector<int> k;

for (int i = 0; i < arr.size(); i++)
for (int j = 0; j < arr.size(); j++)
{
if (arr[i][j])
m++;
}

while (maxflow < b) {
vector<int> path = BellmanFord(modifiedCostsArr, 1, 0);
for (int i = 0; i < path.size() - 1; i++) {
sigma = min(sigma, bandwidthArr[path.at(i)][path.at(i + 1)]);
if (resultVertex.find(pair<int, int>(path.at(i), path.at(i + 1))) == resultVertex.end()) {
resultVertex.insert(pair<int, int>(path.at(i), path.at(i + 1)));
}
}

flow = min(sigma, (b - maxflow));
k.push_back(flow);
int tempCost = 0;
for (int i = 0; i < path.size() - 1; i++)
{
tempCost += costArr[path.at(i)][path.at(i + 1)];
}
if (temporary.find(path) == temporary.end()) {
temporary[path] = pair<int, int>(flow, tempCost);
}
else {
```

```
temporary[path].first += flow;
}
for (int i = 0; i < path.size() - 1; i++) {
f.at(path[i]).at(path[i + 1]) += flow;
if (f.at(path[i]).at(path[i + 1]) == bandwidthArr[path[i]][path[i + 1]]) {
modifiedCostsArr[path[i]][path[i + 1]] = INF;
modifiedCostsArr[path[i + 1]][path[i]] = INF;
}
else
{
if (f.at(path[i]).at(path[i + 1]) >= 0) {
modifiedCostsArr[path[i]][path[i + 1]] = costArr[path[i]][path[i + 1]];
modifiedCostsArr[path[i + 1]][path[i]] = -modifiedCostsArr[path[i]][path[i + 1]];
}
}
}

maxflow += flow;
}
int min = INF;
vector<pair<int, pair<vector<int>, int >>> tempVect;
for (auto i = temporary.begin(); i != temporary.end(); i++)
{
tempVect.push_back(pair<int, pair<vector<int>, int>>(i->second.second, pair<vector<int>, int>(i->
    first, i->second.first)));
}
//
for (int i = 0; i < tempVect.size() - 1; i++)
{
for (int j = 0; j < tempVect.size() - 1; j++)
{
if (tempVect[j].first > tempVect[j + 1].first)
}
}

vector<int> vecPotok;
vector<int> vecMatr;
int ur = 0;
for (int i = 0; i < tempVect.size(); i++)
{
cout << endl;
cout << endl << "Flow path: " << endl;
for (int j = 0; j < tempVect[i].second.first.size() - 1; j++)
{
cout << tempVect[i].second.first[j] + 1 << " - ";
vecMatr.push_back(tempVect[i].second.first[j]);
}
cout << tempVect[i].second.first[tempVect[i].second.first.size() - 1] + 1;
vecMatr.push_back(tempVect[i].second.first[tempVect[i].second.first.size() - 1]);
cout << endl;
cout << endl << "Flow, through path: " << tempVect[i].second.second << endl;
for (int l = 0; l < vecMatr.size() - 1; l++)
{
if (vecMatr[l] < vecMatr[l + 1] && matrPotok[vecMatr[l]][vecMatr[l + 1]] == 0)
matrPotok[vecMatr[l]][vecMatr[l + 1]] = tempVect[i].second.second;
else
if (vecMatr[l] < vecMatr[l + 1])
for (int k = 0; k < tempVect[i].second.first.size() - 1; k++)
{
for (int j = 0; j < tempVect[i].second.first.size() - 1; j++)
{
if (tempVect[k].second.first.at(j) == vecMatr[l] && tempVect[k].second.first.at(j + 1) == vecMatr
    [l + 1] &&
matrPotok[vecMatr[l]][vecMatr[l + 1]] + tempVect[i].second.second <= bandwidthArr[vecMatr[l]][
    vecMatr[l + 1]])
matrPotok[vecMatr[l]][vecMatr[l + 1]] += tempVect[i].second.second;
}
}
}
vecMatr.clear();
vecPotok.push_back(tempVect[i].second.second);
}


cout << endl;
for_each(resultVertex.begin(), resultVertex.end(), [&result, &f, &costArr](pair<int, int> x) {
```

```
    result += f.at(x.first).at(x.second) * costArr[x.first][x.second]; });


cout << endl << "Flow:" << b << endl << endl << endl;
cout << "Minimal cost of the flow: " << result << endl << endl;
cout << "Cost matrix:" << endl;
outArr(costArr);
cout << endl << "Flow matrix:" << endl;
outArr(matrPotok);
}
}
```

3.9 Алгоритм поиска Эйлерова цикла

Сначала необходимо убедиться в том, что граф является Эйлеровым.

Для этого проверяем степени вершин (степени всех вершин должны быть четными). В противном случае нужно привести граф к Эйлерову (т.е. добавить ребро или удалить ребро, связывающее те вершины, степень которых нечетная).

Пусть задан Эйлеров граф G=(V,E). Начинаем с некоторой вершины p и каждый раз вычеркиваем пройденное ребро. Не проходим по ребру, если удаление этого ребра приводит к разбиению графа на две связные компоненты, т.е. необходимо проверять, является ли ребро мостом или нет.

Вход: матрица смежности вершин

Выход: матрица смежности Эйлерова графа

```
int Euler(vector<vector<int>> gr) {
int n = gr.size();
vector<int> degree;
for (int i = 0; i < n; i++) {
int sst = 0;
for (int j = 0; j < n; j++) {
if (gr[i][j] != 0) sst++;
}
degree.push_back(sst);
}

int count = 0;
for (int i = 0; i < n; i++) {

if (degree[i] == 0) {

count = 1;

deg.clear();
for (int i = 0; i < gr.size(); i++)

deg.push_back(i);

break;
}


if (degree[i] % 2 != 0) {

count++;

deg.push_back(i);
}
}

if (count > 0) return 0;

int min = 100; int ind = 0;

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

```
if (min > degree[i]) {
min = degree[i];
ind = i;
}
}

vector<int> res;
stack<int> s;
s.push(ind);

while (!s.empty()) {
int i = s.top();
if (degree[i] == 0) {
s.pop();
res.push_back(i);
}
else {
for (int j = 0; j < n; j++) {

if (gr[i][j] != 0) {

gr[i][j] = 0;
gr[j][i] = 0;

degree[i] -= 1;
degree[j] -= 1;
s.push(j);
break;
}
}
}
}
if (res.size() != 0) {
cout << endl << "Euler cycle: ";
for (size_t i = res.size() - 1; i != 0; i--)
cout << res[i] + 1 << "-";
cout << res[0] + 1;
}
cout << endl << endl;
return res.size();
}
```

3.10 Алгоритм поиска Гамильтонова цикла

Так как не существует простого необходимого условия для проверки графа на гамильтоновость, чтобы установить, является ли граф гамильтоновым, производился поиск гамильтонова цикла.

Если в графе минимальная степень вершины не меньше двух, две вершины с минимальными степенями соединяются ребром и производится поиск гамильтонова цикла.

Вход: матрица смежности вершин

Выход: матрица смежности Гамильтонова графа

```
void Is_Gam(vector<vector<int>>& GamGr, vector<pair<int, int>>& path) {
int versh = GamGr.size();
int n = versh;
int c = 0;
bool flagGam = true;
vector<int> v0;
bool flag = false;
bool add = false;
bool first = 1;
bool f = false;
if (n < 3) {
cout << endl << "\nIt isn't possible to make a Hamilton graph! Graph vertexes less than 3!" <<
    endl << endl;
return;
```

```
}

while (flagGam)
{
if (gamilton(GamGr, path))
{
cout << endl;
cout << "Graph is Hamilton!" << endl;
cout << endl;
flagGam = false;
flag = true;
break;
}
else
{
cout << endl;
cout << "Build a graph to Hamilton!" << endl;
cout << endl;
add = false;
while (!flag)
{

for (int i = 0; i < n; i++)
{

if (path[i].second < n / 2 && !add)
{
v0.push_back(path[i].first);
}
}
for (int i = 0; i < v0.size(); i++)
{
for (int j = 0; j < n; j++)
{
if (!add)
{
if (GamGr[v0[i]][j] == 999 && v0[i] != j)
{
if (v0.size() == 2 && GamGr[v0[i]][v0[i + 1]] == INF)
{
GamGr[v0[i]][v0[i + 1]] = 1;
GamGr[v0[i + 1]][v0[i]] = 1;
if (v0[i] + 1 < v0[i + 1] + 1)
cout << endl << "Add " << v0[i] + 1 << " - " << v0[i + 1] + 1 << endl << endl;
else
cout << endl << "Add " << v0[i + 1] + 1 << " - " << v0[i] + 1 << endl << endl;
add = true;
f = true;
}
else {
GamGr[v0[i]][j] = 1;
GamGr[j][v0[i]] = 1;
if (v0[i] + 1 < j + 1)
cout << endl << "Add " << v0[i] + 1 << " - " << j + 1 << endl << endl;
else
cout << endl << "Add " << j + 1 << " - " << v0[i] + 1 << endl << endl;
add = true;
f = true;
}
}
}
}
}
flag = gamilton(GamGr, path);
add = false;
v0.clear();
}
}
}

if (f)
{
cout << endl;
cout << "Changed matrix:" << endl;
outArr(GamGr);
cout << endl;
```

```
}
cout << endl;
}
```

3.11 Задача коммивояжёра

Задача коммивожёра решается полным перебором всех перестановок последовательности вершин графа (всех, кроме начальной под номером 1). Очередная перестановка – это последовательность вершин в пути. Если такой путь есть в данном графе, существует ребро из первой вершины во вторую вершину пути и ребро из последней вершины в первую, то эта последовательность вершин – гамильтонов цикл – записывается в файл.

Из всех найденных гамильтоновых циклов выбирается один с минимальной длиной.

Вход: матрица смежности вершин гамильтонова графа

Выход: минимальный гамильтонов цикл

```
void TSP(arrWeight)
{
vector<int> GamGrVes = vector<int> (GamGr.size(),
vector<int>(GamGr.size(), 0));
int N = GamGr.size();
for (int i = 0; i < N; ++i)
{
for (int j = i + 1; j < N; j++)
{
if (GamGr[i][j] == INF)
{
GamGrVes[i][j] = 0;
GamGrVes[j][i] = 0;
}
else
{
GamGrVes[i][j] = rand() mod 9 + 1;
GamGrVes[j][i] = rand() mod 9 + 1;
}
}
}
int cost = 0;
int minCost = INF * N;
deque<int> que;
deque<int> res;
ofstream file;
file.open("TSP.txt", ofstream::out);
cout << "Output values:1-screen;2-file";
int what= Input();
bool flag;
switch (what)
{
case 1: flag = false;
break;
case 2: flag = true;
break;
}
bool isHamiltonCicle = true;
for (int i = 0; i < n; i++)
que.push_back(i);
do {
isHamiltonCicle = true;
cost = 0;

if (GamGr[que.at(n - 1)][que.at(0)]>INF)
isHamiltonCicle = false;
else {
for (int i = 0; i < n - 1; i++) {
```

```
49
50 if (GamGr[que.at(i)][que.at(i + 1)]>INF)
51 isHamiltonCicle = false;
52 else
53 cost+=GamGrVes[que.at(i)][que.at(i+1)];
54 }
55 }
56
57 if (cost < minCost) {
58 res.clear();
59 res.resize(que.size());
60
61 copy(que.begin(), que.end(), res.begin());
62 minCost = cost;
63 }
64 }
65 }
66 while (next_permutation(que.begin(), que.end()));
67 cout << endl << "minimum_cycle: " << endl;
68 cout << res.at(0) + 1;
69
70 for (int i = 1; i < res.size(); i++)
71 cout << " - " << res.at(i) + 1;
72 cout << " - " << res.at(0) + 1;
73 cout << "min cost: " << minCost << endl ;
74 }
```

4 Результаты работы программы

На рис. представлена работа программы с графом, состоящим из 4-х вершин.

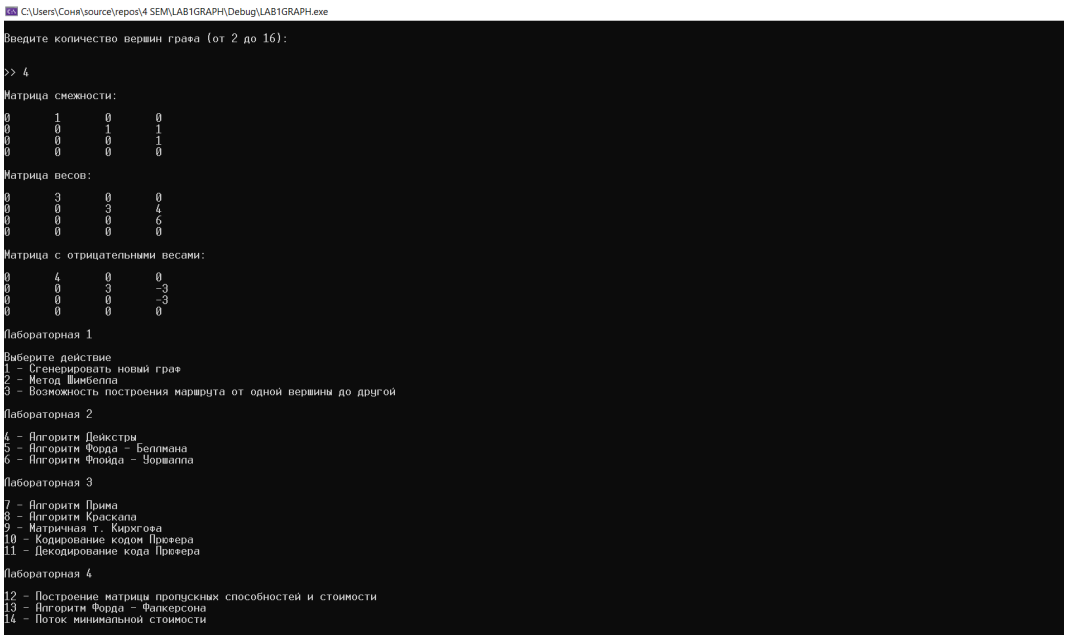


Рис. 1: Основное меню программы

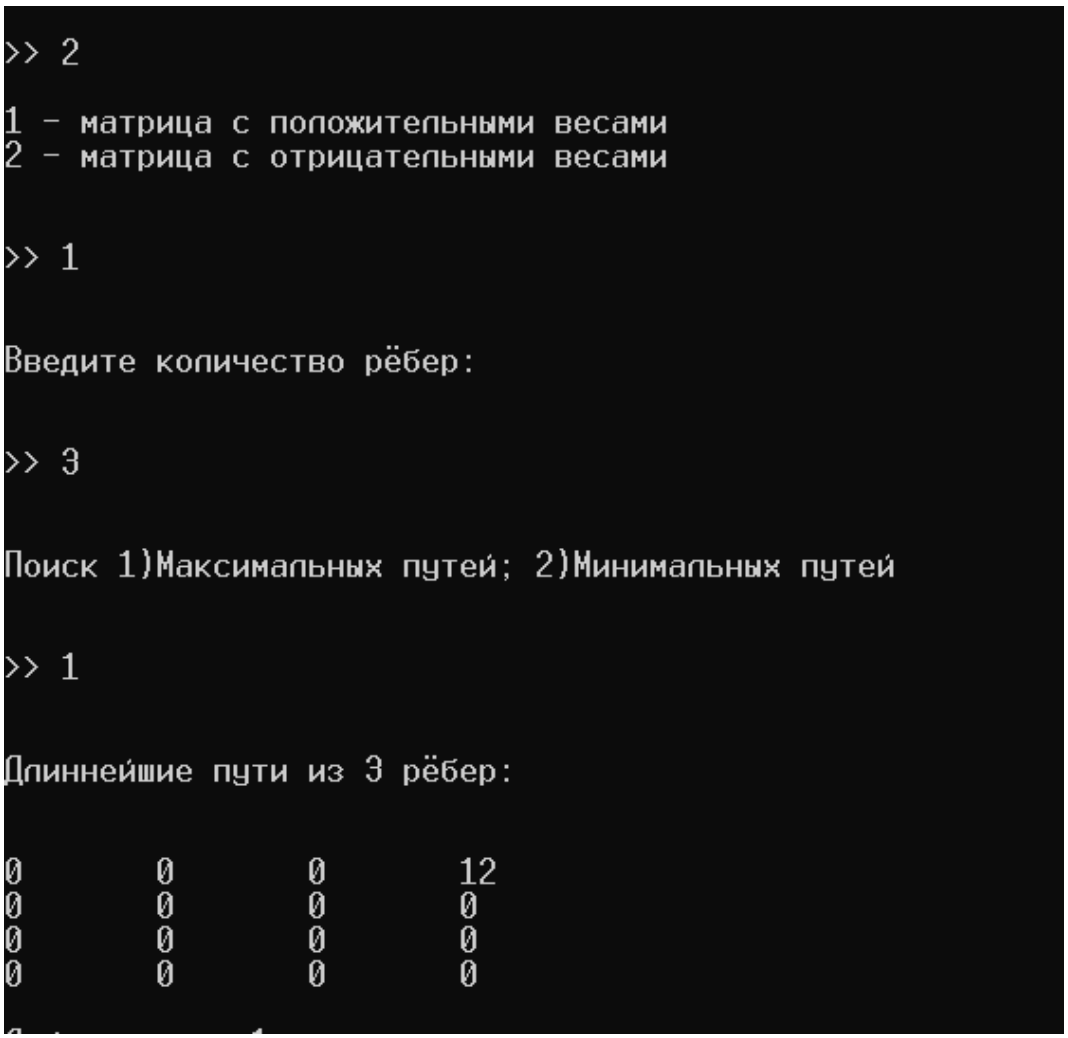


Рис. 2: Метод Шимбелла

```
>> 3

Введите номер первой вершины:

>> 1

Введите номер второй вершины:

>> 2

Путь из вершины 1 в вершину 2 существует
```

Рис. 3: Существование пути из одной вершины в другую

```
>> 4

Введите начальную вершину


>> 1
Введите конечную вершину

>> 3

Алгоритм Дейкстры:
1 - До вершины номер: 1 = 0
1 - До вершины номер: 2 = 3
1 - До вершины номер: 3 = 6
1 - До вершины номер: 4 = 7


Количество итераций:16
```

Рис. 4: Алгоритм Дейкстры

```
>> 5

1 - матрица с положительными весами
2 - матрица с отрицательными весами


>> 1

Введите начальную вершину

1


Алгоритм Беллмана-Форда.

До вершины номер: 1 - Вес ребер: 0

До вершины номер: 2 - Вес ребер: 3

До вершины номер: 3 - Вес ребер: 6

До вершины номер: 4 - Вес ребер: 7


Количество итераций:16
```

Рис. 5: Алгоритм Форда-Беллмана

```
>> 6

1 - матрица с положительными весами
2 - матрица с отрицательными весами


>> 1
0       3       6       7
0       0       3       4
0       0       0       6
0       0       0       0

Количество итераций:64
```

Рис. 6: Алгоритм Флойда-Уоршалла

```
>> 7

1 - матрица с положительными весами
2 - матрица с отрицательными весами


>> 1
1 --> 2
2 --> 3
2 --> 4


Стоимость минимального остова: 10

Количество итераций:16
```

Рис. 7: Алгоритм Прима

```
>> 8

1 - матрица с положительными весами
2 - матрица с отрицательными весами


>> 1
1 --> 2
2 --> 3
2 --> 4


Стоимость минимального остова: 10

Количество итераций:12
```

Рис. 8: Алгоритм Краскала

```
>> 9

1 - матрица с положительными весами
2 - матрица с отрицательными весами


>> 1

Матрица Кирхгофа:
1       -1      0       0
-1      3       -1      -1
0       -1      2       -1
0       -1      -1      2


Количество остовных деревьев графа по теореме Кирхгофа = 3
```

Рис. 9: Матричная т. Кирхгофа

```
>> 10

Код Прюфера:

2 2
```

```
>> 11

Декодирование Прюфера:

1 - 2
3 - 2
2 --> 4
```

Рис. 10 - 11: Кодирование и декодирование Прюфера

```
>> 12

Матрица стоимости:
0       2       0       0
0       0       8       4
0       0       0       5
0       0       0       0

Матрица пропускных способностей:
0       11      0       0
0       0       2       3
0       0       0       8
0       0       0       0
```

Рис. 12: Построение матриц пропускных способностей и стоимости

```
>> 13

Матрица потока :
0       5       0       0
0       0       2       3
0       0       0       2
0       0       0       0


Максимальный поток: 5
```

Рис. 13: Алгоритм Форда-Фалкерсона

```
>> 14


Путь потока:
1 - 2 - 4

Поток, через найденный путь: 3


Заданный поток:3


Минимальная стоимость заданного потока: 18

Матрица стоимости:
0       2       0       0
0       0       8       4
0       0       0       5
0       0       0       0

Матрица потока:
0       3       0       0
0       0       0       3
0       0       0       0
0       0       0       0
```

Рис. 14: Поток минимальной стоимости

```
>> 15


Граф не является эйлеровым!


Граф не является гамильтоновым!
```

Рис. 15: Является ли граф эйлеровым или гамильтоновым

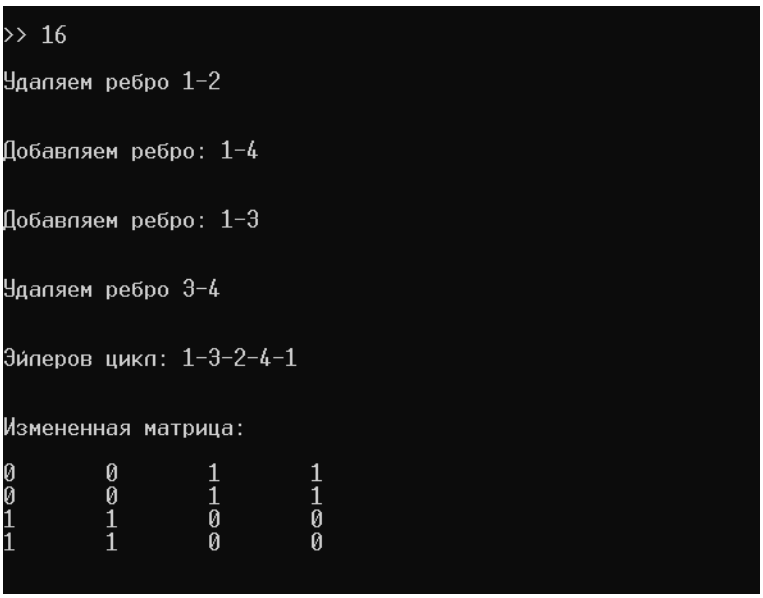


Рис. 16: Эйлеров цикл

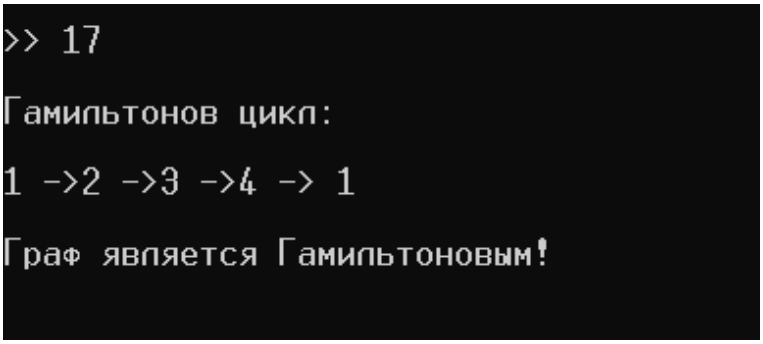


Рис. 17: Гамильтонов цикл

```
>> 18

Матрица стоимости Гамильтонова графа:
0       7       1       3
4       0       6       7
3       7       0       6
7       6       7       0


Вывод промежуточных значений
1 - на экран
2 - в файл

>> 1

Гамильтонов цикл:
1 -> 2 -> 3 -> 4 -> 1


Вес: 26


Гамильтонов цикл:
1 -> 2 -> 4 -> 3 -> 1


Вес: 24


Гамильтонов цикл:
1 -> 3 -> 2 -> 4 -> 1


Вес: 22


Гамильтонов цикл:
1 -> 3 -> 4 -> 2 -> 1


Вес: 17
```

```
Минимальный Гамильтонов цикл:
1 -> 3 -> 4 -> 2 -> 1

Вес: 17
```

Рис. 18: Задача Коммивояжера

Заключение

Результатом работы стала программа, выполняющая все поставленные задачи:

1. поиск экстремальных путей методом Шимбелла;

   • Данный метод не является эффективным с точки зрения расхода памяти, но прост и понятен в реализации.

2. определение возможности построения маршрута;

3. поиск кратчайших путей (алгоритмы Дейкстры, Беллмана-Форда и Флойда-Уоршелла);

   • Алгоритм Дейкстры является самым быстрым, но не применяется при наличии отрицательных весов дуг.

   • Алгоритм Беллмана-Форда медленнее, чем алгоритм Дейкстры ($O(p*q)$), так как рёбер в графе обычно больше, чем вершин. Преимущество данного алгоритма заключается в том, что он может работать с рёбрами отрицательного веса, при условии, что в графе отсутствуют отрицательные циклы.

   • Алгоритм Флойда - Уоршелла имеет самую высокую сложность среди всех перечисленных ($O(n^3)$) и работает намного дольше. Однако преимуществом данного алгоритма является то, что он может работать с ребрами как положительного, так и отрицательного веса.

4. построение минимального по весу остова (алгоритмы Прима и Краскала);

   • Алгоритм Прима

   Недостаток: необходимость просматривать все оставшиеся ребра для поиска минимального. Сложнее в реализации, чем алгоритм Краскала.

   • Алгоритм Краскала

   Недостаток: необходимость в реализации хранения и сортировки списка ребер.

5. поиск числа остовных деревьев (матричная теорема Кирхгофа);

6. кодирование остова с помощью кода Прюфера;

7. поиск максимального потока (алгоритм Форда-Фалкерсона) и потока минимальной стоимости;

8. построение эйлерова цикла;

9. решение задачи коммивояжёра.

Достоинства программы:

1. Алгоритм Флойда-Уоршелла не производит лишних итераций за счёт того, что нет необходимости искать новый путь, проходящий через первую и последнюю вершины.

2. В программе реализованы методы обхода в глубину и ширину и методы нахождения кратчайших путей, которые могут использоваться в других алгоритмах.

3. Все пять лабораторных реализованы в одном проекте с целью избежания повторяющегося кода.

Недостатки программы:

1. Задача коммивояжёра решена полным перебором.

2. Проверка графа на гамильтоновость производится с помощью поиска гамильтонова цикла.

3. Неоптимальное использование памяти: большое количество массивов, которые дублируют друг друга.

4. Во время выполнения кодирования Прюфера не сохранялись веса ребер.

Масштабирование программы:

1. Задачу коммивояжёра можно решить более эффективными методами (метод ветвей и границ, муравьиный алгоритм).

2. Существуют некоторые достаточные условия существования гамильтонова цикла в графе, по которым можно проверить гамильтоновость графа (например, условие Дирака).

3. Можно дополнить реализацию другими алгоритмами решения поставленных задач. Например, поиск кратчайших путей реализовать с помощью волнового алгоритма и алгоритма A*, а поиск кратчайшего остова – алгоритмом Борувки.

Список литературы

[1] Новиков Ф. А. «Дискретная математика для программистов: Учебник для вузов». Санкт-Петербург: Питер, 2008 г. – 384 стр.

[2] Полубенцева М. И. «C/C++. Процедурное программирование». Санкт-Петербург: БХВ, 2015 г. – 448 стр.

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

[4] Востров А. В. Лекции по курсу «Теория Графов» https://tema.spbstu.ru/tgraph/ (дата обращения: 24.03.2020).