Министерство образования и науки РФ

Санкт-Петербургский Политехнический университет Петра Великого

Институт компьютерных наук и технологий

Высшая школа искусственного интеллекта

Направление «Математика и компьютерные науки»

Отчёт по лабораторным работам 1-5 по теории графов
«Реализация алгоритмов на графах»

Студент: Кулыгин Е. А. гр. 3530201/00001

Руководитель: Востров Алексей Владимирович

Санкт – Петербург

2022

Содержание

Введение 4
1 Постановка задачи 5
2 Математическое описание 6
Определение графа 6
Генерация графа 6
Метод Шимбелла 8
Алгоритм Дейкстры 9
Алгоритм Беллмана форда 9
Алгоритм Флойда – Уоршелла 9
Алгоритм Форда – Фалкерсона 9
Вычисление потока минимальной стоимости 10
Поиск минимального остова 10
Алгоритм Краскала 10
Алгоритм Прима 11
Матричная теорема Кирхгофа 11
Код Прюфера 11
Эйлеров граф 12
Гамильтонов граф 12
Задача коммивояжера 12
3 Особенности реализации 14
Генерация графа 14
Алгоритм достижимости 15
Метод Шимбелла 15
Алгоритм Дейкстры 16
Алгоритм Беллмана - Форда 17
Алгоритм Флойда - Уоршелла 19
Алгоритм Краскала 20
Алгоритм Прима 20
Матричная теорема Кирхгофа 22
Кодирование и декодирование Прюфера 23
Алгоритм Форда – Фалкерсона 25
Поток минимальной стоимости 26
Эйлеров граф 27
Гамильтонов граф 29
Задача коммивояжера 31
4 Пример работы программы 33

Генерация графа 33
Метод Шимбелла 34
Определение возможности построения маршрута 34
Алгоритм Дейкстры 35
Алгоритм Беллмана – Форда 36
Алгоритм Флойда – Уоршелла 37
Поток минимальной стоимости 38
Алгоритм Краскала 39
Алгоритм Прима 39
Подсчёт количества остовных деревьев в графе 40
Кодировка графа кодом Прюфера 40
Поиск Эйлерова цикла 41
Поиск Гамильтонова цикла и решение задачи коммивояжера 41
Заключение 43
Список используемой литературы 45

Введение

Теория графов – раздел дискретной математики, изучающий свойства графов. Этот раздел применяется при анализе сложных систем, таких как, железные дороги, компьютерные сети, в различных сферах химии и во многих других областях.

Данный отчёт описывает реализацию алгоритмов, применяемых к связному ациклическому графу. Для построения графа степени его вершин генерируются в соответствие с распределением Паскаля, которое также известно как отрицательное биномиальное распределение 2. Лабораторные работы были реализованы на языке C++ в среде Visual Studio 2019.

1 Постановка задачи

Лабораторная работа 1 :

1. Сформировать случайным образом связный ациклический граф в соответствии с заданными распределением (параметры распределения задаются как константы до компиляции программы).
2. Реализовать метод Шимбелла на полученном графе (пользователь вводит количество ребер по средством консоли).
3. Определить возможность построения маршрута от одной заданной точки до другой (пользователь вводит вершины по средством консоли)

Лабораторная работа 2 :

1. Для сформированных графах (случайно сгенерированных в предыдущей работе) найти кратчайший путь от одной точки до другой, используя алгоритмы Дейкстры, Беллмана-Форда, Флойда Уоршелла.
2. Сравнить скорости работы данных алгоритмов (итерации).

Лабораторная работа 3 :

1. Сформировать связный ациклический граф случайным образом в соответствии с заданным распределением. На его основе построить матрицы пропускных способностей и стоимости.
2. Для полученного графа найти максимальный поток по алгоритму Форда-Фалкерсона (или любого из перечисленных в лекции).
3. Вычислить поток минимальной стоимости (в качестве величины потока брать значение, равное [2/3*max], где *max* – максимальный поток). Использовать ранее реализованные алгоритмы Дейкстры и или Беллмана – Форда.

Лабораторная работа 4 :

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

Лабораторная работа 5 :

1. Для заданных графов (случайно сгенерированных в первой работе) проверить, является ли граф эйлеровым и гамильтоновым. Если граф не является таковым, то отдельно модифицировать граф до эйлерова и отдельно до гамильтонова.
2. Построить эйлеров цикл.
3. Решить задачу коммивояжера на гамильтоновом графе.

2 Математическое описание

Определение графа

Графом G(V,E) называется совокупность двух множеств – непустого множества V (множества вершин) и множества E неупорядоченных пар различных элементов множества V (E – множество ребер).

$$G(V,E) =< V,E >, V \neq 0, E \subset V \times V ; E = E_{-1}$$

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.

Генерация графа

Требуется построить граф, предварительно сгенерировав последовательность степеней вершин для этого графа. Элементы последовательности степеней должны быть сгенерированы с помощью распределения Паскаля.(см. Рис. 1). Пример функции из справочника Вадзинского “Справочник по вероятностным распределениям”.

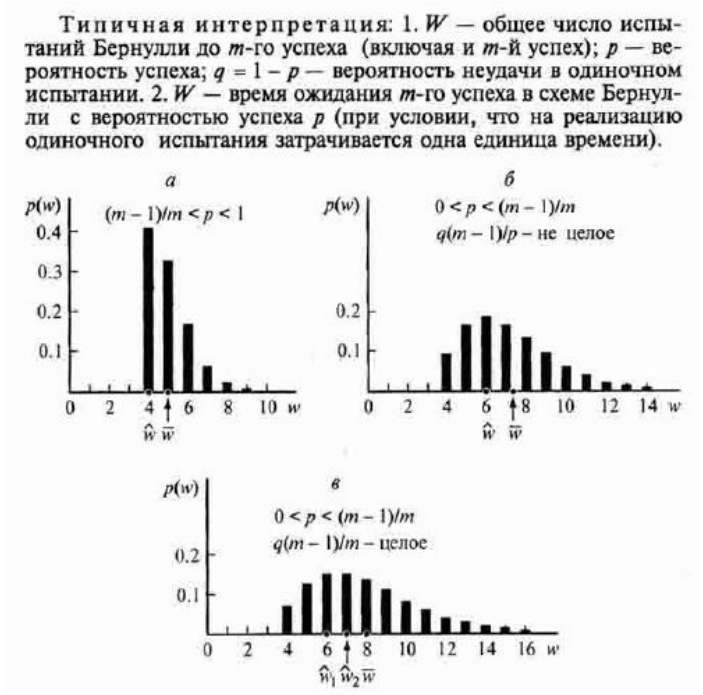


Рисунок 1

Ряд распределения Паскаля:

Ряд распределения $p(w) = C_{w-1}^{m-1} p^m q^{w-m}$,

$w = m, m+1, m+2, \ldots,$

$m \geq 1$ — целое, $0 < p < 1$, $q = 1 - p$

Рисунок 2

Для генерации распределения используется отрицательное биномиальное распределение 1. Так как случайная величина $W(m, p) = Z(m, p) + m$. Где $Z(m, p)$ это случайная величина с отрицательным биномиальным распределением 1. Для генерации случайных величин используется блок – схема на рисунке 3.

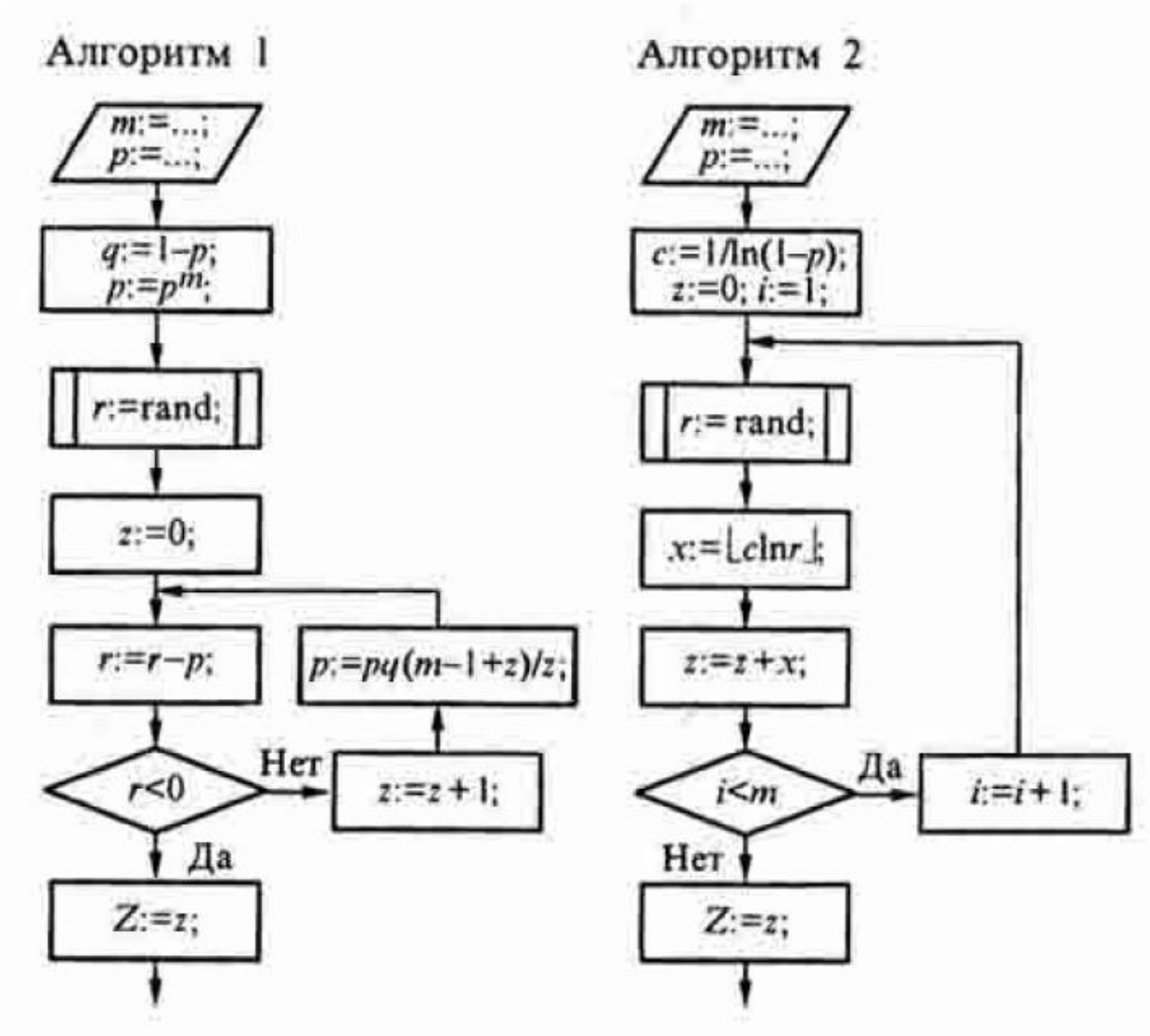
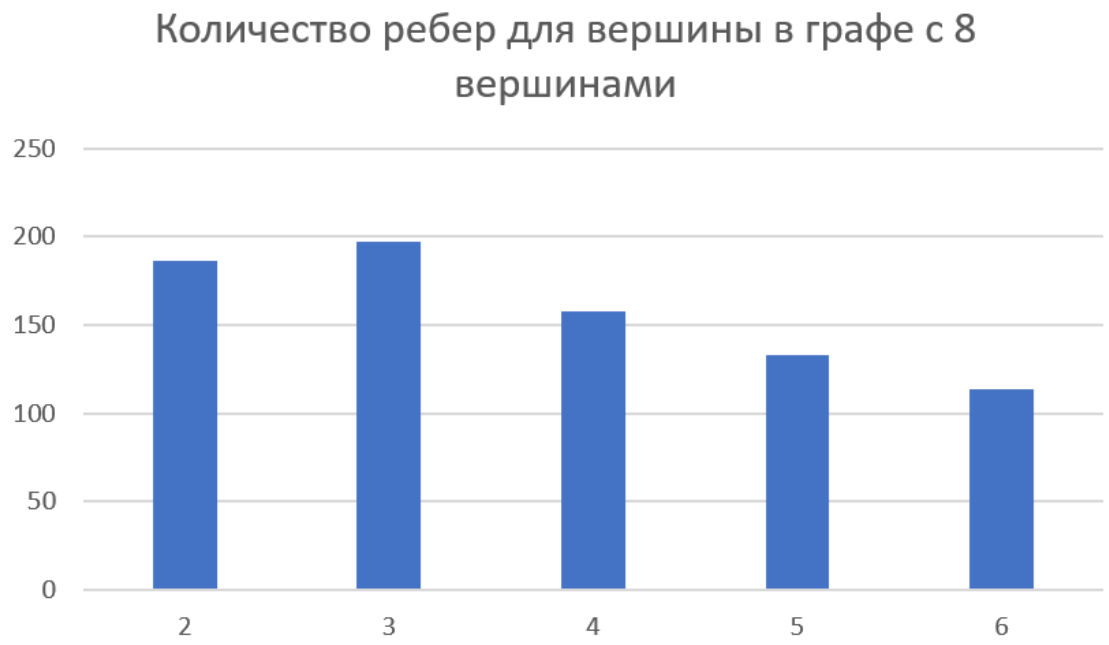


Рисунок 3



Метод Шимбелла

Алгоритм предназначен для определения кратчайших расстояний между всеми парами вершин взвешенного графа и учитывает число ребер, входящих в соответствующие простые цепи. Матрица весов D в исходном виде задает стоимость переходов между вершинами графа по цепям, состоящим из одного звена.

Необходимо определить специальные математические операции:

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

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

2) Операция сложения двух величин заменяется поиском минимального или максимального элемента:

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

Для нахождения экстремальных путей между вершинами необходимо возвести матрицу весов графа в степень, равную количеству ребер, входящих в цепь.

Алгоритм Дейкстры

Алгоритм находит кратчайший между двумя вершина (узлами) в (ор)графе, если длины дуг неотрицательны. Алгоритм Дейкстры является жадным алгоритмом. Сложность алгоритма: O(n).

Алгоритм Беллмана форда

Алгоритм позволяет найти кратчайшие пути из одной вершины графа до всех остальных, даже для графов, в которых веса ребер могут быть отрицательными. Тем не менее, в графе не должны быть циклов отрицательного веса, достижимых из начальной вершины, иначе вопрос о кратчайших путях является бессмысленным. 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.

Алгоритм Флойда – Уоршелла

Алгоритм находит кратчайшие пути между всеми парами вершин (узлов) в (ор)графе. Веса ребер могут быть как положительными, так и отрицательными. Данный алгоритм использует идею динамического программирования. В этом алгоритме для хранения информации о путях используется матрица H: array [1..p,1..p] of 1..pб где

$$\left\{ H[i,\ j] = \begin{matrix} k, если\ k - первая\ вершина, достигаемая\ на\ кратчайшем\ пути\ из\ i\ в\ j; \\ 0, если\ из\ вершины\ i\ в\ вершину\ j\ нет\ пути \end{matrix} \right.$$

Сложность алгоритма: O (n^3) .

Алгоритм Форда – Фалкерсона

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:

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

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

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

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

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

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

Вычисление потока минимальной стоимости

Величиной потока в сети G называют $\omega(f) = div(f,s)$, то есть сумму всех потоков, выходящих из истока.

Задача поиска заданного потока минимальной стоимости в сети состоит в том, чтобы минимизировать целевую функцию:

$$\sum_{u,v \in V} a(u,v) \cdot f(u,v)$$

Где $a(u,v)$ - вес дуги (u,v), $f(u,v)$ - поток через дугу (u,v).

Поиск минимального остова

Пусть $G(V,E)$ - граф. Остовный подграф графа $G(V,E)$ - это подграф, содержащий все вершины. Остовный подграф, являющийся деревом, называется остовом. Любое остовное дерево в графе с n вершинами содержит ровно n-1 ребро. Минимальное остовное дерево – остовное дерево, сумма весов ребер которого минимальна.

Алгоритм Краскала

Алгоритм Краскала относится к жадным алгоритмам, эффективный алгоритм построения минимального остовного дерева взвешенного связного неориентированного графа. Алгоритм состоит из двух фаз.

На подготовительной фазе все дуги удаляются из дерева и упорядочиваются по возрастанию их весов. В графе остаются только вершины, каждая из которых образует отдельную компоненту связности.

Во второй фазе дуги перебираются в порядке возрастания веса. Если начало и конец очередной дуги принадлежит одной и той же компоненте связности, дуга игнорируется. Если же они лежат в разных компонентах связности. Дуга добавляется к графу, а эти две

компоненты связности объединяются в одну. Если число компонент связности дойдет до 1, цикл завершается досрочно.

Сложность алгоритма: $O(n^2)$.

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.

Сложность алгоритма: $O(n^2)$.

Матричная теорема Кирхгофа

Матрицей Кирхгофа называется матрица размера $n \times n$, где n- количество вершин графа. На главной диагонали матрицы находятся степени вершин, а на пересечении i-й строки и j-го столбца $(i \neq j)$ стоит -1, если вершины с номерами i и j смежны, и 0 в противном случае.

Алгебраическое дополнение элемента a_{ij} матрицы A – это число $A_{ij} = (-1)^{i+j} M_{ij}$, где M_{ij} - дополнительный минор, определитель матрицы, получающейся из исходной матрицы A путем вычеркивания i-й строки и j-го столбца.

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

Свойства матрицы Кирхгофа:

1. Суммы элементов в каждой строке и каждом столбце матрице B равны 0.
2. Алгебраические дополнения всех элементов матрицы B равны между собой.

Код Прюфера

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

Кодирования Прюфера переводит помеченные деревья порядка n в последовательность чисел от 1 до n по алгоритму:

Пока количество вершин больше двух:

1. Выбирается лист v с минимальным номером.
2. В код Прюфера добавляется номер вершины, смежной с v.
3. Вершина v и инцидентное ей ребро удаляются из дерева.

Декодирование кода для восстановления дерева. В начале работы алгоритма имеются значения из кода Прюфера и список всех вершин графа. Далее n-2 раза повторяется процедура:

1. Берется первое значение кода Прюфера, и производится поиск наименьшей вершины из списка всех вершин дерева, не содержащейся в массиве с кодом.
2. Найденная вершина и текущее значение кода Прюфера составляют ребро дерева.

В результате получается список всех ребер графа, который был закодирован.

Эйлеров граф

Если граф имеет цикл, содержащий все ребра графа ровно один раз, то такой цикл называется эйлеровым циклом, а граф называется эйлеровым графом. Если граф имеет цепь, содержащую все ребра, то такая цепь называется эйлеровой цепью, а граф называется полуэйлеровым.

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.

Условие эйлеровости графа: для того, чтобы в связном графе существовал эйлеров цикл, необходимо, чтобы в нем все вершины были с четной степенью связности.

Гамильтонов граф

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

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

Достаточное условие гамильтоновости графа (условие Дирака) – если в графе $G(V, E)$ с n вершинами выполняется условие $\deg(v) \geq \frac{n}{2}$ для $\forall v \in V$.

Задача коммивояжера

Формулировка задачи: имеется p городов, расстояния между которым известны. Коммивояжер должен посетить все города по одному разу, вернувшись в тот, с которого начал. Требуется найти такой маршрут движения, при котором суммарное пройденное расстояние будет минимальным.

Проблему коммивояжеру можно представить в виде модели на графе, то есть, используя вершины и ребра между ними. Таким образом, вершины графа соответствуют городам, а рёбра $(i, j)(i, j)$ между вершинами ii и jj пути сообщения между этими городами.

Каждому ребру $(i, j)(i, j)$ можно сопоставить критерий выгодности маршрута $c_{ij} \geq 0$, который можно рассматривать как, расстояние между городами, время или стоимость поездки. Таким образом, задача коммивояжера – задача отыскания кратчайшего гамильтонова цикла в нагруженном полном графе.

3 Особенности реализации

Генерация графа

Граф задается с помощью матрицы смежности. Для генерации графа с помощью распределения Паскаля используется функция PascalRng, которая генерирует случайные значения с учётом параметров, а затем, в зависимости от значений, метод Prepare заполняет одну строку матрицы смежности с последующим перемешиванием.

Вход: два параметра распределения

Выход: значение из распределения

```
int PascalRng(double m, double p) {
	std::random_device rd;
	std::mt19937 mersenne(rd());
	double q = 1 - p;
	p = pow(p, m);
	double r = mersenne() % (100000 - 1) + 1;
	int z = 0;
	r = static_cast<double>(r) / 100000;
	r = r - p;
	while (r >= 0) {
		z++;
		p = p * q * (m - 1 + z) / z;
		r = r - p;
	}
	return int(z + m);
}
```

```
Вход: значение количества связей в рамках одной строки матрицы смежности
Выход: матрица смежности
```

```
vector<int> MyGraph::Prepare() const {
	vector<int> vertexDegrees;
	int tmpVertDeg;
	for (int i = 0; i < vertexCnt - 2; i++) {
		do {
			tmpVertDeg = PascalRng();
		} while (tmpVertDeg >= vertexCnt);
		vertexDegrees.push_back(tmpVertDeg);
	}
	std::sort(vertexDegrees.begin(), vertexDegrees.end(), std::greater<int>());
	for (int i = 0; i < vertexCnt - 2; i++) {
		if (vertexDegrees[i] > (vertexCnt - 1 - i)) {
			vertexDegrees[i] = vertexCnt - 1 - i;
		}
	}
	return vertexDegrees;
}
```

Затем вспомогательная библиотека shuffle случайным образом определяет какие связи именно должны быть в одной строке. Например `tmpVertDeg` возвращает [4,3,2]. И затем случайным образом определяется какие именно вершины должны быть связаны. В первой строке матрицы смежности будет четыре единицы и один ноль. Значит первая вершина будет связана с четырьмя остальными.

Также, для ацикличности графа матрица имеет треугольную форму. В следствие работы метода вызывались методы генерации весов.

Алгоритм достижимости

Вход: матрица смежности, номера вершин

Выход: вывод возможности достижимости

```
void ExecReachability(const MyGraph& graph) {
	int vert1, vert2;
	iMx reachMatrix = graph.GetReachMatrix();
	do {
		cout << "Введите номер первой вершины:\n";
	} while (!GetInputInt(vert1, 1, graph.GetVertexCount()));
	cout << '\n';
	do {
		cout << "Введите номер второй вершины:\n";
	} while (!GetInputInt(vert2, 1, graph.GetVertexCount()));
	cout << '\n';
	vert1--;
	vert2--;
	cout << "Матрица достижомости:\n";
	PrintMatrix(reachMatrix);
	cout << '\n';
	if (reachMatrix[vert1][vert2]) {
		cout << "Существует " << reachMatrix[vert1][vert2] << " маршрутов между
данными вершинами!\n";
	}
	else {
		cout << "Не существует маршрутов между данными вершинами!\n";
	}
	cout << '\n';
}
```

Метод Шимбелла

В соответствии с описанным алгоритмом в математической модели, было реализовано два метода. `CalcShimbell() – для генерации матрицы и ShimbellMult() для перемножения матриц.`

Вход: матрица смежности вершин, количество вершин

Выход: матрица Шимбелла

```
iMx MyGraph::CalcShimbell(int edgeCnt, ShimbellMode mode) const {
	iMx resMatrix = posWeightsMatrix;
	for (int i = 0; i < edgeCnt - 1; i++) {
		resMatrix = ShimbellMult(resMatrix, posWeightsMatrix, mode);
	}
	return resMatrix;
}
```

```
iMx MyGraph::ShimbellMult(const iMx mxA, const iMx mxB, ShimbellMode mode) const {
	iMx resMatrix(vertexCnt, vector<int>(vertexCnt, 0));
	vector<int> buf;
	bool isNotZero;
	for (int i = 0; i < vertexCnt; i++) {
		for (int j = 0; j < vertexCnt; j++) {
			buf.clear();
			isNotZero = false;
			for (int k = 0; k < vertexCnt; 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 == ShimbellMode::kShortest) {
					resMatrix[i][j] = *std::min_element(buf.begin(),
buf.end());
				}
				else {
					resMatrix[i][j] = *std::max_element(buf.begin(),
buf.end());
				}
			}
			else {
				resMatrix[i][j] = 0;
			}
		}
	}
	return resMatrix;
}
```

Алгоритм Дейкстры

Данный алгоритм указывает все возможные пути от одной вершины до всех прочих вершин.

Вход: номер исходной вершины

Выход: расстояния до каждой прочей вершины

Также, алгоритм применяет и нахождение с помощью подбора, и с помощью перебора.

```
void ExecDijkstra(const MyGraph& graph) {
	int inpVert, counter = 0;

	do {
		cout << "Введите исходную вершину:\n";;
	} while (!GetInputInt(inpVert, 1, graph.GetVertexCount()));
	cout << '\n';
	inpVert--;

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

	cout << "П Е Р Е Б О Р:\n";
```

```
	vector<int> distances = graph.Dijkstra(inpVert, counter);
	iMx paths = graph.RestorePaths(inpVert, distances,
graph.GetWeightsMatrix(WeightsType::kModifiedPos));

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

	cout << "О Ч Е Р Е Д Ь:\n";

	distances = graph.Dijkstra_queue(inpVert, counter);
	paths = graph.RestorePaths(inpVert, distances,
graph.GetWeightsMatrix(WeightsType::kModifiedPos));

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

}
```

Алгоритм Беллмана - Форда

Как и алгоритм Дейкстры, алгоритм Беллмана–Форда работает путем релаксации, при которой приближения к правильному расстоянию заменяются лучшими, пока они в конечном итоге не достигнут решения. В обоих алгоритмах приблизительное расстояние до каждой вершины всегда является завышением истинного расстояния и заменяется минимальным его старым значением и длиной вновь найденного пути. Также, алгоритм

Беллмана – Форда может работать с отрицательными весами, для демонстрации, некоторые веса изначальной матрицы получают отрицательные значения.

Вход: матрица смежности,

Выход:

```
void ExecBellmanFord(const MyGraph& graph) {
	int inpVert, counter = 0;

	do {
		cout << "Введите исходную вершину:\n";;
	} while (!GetInputInt(inpVert, 1, graph.GetVertexCount()));
	cout << '\n';
	inpVert--;

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

	vector<int> distances = graph.BellmanFord(inpVert,
graph.GetWeightsMatrix(WeightsType::kMixed), counter);
	iMx paths = graph.RestorePaths(inpVert, distances,
graph.GetWeightsMatrix(WeightsType::kModifiedMixed));

	for (int i = 0; i < graph.GetVertexCount(); i++) {
		if (i != inpVert) {
			if (paths[i][0] != INF) {
				cout << "Кратчайший путь длиной " << std::setw(3) <<
distances[i] << " до вершины " << std::setw(2) << i + 1 << ": ";
				for (int j = paths[i].size() - 1; j > 0; j--) {
					cout << std::setw(2) << paths[i][j] + 1 << " -> ";
				}
				cout << std::setw(2) << paths[i][0] + 1 << '\n';
			}
			else {
				cout << "До вершины " << std::setw(2) << i + 1 << " пути
нет!\n";
			}
		}
	}
	cout << '\n';
	cout << "Количество итераций: " << counter << '\n';
	cout << '\n';
}
```

Также, вспомогательный метод BellmanFord(), который возвращает вектор означающий дистанцию от одной вершины до другой.

```
vector<int> MyGraph::BellmanFord(int inpVert, iMx wieghtsMx, int& counter) const {
	counter = 0;
	vector<int> distances(vertexCnt, INF);
	distances[inpVert] = 0;

	int curVert, newDistance;

	deque<int> dq;
	dq.push_back(inpVert);

	while (!dq.empty()) {
		curVert = dq.front();
```

```
			dq.pop_front();
			for (int i = curVert + 1; i < vertexCnt; i++, counter++) {
				if (wieghtsMx[curVert][i] != INF) {
					newDistance = distances[curVert] + wieghtsMx[curVert][i];
					if (newDistance < distances[i]) {
						distances[i] = newDistance;
						if (std::find(dq.begin(), dq.end(), i) == dq.end())
{
							dq.push_back(i);

						}
						else {

							std::remove(dq.begin(), dq.end(), i);

							dq.push_front(i);

						}
					}
				}
			}
		}

		return distances;
}
```

Алгоритм Флойда - Уоршелла

Алгоритм Флойда–Варшалла сравнивает все возможные пути через граф между каждой парой вершин. Он способен выполнять это с помощью сравнений в графе, даже если в графе может быть до $\Omega(|V|^2)$ ребер, и каждая комбинация ребер проверяется. Он делает это путем постепенного улучшения оценки кратчайшего пути между двумя вершинами, пока оценка не станет оптимальной.

Вход: матрица смежности

Выход: матрица расстояний между вершинами

```
void ExecFloydWarshall(const MyGraph& graph) {
	int counter = 0;

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

	iMx distancesMx = graph.FloydWarshall(counter);

	cout << "Матрица расстояний:\n";
	for (int i = 0; i < graph.GetVertexCount(); i++) {
		for (int j = 0; j < graph.GetVertexCount(); j++) {
			if (distancesMx[i][j] != INF) {
				cout << std::setw(3) << distancesMx[i][j] << " ";
			}
			else {
				cout << std::setw(3) << "INF" << " ";
			}
		}
```

```
			cout << '\n';
		}
		cout << '\n';

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

Алгоритм Краскала

В начале текущее множество рёбер устанавливается пустым. Затем, пока это возможно, проводится следующая операция: из всех рёбер, добавление которых к уже имеющемуся множеству не вызовет появление в нём цикла, выбирается ребро минимального веса и добавляется к уже имеющемуся множеству. Когда таких рёбер больше нет, алгоритм завершён. Подграф данного графа, содержащий все его вершины и найденное множество рёбер, является его остовным деревом минимального веса.

Вход: матрица весов

Выход: матрица весов кратчайшего остова

```
iMx MyGraph::Kruskal(iMx weightsMx, int* counter, int* sum) const {
		iMx minSpanTree = iMx(vertexCnt, vector<int>(vertexCnt, 0));
		priority_queue<Edge, vector<Edge>, std::greater<Edge>> pq = SortEdges(weightsMx);
		Edge edge;
		if (counter) {
			(*counter) = 0;
		}

		while (!pq.empty()) {
			edge = pq.top();
			pq.pop();
			if (!bfs(minSpanTree, edge.vert1, edge.vert2, nullptr, nullptr)) {
				minSpanTree[edge.vert1][edge.vert2] = weightsMx[edge.vert1][edge.vert2];
				minSpanTree[edge.vert2][edge.vert1] = weightsMx[edge.vert1][edge.vert2];
			}
			if (counter) {
				(*counter)++;
			}
		}

		if (sum) {
			(*sum) = 0;
			for (int i = 0; i < vertexCnt; i++) {
				for (int j = i + 1; j < vertexCnt; j++) {
					if (minSpanTree[i][j] != 0) {
						(*sum) += minSpanTree[i][j];
					}
				}
			}
		}

		return minSpanTree;
}
```

Алгоритм Прима

На вход алгоритма подаётся связный неориентированный граф. Для каждого ребра задаётся его стоимость.

Сначала берётся произвольная вершина и находится ребро, инцидентное данной вершине и обладающее наименьшей стоимостью. Найденное ребро и соединяемые им две вершины образуют дерево. Затем, рассматриваются рёбра графа, один конец которых — уже принадлежащая дереву вершина, а другой — нет; из этих рёбер выбирается ребро наименьшей стоимости. Выбираемое на каждом шаге ребро присоединяется к дереву. Рост дерева происходит до тех пор, пока не будут исчерпаны все вершины исходного графа.

Результатом работы алгоритма является остовное дерево минимальной стоимости.

Вход: матрица весов

Выход: остовное дерево минимальной стоимости

```
iMx MyGraph::Prim(iMx weightsMx, int* counter, int* sum) const {
	iMx minSpanTree = iMx(vertexCnt, vector<int>(vertexCnt, 0));
	vector<int> mstPath(vertexCnt, 0);
	mstPath[0] = -1;
	vector<int> mstKeys(vertexCnt, INF);
	mstKeys[0] = 0;
	vector<bool> isInMst(vertexCnt, false);
	int minKey, minIndex;
	if (counter) {
		(*counter) = 0;
	}

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

	for (int i = 0; i < vertexCnt - 1; i++) {
		minKey = INF;
		for (int j = 0; j < vertexCnt; j++) {
			if ((isInMst[j] == false) && (mstKeys[j] < minKey)) {
				minIndex = j;
				minKey = mstKeys[j];
			}
			if (counter) {
				(*counter)++;
			}
		}
		isInMst[minIndex] = true;

		for (int j = 0; j < vertexCnt; j++) {
			if ((weightsMx[minIndex][j] != 0) && (isInMst[j] == false) &&
(weightsMx[minIndex][j] < mstKeys[j])) {
				mstKeys[j] = weightsMx[minIndex][j];
				mstPath[j] = minIndex;
			}
			if (counter) {
				(*counter)++;
			}
		}
```

```
	}

	for (int i = 1; i < vertexCnt; i++) {
		minSpanTree[i][mstPath[i]] = weightsMx[i][mstPath[i]];
		minSpanTree[mstPath[i]][i] = weightsMx[i][mstPath[i]];
	}

	if (sum) {
		(*sum) = 0;
		for (int i = 0; i < vertexCnt; i++) {
			for (int j = i + 1; j < vertexCnt; j++) {
				if (minSpanTree[i][j] != 0) {
					(*sum) += minSpanTree[i][j];
				}
			}
		}
	}

	return minSpanTree;
}
```

Матричная теорема Кирхгофа

Пусть G — связный помеченный граф с матрицей Кирхгофа M. Все алгебраические дополнения матрицы Кирхгофа M равны между собой и их общее значение равно количеству остовных деревьев графа G.

Вход: матрица смежности вершин

Выход: матрица Кирхгофа

```
iMx MyGraph::GenKirchhoff() const {
	iMx kirchhoffMx = iMx(vertexCnt, vector<int>(vertexCnt, 0));
	int vertDeg;
	iMx modAdjMx = adjacencyMatrix;

	for (int i = 0; i < vertexCnt; i++) {
		for (int j = 0; j < i; j++) {
			modAdjMx[i][j] = adjacencyMatrix[j][i];
		}
	}

	for (int i = 0; i < vertexCnt; i++) {
		vertDeg = 0;
		for (int j = 0; j < vertexCnt; j++) {
			if (modAdjMx[i][j]) {
				kirchhoffMx[i][j] = -1;
				vertDeg++;
			}
		}
		kirchhoffMx[i][i] = vertDeg;
	}
	return kirchhoffMx;
}
```

Кодирование и декодирование Прюфера

Кодирование Прюфера переводит помеченные деревья порядка n в последовательность чисел от 1 до n по алгоритму:

Пока количество вершин больше двух:

1. Выбирается лист v с минимальным номером.
2. В код Прюфера добавляется номер вершины, смежной с v
3. Вершина v и инцидентное ей ребро удаляются из дерево.

Полученная последовательность – код Прюфера для заданного дерева.

Вход: матрица весов

Выход: код Прюфера и проверка его Декодирования

```
void MyGraph::PruferEncode(iMx& weightsMx, vector<int>& pruferCode, vector<int>&
pruferWeights) const {
	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 edge;
	iMx 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 {
					edge.vert2 = j;
					cnt++;
				}
			}
		}
		if (cnt == 1) {
			edge.vert1 = i;
			edge.weight = weightsMx[edge.vert1][edge.vert2];
			break;
		}
	}

	pruferCode.push_back(edge.vert2 + 1);
	pruferWeights.push_back(edge.weight);
    tmpWeightsMx[edge.vert1][edge.vert2] = tmpWeightsMx[edge.vert2][edge.vert1] = 0;
```

```
	PruferEncode(tmpWeightsMx, pruferCode, pruferWeights);
}
```

И декодирование соответственно :

1. Выбирается вершина v с непомеченным наименьшим номером из V, которая не встречается в массива кода Прюфера.
2. Ребро добавляется в дерево, соединяющее вершину v и первый элемент из кода Прюфера.
3. Из кода Прюфера удаляется первая вершина, в массиве вершина v помечается как использованная. Алгоритм выполняется до тех пор, пока в коде Прюфера не останется ни одной вершины.

Вход: код Прюфера

Выход: восстановленная матрица весов кратчайшего остова

```
iMx MyGraph::PruferDecode(vector<int>& pruferCode, vector<int>& pruferWeights) const
{
	iMx weightMx = iMx(pruferCode.size() + 1, vector<int>(pruferCode.size() + 1,
0));
	vector<bool> isUsed(weightMx.size(), false);
	vector<int> qPruferCode, qPruferWeights;
	for (int i = 0; i < pruferCode.size(); i++) {
		qPruferCode.push_back(pruferCode[i] - 1);
		qPruferWeights.push_back(pruferWeights[i]);
	}
	int tmpVert, tmpWeight;

	while (!qPruferCode.empty()) {
		for (int i = 0; i < weightMx.size(); i++) {
			if (qPruferCode.end() == std::find(qPruferCode.begin(),
qPruferCode.end(), i)) {
				if (isUsed[i] == false) {
					isUsed[i] = true;
					tmpVert = qPruferCode.front(); tmpWeight =
qPruferWeights.front();
					weightMx[i][tmpVert] = weightMx[tmpVert][i] =
tmpWeight;
					qPruferCode.erase(qPruferCode.begin());
qPruferWeights.erase(qPruferWeights.begin());
					break;
				}
			}
		}
	}

	return weightMx;
}
```

Алгоритм Форда – Фалкерсона

Изначально величине потока присваивается значение 0: f(u,v)=0 для всех u,v из V. Затем величина потока итеративно увеличивается посредством поиска увеличивающего пути (путь от источника s к стоку t, вдоль которого можно послать больший поток). Процесс повторяется, пока можно найти увеличивающий путь.

Вход: матрица пропускных способностей

Выход: максимальный поток

Матрица пропускных способностей :

```
void MyGraph::GenMaxFlowMx() {
	std::random_device rd;
	std::mt19937 mersenne(rd());
	maxFlowMx = adjacencyMatrix;
	for (int i = 0; i < vertexCnt; i++) {
		for (int j = i + 1; j < vertexCnt; j++) {
			if (adjacencyMatrix[i][j]) {
				maxFlowMx[i][j] = mersenne() % (FLOW_MAX - 1) + 1;
			}
		}
	}
}
```

Максимальный поток:

```
int MyGraph::fordFulkerson(int source, int sink) const {
	int tmpSink = sink;
	iMx residualGraph = AddFictVert();
	if (residualGraph.size() != vertexCnt) {
		source++;
		tmpSink++;
	}

	vector<int> path(residualGraph.size(), 0);
	int maxFlow = 0;
	int curFlow;
	while (bfs_FordFulkerson(residualGraph, source, tmpSink, path)) {
		curFlow = INF;
		for (int i = tmpSink; i != source; i = path[i]) {
			curFlow = std::min(curFlow, residualGraph[path[i]][i]);
		}
		for (int i = tmpSink; i != source; i = path[i]) {
			residualGraph[path[i]][i] -= curFlow;
			residualGraph[i][path[i]] += curFlow;
		}
		maxFlow += curFlow;
	}
	return maxFlow;
}
```

Поток минимальной стоимости

Заданное значение потока равно [2/3*max], где max – значение максимального потока в сети.

1. Находится минимальный по весу увеличивающий путь из истока в сток с помощью алгоритма Беллмана – Форда.
2. Вычисляется максимальный поток по найденному пути.
3. Далее полученное значение прибавляется к значению потока каждой дуги найденного увеличивающего пути и отнимается от заданного значения потока.
4. Пункты 1 – 3 повторяются, пока заданное значение потока не станет равным нулю.

Вход: матрица пропускных способностей

Выход: поток минимальной стоимости

```
int MyGraph::CalcMinCostFlow(int source, int sink, int flow, McfRetVals& retVals)
const {
	int curCost = 0, bottleNeck = INF, minCostFlow = 0;
	int counter;
	iMx costMx = modMixedWeightsMx, flowMx = maxFlowMx;
	vector<int> path;
	vector<i2Pair> edgesToRemove;

	while (flow) {
		curCost = 0;
		bottleNeck = INF;
		edgesToRemove.clear();

		path = RestorePaths(source, BellmanFord(source, costMx, counter),
costMx)[sink];
		retVals.paths.push_back(path);

		for (int i = path.size() - 1; i > 0; i--) {
			bottleNeck = std::min(bottleNeck, flowMx[path[i]][path[i - 1]]);
		}
		bottleNeck = std::min(bottleNeck, flow);
		retVals.flows.push_back(bottleNeck);

		for (int i = path.size() - 1; i > 0; i--) {
			flowMx[path[i]][path[i - 1]] -= bottleNeck;
			curCost += costMx[path[i]][path[i - 1]];
			if (flowMx[path[i]][path[i - 1]] == 0) {
				edgesToRemove.push_back(std::make_pair(path[i], path[i -
1]));
			}
		}
		retVals.costsPerPath.push_back(curCost);

		minCostFlow += bottleNeck * curCost;

		for (auto it = edgesToRemove.begin(); it != edgesToRemove.end(); ++it) {
			costMx[it->first][it->second] = INF;
		}

		flow -= bottleNeck;
	}
```

```
	retVals.modCostMx = costMx;
	retVals.modFlowMx = flowMx;

	return minCostFlow;
}
```

Эйлеров граф

Проверка, является ли граф эйлеровым, производится с помощью условия четности вершин. Далее пока степень вершин нечетна: если между вершинами есть ребро, то удалить его, если

нет – добавить.

Производить проверку на четность вершин до тех пор, пока все вершины не станут четными.

Вход: матрица весов

Выход: эйлеров граф и эйлеров цикл

```
vector<int> MyGraph::EulerCycles(iMx weightsMx, iMx& modWeightsMx, IsEulerOrHamilton&
isEulerRes) const {
	isEulerRes = IsEuler(weightsMx);
	if (isEulerRes == IsEulerOrHamilton::kFalse2Vert) {
		return vector<int>();
	}

	modWeightsMx = MakeUnoriented(weightsMx);
	vector<int> vertDeg = CalcDegrees(modWeightsMx);

	if (isEulerRes == IsEulerOrHamilton::kFalseModifiable) {
		bool isEuler = false, isChanged = false;
		int vertToConnect = -1;
		std::random_device rd;
		std::mt19937 mersenne(rd());

		while (!isEuler) {
			isChanged = false;
			for (int i = 0; i < modWeightsMx.size(); i++) {
				if ((vertDeg[i] % 2) == 1) {

					vertToConnect = -1;
					for (int j = 0; j < modWeightsMx.size(); j++) {
						if ((modWeightsMx[i][j] == 0) && (i != j)) {
							if (vertToConnect == -1) {

{

modWeightsMx.size() - 1) {

```

```
if (modWeightsMx.size() % 2 == 0)

	if (vertDeg[j] !=

		vertToConnect = j;
	}
}
else {
	vertToConnect = j;
}
```

```
                                        }
                                        if ((vertDeg[j] % 2) == 1) {
                                              isChanged = true;
                                              vertDeg[i]++;
                                              vertDeg[j]++;
                                              modWeightsMx[i][j] =
modWeightsMx[j][i] = mersenne() % (WEIGHT_MAX - 1) + 1;
                                              break;
                                        }
                                  }
                            }

                            if (!isChanged && (vertToConnect != -1)) {
                                  isChanged = true;
                                  vertDeg[i]++;
                                  vertDeg[vertToConnect]++;
                                  modWeightsMx[i][vertToConnect] =
modWeightsMx[vertToConnect][i] = mersenne() % (WEIGHT_MAX - 1) + 1;
                            }

                            if (vertToConnect == -1) {
                                  isEulerRes =
IsEulerOrHamilton::kFalseUnmodifiable;
                            }
                      }
                }

                if (!isChanged) {
                      isEuler = true;
                }
          }
    }

    if (isEulerRes == IsEulerOrHamilton::kFalseUnmodifiable) {
          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;
    std::stack<int> vertices;
    vertices.push(0);
    iMx weightsForDec = modWeightsMx;

    while (!vertices.empty()) {
          curVert = vertices.top();
          if (vertDeg[curVert] == 0) {
                vertices.pop();
```

```
                eulerPath.push_back(curVert);
            }
            else {
                for (int i = 0; i < vertexCnt; i++) {
                    if (weightsForDec[curVert][i] != 0) {
                        vertices.push(i);
                        vertDeg[i]--;
                        vertDeg[curVert]--;
                        weightsForDec[curVert][i] = 0;
                        weightsForDec[i][curVert] = 0;
                        break;
                    }
                }
            }
        }

        return eulerPath;
}
```

Гамильтонов граф

Проверка является ли граф гамильтоновым осуществляется на основе условия Дирака. Есть степень каждой вершины не меньше чем число вершин, деленное пополам, то граф называется графом Дирака. Каждый граф Дирака является гамильтоновым.

Если граф не является гамильтоновым:

Добавляем ребро из вершины, степень которой меньше p/2, где p – число вершин.

Далее снова проверяем на гамильтоновость.

Вход: матрица весов

Выход: весовая матрица гамильтонова графа и гамильтонов цикл

```
vector<int> MyGraph::Hamilton(iMx weightsMx, iMx& modWeightsMx, IsEulerOrHamilton&
isHamRes, int& minLen) const {
        if (weightsMx.size() == 2) {
            isHamRes = IsEulerOrHamilton::kFalse2Vert;
            return vector<int>();
        }


        modWeightsMx = MakeUnoriented(weightsMx);
        vector<int> vertDeg;
        bool isHamilton = true, isChanged = false;
        isHamRes = IsEulerOrHamilton::kTrue;
        int vertToConnect = -1;
        std::random_device rd;
        std::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()
% (WEIGHT_MAX - 1) + 1;
                        }
                  }
            }
      }
      else {
            vertDeg = CalcDegrees(modWeightsMx);
            for (int i = 0; i < weightsMx.size(); i++) {
                  if (vertDeg[i] < (vertexCnt / 2)) {
                        isHamilton = false;
                        isHamRes = IsEulerOrHamilton::kFalseModifiable;
                        break;
                  }
            }

            while (!isHamilton) {
                  isChanged = false;
                  for (int i = 0; i < weightsMx.size(); i++) {
                        vertToConnect = -1;
                        if (vertDeg[i] < (vertexCnt / 2)) {
                              for (int j = 0; j < weightsMx.size(); j++) {
                                    if ((modWeightsMx[i][j] == 0) && (i != j)) {
                                          vertToConnect = j;
                                          if (vertDeg[j] < (vertexCnt / 2)) {
                                                isChanged = true;
                                                vertDeg[i]++;
                                                vertDeg[j]++;
                                                modWeightsMx[i][j] =
modWeightsMx[j][i] = mersenne() % (WEIGHT_MAX - 1) + 1;
                                                break;
                                          }
                                    }
                              }
                              if (!isChanged && (vertToConnect != -1)) {
                                    isChanged = true;
                                    vertDeg[i]++;
                                    modWeightsMx[i][vertToConnect] =
modWeightsMx[vertToConnect][i] = mersenne() % (WEIGHT_MAX - 1) + 1;
                              }
                        }
                  }
                  if (!isChanged) {
                        isHamilton = true;
                  }
            }
      }

      ofstream ofs(OUTPUT_FILE_NAME);
      if (ofs.is_open()) {
      }
      vector<int> path;
      path.push_back(0);
      vector<int> minPath;
      int len = 0;
      minLen = INT_MAX;

      FindHamiltonCycles(ofs, modWeightsMx, path, minPath, len, minLen);
```

```
	ofs.close();
	return minPath;
}
```

Функция для поиска гамильтонова цикла. Она не возвращает значение, но находит и изменяет минимальный гамильтоновый цикл.

```
void MyGraph::FindHamiltonCycles(ofstream& ofs, iMx& weightsMx, vector<int>& path,
vector<int>& minPath, int& len, int& minLen) const {
	if (path.size() == vertexCnt) {
		if (weightsMx[path[path.size() - 1]][0] != 0) {
			len += weightsMx[path[path.size() - 1]][0];
			path.push_back(0);
			if (len < minLen) {
				minLen = len;
				minPath = path;
			}

			for (int i = 0; i < path.size() - 1; i++) {
				ofs << path[i] + 1 << " -> ";
			}
			ofs << path[path.size() - 1] + 1 << '\t';
			ofs << "Weight: " << 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 tmpLen = len + weightsMx[path[path.size() - 1]][i];
				path.push_back(i);
				FindHamiltonCycles(ofs, weightsMx, path, minPath, tmpLen,
minLen);
				path.pop_back();
			}
		}
	}
}
```

Задача коммивояжера

Решение задачи является минимальный по стоимости гамильтонов цикл. Поэтому сначала выполняется поиск всех гамильтоновых циклов и их стоимости, а затем из них выбирается цикл с минимальной стоимостью. Если граф не является гамильтоновым, то решить задачу не представляется возможным и мы дополняем граф до гамильтонова моделируя ситуацию.

Реализация лежит в поиске гамильтонова цикла. Функция Hamilton дополнена выводом в файл. Таким образом, выполняя поиск гамильтонова цикла все гамильтоновы циклы выведутся в файл, а наименьший станет решением задачи.

4 Пример работы программы

Генерация графа

```
Введите количество вершин графа n (2 <= n <= 50):
5

Матрица смежности вершин:
      1   2   3   4   5
  1   0   1   0   1   0
  2   0   0   1   0   1
  3   0   0   0   1   1
  4   0   0   0   0   1
  5   0   0   0   0   0
Матрица весов
      1   2   3   4   5
  1   0  14   0   2   0
  2   0   0   9   0   2
  3   0   0   0  11   5
  4   0   0   0   0   2
  5   0   0   0   0   0
```

Рисунок 4: Пример генерации графа

Метод Шимбелла

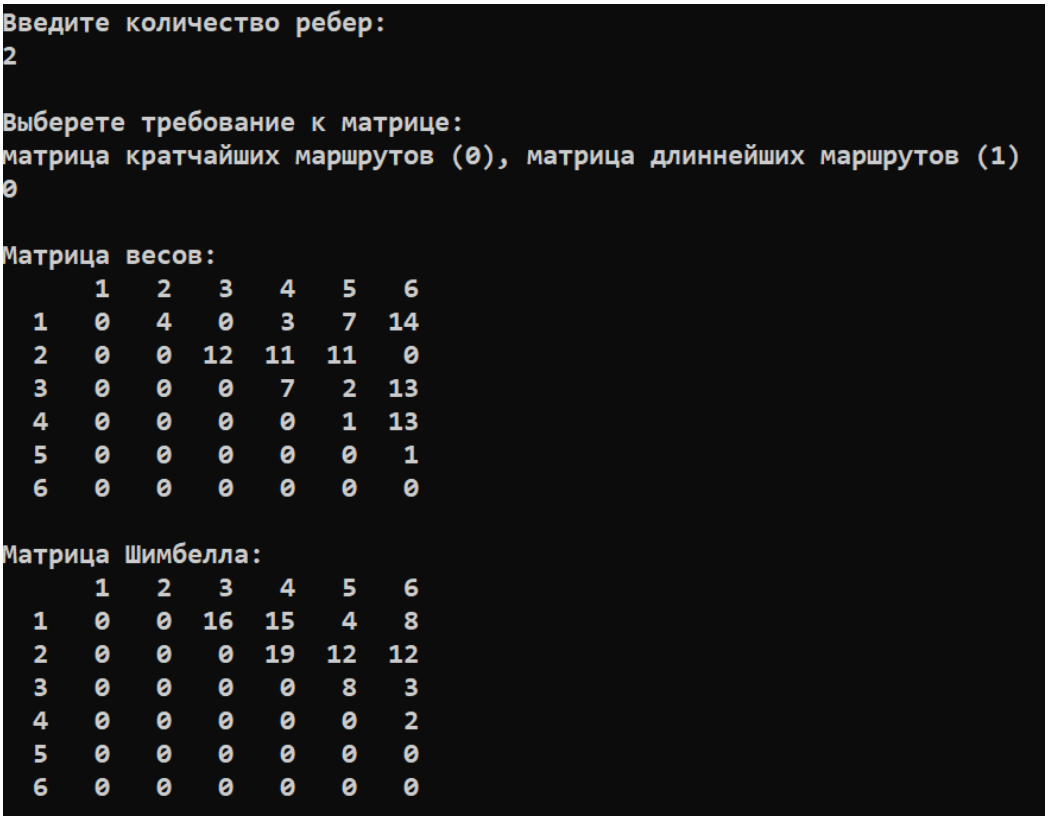


Рисунок 5: Пример работы метода Шимбелла

Определение возможности построения маршрута

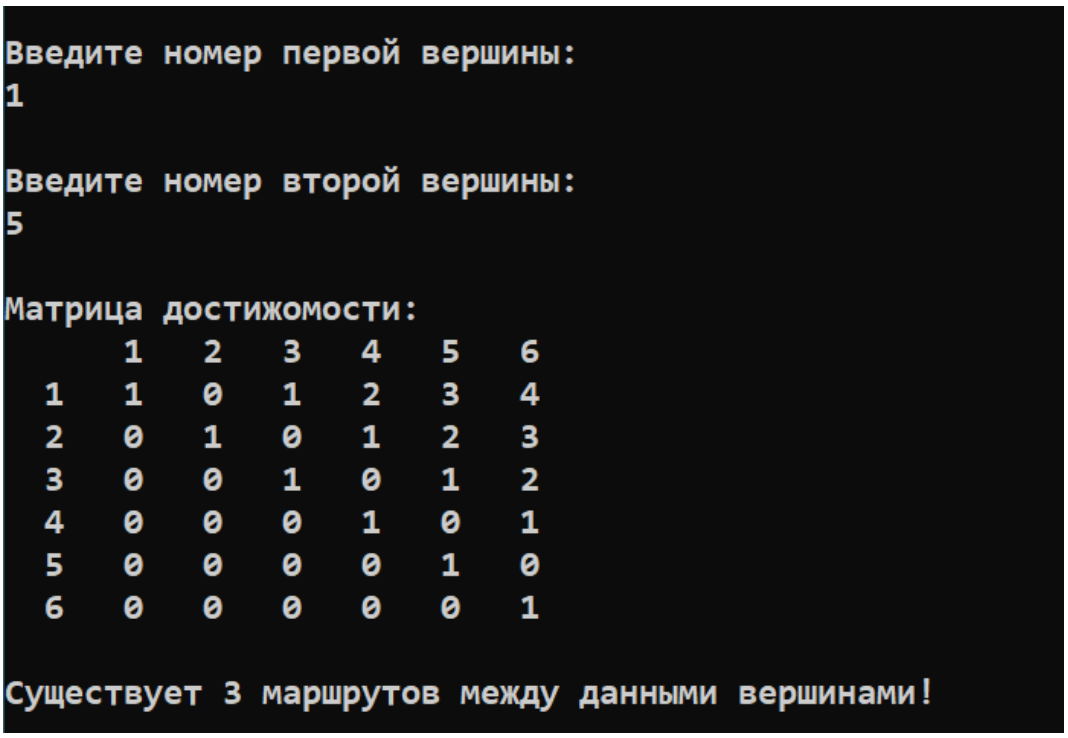


Рисунок 6: Пример определения возможности построения маршрута на том же графе

Алгоритм Дейкстры

```
Матрица весов:
     1   2   3   4   5   6
  1  0   4   0   3   7  14
  2  0   0  12  11  11   0
  3  0   0   0   7   2  13
  4  0   0   0   0   1  13
  5  0   0   0   0   0   1
  6  0   0   0   0   0   0

П Е Р Е Б О Р:
До вершины  1 пути нет!
Кратчайший путь длиной  8 до вершины  3:  2 ->  3
Кратчайший путь длиной  8 до вершины  4:  2 ->  4
Кратчайший путь длиной  9 до вершины  5:  2 ->  3 ->  5
Кратчайший путь длиной 15 до вершины  6:  2 ->  3 ->  6

Количество итераций: 72

О Ч Е Р Е Д Ь:
До вершины  1 пути нет!
Кратчайший путь длиной  8 до вершины  3:  2 ->  3
Кратчайший путь длиной  8 до вершины  4:  2 ->  4
Кратчайший путь длиной  9 до вершины  5:  2 ->  3 ->  5
Кратчайший путь длиной 15 до вершины  6:  2 ->  3 ->  6

Количество итераций: 11
```

Рисунок 7: Пример работы алгоритма Дейкстры на том же графе

Алгоритм Беллмана – Форда

```
Введите исходную вершину:
2

Хотите ли вы дополнить матрицу весов отрицательными значениями? (Y/N)
Y
Матрица весов:
     1   2   3   4   5   6
  1  0  10   0   8  12 -10
  2  0   0  -8  -8  10   0
  3  0   0   0 -14  -1  -7
  4  0   0   0   0   1  -7
  5  0   0   0   0   0  -9
  6  0   0   0   0   0   0

До вершины  1 пути нет!
Кратчайший путь длиной  -8 до вершины  3:  2 ->  3
Кратчайший путь длиной -22 до вершины  4:  2 ->  3 ->  4
Кратчайший путь длиной -21 до вершины  5:  2 ->  3 ->  4 ->  5
Кратчайший путь длиной -30 до вершины  6:  2 ->  3 ->  4 ->  5 ->  6

Количество итераций: 13
```

Рисунок 8: Пример работы алгоритма Беллмана - Форда с отрицательными значениями

Алгоритм Флойда – Уоршелла

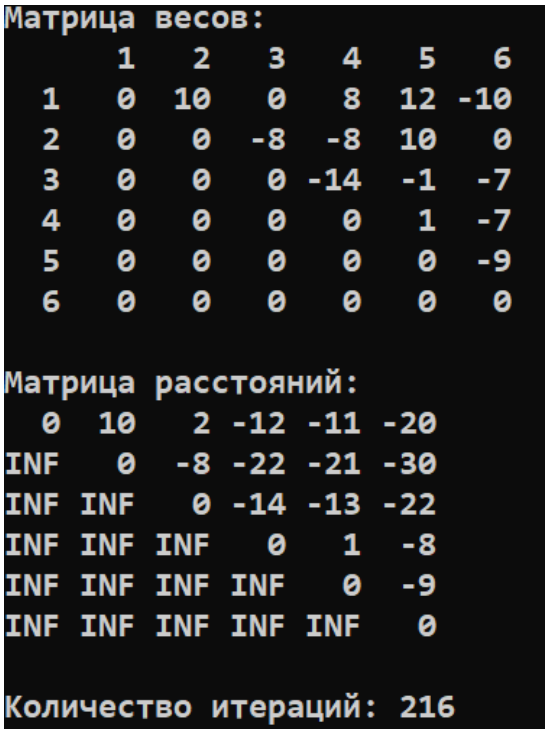


Рисунок 9: Алгоритм Флойда - Уоршелла с отрицательными весами

Поток минимальной стоимости

```
Матрица потоков:
      1   2   3   4   5   6
  1   0  17   0  16   2  19
  2   0   0   5  19  10   0
  3   0   0   0  19  17   1
  4   0   0   0   0   8   4
  5   0   0   0   0   0   6
  6   0   0   0   0   0   0

Матрица стоимостей за единицу потока:
      1   2   3   4   5   6
  1 999  10 999   8  12 -10
  2 999 999  -8  -8  10 999
  3 999 999 999 -14  -1  -7
  4 999 999 999 999   1  -7
  5 999 999 999 999 999  -9
  6 999 999 999 999 999 999

Поток величины 8 со стоимостью -8 за единицу потока
по пути: 2 -> 3
Итоговая стоимость: -64

Величина потока минимальной стоимости: -64
```

Рисунок 10: Пример нахождения потока минимальной стоимости между заданными вершинами

Алгоритм Краскала

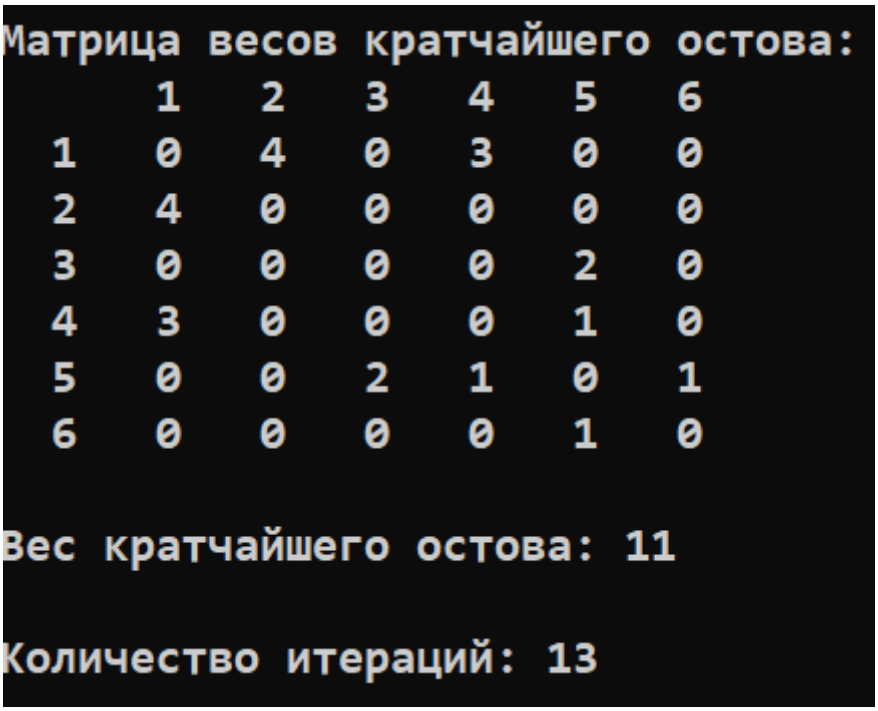


Рисунок 11: Пример работы алгоритма Краскала

Алгоритм Прима

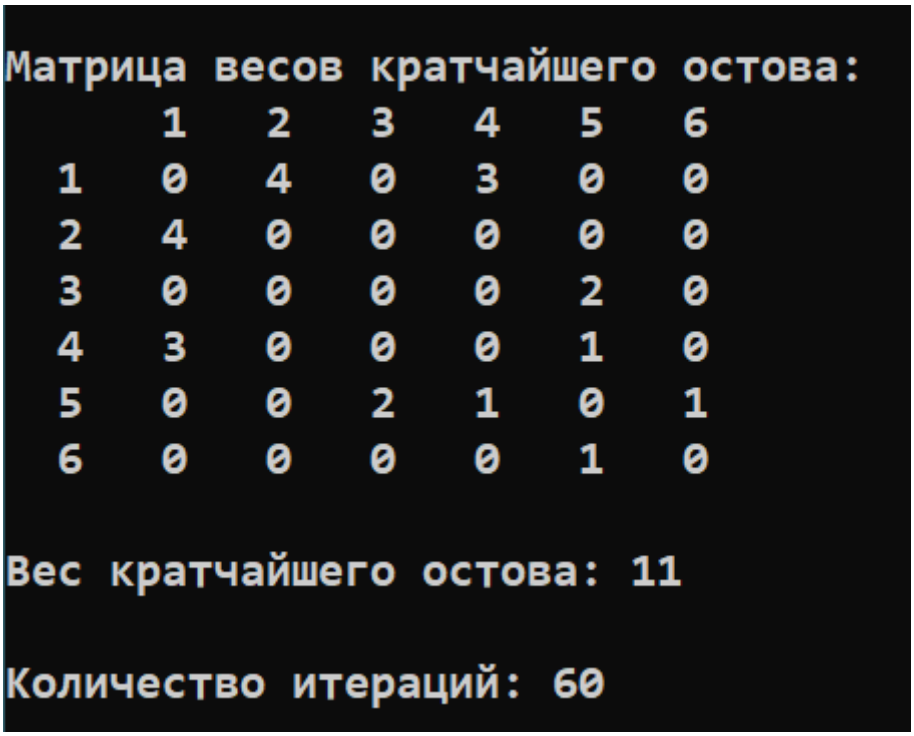


Рисунок 12: Пример работы алгоритма Прима

Подсчёт количества остовных деревьев в графе

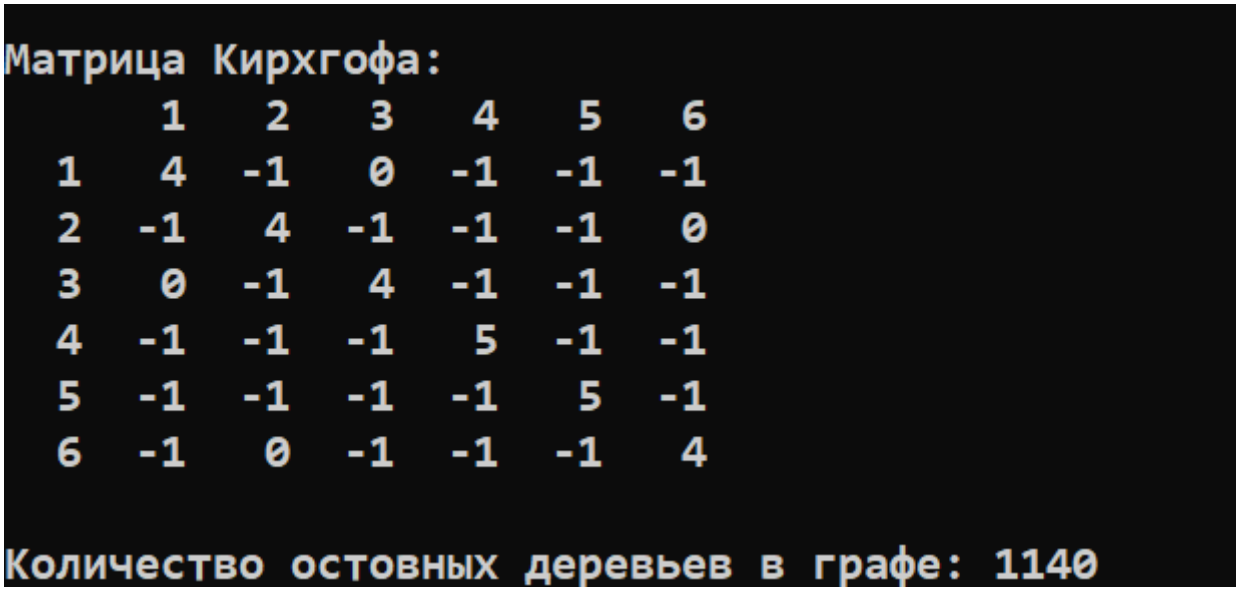


Рисунок 13: Подсчёт количества остовных деревьев в графе

Кодировка графа кодом Прюфера

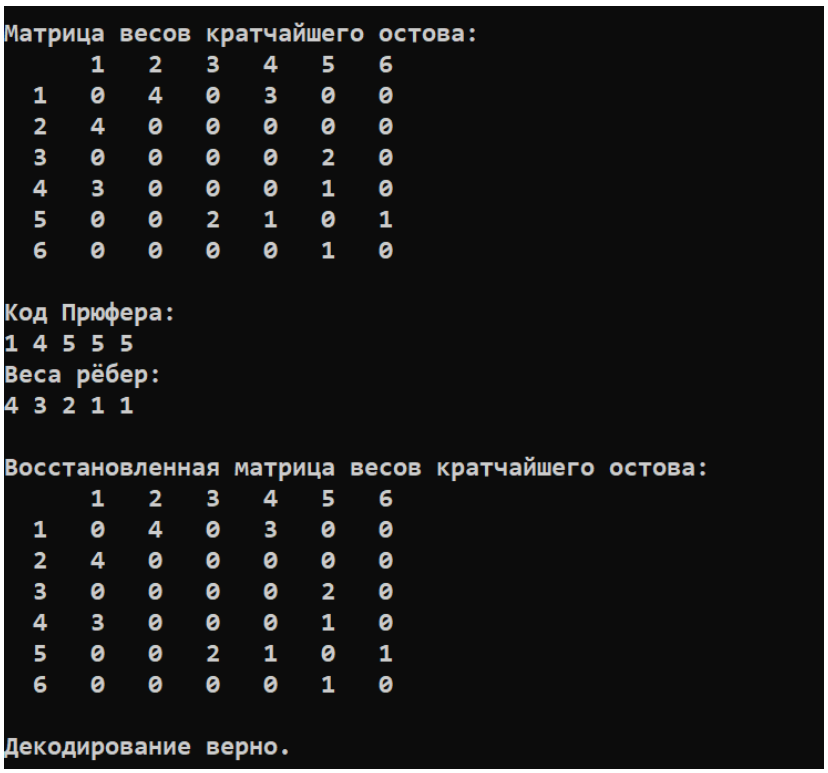


Рисунок 14: Пример кодирования и декодирования кода Прюфера

Поиск Эйлерова цикла

```
Граф не является эйлеровым. Модифицификация до эйлерова графа возможна.

Весовая матрица эйлерова графа:
     1   2   3   4   5   6   7   8
  1  0  14   0   8   5   0  14   0
  2 14   0   0  14  10   5   3   2
  3  0   0   0   0   0  13   8   0
  4  8  14   0   0   0   7   0   8
  5  5  10   0   0   0   3  13   0
  6  0   5  13   7   3   0  13   7
  7 14   3   8   0  13  13   0   7
  8  0   2   0   8   0   7   7   0

Эйлеров цикл:
1 -> 7 -> 8 -> 6 -> 7 -> 5 -> 6 -> 4 -> 8 -> 2 -> 7 -> 3 -> 6 -> 2 -> 5 -> 1 -> 4 -> 2 -> 1
```

Рисунок 15: Поиск Эйлерова цикла и модификация до Эйлерова графа

Поиск Гамильтонова цикла и решение задачи коммивояжера

```
Граф не является гамильтоновым. Модифицификация до гамильтонова графа возможна.

Весовая матрица гамильтонова графа:
     1   2   3   4   5   6   7   8
  1  0  12   8   0  11   0   8   0
  2 12   0   0   8   0   0  13   8
  3  8   0   0   0  13   6   8   0
  4  0   8   0   0   0  12  14   5
  5 11   0  13   0   0   4   8   0
  6  0   0   6  12   4   0   6   8
  7  8  13   8  14   8   6   0   9
  8  0   8   0   5   0   8   9   0

Минимальный гамильтонов цикл:
1 -> 2 -> 4 -> 8 -> 7 -> 5 -> 6 -> 3 -> 1
Вес пути: 60
```

Рисунок 16: Пример нахождения минимального гамильтонового цикла

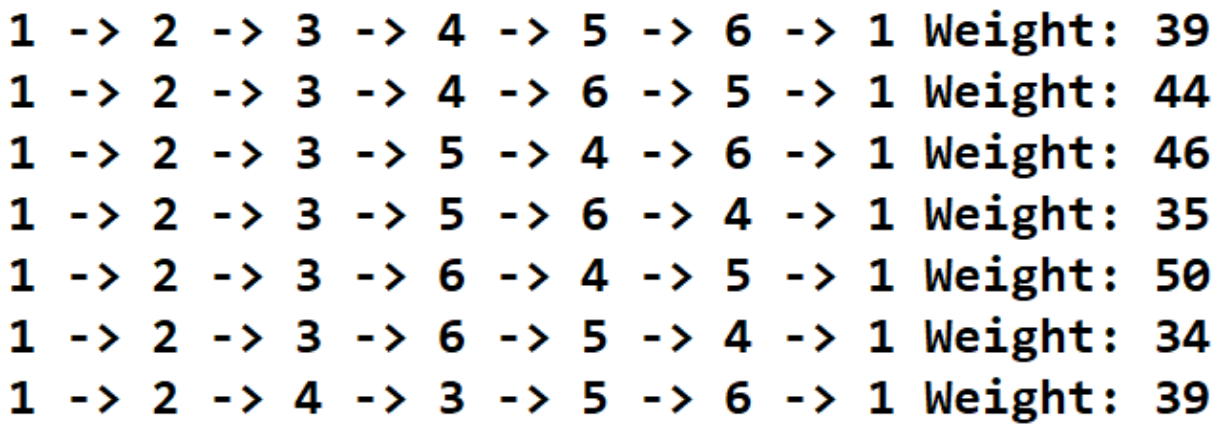


Рисунок 17: Пример части файла с перечисленными гамильтоновыми циклами

Заключение

В результате работы был реализован вспомогательный класс, представляющий собой модель матрицы графа. Также, были реализованы функции на основе существующих и разобранных алгоритмов.

1. Определение экстремальных путей. Метод Шимбелла
   В ходе работы был реализован алгоритм Шимбелла для поиска минимальных и максимальных путей. Алгоритм данного метода широко известен. Метод не является наиболее эффективным и уступает прочим алгоритмам по скорости выполнения и расходу памяти.
2. Поиска кратчайших путей в графе
   Были реализованы алгоритмы: Дейстры, Беллмана – Форда, Флойда – Уоршела.
   Алгоритм Дейстры является самым быстрым из перечисленных $(O(n^2))$, но имеет недостаток, так как может использоваться исключительно для графов с положительными весами.
   Алгоритм Беллмана – Форда в общем случае медленнее, чем алгоритм Дейкстры $((O(|E| * |V|))$, так как ребер в графе больше, чем вершин. Но, тем не менее, данный алгоритм может работать с отрицательными весами, при условии отсутствия отрицательных циклов.
   Алгоритм Флойда – Уоршела имеет самую высокую сложность среди всех перечисленных алгоритмов $(O(n^3))$, но не накладывает условия на веса ребер и ищет кратчайшие пути не от одной вершины до всех, а от всех вершин сразу же до всех.
3. Поиск остовов минимального веса. Код Прюфера.
   Были реализованы алгоритмы Прима и Краскала для поиска остовов минимального веса.
   Недостаток алгоритма Прима заключается в том, что в нем на каждом шаге необходимо просматривать все оставшиеся ребра для поиска минимального. В случае алгоритма Краскала его недостатком является то, что необходимо реализовывать хранение и сортировку списка ребер.
   С помощью кода Прюфера было произведено кодирование полученных остовных деревьев. Данный метод хранения остовов удобен для использования, так как является более минимальным представлением в виде n мерного массива вместо матрицы размера n*n. Однако, недостатком хранения является отсутствие возможности указать веса ребер.
4. Поиск максимального потока. Поиск потока минимальной стоимости.
   Для поиска максимального потока использовался алгоритм Форда – Фалкерсона. Данный алгоритм основан на теореме Форда – Фалкерсона. Для поиска потока минимальной стоимости используется ранее реализованный алгоритм Беллмана – Форда.
5. Поиск и построение эйлеровых и гамильтоновых циклов. Решение задачи коммивояжера.
   Эйлеровы графы строились в соответствии с теоремой Эйлера, гамильтоновы – в соответствии с ранее описанным условием Дирака. Было установлено, что поиск

эйлеровых циклов в графе является более простой задачей, чем гамильтоновых, так как для поиска эйлеровых циклов существует полиномиальный алгоритм, в то время как с гамильтоновыми циклами необходимо использовать перебор экспоненциальной сложности.
Для решения задачи коммивояжера находятся все гамильтоновы циклы и из них выбирается наименьший.

Список используемой литературы

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

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

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