МИНОБРНАУКИ РОССИИ
Федеральное государственное бюджетное образовательное учреждение
высшего образования

«САРАТОВСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ ИМЕНИ Н. Г. ЧЕРНЫШЕВСКОГО»

ТЕМА РАБОТЫ

РЕФЕРАТ

студента 3 курса 351 группы
направления 09.03.04 — Программная инженерия
факультета КНиИТ
Карасева Вадима Дмитриевича

Саратов 2025

СОДЕРЖАНИЕ

1 Минимальные требования для класса Граф

1.1 Условие

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 иили 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.

   Замечание:

В зависимости от выбранного способа хранения графа могут появиться дополнительные трудности при удалении-добавлении, например, необходимость переименования вершин, если граф хранится списком (например, vector C++, List C). Этого можно избежать, если хранить в списке пару (имя вершины, список смежных вершин), или хранить в другой структуре (например, Dictionary C, map в C++, при этом список смежности вершины может также храниться в виде словаря с ключами - смежными вершинами и значениями - весами соответствующих ребер). Идеально, если в качестве вершины реализуется обобщенный тип (generic), но достаточно использовать строковый тип или свой класс.

1.2 код (фрагменты кода)

```
struct Edge {
    string to;   // адрес вершины назначения
    int weight;  // вес ребра
    Edge(string t, int w = 1) : to(t), weight(w) {}
};

struct Point {
    string adress;          // имя вершины
    vector<Edge> adj;       // список смежных вершин (ребер)

    Point(string adr = "") : adress(adr) {}
};

class Graph {
private:
    bool directed;
public:
    vector<Point> adjList;

    // конструкторы
    Graph(bool dir = false) : directed(dir) {}
    Graph(const string& filePath, bool dir = false);
    Graph(const Graph& other);

    void addPoint(const string& name);
    void addEdge(const string& from, const string& to, int weight = 1);
    void removePoint(const string& name);
```

```
    void removeEdge(const string& from, const string& to);
    void printAdjList(const string& filePath) const;
    void saveToFile(const string& filePath) const;
    int findVertex(const string& name) const;
};

// реализация

Graph::Graph(const string& filePath, bool dir) : directed(dir) {
    ifstream fin(filePath);
    if (!fin.is_open()) throw runtime_error("Не удалось открыть файл");

    string from, to;
    int w;
    while (fin >> from >> to >> w) {
        addPoint(from);
        addPoint(to);
        addEdge(from, to, w);
    }
}

Graph::Graph(const Graph& other) : directed(other.directed),
adjList(other.adjList) {}

int Graph::findVertex(const string& name) const {
    for (int i = 0; i < (int)adjList.size(); i++)
        if (adjList[i].adress == name) return i;
    return -1;
}

// добавить вершину
void Graph::addPoint(const string& name) {
    if (findVertex(name) != -1) {
        cout << "Вершина \"" << name << "\" уже существует.\n";
        return;
    }
    adjList.push_back(Point(name));
    cout << "Вершина \"" << name << "\" успешно добавлена.\n";
}
```

```
// добавить ребро
void Graph::addEdge(const string& from, const string& to, int weight) {
    int i = findVertex(from);
    int j = findVertex(to);

    // проверяем существование вершин
    if (i == -1 && j == -1) {
        cout << "Вершины \"" << from << "\" и \"" << to << "\" не
существуют. Ребро добавить невозможно.\n";
        return;
    } else if (i == -1) {
        cout << "Вершина \"" << from << "\" не существует. Ребро
добавить невозможно.\n";
        return;
    } else if (j == -1) {
        cout << "Вершина \"" << to << "\" не существует. Ребро добавить
невозможно.\n";
        return;
    }

    // проверяем, существует ли уже ребро
    auto& edges = adjList[i].adj;
    bool exists = any_of(edges.begin(), edges.end(), [&](const Edge& e)
{ return e.to == to; });

    if (exists) {
        cout << "Ребро \"" << from << " -> " << to << "\" уже
существует. Добавление не выполнено.\n";
        return;
    }

    // добавляем ребро
    edges.push_back(Edge(to, weight));

    if (!directed && from != to) {
        adjList[j].adj.push_back(Edge(from, weight));
    }

    cout << "Ребро \"" << from << " -> " << to << "\" добавлено.\n";
}
```

```
// удалить вершину
void Graph::removePoint(const string& name) {
    int idx = findVertex(name);
    if (idx == -1) {
        cout << "Вершина \"" << name << "\" не существует.\n";
        return;
    }

    adjList.erase(adjList.begin() + idx);

    // удаляем все рёбра, ведущие к этой вершине
    for (auto& v : adjList) {
        v.adj.erase(remove_if(v.adj.begin(), v.adj.end(),
                              [&](Edge& e) { return e.to == name; }),
                    v.adj.end());
    }

    cout << "Вершина \"" << name << "\" удалена.\n";
}

// удалить ребро
void Graph::removeEdge(const string& from, const string& to) {
    int i = findVertex(from);
    int j = findVertex(to);

    // проверяем существование вершин
    if (i == -1 && j == -1) {
        cout << "Вершины \"" << from << "\" и \"" << to << "\" не
существуют. Ребро удалить невозможно.\n";
        return;
    } else if (i == -1) {
        cout << "Вершина \"" << from << "\" не существует. Ребро
удалить невозможно.\n";
        return;
    } else if (j == -1) {
        cout << "Вершина \"" << to << "\" не существует. Ребро удалить
невозможно.\n";
        return;
```

```
    }

    auto& edgesFrom = adjList[i].adj;
    auto it = remove_if(edgesFrom.begin(), edgesFrom.end(), [&](Edge&
e) { return e.to == to; });

    if (it == edgesFrom.end()) { // ребро не найдено
        cout << "Ребро \"" << from << " -> " << to << "\" не
существует.\n";
    } else {
        edgesFrom.erase(it, edgesFrom.end());
        cout << "Ребро \"" << from << " -> " << to << "\" удалено.\n";
    }

    if (!directed) {
        auto& edgesTo = adjList[j].adj;
        edgesTo.erase(remove_if(edgesTo.begin(), edgesTo.end(), [&]
(Edge& e) { return e.to == from; }),
                      edgesTo.end());
    }
}

void Graph::saveToFile(const string& filePath) const {
    ofstream fout(filePath);
    if (!fout.is_open()) throw runtime_error("Не удалось открыть
файл");

    for (const auto& v : adjList) {
        for (const auto& e : v.adj) {
            if (directed) {
                // для ориентированного графа сохраняем всё
                fout << v.adress << " " << e.to << " " << e.weight <<
"\n";
            } else {
                // для неориентированного графа:
                // записываем ребро, если from < to или это петля (from
== to)
                if (v.adress < e.to || v.adress == e.to) {
                    fout << v.adress << " " << e.to << " " << e.weight
<< "\n";
```

```
                }
            }
        }
    }
}

// вывести список смежности в файл
void Graph::printAdjList(const string& filePath) const {
    ofstream fout(filePath);
    if (!fout.is_open()) throw runtime_error("Cannot open file.");
    for (const auto& v : adjList) {
        fout << v.adress << ": ";
        for (const auto& e : v.adj)
            fout << "(" << e.to << "," << e.weight << ") ";
        fout << "\n";
    }
}

struct GraphRecord {
    string name;
    Graph* g;
};
```

1.3 Пример интерфейса в консоли

Примечание: в меню уже добавлены пункты, которые выполняли задания, которые были выданы на практике.

```
=== Меню ===
1. Создать новый пустой граф
2. Загрузить граф из файла
3. Переключиться на другой граф
4. Добавить вершину
5. Добавить ребро
6. Показать список смежности текущего графа
7. Сохранить текущий граф в файл
8. Удалить вершину
9. Удалить ребро
10. Найти вершину, в которую ведут дуги из u и v
11. Вывести степени всех вершин
12. Построить обращённый орграф
13. Классифицировать текущий граф
```

```
14. Найти вершины, до всех остальных достижимые за ≤ k шагов
15. Построить минимальный остров (Краскал)
16. Найти вершины, из которых все минимальные пути до остальных ≤ N
(Дейкстра)
17. Найти кратчайшие пути из заданной вершины (Беллман–Форд)
18. Определить N-периферию для заданной вершины (Флойд–Уоршелл)
19. Найти максимальный поток (Эдмондс–Карп)
0. Выход
Введите ваш выбор:

Введите ваш выбор: 2
Введите имя нового графа: aboba
Имя файла: graph2.txt
Ориентированный? (1 = да, 0 = нет): 1
Вершина "A" успешно добавлена.
Вершина "B" успешно добавлена.
Ребро "A -> B" добавлено.
Вершина "A" уже существует.
Вершина "C" успешно добавлена.
Ребро "A -> C" добавлено.
Вершина "B" уже существует.
Вершина "D" успешно добавлена.
Ребро "B -> D" добавлено.
Вершина "C" уже существует.
Вершина "D" уже существует.
Ребро "C -> D" добавлено.
Вершина "C" уже существует.
Вершина "E" успешно добавлена.
Ребро "C -> E" добавлено.
Вершина "E" уже существует.
Вершина "E" уже существует.
Ребро "E -> E" добавлено.
Вершина "F" успешно добавлена.
Вершина "G" успешно добавлена.
Ребро "F -> G" добавлено.
Вершина "H" успешно добавлена.
Вершина "H" уже существует.
Ребро "H -> H" добавлено.
Граф "aboba" загружен из graph2.txt и выбран как текущий.
```

```
Введите ваш выбор: 3
Доступные графы:
0. aboba (текущий)
Введите номер графа для переключения:

Введите ваш выбор: 4
Введите имя вершины: S
Вершина "S" успешно добавлена.

Введите ваш выбор: 5
Введите вершину-источник: A
Введите вершину-назначение: S
Введите вес ребра: 12
Ребро "A -> S" добавлено.

Введите ваш выбор: 6
Список смежности графа "aboba":
A: (B,4) (C,7) (S,12)
B: (D,3)
C: (D,6) (E,2)
D:
E: (E,9)
F: (G,5)
G:
H: (H,8)
S:

Введите ваш выбор: 7
Граф "aboba" сохранён в файл aboba_export.txt

Введите ваш выбор: 8
Введите вершину для удаления: S
Вершина "S" удалена.

Введите ваш выбор: 9
Введите вершину-источник: A
Введите вершину-назначение: S
Вершина "S" не существует. Ребро удалить невозможно.

Введите ваш выбор: 9
```

```
Введите вершину-источник: A
Введите вершину-назначение: B
Ребро "A -> B" удалено.

Введите ваш выбор: 0
Выход...
```

2 Список смежности Ia

2.1 Условие

Определить, существует ли вершина, в которую есть дуга как из вершины u, так и из вершины v. Вывести такую вершину.

2.2 Код (фрагменты кода)

```
// найти общую вершину назначения для двух вершин-источников
void Graph::findCommonTarget(const string& u, const string& v) const {
    int idxU = findVertex(u);
    int idxV = findVertex(v);

    if (idxU == -1 && idxV == -1) {
        cout << "Вершины \"" << u << "\" и \"" << v << "\" не
существуют.\n";
        return;
    } else if (idxU == -1) {
        cout << "Вершина \"" << u << "\" не существует.\n";
        return;
    } else if (idxV == -1) {
        cout << "Вершина \"" << v << "\" не существует.\n";
        return;
    }

    const auto& edgesU = adjList[idxU].adj;
    const auto& edgesV = adjList[idxV].adj;

    vector<string> common;
    for (const auto& e1 : edgesU) {
        for (const auto& e2 : edgesV) {
            if (e1.to == e2.to) {
                common.push_back(e1.to);
            }
        }
    }

    if (common.empty()) {
        cout << "Нет вершин, в которые идут дуги и из \"" << u << "\",
и из \"" << v << "\".\n";
    } else {
```

```
        cout << "Вершины, в которые идут дуги из \"" << u << "\" и \""
<< v << "\": ";
        for (const auto& name : common) {
            cout << name << " ";
        }
        cout << "\n";
    }
}
```

2.3 Краткое описание алгоритма

Данный алгоритм находит общие вершины назначения для двух заданных вершин в ориентированном графе.

2.3.1 Что делает

1. Проверяет существование вершин u и v в графе
2. Находит все исходящие дуги из обеих вершин
3. Ищет пересечение - вершины, в которые ведут дуги И из u, И из v

2.4 Примеры входных и выходных данных

2.4.1 Входные данные

```
A B 4
A C 7
B D 3
C D 6
C E 2
E E 9
F G 5
H H 8
```

2.4.2 Выходные данные

```
Введите имя вершины u: C
Введите имя вершины v: E
Вершины, в которые идут дуги из "C" и "E": E
```

3 Список смежности Ia

3.1 Условие

Для каждой вершины графа вывести её степень.

3.2 Код (фрагменты кода)

```
// вывести степени вершин
void Graph::printDegrees() const {
    cout << "\nСтепени вершин:\n";

    for (const auto& v : adjList) {
        int outDeg = v.adj.size(); // исходящая степень
        int inDeg = 0;             // входящая степень

        // считаем входящие рёбра
        for (const auto& u : adjList) {
            for (const auto& e : u.adj) {
                if (e.to == v.adress) {
                    inDeg++;
                }
            }
        }

        if (directed) {
            cout << v.adress << ": входящая = " << inDeg
                 << ", исходящая = " << outDeg << "\n";
        } else {
            // неориентированный граф: петля добавляет ещё 1
            int degree = outDeg;
            for (const auto& e : v.adj) {
                if (e.to == v.adress) degree++;
            }
            cout << v.adress << ": степень = " << degree << "\n";
        }
    }
}
```

3.3 Краткое описание алгоритма

Данный алгоритм вычисляет и выводит степени вершин графа.

3.3.1 Что делает

Для ориентированного графа:

1. Исходящая степень - количество дуг, исходящих из вершины (просто размер списка смежности)
2. Входящая степень - количество дуг, входящих в вершину (перебирает все рёбра всех вершин)

Для неориентированного графа:

1. Общая степень - количество инцидентных рёбер
2. Особый случай: петли учитываются дважды (по стандарту теории графов)

3.4 Примеры входных и выходных данных

3.4.1 Входные данные

```
A B 4
A C 7
B D 3
C D 6
C E 2
E E 9
F G 5
H H 8
```

3.4.2 Выходные данные

```
Степени вершин:
A: входящая = 0, исходящая = 2
B: входящая = 1, исходящая = 1
C: входящая = 1, исходящая = 2
D: входящая = 2, исходящая = 0
E: входящая = 2, исходящая = 1
F: входящая = 0, исходящая = 1
G: входящая = 1, исходящая = 0
H: входящая = 1, исходящая = 1
```

4 Список смежности Iб: несколько графов

4.1 Условие

Построить орграф, являющийся обращением данного орграфа (каждая дуга перевёрнута).

4.2 Код (фрагменты кода)

```
Graph Graph::getReversed() const {
    if (!directed) {
        throw runtime_error("Операция обращённого графа применима
только к ориентированным графам!");
    }

    Graph reversed(true); // создаём новый ориентированный граф

    // добавляем все вершины
    for (const auto& v : adjList) {
        reversed.addPoint(v.adress);
    }

    // добавляем рёбра в обратном направлении
    for (const auto& v : adjList) {
        for (const auto& e : v.adj) {
            reversed.addEdge(e.to, v.adress, e.weight);
        }
    }

    return reversed;
}
```

4.3 Краткое описание алгоритма

4.3.1 Что делает

1. Проверяет, что исходный граф ориентированный
2. Создает новый ориентированный граф
3. Копирует все вершины без изменений
4. Разворачивает все ребра - меняет направление дуг на противоположное

4.4 Примеры входных и выходных данных

4.4.1 Входные данные

```
A B 4
A C 7
B D 3
C D 6
C E 2
E E 9
F G 5
H H 8
```

4.4.2 Выходные данные

```
Обращённый граф создан. Его список смежности:
A:
B: (A,4)
C: (A,7)
D: (B,3) (C,6)
E: (C,2) (E,9)
F:
G: (F,5)
H: (H,8)
```

5 Обходы графа II

5.1 Условие

Проверить, является ли граф деревом, или лесом, или не является ни тем, ни другим.

5.2 Код (фрагменты кода)

```
    // вспомогательные: подсчёт числа вершин и рёбер
    // (для неориентированного учитываем каждое неориентир. ребро 1
раз)
    int vertexCount() const {
        return (int)adjList.size();
    }

    int edgeCount() const {
        int cnt = 0;
        for (const auto& v : adjList) cnt += (int)v.adj.size();
        if (!directed) cnt /= 2; // в неориентированном случае рёбра
хранятся дважды
        return cnt;
    }

    // DFS для подсчёта компонент (рассматриваем граф как
неориентированный)
    void dfsUndir(int v, vector<char>& used) const {
        used[v] = 1;
        for (const auto& e : adjList[v].adj) {
            int to = findVertex(e.to);
            if (to != -1 && !used[to]) dfsUndir(to, used);
        }
    }

    // проверка на циклы в неориентированном графе (DFS с родителем)
    bool hasCycleUndirUtil(int v, int parent, vector<char>& used) const
{
        used[v] = 1;
        for (const auto& e : adjList[v].adj) {
            int to = findVertex(e.to);
            if (to == -1) continue;
            if (!used[to]) {
```

```
                if (hasCycleUndirUtil(to, v, used)) return true;
            } else if (to != parent) {
                // нашли обратное посещённое ребро (и не родитель) ->
цикл
                return true;
            }
        }
        return false;
    }

    bool hasCycleUndir() const {
        int n = vertexCount();
        vector<char> used(n, 0);
        for (int i = 0; i < n; ++i) {
            if (!used[i]) {
                if (hasCycleUndirUtil(i, -1, used)) return true;
            }
        }
        return false;
    }

    // проверка на циклы в ориентированном графе
    // (DFS с раскраской: 0=white,1=gray,2=black)
    bool hasCycleDirUtil(int v, vector<int>& color) const {
        color[v] = 1; // gray
        for (const auto& e : adjList[v].adj) {
            int to = findVertex(e.to);
            if (to == -1) continue;
            if (color[to] == 0) {
                if (hasCycleDirUtil(to, color)) return true;
            } else if (color[to] == 1) {
                // нашли обратную (серую) вершину -> цикл
                return true;
            }
        }
        color[v] = 2; // black
        return false;
    }

    bool hasCycleDir() const {
```

```
    int n = vertexCount();
    vector<int> color(n, 0);
    for (int i = 0; i < n; ++i) {
        if (color[i] == 0) {
            if (hasCycleDirUtil(i, color)) return true;
        }
    }
    return false;
}

// подсчёт компонент (через неориентированный просмотр)
int countComponents() const {
    int n = vertexCount();
    vector<char> used(n, 0);
    int comps = 0;
    for (int i = 0; i < n; ++i) {
        if (!used[i]) {
            ++comps;
            dfsUndir(i, used);
        }
    }
    return comps;
}

// подсчёт входных степеней (для ориентированного графа)
vector<int> indegrees() const {
    int n = vertexCount();
    vector<int> indeg(n, 0);
    for (int i = 0; i < n; ++i) {
        for (const auto& e : adjList[i].adj) {
            int to = findVertex(e.to);
            if (to != -1) indeg[to]++;
        }
    }
    return indeg;
}

// проверка: неориентированный лес (acyclic)
bool isForestUndirected() const {
    // лес = ацикличный неориентированный граф
```

```
        return !hasCycleUndir();
    }

    bool isTreeUndirected() const {
        if (directed) return false;
        int n = vertexCount();
        if (n == 0) return false; // пустой граф — трактуем как не-
дерево (по задаче можно считать особым случаем)
        // дерево <=> связный и edges == n-1 (и ацикличный)
        int edges = edgeCount();
        if (edges != n - 1) return false;
        int comps = countComponents();
        return comps == 1 && !hasCycleUndir();
    }

    // ориентированная арборесценция (ориент. дерево с корнем)
    bool isArborescence() const {
        if (!directed) return false;
        int n = vertexCount();
        if (n == 0) return false;
        // 1) нет ориентированных циклов
        if (hasCycleDir()) return false;
        // 2) ровно один корень (indegree == 0), все остальные indeg ==
1
        auto indeg = indegrees();
        int rootCount = 0;
        for (int d : indeg) {
            if (d == 0) ++rootCount;
            else if (d != 1) return false;
        }
        if (rootCount != 1) return false;
        // 3) корень должен быть способен достичь все вершины (проверим
достижимость из найденного корня)
        int root = -1;
        for (int i = 0; i < n; ++i) if (indeg[i] == 0) { root = i;
break; }
        // BFS/DFS по ориентированным рёбрам
        vector<char> used(n, 0);
        // простой стек DFS:
        vector<int> st; st.push_back(root); used[root] = 1;
```

```
        while (!st.empty()) {
            int v = st.back(); st.pop_back();
            for (const auto& e : adjList[v].adj) {
                int to = findVertex(e.to);
                if (to != -1 && !used[to]) {
                    used[to] = 1;
                    st.push_back(to);
                }
            }
        }
        for (int i = 0; i < n; ++i) if (!used[i]) return false;
        return true;
    }

    // ориентированный лес арборесценций: нет ориентированных циклов и
indeg <= 1 for all vertices
    bool isDirectedForest() const {
        if (!directed) return false;
        if (hasCycleDir()) return false;
        auto indeg = indegrees();
        for (int d : indeg) if (d > 1) return false;
        return true;
    }

    // основная классификация: возвращает
    // "Tree", "Forest", "DirectedArborescence", "DirectedForest" или
"Other"
    string classify() const {
        if (!directed) {
            if (isTreeUndirected()) return "Tree";
            if (isForestUndirected()) return "Forest";
            return "Other";
        } else {
            if (isArborescence()) return "DirectedArborescence";
            if (isDirectedForest()) return "DirectedForest";
            return "Other";
        }
    }

    vector<string> verticesWithinK(int k) const {
```

```
    vector<string> result;
    int n = vertexCount();

    for (int i = 0; i < n; ++i) {
        vector<int> dist(n, -1); // -1 = не достигнута
        queue<int> q;
        dist[i] = 0;
        q.push(i);

        while (!q.empty()) {
            int v = q.front(); q.pop();
            for (const auto& e : adjList[v].adj) {
                int to = findVertex(e.to);
                if (to != -1 && dist[to] == -1) {
                    dist[to] = dist[v] + 1;
                    q.push(to);
                }
            }
        }
    }
}
```

5.3 Краткое описание алгоритма

Данный код содержит набор алгоритмов для анализа графов и их классификации. Основные функции:

5.3.1 Что делает

1. Базовые операции
    1. vertexCount() - количество вершин
    2. edgeCount() - количество ребер (для неориентированных делит на 2)
2. Поиск циклов
    1. hasCycleUndir() - проверяет циклы в неориентированном графе (DFS с родителем)
    2. hasCycleDir() - проверяет циклы в ориентированном графе (DFS с раскраской)
3. Анализ связности
    1. countComponents() - подсчет компонент связности (как неориентированный граф)

2. dfsUndir() - DFS для обхода компонент

4. Классификация графов

Для неориентированных графов:

1. isTreeUndirected() - проверяет, является ли деревом (связный + n-1 ребер + ацикличный)
2. isForestUndirected() - проверяет, является ли лесом (ацикличный)

Для ориентированных графов:

isArborescence() - ориентированное дерево с корнем:

1. Нет циклов
2. Один корень (входящая степень = 0)
3. Все остальные: входящая степень = 1
4. Корень достижим до всех вершин

isDirectedForest() - ориентированный лес:

1. Нет циклов
2. Входящая степень ≤ 1 для всех вершин

5. Основная классификация

classify() возвращает тип графа:

1. «Tree» - неориентированное дерево
2. «Forest» - неориентированный лес
3. «DirectedArborescence» - ориентированное дерево
4. «DirectedForest» - ориентированный лес
5. «Other» - другой тип

5.4 Примеры входных и выходных данных

5.4.1 Входные данные

```
A B 1
A C 1
B D 1
B E 1
C F 1
```

5.4.2 Выходные данные

```
Тип графа: DirectedArborescence
```

6 Обходы графа II

6.1 Условие

Вывести все вершины, длины кратчайших (по числу дуг) путей от которых до всех остальных не превосходят k.

6.2 Код (фрагменты кода)

```
vector<string> verticesWithinK(int k) const {
vector<string> result;
int n = vertexCount();

for (int i = 0; i < n; ++i) {
    vector<int> dist(n, -1); // -1 = не достигнута
    queue<int> q;
    dist[i] = 0;
    q.push(i);

    while (!q.empty()) {
        int v = q.front(); q.pop();
        for (const auto& e : adjList[v].adj) {
            int to = findVertex(e.to);
            if (to != -1 && dist[to] == -1) {
                dist[to] = dist[v] + 1;
                q.push(to);
            }
        }
    }

    // проверяем, все расстояния ≤ k
    bool ok = true;
    for (int d : dist) {
        if (d == -1 || d > k) {
            ok = false;
            break;
        }
    }

    if (ok) result.push_back(adjList[i].adress);
}
```

```
return result;
}
```

6.3 Краткое описание алгоритма

Данный алгоритм находит вершины, из которых все другие вершины достижимы на расстоянии ≤ K.

6.3.1 Что делает

Для каждой вершины графа:

1. Запускает BFS для вычисления кратчайших расстояний до всех других вершин
2. Проверяет, что все вершины достижимы и расстояние до них не превышает K
3. Добавляет вершину в результат, если условие выполняется

6.4 Примеры входных и выходных данных

6.4.1 Входные данные

```
A B 4
A C 2
B C 5
B D 10
C E 3
E D 4
D F 11
E F 5
F G 7
G H 1
H I 6
I J 2
C J 9
```

6.4.2 Выходные данные

```
Введите k: 3
Вершины, из которых все другие достижимы за ≤ 3 шагов: C E F J
```

7 Каркас III

7.1 Условие

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

Алгоритм, который необходимо реализовать для решения задачи (Прима или Краскала), выдает преподаватель.

7.2 Код (фрагменты кода)

```
void Graph::kruskalMST() const {
    // алгоритм Краскала работает только для неориентированных графов
    if (directed) {
        cout << "Kruskal: граф ориентированный — алгоритм применим
только к неориентированным графам.\n";
        return;
    }

    // вспомогательная запись ребра: индексы вершин + вес
    struct ERec { int u, v, w; };

    int n = (int)adjList.size();
    if (n == 0) {
        cout << "Граф пустой.\n";
        return;
    }

    // 1) собираем все рёбра (для неориентированного — только один раз:
i < j)
    vector<ERec> edges;
    for (int i = 0; i < n; ++i) {
        for (const auto& e : adjList[i].adj) {
            int j = findVertex(e.to);
            if (j == -1) continue; // защита на случай
неконсистентности
            if (i < j) { // добавляем только один экземпляр ребра для
неориентированного графа
                edges.push_back({i, j, e.weight});
            }
        }
```

```
    }

    if (edges.empty()) {
        cout << "В графе нет рёбер.\n";
        return;
    }

    // 2) сортируем рёбра по весу
    sort(edges.begin(), edges.end(), [](const ERec& a, const ERec& b) {
        return a.w < b.w;
    });

    // 3) DSU (Union-Find) по индексам 0..n-1
    struct DSU {
        vector<int> p, r;
        DSU(int n=0) { p.resize(n); r.assign(n,0); for (int i=0;i<n;+
+i) p[i]=i; }
        int find(int a) { return p[a]==a ? a : p[a]=find(p[a]); }
        bool unite(int a, int b) {
            a = find(a); b = find(b);
            if (a==b) return false;
            if (r[a] < r[b]) swap(a,b);
            p[b] = a;
            if (r[a]==r[b]) ++r[a];
            return true;
        }
    } dsu(n);

    // 4) построим MST в новом графе mst
    Graph mst(false); // неориентированный
    for (const auto& pt : adjList) mst.addPoint(pt.adress); // добавим
все вершины в MST

    int totalWeight = 0;
    vector<ERec> mstEdges;

    cout << "\nАлгоритм Краскала\n";
    for (const auto& er : edges) {
        if (dsu.find(er.u) != dsu.find(er.v)) {
            dsu.unite(er.u, er.v);
```

```
            mst.addEdge(adjList[er.u].adress, adjList[er.v].adress,
er.w);
            mstEdges.push_back(er);
            totalWeight += er.w;
            cout << "Добавлено ребро: " << adjList[er.u].adress
                 << " - " << adjList[er.v].adress
                 << " (вес = " << er.w << ")\n";
        }
    }

    cout << "Суммарный вес минимального остова: " << totalWeight <<
"\n";

    // сохраним результат в файл
    try {
        mst.saveToFile("mst_output.txt");
        cout << "MST сохранён в mst_output.txt\n";
    } catch (const exception& ex) {
        cout << "Не удалось сохранить MST в файл: " << ex.what() <<
"\n";
    }
}
```

7.3 Краткое описание алгоритма

Данный алгоритм реализует алгоритм Краскала для построения минимального остовного дерева (MST).

7.3.1 Что делает

Находит минимальное остовное дерево - подграф, который:

Содержит все вершины исходного графа

Является деревом (связный и ацикличный)

Имеет минимальный суммарный вес рёбер

Шаги алгоритма:

1. Проверка условий

   Работает только для неориентированных графов

   Проверяет, что граф не пустой

2. Сбор всех рёбер

   Собирает все рёбра графа, избегая дублирования (только i < j)

Каждое ребро: (индекс_вершины1, индекс_вершины2, вес)

3. Сортировка рёбер

   Сортирует рёбра по возрастанию веса

4. Система непересекающихся множеств (DSU)

   Использует Union-Find для эффективной проверки циклов

   find() - находит представителя множества

   unite() - объединяет два множества

5. Построение MST

   Проходит по отсортированным рёбрам

   Добавляет ребро в MST, если оно не создаёт цикл

   Выводит процесс добавления рёбер

6. Сохранение результата

   Сохраняет полученное MST в файл mst_output.txt

7.4 Примеры входных и выходных данных

7.4.1 Входные данные

```
A B 4
A C 2
B C 5
B D 10
C E 3
E D 4
D F 11
E F 5
F G 7
G H 1
H I 6
I J 2
C J 9
```

7.4.2 Выходные данные

```
Алгоритм Краскала
Ребро "G -> H" добавлено.
Добавлено ребро: G - H (вес = 1)
Ребро "A -> C" добавлено.
Добавлено ребро: A - C (вес = 2)
Ребро "I -> J" добавлено.
```

```
Добавлено ребро: I - J (вес = 2)
Ребро "C -> E" добавлено.
Добавлено ребро: C - E (вес = 3)
Ребро "A -> B" добавлено.
Добавлено ребро: A - B (вес = 4)
Ребро "D -> E" добавлено.
Добавлено ребро: D - E (вес = 4)
Ребро "E -> F" добавлено.
Добавлено ребро: E - F (вес = 5)
Ребро "H -> I" добавлено.
Добавлено ребро: H - I (вес = 6)
Ребро "F -> G" добавлено.
Добавлено ребро: F - G (вес = 7)
Суммарный вес минимального остова: 34
MST сохранён в mst_output.txt
```

8 Веса IV а

8.1 Условие

Определить, есть ли в графе вершина, каждая из минимальных стоимостей пути от которой до остальных не превосходит N.

8.2 Код (фрагменты кода)

```
void Graph::verticesAllDistances() const {
    if (adjList.empty()) {
        cout << "Граф пуст.\n";
        return;
    }

    string startName;
    cout << "Введите начальную вершину: ";
    cin >> startName;

    int s = findVertex(startName);
    if (s == -1) {
        cout << "Вершина \"" << startName << "\" не найдена.\n";
        return;
    }

    const int INF = INT_MAX / 4;
    int n = vertexCount();
    vector<int> dist(n, INF);
    vector<int> parent(n, -1); // опционально: чтобы восстановить пути
    dist[s] = 0;

    // min-куча: (dist, vertex_index)
    priority_queue<pair<int,int>, vector<pair<int,int>>,
greater<pair<int,int>>> pq;
    pq.push({0, s});

    while (!pq.empty()) {
        auto [d, v] = pq.top(); pq.pop();
        if (d != dist[v]) continue; // устаревшая запись в куче

        // проходим все вершины-соседи v (используем adjList)
        for (const auto& e : adjList[v].adj) {
```

```
            int to = findVertex(e.to);
            if (to == -1) continue; // защита от неконсистентности
            int w = e.weight;
            if (dist[v] != INF && dist[v] + w < dist[to]) {
                dist[to] = dist[v] + w;
                parent[to] = v;
                pq.push({dist[to], to});
            }
        }
    }

    // вывод расстояний
    cout << "\nКратчайшие расстояния от вершины " << startName << ":
\n";
    for (int i = 0; i < n; ++i) {
        cout << adjList[i].adress << " : ";
        if (dist[i] == INF) cout << "недостижима\n";
        else cout << dist[i] << "\n";
    }

    // если тебе нужно проверить условие "все расстояния <= N",
    // можно спросить N и проверить:
    char ask;
    cout << "\nПроверить, что все расстояния ≤ N? (y/n): ";
    cin >> ask;
    if (ask == 'y' || ask == 'Y') {
        int N;
        cout << "Введите N: ";
        cin >> N;
        bool ok = true;
        for (int i = 0; i < n; ++i) {
            if (i == s) continue;
            if (dist[i] == INF || dist[i] > N) { ok = false; break; }
        }
        if (ok) cout << "Все расстояния от " << startName << " до
остальных ≤ " << N << "\n";
        else cout << "Не все расстояния ≤ " << N << "\n";
    }
}
```

8.3 Краткое описание алгоритма

Находит кратчайшие расстояния от заданной стартовой вершины до всех других вершин в графе.

8.3.1 Что делает

1. Инициализация

   Запрашивает у пользователя начальную вершину

   Проверяет её существование

   Инициализирует массив расстояний INF (бесконечность)

   Устанавливает расстояние до стартовой вершины = 0

2. Приоритетная очередь

   Использует min-кучу для эффективного выбора вершины с минимальным расстоянием

   Хранит пары (расстояние, индекс_вершины)

3. Основной цикл Дейкстры

   Извлекает вершину с минимальным расстоянием

   Для каждого соседа: проверяет, можно ли улучшить расстояние

   Если да - обновляет расстояние и добавляет в очередь

4. Вывод результатов

   Показывает расстояния до всех вершин

   Отмечает недостижимые вершины

5. Дополнительная проверка

   Спрашивает пользователя, проверить ли условие «все расстояния ≤ N»

   Выводит результат проверки

8.4 Примеры входных и выходных данных

8.4.1 Входные данные

```
A B 4
A C 2
B C 5
B D 10
C E 3
E D 4
D F 11
E F 5
```

```
F G 7
G H 1
H I 6
I J 2
C J 9
```

8.4.2 Выходные данные

```
Введите начальную вершину: A

Кратчайшие расстояния от вершины A:
A : 0
B : 4
C : 2
D : 9
E : 5
F : 10
G : 17
H : 18
I : 24
J : 11

Проверить, что все расстояния ≤ N? (y/n): y
Введите N: 10
Не все расстояния ≤ 10
```

9 Веса IV b

9.1 Условие

Вывести кратчайшие пути из вершины u во все остальные вершины.

9.2 Код (фрагменты кода)

```
void Graph::bellmanFord(const string& start) {
    int n = vertexCount();
    int startIndex = findVertex(start);

    if (startIndex == -1) {
        cout << "Вершина " << start << " не найдена.\n";
        return;
    }

    const long long INF = LLONG_MAX / 4;
    vector<long long> dist(n, INF);
    dist[startIndex] = 0;

    // Алгоритм Беллмана–Форда
    for (int i = 0; i < n - 1; ++i) {
        bool updated = false;
        for (int u = 0; u < n; ++u) {
            if (dist[u] == INF) continue;
            for (const auto& e : adjList[u].adj) {
                int v = findVertex(e.to);
                if (v == -1) continue;
                long long w = e.weight;
                if (dist[u] + w < dist[v]) {
                    dist[v] = dist[u] + w;
                    updated = true;
                }
            }
        }
        if (!updated) break; // оптимизация
    }

    // Проверка на отрицательные циклы (опционально)
    for (int u = 0; u < n; ++u) {
        if (dist[u] == INF) continue;
```

```
        for (const auto& e : adjList[u].adj) {
            int v = findVertex(e.to);
            if (v == -1) continue;
            if (dist[u] + e.weight < dist[v]) {
                cout << "Граф содержит цикл отрицательного веса!\n";
                return;
            }
        }
    }

    // Вывод результатов
    cout << "Кратчайшие расстояния от вершины " << start << ":\n";
    for (int i = 0; i < n; ++i) {
        cout << adjList[i].adress << " : ";
        if (dist[i] == INF)
            cout << "недостижима\n";
        else
            cout << dist[i] << "\n";
    }
}
```

9.3 Краткое описание алгоритма

Этот алгоритм реализует алгоритм Беллмана-Форда для поиска кратчайших путей от одной вершины до всех остальных.

9.3.1 Что делает

Основная цель:

Находит кратчайшие расстояния от заданной вершины до всех других вершин в графе, работая с отрицательными весами рёбер. Шаги алгоритма:

1. Инициализация

   Находит стартовую вершину

   Инициализирует массив расстояний INF (бесконечность)

   Устанавливает расстояние до стартовой вершины = 0

2. Основной цикл релаксации

   Выполняет n-1 итераций (максимальная длина пути без циклов)

   Для каждого ребра u→v проверяет: dist[u] + weight < dist[v]

   Если да - обновляет расстояние до v

   Оптимизация: досрочный выход, если на итерации не было изменений

3. Проверка на отрицательные циклы

   Выполняет дополнительную итерацию релаксации

   Если можно улучшить расстояние - обнаруживает отрицательный цикл

4. Вывод результатов

   Показывает расстояния до всех вершин

   Отмечает недостижимые вершины

9.4 Примеры входных и выходных данных

9.4.1 Входные данные

```
A B 2
A C 5
B C 1
B D 4
C E 3
D E 2
E F 6
B A -1
```

9.4.2 Выходные данные

```
Введите имя начальной вершины: A
Кратчайшие расстояния от вершины A:
A : 0
B : 2
C : 3
D : 6
E : 6
F : 12
```

10 Веса IV с

10.1 Условие

N-периферией для вершины называется множество вершин, расстояние от которых до заданной вершины больше N. Определить N-периферию для заданной вершины графа.

10.2 Код (фрагменты кода)

```
void Graph::floydPeriphery(const string& start, int N) const {
    int n = adjList.size();
    if (n == 0) {
        cout << "Граф пуст.\n";
        return;
    }

    // Сопоставляем вершины индексам
    unordered_map<string, int> idx;
    for (int i = 0; i < n; ++i)
        idx[adjList[i].adress] = i;

    if (!idx.count(start)) {
        cout << "Вершина " << start << " не найдена в графе.\n";
        return;
    }

    // Инициализация матрицы расстояний
    vector<vector<int>> dist(n, vector<int>(n, INT_MAX / 2));

    for (int i = 0; i < n; ++i)
        dist[i][i] = 0;

    for (int i = 0; i < n; ++i)
        for (const auto& e : adjList[i].adj)
            dist[i][idx[e.to]] = min(dist[i][idx[e.to]], e.weight);

    // Алгоритм Флойда–Уоршелла
    for (int k = 0; k < n; ++k)
        for (int i = 0; i < n; ++i)
            for (int j = 0; j < n; ++j)
                if (dist[i][j] > dist[i][k] + dist[k][j])
```

```
                dist[i][j] = dist[i][k] + dist[k][j];

    // Определяем N-периферию
    int s = idx[start];
    vector<string> periphery;
    for (int i = 0; i < n; ++i)
        if (dist[s][i] > N && dist[s][i] < INT_MAX / 2)
            periphery.push_back(adjList[i].adress);

    // Вывод результата
    cout << "N-периферия вершины " << start << " (N = " << N << "): ";
    if (periphery.empty())
        cout << "пусто\n";
    else {
        for (const auto& v : periphery)
            cout << v << " ";
        cout << "\n";
    }
}
```

10.3 Краткое описание алгоритма

Этот алгоритм находит N-периферию вершины с использованием алгоритма Флойда-Уоршелла.

10.3.1 Что делает

Находит все вершины, расстояние до которых от стартовой вершины превышает заданное значение N.

Шаги алгоритма:

1. Подготовка данных

   Создает словарь для сопоставления имен вершин с индексами

   Проверяет существование стартовой вершины

2. Инициализация матрицы расстояний

   Заполняет матрицу большими значениями (INT_MAX / 2)

   Диагональ (расстояние до себя) = 0

   Заполняет прямые связи из списка смежности

3. Алгоритм Флойда-Уоршелла

Тройной цикл для нахождения кратчайших путей между всеми парами вершин

Для каждой тройки (i, j, k) проверяет: dist[i][j] > dist[i][k] + dist[k][j]

Если да - улучшает расстояние

4. Поиск N-периферии

Для стартовой вершины s находит все вершины i, где: dist[s][i] > N и dist[s][i] < INF

Добавляет их в результат.

10.4 Примеры входных и выходных данных

10.4.1 Входные данные

```
A B 2
A C 5
B C 1
B D 4
C E 3
D E 2
E F 6
B A -1
```

10.4.2 Выходные данные

```
Введите вершину: A
Введите N: 6
N-периферия вершины A (N = 6): F
```

11 Максимальный поток V

11.1 Условие

Решить задачу на нахождение максимального потока любым алгоритмом. Подготовить примеры, демонстрирующие работу алгоритма в разных случаях.

11.2 Код (фрагменты кода)

```
int Graph::edmondsKarp(const string& sourceName, const string&
sinkName) const {
    int n = vertexCount();
    int s = findVertex(sourceName);
    int t = findVertex(sinkName);
    if (s == -1 || t == -1) {
        cout << "Ошибка: источник или сток не найдены.\n";
        return 0;
    }

    // Матрица пропускных способностей
    vector<vector<int>> capacity(n, vector<int>(n, 0));
    for (int i = 0; i < n; ++i) {
        for (const auto& e : adjList[i].adj) {
            int j = findVertex(e.to);
            if (j != -1)
                capacity[i][j] += e.weight; // если несколько рёбер —
суммируем
        }
    }

    vector<vector<int>> flow(n, vector<int>(n, 0));
    int maxFlow = 0;

    while (true) {
        // BFS для поиска пути с остаточной ёмкостью
        vector<int> parent(n, -1);
        queue<int> q;
        q.push(s);
        parent[s] = s;

        while (!q.empty() && parent[t] == -1) {
            int u = q.front();
```

```
            q.pop();
            for (int v = 0; v < n; ++v) {
                if (parent[v] == -1 && capacity[u][v] - flow[u][v] > 0)
{
                    parent[v] = u;
                    q.push(v);
                }
            }
        }

        // если пути нет — всё, закончили
        if (parent[t] == -1)
            break;

        // находим минимальную пропускную способность по пути
        int increment = INT_MAX;
        for (int v = t; v != s; v = parent[v]) {
            int u = parent[v];
            increment = min(increment, capacity[u][v] - flow[u][v]);
        }

        // обновляем потоки вдоль пути
        for (int v = t; v != s; v = parent[v]) {
            int u = parent[v];
            flow[u][v] += increment;
            flow[v][u] -= increment; // обратное ребро
        }

        maxFlow += increment;
    }

    cout << "Максимальный поток из " << sourceName << " в " << sinkName
<< " = " << maxFlow << "\n";
    return maxFlow;
}
```

11.3 Краткое описание алгоритма

Находит максимальный поток от истока (source) к стоку (sink) в ориентированном графе, где рёбра имеют пропускные способности.

11.3.1 Что делает

1. Инициализация

   Находит вершины истока и стока

   Создает матрицу пропускных способностей из весов рёбер

   Инициализирует матрицу потоков нулями
2. Основной цикл

   Пока существует увеличивающий путь:
3. Поиск пути (BFS)

   Ищет путь от истока к стоку в остаточной сети

   Остаточная пропускная способность: capacity[u][v] - flow[u][v]

   Запоминает путь через массив parent
4. Вычисление минимального потока

   Находит минимальную остаточную способность вдоль найденного пути

   Это максимальный поток, который можно пропустить по этому пути
5. Обновление потоков

   Увеличивает поток по прямым рёбрам пути

   Уменьшает поток по обратным рёбрам (для возможности отмены)
6. Накопление результата

   Добавляет найденный поток к общему максимальному потоку

11.4 Примеры входных и выходных данных

11.4.1 Входные данные

```
S A 10
S C 10
A B 4
A C 2
C D 15
B T 10
D B 6
D T 10
```

11.4.2 Выходные данные

```
=== Edmond-Karp algorithm - Max Flow ===
Введите имя источника: S
```

```
Введите имя стока: T
Максимальный поток из S в T = 16
```