Министерство образования и науки Российской Федерации

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ «САРАТОВСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

ИМЕНИ Н.Г. ЧЕРНЫШЕВСКОГО»

**ОТЧЕТ О ПРАКТИКЕ**

студента 3 курса 311 группы

направления 02.03.02 Фундаментальная информатика и информационные технологии

факультета компьютерных наук и информационных технологий

Живодерова Дмитрия Александровича

Саратов 2024

**СОДЕРЖАНИЕ**

[1. Класс граф 3](#_Toc182813847)

[2. Список смежности Ia.1 3](#_Toc182813848)

[3. Список смежности Ia.2 3](#_Toc182813849)

[4. Список смежности Iб 3](#_Toc182813850)

[5. Обходы графа II (обход в глубину) 3](#_Toc182813851)

[6. Обходы графа II (обход в ширину) 3](#_Toc182813852)

[7. Каркас III 3](#_Toc182813853)

[8. Веса IV a 3](#_Toc182813854)

[9. Веса IV b 3](#_Toc182813855)

[10. Веса IV c 3](#_Toc182813856)

[11. Максимальный поток 3](#_Toc182813857)

[12. Творческое задание 3](#_Toc182813858)

# **Класс граф**

Задание. Для решения всех задач курса необходимо создать класс (или иерархию классов - на усмотрение разработчика), содержащий:

1. Структуру для хранения списка смежности графа (не работать с графом через матрицы смежности, если в некоторых алгоритмах удобнее использовать список ребер - реализовать метод, создающий список рёбер на основе списка смежности);

2. Конструкторы (не менее 3-х):

* конструктор по умолчанию, создающий пустой граф
* конструктор, заполняющий данные графа из файла
* конструктор-копию (аккуратно, не все сразу делают именно копию)
* специфические конструкторы для удобства тестирования

3. Методы:

* добавляющие вершину,
* добавляющие ребро (дугу),
* удаляющие вершину,
* удаляющие ребро (дугу),
* выводящие список смежности в файл (в том числе в пригодном для чтения конструктором формате).

Не выполняйте некорректные операции, сообщайте об ошибках.

4. Должны поддерживаться как ориентированные, так и неориентированные графы. Заранее предусмотрите возможность добавления меток и\или весов для дуг. Поддержка мультиграфа не требуется.

5. Добавьте минималистичный консольный интерфейс пользователя (не смешивая его с реализацией!), позволяющий добавлять и удалять вершины и рёбра (дуги) и просматривать текущий список смежности графа.

6. Сгенерируйте не менее 4 входных файлов с разными типами графов (балансируйте на комбинации ориентированность-взвешенность) для тестирования класса в этом и последующих заданиях. Графы должны содержать не менее 7-10 вершин, в том числе петли и изолированные вершины.

Инициализация класса Graph:

class Graph

{

    constructor(directed = false, weighted = false)

    {

        this.adjacencyList = {};

        this.transposedList = {};

        this.directed = directed;

        this.weighted = weighted;

    }

    isGraph()

    {

        const direction = this.directed ? "ориентированный" : "НЕориентированный";

        const weight = this.weighted ? "взвешенный" : "НЕвзвешенный";

        return `${direction} | ${weight}`;

    }

}

Список смежности *adjacencyList* представляет собой структуру данных, используемую для хранения информации о графах в виде узлов и их соседей. Каждый узел связан со списком своих соседей, что подразумевает наличие рёбер между узлами. Данная структура предоставляет возможность учета весов ребер, что позволяет представлять как взвешенные, так и невзвешенные графы.

Реализация класса Graph:

    // Добавление вершины

    addVertex(vertex)

    {

        if (!this.adjacencyList[vertex])

        {

            this.adjacencyList[vertex] = [];

            console.log(`Вершина ${vertex} добавлена.`);

            this.transposedList[vertex] = [];

        }

        else console.log(`Вершина "${vertex}" уже существует`);

    }

    // Добавление ребра

    addEdge(edgeFrom, edgeTo, weight = 1)

    {

        let ok = true;

        if (!this.adjacencyList[edgeFrom])

        {

            console.log(`Вершины ${edgeFrom} не существует`);

            ok = false;

        }

        if (!this.adjacencyList[edgeTo])

        {

            console.log(`Вершины ${edgeTo} не существует`);

            ok = false;

        }

        if (ok)

        {

            const edgeExists = this.adjacencyList[edgeFrom].some(edge => edge.node === edgeTo);

            if (!edgeExists)

            {

                if (this.weighted)

                {

                        this.adjacencyList[edgeFrom].push({ node: edgeTo, weight });

                        this.transposedList[edgeTo].push({ node: edgeFrom }); // в инвертированный граф

                        if (!this.directed)

                        {

                            this.adjacencyList[edgeTo].push({ node: edgeFrom, weight });

                            this.transposedList[edgeFrom].push({ node: edgeTo }); // в инвертированный граф

                        }

                        console.log(`Ребро от ${edgeFrom} до ${edgeTo} с весом ${weight} добавлено.`);

                }

                else

                {

                    this.adjacencyList[edgeFrom].push({ node: edgeTo});

                    this.transposedList[edgeTo].push({ node: edgeFrom }); // в инвертированный граф

                    if (!this.directed)

                    {

                        this.adjacencyList[edgeTo].push({ node: edgeFrom});

                        this.transposedList[edgeFrom].push({ node: edgeTo }); // в инвертированный граф

                    }

                    console.log(`Ребро от ${edgeFrom} до ${edgeTo} добавлено.`);

                }

            }

            else console.log(`Данное ребро уже существует`);

        }

    }

    // Удаление вершины

    removeVertex(vertex)

    {

        if (!this.adjacencyList[vertex])

            console.log(`Вершины ${vertex} не существует.`);

        else

        {

            delete this.adjacencyList[vertex];

            for (const key in this.adjacencyList)

                this.adjacencyList[key] = this.adjacencyList[key].filter(edge => edge.node !== vertex);

            console.log(`Вершина ${vertex} удалена.`);

        }

    }

    // Удаление ребра

    removeEdge(edgeFrom, edgeTo)

    {

        let ok = true;

        if (!this.adjacencyList[edgeFrom])

        {

            console.log(`Вершины ${edgeFrom} не существует.`);

            ok = false;

        }

        if (!this.adjacencyList[edgeTo])

        {

            console.log(`Вершины ${edgeTo} не существует.`);

            ok = false;

        }

        if (ok)

        {

            const edgeExists = this.adjacencyList[edgeFrom].some(edge => edge.node === edgeTo);

            if (edgeExists)

            {

                if (this.adjacencyList[edgeFrom])

                    this.adjacencyList[edgeFrom] = this.adjacencyList[edgeFrom].filter(edge => edge.node !== edgeTo);

                if (this.transposedList[edgeTo])

                    this.transposedList[edgeTo] = this.transposedList[edgeTo].filter(edge => edge.node !== edgeFrom);

                if (!this.directed && this.adjacencyList[edgeTo])

                    this.adjacencyList[edgeTo] = this.adjacencyList[edgeTo].filter(edge => edge.node !== edgeFrom);

                console.log(`Ребро от ${edgeFrom} до ${edgeTo} удалено.`);

            }

            else console.log(`Данного ребра не существует`);

        }

    }

    // Вывод списка смежности

    toString()

    {

        let result = '';

        if (this.weighted)

        {

            for (const vertex in this.adjacencyList)

                result += `${vertex} -> ${this.adjacencyList[vertex].map(edge => `${edge.node} (${edge.weight})`).join(', ')}\n`;

        }

        else

        {

            for (const vertex in this.adjacencyList)

                result += `${vertex} -> ${this.adjacencyList[vertex].map(edge => `${edge.node}`).join(', ')}\n`;

        }

        return result;

    }

    // Импорт из файла

    static async fromFile(filePath)

    {

        const graph = new Graph();

        const fileStream = fs.createReadStream(filePath);

        const rl = readline.createInterface

        ({

            input: fileStream,

            crlfDelay: Infinity

        });

        let isDirected = false;

        let isWeighted = false;

        for await (const line of rl)

        {

            const trimmedLine = line.trim();

            if (trimmedLine.startsWith('#'))

            {

                if (trimmedLine.includes('dir\_w'))

                {

                    isDirected = true;

                    isWeighted = true;

                }

                else if (trimmedLine.includes('und\_w'))

                {

                    isDirected = false;

                    isWeighted = true;

                }

                else if (trimmedLine.includes('dir\_unw'))

                {

                    isDirected = true;

                    isWeighted = false;

                }

                else if (trimmedLine.includes('und\_unw'))

                {

                    isDirected = false;

                    isWeighted = false;

                }

                continue;

            }

            graph.directed = isDirected;

            graph.weighted = isWeighted;

            if (trimmedLine === '' || trimmedLine.startsWith('#')) continue;

            const parts = trimmedLine.split(',');

            if (parts.length < 2) continue;

            const edgeFrom = parts[0].trim();

            const edgeTo = parts[1].trim();

            const weight = parts.length === 3 ? parseInt(parts[2].trim()) : 1;

            if (edgeTo.length > 0) graph.addEdge(edgeFrom, edgeTo, weight);

            else graph.addVertex(edgeFrom);

        }

        return graph;

    }

    // Экспорт в файл

    exportToFile(filePath)

    {

        let result = '';

        const vertices = Object.keys(this.adjacencyList);

        result += vertices.join(',\n') + ',\n';

        if (this.weighted)

            for (const vertex in this.adjacencyList)

                for (const edge of this.adjacencyList[vertex])

                    result += `${vertex},${edge.node},${edge.weight}\n`;

        else

            for (const vertex in this.adjacencyList)

                for (const edge of this.adjacencyList[vertex])

                    result += `${vertex},${edge.node}\n`;

        fs.writeFile(filePath, result, (err) =>

        {

            if (err)

            {

                console.error('Ошибка при записи в файл:', err);

                handleUserInput();

            }

            else

            {

                console.log(`Список смежности успешно сохранен в файл: ${filePath}`);

                handleUserInput();

            }

        });

    }

    // Копирование графа

    copy()

    {

        const newGraph = new Graph(this.directed, this.weighted);

        for (const vertex in this.adjacencyList)

            newGraph.addVertex(vertex);

        for (const vertex in this.adjacencyList)

            for (const edge of this.adjacencyList[vertex])

                newGraph.addEdge(vertex, edge.node);

        return newGraph;

    }

    // Полный граф

    createCompleteGraph(numVertices, directed = false, weighted = false) {

        const graph = new Graph(directed, weighted);

        for (let i = 0; i < numVertices; i++)

            graph.addVertex(`V${i + 1}`);

        for (let i = 0; i < numVertices; i++)

            {

            for (let j = 0; j < numVertices; j++)

            {

                if (i !== j)

                {

                    const weight = weighted ? Math.floor(Math.random() \* 10) + 1 : 1; // Случайный вес от 1 до 10

                    graph.addEdge(`V${i + 1}`, `V${j + 1}`, weight);

                }

            }

        }

        return graph;

    }

Конструктор, принимающий строку, которая является путем до файла, строит граф на основе содержимого этого файла. Файл должен иметь следующий формат: на первой строке после знака # указана ориентированность и взвешенность графа в виде dir/und для ориентированности и w/unw для взвешенности, соединенные знаком нижнего подчеркивания (например: # dir\_w). На следующих строках должен задаваться список смежности, в формате: «A,B,N», где A и B – вершины, а N – вес ребра.

Конструктор копирования используется для создания копии существующего графа. Конструктор дублирует все данные, включая вершины и ребра.

Присутствуют также конструктор по умолчанию, создающий пустой граф с заданными атрибутами ориентированности и взвешенности, и специфический конструктор, создающий полный граф, с одним атрибутом – количество вершин.

Фнукция *addVertex* добавляет вершину в граф. В функции проводится проверка на отсутствие вершины в графе, и если она отсутствует, то функция добавляет вершину в список смежности, иначе выводится сообщение об ошибке «Вершина уже существует».

Функция *removeVertex* удаляет вершину из графа. В функции проводится проверка на наличие вершины в графе. Если вершина не найдена, выводится сообщение об ошибке «Вершины не существует». Иначе, она удаляется из списка смежности, а также из всех ребер, которые ссылаются на нее.

Функция *addEdge* предназначена для добавления ребра в граф между двумя вершинами. Она принимает три аргумента: начальная вершина, конечная вершина и вес ребра, по умолчанию равный 1. В функции проводится проверка, существуют ли указанные вершины в списке смежности. Если одна из вершин отсутствует, выводится сообщение об ошибке. Затем функция проверяет, существует ли уже ребро между этими вершинами. Если ребро не существует, оно добавляется в список смежности начальной вершины, и конечной, если граф неориентированный, а также в транспонированный список для обратного направления. В случае взвешенного графа сохраняется вес ребра.

Функция *removeEdge* удаляет ребро между двумя вершинами в графе. Сначала она проверяет, существуют ли указанные вершины в списке смежности. Если одна из вершин отсутствует, выводится сообщение об ошибке. Затем функция проверяет, существует ли ребро между этими вершинами. Если ребро найдено, оно удаляется из списка смежности начальной вершины, и конечной, если граф неориентированный, а также, из обратного ребра в транспонированном списке.

Реализация консольного интерфейса:

function showMenu()

{

    console.log('\n===================================');

    console.log(`Граф:  ${graph.isGraph()}`);

    console.log('===================================');

    console.log('Выберите действие:');

    console.log('1. Добавить вершину');

    console.log('2. Добавить ребро');

    console.log('3. Удалить вершину');

    console.log('4. Удалить ребро');

    console.log('5. Показать список смежности');

    console.log('6. Создать тестовый граф');

    console.log('7. Открыть файл и загрузить граф');

    console.log('8. Сохранить список смежности в файл');

    console.log('9. Сохранить список смежности в файл (визуал)');

    console.log('10. Сменить ориентированность');

    console.log('11. Сменить взвешенность');

    console.log('12. Сделать копию');

    console.log('----- Задания -----');

    console.log('13. Изолированые вершины орграфа');

    console.log('14. Полустепень исхода орграфа');

    console.log('15. Орграф, являющийся дополнением');

    console.log('-------------------');

    console.log('16. Обход в глубину');

    console.log('17. Обход в ширину');

    console.log('-------------------');

    console.log('18. Минимальное остовное дерево (Краскал)');

    console.log('19. Длины кратч. путей от u до всех остальных вершин (Дейкстры)');

    console.log('20. Кратч. пути из вершины u во все остальные вершины (Беллмана-Форда)');

    console.log('21. Вершина, каждая из минимальных стоимостей пути от которой до остальных не превосходит N (Флойд)');

    console.log('22. Максимальный поток');

    console.log('-------------------');

    console.log('0. Выйти');

    console.log('==============================\n');

}

// Функция обработки пользовательского ввода

function handleUserInput()

{

    showMenu();

    rl.question('---Введите номер действия: ', (choice) =>

    {

        switch (choice)

        {

            case '0': ...

            case '1': ...

            case '2': ...

            ...

            default: ...

        }

    });

}

# **Список смежности Ia (1)**

Вывести все изолированные вершины орграфа (степени 0).

Решением является функция *findIsolatedVertices,* которая использует две вспомогательные функции: *inDegree*: рассчитывает полустепень захода, и *outDegree*, рассчитывающая полустепень исхода.

    // поиск изолированных вершин

    findIsolatedVertices()

    {

        const isolatedVertices = [];

        for (const vertex in this.adjacencyList)

        {

            const degreeOut = this.outDegree(vertex);

            const degreeIn = this.inDegree(vertex);

            if (degreeOut === 0 && degreeIn === 0)

                isolatedVertices.push(vertex);

        }

        return isolatedVertices;

    }

Создаем пустой массив *isolatedVertices* для хранения изолированных вершин. С помощью цикла for вычисляем для каждой вершины ее полустепень исхода и захода. Если они равны нулю, то соответственно и степень равна нулю, значит добавляем вершину в массив.

    // полустепень захода

    inDegree(vertex)

    {

        if (!this.adjacencyList[vertex])

        {

            console.log(`Вершины ${vertex} не существует.`);

            return -1;

        }

        let count = 0;

        for (let v in this.adjacencyList)

            if (this.adjacencyList[v].some(edge => edge.node === vertex))

                count++;

        return count;

    }

Функция *inDegree* вычисляет, сколько входящих ребер есть у заданной вершины, то есть сколько других вершин указывают на нее. Сначала проверяем, существует ли вершина в списке смежности. Если нет, то вывод будет ошибкой. Иначе, с помощью цикла for, проходим по всем вершинам графа, и для каждой вершины проверяем, содержит ли ее массив соседей (исходящих связей) ребро, указывающее на нужную нам вершину.

Функция *outDegree* описана в следующем пункте.

# **3.** **Список смежности Ia (2)**

Вывести полустепень исхода данной вершины орграфа.

    // полустепень исхода

    outDegree(vertex)

    {

        if (!this.adjacencyList[vertex])

        {

            console.log(`Вершины ${vertex} не существует.`);

            return -1;

        }

        return this.adjacencyList[vertex].length;

    }

Функция *outDegree* вычисляет, сколько исходящих ребер есть у заданной вершины в графе. Она возвращает количество соседей вершины, то есть сколько других вершин соединены с ней исходящими связями. Также, как и алгоритме поиска полустепени захода, проверяем, существует ли вершина в списке смежности. Если вершина существует, просто считаем количество записей (соседей) в массиве, который соответствует этой вершине в списке смежности. Иначе, выводом будет ошибка.

# 4. **Список смежности Iб**

Построить орграф, являющийся дополнением данного орграфа.

С помощью функции *getComplement*, мы можем построить дополнение орграфа, что является графом с теми же вершинами, но содержащий все возможные направленные ребра, которых нет в исходном графе, за исключением петель.

    // дополнение орграфа

    getComplement() {

        const vertices = Object.keys(this.adjacencyList);

        for (let i = 0; i < vertices.length; i++)

        {

            for (let j = 0; j < vertices.length; j++)

            {

                if (i !== j)

                {

                    const fromVertex = vertices[i];

                    const toVertex = vertices[j];

                    this.addOrDelEdge(fromVertex, toVertex);

                }

            }

        }

    }

Из списка смежности извлекается массив всех вершин графа. Используются два цикла для перебора вершин, в котором внешний цикл проходит по всем вершинам, задавая начальную вершину, а внутренний цикл проходит по всем вершинам, задавая конечную вершину. Во внутреннем цикле происходит проверка на петлю, то есть если вершины не совпадают, то алгоритм продолжается, иначе ребро пропускается, так как петли не учитываются в дополнении.

addOrDelEdge(fromVertex, toVertex)

    {

        if (this.adjacencyList[fromVertex] && this.adjacencyList[toVertex])

        {

            const edgeIndex = this.adjacencyList[fromVertex].findIndex(edge => edge.node === toVertex);

            if (edgeIndex === -1) this.adjacencyList[fromVertex].push({ node: toVertex });

            else this.adjacencyList[fromVertex].splice(edgeIndex, 1);

        }

    }

Вызов функции *addOrDelEdge* добавляет ребро, если его нет, или удаляет, если оно уже существует. Таким образом, обеспечивается инверсия ребер графа.

# **5. Обходы графа II (обход в глубину)**

Найти путь, соединяющий вершины ***u1*** и ***u2*** и не проходящий через вершину ***v***.

Для решения данного задания, была реализована функция *findPath*.

    // Найти путь, соединяющий вершины u1 и u2 и не проходящий через вершину v

    findPath(u1, u2, v) {

        const visited = new Set();

        const path = [];

        const dfs = (current) =>

        {

            if (current === u2)

            {

                path.push(current);

                return true; // путь найден

            }

            visited.add(current);

            path.push(current);

            if (!this.adjacencyList[current])

            {

                console.log(`Вершина ${current} отсутствует в графе.`);

                return false;

            }

            for (const edge of this.adjacencyList[current])

            {

                const neighbor = edge.node;

                if (!visited.has(neighbor) && neighbor !== v)

                    if (dfs(neighbor))

                        return true;

            }

            path.pop();

            return false;

        };

        dfs(u1);

        return path.length > 0 && path[path.length - 1] === u2 ? path : null;

    }

Создается пустой набор *visited* для хранения посещенных вершин, чтобы избежать зацикливания, и массив *path* для хранения текущего пути. Реализация поиска в глубину представлена как вложенная функция *dfs*, которая выполняет рекурсивный обход графа от текущей вершины.

Если текущая вершина совпадает с конечной, то путь найден: текущая вершина добавляется в массив *path*, и функция возвращает true, чтобы остановить дальнейший поиск.

Текущая вершина добавляется в *visited* и *path*.

Проверяется наличие соседей у текущей вершины в списке смежности. И для каждого соседа проверяется: не посещен ли он ранее, не совпадает ли он с вершиной, которую нельзя включать в путь. Если сосед удовлетворяет условиям, то для продолжения поиска, рекурсивно вызывается *dfs* от соседа. Если в результате рекурсии путь найден, функция возвращает true. А если путь от текущей вершины не найден, она удаляется из *path*, и поиск возвращает false.

# **6. Обходы графа II (обход в ширину)**

Найти сильно связные компоненты орграфа.

Решение использует функции *findStronglyConnectedComponents*, *bfsFirstPass* и *bfsSecondPass*, которые реализуют алгоритм для поиска сильно связных компонент орграфа, и два прохода в ширину соответственно.

Первый проход необходим для того, чтобы определить порядок выхода вершин из обхода графа для дальнейшего анализа в транспонированном графе.

Второй проход необходим для того, чтобы определить сильно связные компоненты, используя транспонированный граф.

    // Найти сильно связные компоненты орграфа

    // Первый проход BFS для получения порядка выхода

    bfsFirstPass(vertex, visited, stack)

    {

        const queue = [vertex];

        visited.add(vertex);

        while (queue.length > 0)

        {

            const current = queue.shift();

            for (const edge of this.adjacencyList[current])

            {

                const neighbor = edge.node;

                if (!visited.has(neighbor))

                {

                    visited.add(neighbor);

                    queue.push(neighbor);

                }

            }

            stack.push(current);

        }

    }

В начале добавляем вершину в очередь и помечаем как посещенную. Далее, пока очередь не пуста, извлекаем текущую вершину из очереди, проходим по всем ее соседям (исходящим ребрам), используя список смежности. Если сосед еще не был посещен, то помечаем его как посещенного, и добавляем в очередь для дальнейшего обхода.

Каждую вершину, обработанную полностью, добавляем в стек. Этот стек фиксирует порядок выхода вершин, который будет использоваться во втором проходе.

    // Второй проход BFS для нахождения ССК

    bfsSecondPass(vertex, visited, component)

    {

        const queue = [vertex];

        visited.add(vertex);

        component.push(vertex);

        while (queue.length > 0)

        {

            const current = queue.shift();

            for (const edge of this.transposedList[current])

            {

                const neighbor = edge.node;

                if (!visited.has(neighbor))

                {

                    visited.add(neighbor);

                    queue.push(neighbor);

                    component.push(neighbor);

                }

            }

        }

    }

Начинаем обход с вершины в транспонированном графе. Добавляем ее в очередь, множество компонент, и помечаем как посещенную.

Пока очередь не пуста, извлекаем текущую вершину из очереди, проходим по всем ее соседям (входящим ребрам), используя список смежности транспонированного графа. Если сосед еще не был посещен, то помечаем его как посещенный, помещаем в очередь и записываем в текущую компоненту.

После завершения обхода сохраняем собранную компоненту как одну из сильно связных компонент графа.

    findStronglyConnectedComponents()

    {

        const visited = new Set();

        const stack = [];

        // Первый проход

        for (const vertex in this.adjacencyList)

            if (!visited.has(vertex))

                this.bfsFirstPass(vertex, visited, stack);

        // Второй проход

        visited.clear();

        const stronglyConnectedComponents = [];

        while (stack.length > 0)

        {

            const vertex = stack.shift();

            if (!visited.has(vertex))

            {

                const component = [];

                this.bfsSecondPass(vertex, visited, component);

                stronglyConnectedComponents.push(component);

            }

        }

        // Добавляем отдельные компоненты для изолированных вершин

        for (const vertex in this.adjacencyList)

            if (!visited.has(vertex))

                stronglyConnectedComponents.push([vertex]);

        return stronglyConnectedComponents;

    }

Создается множество для хранения посещенных вершин и стэк для сохранения порядка обработки вершин во время первого прохода.

Первый проход (по прямому графу): проходим по всем вершинам из списка смежности, если вершина еще не было посещена, то выполняется обход в ширину от текущей вершины, с помощью ранее описанной функции *bfsFirstPass*.

Второй проход (по транспонированному графу): очищаем множество посещенных вершин для нового обхода. Создаем массив для хранения найденных ССК. Пока стэк не пуст, извлекаем из него вершину. Если вершина еще не была посещена, то вызываем функцию *bfsSecondPass*, для обхода в ширину по транспонированному графу. Сохраняем найденную компоненту в массив ССК.

После второго прохода проверяем, остались ли еще не посещенные вершины, то есть те, которые изолированы. А каждая изолированная вершина формирует свою собственную ССК и добавляется в соответствующий массив, который является результатом.

# **7. Каркас III**

Дан взвешенный неориентированный граф из N вершин и M ребер. Требуется найти в нем каркас минимального веса.

Для решения используется алгоритм Краскала.

Решением является функция *findMinimumSpanningTree*.

    findMinimumSpanningTree()

    {

        const edges = [];

        for (let vertex in this.adjacencyList)

            for (let edge of this.adjacencyList[vertex])

                if (vertex < edge.node)

                    edges.push({ weight: edge.weight, vertices: [vertex, edge.node] });

        edges.sort((a, b) => a.weight - b.weight);

        const parent = {};

        const rank = {};

        for (let vertex in this.adjacencyList)

        {

            parent[vertex] = vertex;

            rank[vertex] = 0;

        }

        const find = (v) =>

        {

            if (parent[v] !== v) parent[v] = find(parent[v]);

            return parent[v];

        };

        const union = (v1, v2) =>

        {

            const root1 = find(v1);

            const root2 = find(v2);

            if (root1 !== root2)

            {

                if (rank[root1] > rank[root2]) parent[root2] = root1;

                else if (rank[root1] < rank[root2]) parent[root1] = root2;

                else

                {

                    parent[root2] = root1;

                    rank[root1]++;

                }

            }

        };

        const mstEdges = [];

        for (let edge of edges)

        {

            const { vertices: [v1, v2], weight } = edge;

            if (find(v1) !== find(v2))

            {

                union(v1, v2);

                mstEdges.push(edge);

            }

        }

        const mstGraph = new Graph(false, true);

        for (let edge of mstEdges)

        {

            const [v1, v2] = edge.vertices;

            mstGraph.addVertex(v1);

            mstGraph.addVertex(v2);

            mstGraph.addEdge(v1, v2, edge.weight);

        }

        return mstGraph;

    }

Происходит обход списка смежности для сбора всех ребер графа. Каждое ребро записывается в массив *edges* в виде объекта {вес, [вершина 1, вершина 2]}. При помощи условия *vertex < edge.node*, ребра добавляются только один раз. Массив ребер сортируется по возрастанию веса ребер.

Создаются два вспомогательных объекта: *parent* - для хранения родительских вершин, и *rank* – для оптимизации объединения по рангу. Сначала каждая вершина является своим собственным родителем, а ранг равен нулю.

Далее описываем функции для работы с объединением множеств: *find(v)* возвращает корневую вершину множества, к которому принадлежит вершина *v*. И *union(v1, v2)* – объединяет множества двух вершин, использует ранги для минимизации глубины дерева.

И начинаем формировать минимальное остовное дерево. Перебираются ребра из массива *edges* в порядке возрастания веса. Для каждого ребра проверяется, принадлежат ли его вершины разным множествам с помощью функции *find*. Если вершины принадлежат разным множества, то ребро добавляется в остовное дерево, а множества этих вершин объединяются с помощью *union*.

Создается новый объект графа, в него добавляются новые вершины и ребра из сформированного массива. Функция возвращает граф, представляющий минимальное остовное дерево.

# **8. Веса IV a**

Вывести длины кратчайших путей от u до всех остальных вершин.

Для решения используем алгоритм Дейкстры.

Алгоритм предназначен для взвешенных графов (с положительными весами ребер). Он использует подход, чтобы постепенно находить кратчайший путь к каждой вершине.

Начинаем с заданной стартовой вершины, устанавливая расстояние до самой себя равной 0, а до всех остальных вершин — бесконечность. Используем приоритетную очередь для обработки вершин, всегда выбирая вершину с минимальным текущим расстоянием. Обновляем расстояния до соседних вершин, если через текущую вершину найден путь короче. Продолжаем до тех пор, пока не обработаем все вершины.

Решение представлено в функции *deikstra*, с использованием дополнительного класса *MinPriorityQueue*, реализующий минимальную приоритетную очередь.

deikstra(start)

    {

        const distances = {};

        const priorityQueue = new MinPriorityQueue();

        for (let vertex in this.adjacencyList) distances[vertex] = Infinity;

        distances[start] = 0;

        priorityQueue.enqueue(start, 0);

        while (!priorityQueue.isEmpty())

        {

            const { element: currentVertex } = priorityQueue.dequeue();

            for (let neighbor of this.adjacencyList[currentVertex])

            {

                const { node: neighborVertex, weight } = neighbor;

                const newDistance = distances[currentVertex] + weight;

                // Если найдено более короткое расстояние до соседней вершины

                if (newDistance < distances[neighborVertex])

                {

                    distances[neighborVertex] = newDistance;

                    priorityQueue.enqueue(neighborVertex, newDistance);

                }

            }

        }

        return distances;

    }

Создается объект дистанций, где каждому узлу из графа устанавливается начальное расстояние – бесконечность, а расстояние до самой себя устанавливается в 0. Используется приоритетная очередь, чтобы отслеживать вершины с минимальными расстояниями, где стартовая вершина добавляется в очередь с приоритетом 0.

Дальше происходит обработка вершин. Пока очередь не пуста: извлекается вершина с минимальным текущим расстоянием, и для каждого соседа этой вершины рассчитывается потенциальное новое расстояние через текущую вершину. Если новое расстояние меньше текущего известного расстояния до соседа, то обновляется минимальное расстояние до соседа в объекте дистанций, и сосед добавляется в очередь с обновленным приоритетом.

После обработки всех вершин объект дистанций содержит минимальные расстояния от стартовой вершины до всех остальных.

class MinPriorityQueue

{

    constructor()

    {

        this.elements = [];

    }

    enqueue(element, priority)

    {

        this.elements.push({ element, priority });

        this.elements.sort((a, b) => a.priority - b.priority);

    }

    dequeue()

    {

        return this.elements.shift();

    }

    isEmpty()

    {

        return this.elements.length === 0;

    }

}

Данный класс позволяет работать с элементами в порядке их приоритета. Элементы с меньшим значением приоритета извлекаются первыми.

В конструкторе инициализируется пустой массив, который хранит элементы очереди. Каждый объект содержит: сам элемент, числовой параметр элемента (чем меньше значение, тем выше приоритет).

Метод *enqueue* добавляет новый элемент с указанным приоритетом в очередь. После добавления, выполняет сортировку массива элементов по возрастанию значения приоритета.

Таким образом, элемент с минимальным приоритетом всегда оказывается первым в очереди.

Метод *dequeue* удаляет и возвращает элемент с минимальным приоритетом (первый элемент в массиве). Если очередь пуста, возвращает undefined.

Метод *isEmpty* проверяет, пуста ли очередь, возвращая true если в массиве элементов нет элементов, и false в противном случае.

# 9. Веса IV b

# 10. Веса IV c

# 11. Максимальный поток

# 12. Творческое задание