

**МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ**  
**ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ БЮДЖЕТНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ**  
**ВЫСШЕГО ОБРАЗОВАНИЯ**  
**«БЕЛГОРОДСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНОЛОГИЧЕСКИЙ**  
**УНИВЕРСИТЕТ им. В. Г. ШУХОВА»**  
**(БГТУ им. В.Г. Шухова)**



**ИНСТИТУТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ И УПРАВЛЯЮЩИХ СИСТЕМ**

**Лабораторная работа №4.1**

по дисциплине: Дискретная математика  
тема: «Маршруты»

Выполнил: ст. группы ПВ-223  
Пахомов Владислав Андреевич

Проверили:  
ст. пр. Рязанов Юрий Дмитриевич  
ст. пр. Бондаренко Татьяна Владими-  
ровна

Белгород 2023 г.

## Лабораторная работа №4.1

Маршруты

Вариант 10

**Цель работы:** изучить основные понятия теории графов, способы задания графов, научиться программно реализовывать алгоритмы получения и анализа маршрутов в графах.

1. Представить графы  $G_1$  и  $G_2$  матрицей смежности, матрицей инцидентности, диаграммой.

$G_1$ :

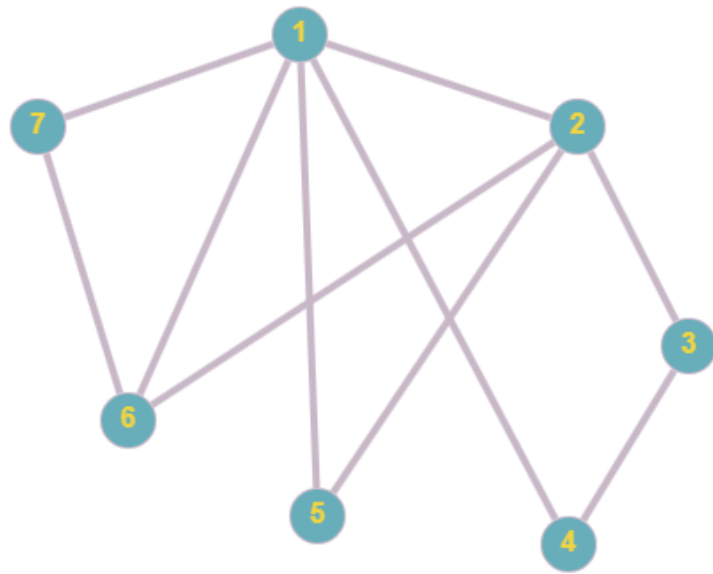
Матрица смежности:

$$\begin{pmatrix} 0 & 1 & 0 & 1 & 1 & 1 & 1 \\ 1 & 0 & 1 & 0 & 1 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

Матрица инцидентности:

$$\begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

Диаграмма:



$G_2$ :

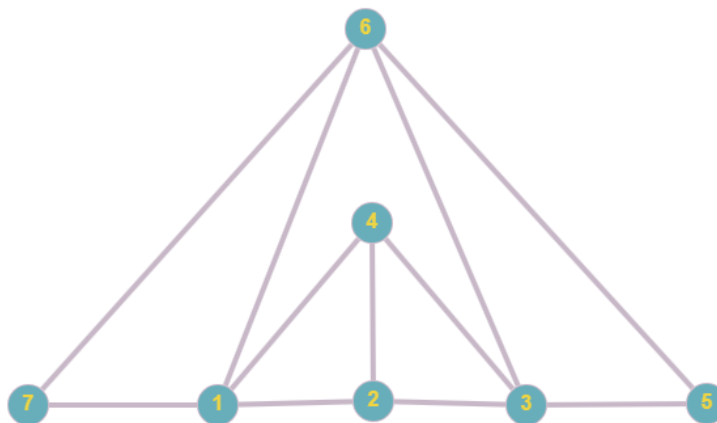
Матрица смежности:

$$\begin{pmatrix} 0 & 1 & 0 & 1 & 0 & 1 & 1 \\ 1 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$

Матрица инцидентности:

$$\begin{pmatrix} 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix}$$

Диаграмма:



2. Определить, являются ли последовательности вершин маршрутом, цепью, простой цепью, циклом, простым циклом в графах  $G_1$  и  $G_2$ .

(2, 3, 4, 1, 7, 6)

В  $G_1$  последовательность вершин является маршрутом, цепью, простой цепью. Не является циклом, следовательно, и простым циклом тоже не является. В  $G_2$  последовательность вершин является маршрутом, цепью, простой цепью. Не является циклом, следовательно, и простым циклом тоже не является.

(4, 1, 6, 7, 1, 2)

В  $G_1$  последовательность вершин является маршрутом, цепью, не является простой цепью (вершина 1 повторяется дважды). Не является циклом, следовательно, и простым циклом тоже не является. В  $G_2$  последовательность вершин является маршрутом, цепью, не является простой цепью (вершина 1 повторяется дважды). Не является циклом, следовательно, и простым циклом тоже не является.

(1, 4, 3, 2, 1)

В  $G_1$  последовательность вершин является маршрутом, цепью, не является простой цепью (вершина 1 повторяется дважды). Является циклом и простым циклом. В  $G_2$  последовательность вершин является маршрутом, цепью, не является простой цепью (вершина 1 повторяется дважды). Является циклом и простым циклом.

(1, 2, 4, 3, 2, 1)

В  $G_1$  последовательность вершин не является маршрутом (2 и 4 не смежны, следовательно не найдётся такого инцидентного общего ребра для обеих вершин), следовательно не обладает другими свойствами. В  $G_2$  последовательность вершин является маршрутом, не является цепью (ребро 1-2 повторяется дважды), следовательно, не является и простой цепью, циклом и простым циклом.

(2, 4, 1, 7, 6, 1, 2)

В  $G_1$  последовательность вершин не является маршрутом (2 и 4 не смежны, следовательно не найдётся такого инцидентного общего ребра для обеих вершин), следовательно не обладает другими свойствами. В  $G_2$  последовательность вершин является маршрутом, цепью, не является простой цепью (1, 2 повторяются дважды). Является циклом, не является простым циклом (1 повторяется дважды).

3. Написать программу, определяющую, является ли заданная последовательность вершин маршрутом, цепью, простой цепью, циклом, простым циклом в графах  $G_1$  и  $G_2$ .

Напишем вспомогательный классы, определяющие объекты графа, вершины, ребра, точечного маршрута. В дальнейшем будем использовать его.

*node.hpp*

```
#ifndef GRAPH_NODE
#define GRAPH_NODE

template <typename T, typename V>
class NamedNode {
protected:
    T value;

public:
    using NodeValueType = T;

    V name;

    NamedNode(T value, V name) {
        this->value = value;
        this->name = name;
    };

    NamedNode<T, V>* clone() {
        return new NamedNode<T, V>(value, name);
    }

    virtual bool equals(const NamedNode<T, V>& b) const {
        return this->value == b.value;
    }

    const T& getValue() const {
        return value;
    }
};

template <typename T>
class Node : public NamedNode<T, T> {
public:
    Node(T value) : NamedNode<T, T>(value, value) {};

    Node<T>* clone() {
        return new Node<T>(this->value);
    }
};

#endif
```

*edge.hpp*

```

#ifndef GRAPH_EDGE
#define GRAPH_EDGE

template <typename T, typename V>
class NamedEdge {
public:
    using NodeType = T;
    using NameType = V;

    std::vector<T*> nodes;
    V name;
    bool isDirected;

    NamedEdge(){};

    NamedEdge(std::initializer_list<T*> nodes, V name, bool isDirected) {
        if (nodes.size() < 2) throw std::invalid_argument("You can't create edge with less than two nodes");
        this->nodes.insert(this->nodes.begin(), nodes.begin(), nodes.end());
        this->name = name;
        this->isDirected = isDirected;
    }

    virtual bool equals(const NamedEdge<T, V>& b) const {
        if (isDirected != b.isDirected || nodes.size() != b.nodes.size() || name != b.name) return false;

        for (int i = 0; i < nodes.size(); i++)
            if (!(*this->nodes[i]).equals(*b.nodes[i])) return false;

        return true;
    }
};

template <typename T>
class Edge : public NamedEdge<T, std::string> {
public:
    Edge(){};

    Edge(std::initializer_list<T*> nodes, bool isDirected) : NamedEdge<T, std::string>(nodes, "", isDirected) {};
};

#endif

```

## *routes.hpp*

```

#ifndef GRAPH_ROUTE
#define GRAPH_ROUTE

template <typename N>
class PointRoute {
public:
    std::vector<N> route;

    PointRoute(std::vector<N> route) {

```

```

        this->route = route;
    }
};

#endif

```

## graph.hpp

```

#ifndef GRAPH
#define GRAPH

// Abandon hope, all ye who enter here

#include "node.hpp"
#include "edge.hpp"
#include "routes.hpp"

#include <set>

// Абстрактный класс граф, содержит виртуальные методы
template <typename E, typename N = typename E::NodeType>
class Graph {

public:
    using NodeValueType = typename N::NodeValueType;

    virtual void addNode(N& node) = 0;
    virtual void addEdge(E edge) = 0;
    virtual N* operator[] (const NodeValueType value) = 0;
    virtual void deleteEdge(E edge) = 0;
    virtual void deleteNode(const N& node) = 0;

    virtual bool isRoute(std::vector<N*> route) = 0;
    virtual bool isChain(std::vector<N*> chain) = 0;
    virtual bool isSimpleChain(std::vector<N*> simpleChain) = 0;
    virtual bool isCycle(std::vector<N*> cycle) = 0;
    virtual bool isSimpleCycle(std::vector<N*> simpleCycle) = 0;

    virtual const std::vector<N*> getAdjacentNodes(N* node) = 0;
    virtual std::vector<PointRoute<N*>> getAllRoutes(N* start, int steps) = 0;

    virtual int countAllRoutesBetweenAllNodes(int steps) = 0;

    virtual std::vector<PointRoute<N*>> getAllRoutesBetweenTwoNodes(N* start, N* end, int steps) = 0;

    virtual std::vector<PointRoute<N*>> getAllSimpleMaximumChains(N* start) = 0;
};

template <typename E, typename N = typename E::NodeType>
class AdjacencyMatrixGraph : public Graph<E, N> {
public:
    class EdgeContainer {
    public:

```

```

    E current;
    EdgeContainer* linked = nullptr;

    EdgeContainer(){};

    EdgeContainer(E curr) : current(curr) {};

    EdgeContainer(E current, EdgeContainer* linked) {
        this->current = current;
        this->linked = linked;
    }

    bool operator==(const EdgeContainer& b) {
        return current.equals(b.current) && this->linked == b.linked;
    }
};

using NodeValueType = typename N::NodeValueType;

std::vector<N*> nodes;
std::vector<std::vector<std::vector<EdgeContainer*>>> edges;

void addNode(N& node) override {
    for (auto &pNode : nodes)
        if (pNode->equals(node))
            throw std::invalid_argument("Node is already present");

    nodes.push_back(node.clone());

    for (int i = 0; i < edges.size(); i++) {
        edges[i].resize(edges.size() + 1);
        edges[i][edges[i].size() - 1] = nullptr;
    }

    edges.push_back(std::vector<std::vector<EdgeContainer*>>(edges.size() + 1, nullptr));
}

void addEdge(E edge) {
    reinterpretNodes(edge.nodes);

    std::vector<EdgeContainer*>& edgePos = this->at(edge.nodes.front(), edge.nodes.back());
    if (!edgePos) edgePos = new std::vector<EdgeContainer*>();

    (*edgePos).push_back(EdgeContainer(edge));
    EdgeContainer* added = &(*edgePos).back();
    if (!edge.isDirected) {
        std::reverse(edge.nodes.begin(), edge.nodes.end());

        std::vector<EdgeContainer*>& revEdgePos = this->at(edge.nodes.front(), edge.nodes.back());
        if (!revEdgePos) revEdgePos = new std::vector<EdgeContainer*>();

        (*revEdgePos).push_back(EdgeContainer(edge));

        EdgeContainer* nonDirectAdded = &(*revEdgePos).back();
    }
}

```



```

        added->linked = nonDirectAdded;
        nonDirectAdded->linked = added;
    }
}

void deleteEdge(E edge) {
    deleteEdge(edge, false);
}

void deleteNode(const N& node) {
    int deleteIndex = this->nodes.size();
    for (int i = 0; i < this->nodes.size() && deleteIndex == this->nodes.size(); i++) {
        if ((*this->nodes[i]).getValue() == node.getValue())
            deleteIndex = i;
    }

    if (deleteIndex == this->nodes.size()) throw std::invalid_argument("Node does not belongs to graph");

    for (auto &colEdge : this->edges[deleteIndex]) {
        delete colEdge;
    }
    this->nodes.erase(this->nodes.begin() + deleteIndex);

    for (auto &rowEdge : this->edges) {
        delete rowEdge[deleteIndex];
        rowEdge.erase(rowEdge.begin() + deleteIndex);
    }
}

bool isRoute(std::vector<N*> route) {
    reinterpretNodes(route);

    // Если две вершины смежны, то они инцидентны одному ребру.
    // Проверкой на смежность будем сразу проверять, что вершина i
    // инцидентна какому-то ребру, и вершина i + 1 тоже инцидентна
    // тому же ребру.
    for (int i = 0; i < route.size() - 1; i++) {
        if (!isNodesAdjacent(*route[i], *route[i + 1])) return false;
    }

    return true;
}

bool isChain(std::vector<N*> chain) {
    reinterpretNodes(chain);

    std::vector<EdgeContainer> visitedEdges;
    for (int i = 0; i < chain.size() - 1; i++) {
        auto pedges = findByBeginEndPoint((*chain[i]).getValue(), (*chain[i + 1]).getValue());
        if (pedges == nullptr) return false;
        auto edges = *pedges;

        // Отдаём приоритет направленным рёбрам

```

```

    // Можем обратиться к 0 элементу безопасно из-за проверки на маршрут выше.
    EdgeContainer searchEdge;
    bool found = false;
    bool foundDirected = false;
    for (int j = 0; j < edges.size(); j++) {
        if (edges[j].linked == nullptr && std::find(visitedEdges.begin(), visitedEdges.end(), edges[j])
            ↪ == visitedEdges.end()) {
            foundDirected = true;
            found = true;
            visitedEdges.push_back(edges[j]);
            break;
        } else if (edges[j].linked != nullptr && std::find(visitedEdges.begin(), visitedEdges.end(),
            ↪ edges[j]) == visitedEdges.end()) {
            found = true;
            searchEdge = edges[j];
        }
    }

    if (!found)
        return false;
    else if (found && !foundDirected) {
        visitedEdges.push_back(searchEdge);
        visitedEdges.push_back(*searchEdge.linked);
    }
}

return true;
}

bool isSimpleChain(std::vector<N*> simpleChain) {
    // Проверяем, что рёбра не повторяются
    if (!isRoute(simpleChain)) return false;

    for (int i = 0; i < simpleChain.size() - 1; i++) {
        for (int j = i + 1; j < simpleChain.size(); j++) {
            if ((simpleChain[i])->equals(*(simpleChain[j]))) return false;
        }
    }

    return true;
}

bool isCycle(std::vector<N*> cycle) {
    if (!isChain(cycle)) return false;

    return (*(cycle.front())).equals(*(cycle.back()));
}

bool isSimpleCycle(std::vector<N*> cycle) {
    if (!isCycle(cycle)) return false;

    for (int i = 0; i < cycle.size() - 2; i++) {
        for (int j = i + 1; j < cycle.size() - 1; j++) {
            if ((cycle[i])->equals(*(cycle[j]))) return false;
        }
    }
}

```

```

    }
}

return true;
}

// Ищет среди вершин нужную с value. Если вершина не найдена - возвращает null. Иначе - возвращает ссылку на
↳ него.
N* operator[](const NodeValueType value) {
    for (auto &n : this->nodes)
        if (n->getValue() == value)
            return n;

    return nullptr;
}

// Ищет и возвращает указатель на массив, все грани в котором начинаются с точки с значением a и
↳ заканчивается b
const std::vector<EdgeContainer>* findByBeginEndPoint(const NodeValueType begin, const NodeValueType end) {
    return at((*this)[begin], (*this)[end]);
}

~AdjacencyMatrixGraph() {
    // Удаляем рёбра
    for (auto row : this->edges) {
        for (auto element : row) {
            delete element;
        }
    }

    // Удаляем вершины
    for (auto node : this->nodes) {
        delete node;
    }
}

const std::vector<N*> getAdjacentNodes(N* node) {
    int nodeIndex = -1;
    for (int i = 0; i < this->nodes.size(); i++) {
        if ((*this->nodes[i]).equals(*node)) {
            nodeIndex = i;

            break;
        }
    }
}

if (nodeIndex == -1) throw std::invalid_argument("Node does not belongs to graph");

std::vector<N*> result;
for (int i = 0; i < this->nodes.size(); i++) {
    if (this->edges[nodeIndex][i] != nullptr)
        result.push_back(this->nodes[i]);
}

```

```

        return result;
    }

std::vector<PointRoute<N*>> getAllRoutes(N* start, int steps) {
    std::vector<PointRoute<N*>> result;
    if (steps < 1) {
        result.push_back(PointRoute<N*>({start}));
        return result;
    }

    for (auto &adjNode : getAdjacentNodes(start)) {
        auto routes = getAllRoutes(adjNode, steps - 1);
        for (auto &route : routes) {
            route.route.insert(route.route.begin(), start);

            result.push_back(route);
        }
    }

    return result;
}

int countAllRoutesBetweenAllNodes(int steps) {
    int count = 0;
    for (int i = 0; i < nodes.size(); i++)
        for (int j = 0; j < nodes.size(); j++)
            count += countAllRoutesBetweenTwoNodes(i, j, steps);

    return count;
}

std::vector<PointRoute<N*>> getAllRoutesBetweenTwoNodes(N* start, N* end, int steps) {
    std::vector<PointRoute<N*>> result;
    if (steps < 1) {
        if (start->equals(*end))
            result.push_back(PointRoute<N*>({start}));

        return result;
    }

    for (auto &adjNode : getAdjacentNodes(start)) {
        auto routes = getAllRoutesBetweenTwoNodes(adjNode, end, steps - 1);

        for (auto &route : routes) {
            route.route.insert(route.route.begin(), start);

            result.push_back(route);
        }
    }

    return result;
}

```

```

std::vector<PointRoute<N*>> getAllSimpleMaximumChains(N* start) {
    return getAllSimpleMaximumChains(start, {});
}

private:
int countAllRoutesBetweenTwoNodes(int start, int end, int steps) {
    if (steps <= 1) {
        return edges[start][end] != nullptr;
    }

    int count = 0;
    for (int i = 0; i < edges.size(); i++) {
        count += countAllRoutesBetweenTwoNodes(start, i, steps - 1) * (edges[i][end] != nullptr);
    }

    return count;
}

std::vector<PointRoute<N*>> getAllSimpleMaximumChains(N* start, std::vector<N*> takenNodes) {
    std::vector<PointRoute<N*>> result;
    takenNodes.push_back(start);

    bool anyElementFound = false;
    for (auto &adjNode : getAdjacentNodes(start)) {
        if (std::find(takenNodes.begin(), takenNodes.end(), adjNode) != takenNodes.end()) continue;

        anyElementFound = true;
        auto adjacentNodesForCurrentNode = getAdjacentNodes(adjNode);

        auto routes = getAllSimpleMaximumChains(adjNode, takenNodes);
        for (auto &route : routes) {
            route.route.insert(route.route.begin(), start);

            result.push_back(route);
        }
    }

    if (!anyElementFound) {
        result.push_back(PointRoute<N*>({start}));
    }

    return result;
}

void deleteEdge(E edge, bool nonDirectionalDelete) {
    reinterpretNodes(edge.nodes);

    std::vector<EdgeContainer*> edgePos = this->at(edge.nodes.front(), edge.nodes.back());
    if (!edgePos) throw std::invalid_argument("Edge doesn't exist in graph");

    for (int i = 0; i < (*edgePos).size(); i++) {
        EdgeContainer& currentEdge = (*edgePos)[i];
    }
}

```

```

        if (!currentEdge.current.equals(edge)) continue;

        if (!edge.isDirected && !nonDirectionalDelete) {
            deleteEdge(currentEdge.linked->current, true);
        }

        (*edgePos).erase((*edgePos).begin() + i);
        if ((*edgePos).size() == 0) {
            delete edgePos;
            edgePos = nullptr;
        }

        return;
    }

    throw std::invalid_argument("Edge doesn't exist in graph");
}

bool isNodesAdjacent(const N& a, const N& b) {
    try {
        return findByBeginEndPoint(a.getValue(), b.getValue()) != nullptr;
    } catch (std::invalid_argument& ignore) {}

    return false;
}

// Обновляет вершины в edge, делая так, чтобы они принадлежали graph. Выбрасывает ошибку, если вершины не
↳ существует.
void reinterpretNodes(std::vector<N*>& nodes) {
    for (int i = 0; i < nodes.size(); i++) {
        auto pNode = this->operator[](nodes[i]->getValue());

        if (pNode == nullptr)
            throw std::invalid_argument("Node does not belongs to graph");

        nodes[i] = pNode;
    }
}

std::vector<EdgeContainer*> at(N* from, N* to) {
    if (from == nullptr || to == nullptr) throw std::invalid_argument("Node does not belongs to graph");

    int fromIndex = -1;
    int toIndex = -1;
    for (int i = 0; i < nodes.size() && (fromIndex == -1 || toIndex == -1); i++) {
        if (nodes[i] == from) fromIndex = i;
        if (nodes[i] == to) toIndex = i;
    }

    if (fromIndex == -1 || toIndex == -1) {
        throw std::invalid_argument("Node is not present");
    }

    return edges[fromIndex][toIndex];
}

```

```
    }  
};  
  
#endif
```

В данном задании были использованы методы isRoute, isChain, isSimpleChain, isCycle, isSimpleCycle.

*util.h*

```
#pragma once  
  
#include "../libs/alg/alg.h"  
  
typedef Node<int> IntNode;  
  
Graph<Edge<IntNode>>* constructGraph1() {  
    auto G1 = new AdjacencyMatrixGraph<Edge<IntNode>>();  
  
    IntNode N1(1);  
    IntNode N2(2);  
    IntNode N3(3);  
    IntNode N4(4);  
    IntNode N5(5);  
    IntNode N6(6);  
    IntNode N7(7);  
  
    G1->addNode(N1);  
    G1->addNode(N2);  
    G1->addNode(N3);  
    G1->addNode(N4);  
    G1->addNode(N5);  
    G1->addNode(N6);  
    G1->addNode(N7);  
  
    G1->addEdge({&N1, &N2}, false);  
    G1->addEdge({&N1, &N4}, false);  
    G1->addEdge({&N1, &N5}, false);  
    G1->addEdge({&N1, &N6}, false);  
    G1->addEdge({&N1, &N7}, false);  
    G1->addEdge({&N2, &N3}, false);  
    G1->addEdge({&N2, &N5}, false);  
    G1->addEdge({&N2, &N6}, false);  
    G1->addEdge({&N3, &N4}, false);  
    G1->addEdge({&N6, &N7}, false);  
  
    return G1;  
}  
  
Graph<Edge<IntNode>>* constructGraph2() {  
    auto G2 = new AdjacencyMatrixGraph<Edge<IntNode>>();  
  
    IntNode N1(1);
```

```

IntNode N2(2);
IntNode N3(3);
IntNode N4(4);
IntNode N5(5);
IntNode N6(6);
IntNode N7(7);

G2->addNode(N1);
G2->addNode(N2);
G2->addNode(N3);
G2->addNode(N4);
G2->addNode(N5);
G2->addNode(N6);
G2->addNode(N7);

G2->addEdge({{&N7, &N6}, false});
G2->addEdge({{&N1, &N6}, false});
G2->addEdge({{&N4, &N1}, false});
G2->addEdge({{&N4, &N2}, false});
G2->addEdge({{&N4, &N3}, false});
G2->addEdge({{&N3, &N6}, false});
G2->addEdge({{&N5, &N6}, false});
G2->addEdge({{&N7, &N1}, false});
G2->addEdge({{&N2, &N1}, false});
G2->addEdge({{&N2, &N3}, false});
G2->addEdge({{&N5, &N3}, false});

return G2;
}

```

```

#include "../util.h"

```

```

void testRoute(Graph<Edge<IntNode>>* G, std::string graphName, std::vector<int>& route) {
    std::cout << "Testing route {";
    for (int i = 0; i < route.size(); i++) {
        if (i == route.size() - 1) {
            std::cout << route[i];
        } else {
            std::cout << route[i] << ", ";
        }
    }
    std::cout << "} for graph " << graphName << ":\n\n";
    std::vector<IntNode*> nodes;
    for (auto &element : route) {
        nodes.push_back((*G)[element]);
    }

    std::cout << (G->isRoute(nodes) ? "Is route\n" : "Is not route\n");
    std::cout << (G->isChain(nodes) ? "Is chain\n" : "Is not chain\n");
    std::cout << (G->isSimpleChain(nodes) ? "Is simple chain\n" : "Is not simple chain\n");
    std::cout << (G->isCycle(nodes) ? "Is cycle\n" : "Is not cycle\n");
    std::cout << (G->isSimpleCycle(nodes) ? "Is simple cycle\n" : "Is not simple cycle\n\n");
}

```



```

}

int main() {
    Graph<Edge<IntNode>> *G1 = constructGraph1();
    Graph<Edge<IntNode>> *G2 = constructGraph2();

    std::vector<int> route1 {2, 3, 4, 1, 7, 6};
    std::vector<int> route2 {4, 1, 6, 7, 1, 2};
    std::vector<int> route3 {1, 4, 3, 2, 1};
    std::vector<int> route4 {1, 2, 4, 3, 2, 1};
    std::vector<int> route5 {2, 4, 1, 7, 6, 1, 2};

    testRoute(G1, "G1", route1);
    testRoute(G1, "G1", route2);
    testRoute(G1, "G1", route3);
    testRoute(G1, "G1", route4);
    testRoute(G1, "G1", route5);

    testRoute(G2, "G2", route1);
    testRoute(G2, "G2", route2);
    testRoute(G2, "G2", route3);
    testRoute(G2, "G2", route4);
    testRoute(G2, "G2", route5);

    return 0;
}

```

## Результат работы программы:

Testing route {2, 3, 4, 1, 7, 6} for graph G1:

Is route  
 Is chain  
 Is simple chain  
 Is not cycle  
 Is not simple cycle

Testing route {4, 1, 6, 7, 1, 2} for graph G1:

Is route  
 Is chain  
 Is not simple chain  
 Is not cycle  
 Is not simple cycle

Testing route {1, 4, 3, 2, 1} for graph G1:

Is route  
 Is chain  
 Is not simple chain  
 Is cycle  
 Is simple cycle

Testing route {1, 2, 4, 3, 2, 1} for graph G1:

Is not route  
Is not chain  
Is not simple chain  
Is not cycle  
Is not simple cycle

Testing route {2, 4, 1, 7, 6, 1, 2} for graph G1:

Is not route  
Is not chain  
Is not simple chain  
Is not cycle  
Is not simple cycle

Testing route {2, 3, 4, 1, 7, 6} for graph G2:

Is route  
Is chain  
Is simple chain  
Is not cycle  
Is not simple cycle

Testing route {4, 1, 6, 7, 1, 2} for graph G2:

Is route  
Is chain  
Is not simple chain  
Is not cycle  
Is not simple cycle

Testing route {1, 4, 3, 2, 1} for graph G2:

Is route  
Is chain  
Is not simple chain  
Is cycle  
Is simple cycle

Testing route {1, 2, 4, 3, 2, 1} for graph G2:

Is route  
Is not chain  
Is not simple chain  
Is not cycle  
Is not simple cycle

Testing route {2, 4, 1, 7, 6, 1, 2} for graph G2:

Is route  
Is chain  
Is not simple chain  
Is cycle  
Is not simple cycle

Результат выполнения программы совпали с ручными вычислениями.

4. Написать программу, получающую все маршруты заданной длины, выходящие из заданной вершины. Использовать программу для получения всех маршрутов заданной длины в графах  $G_1$  и  $G_2$ .

В данном задании использовался метод `getAllRoutes`.

*main.cpp*

```
#include "../util.h"

int main() {
    Graph<Edge<IntNode>> *G1 = constructGraph1();
    Graph<Edge<IntNode>> *G2 = constructGraph2();

    std::cout << "All routes that start in 5 in G1 and has length of 2: \n";
    auto r = G1->getAllRoutes((*G1)[5], 2);

    for (auto &route : r) {
        for (int i = 0; i < route.route.size(); i++) {
            if (i == route.route.size() - 1) {
                std::cout << route.route[i]->name;
            } else {
                std::cout << route.route[i]->name << ", ";
            }
        }

        std::cout << std::endl;
    }

    std::cout << "All routes that start in 2 in G2 and has length of 3: \n";
    r = G2->getAllRoutes((*G1)[2], 3);

    for (auto &route : r) {
        for (int i = 0; i < route.route.size(); i++) {
            if (i == route.route.size() - 1) {
                std::cout << route.route[i]->name;
            } else {
                std::cout << route.route[i]->name << ", ";
            }
        }

        std::cout << std::endl;
    }

    return 0;
}
```

Результат выполнения программы:

```
All routes that start in 5 in G1 and has length of 2:
5, 1, 2
5, 1, 4
5, 1, 5
```

```
5, 1, 6
5, 1, 7
5, 2, 1
5, 2, 3
5, 2, 5
5, 2, 6
All routes that start in 2 in G2 and has length of 3:
2, 1, 2, 1
2, 1, 2, 3
2, 1, 2, 4
2, 1, 4, 1
2, 1, 4, 2
2, 1, 4, 3
2, 1, 6, 1
2, 1, 6, 3
2, 1, 6, 5
2, 1, 6, 7
2, 1, 7, 1
2, 1, 7, 6
2, 3, 2, 1
2, 3, 2, 3
2, 3, 2, 4
2, 3, 4, 1
2, 3, 4, 2
2, 3, 4, 3
2, 3, 5, 3
2, 3, 5, 6
2, 3, 6, 1
2, 3, 6, 3
2, 3, 6, 5
2, 3, 6, 7
2, 4, 1, 2
2, 4, 1, 4
2, 4, 1, 6
2, 4, 1, 7
2, 4, 2, 1
2, 4, 2, 3
2, 4, 2, 4
2, 4, 3, 2
2, 4, 3, 4
2, 4, 3, 5
2, 4, 3, 6
```

5. Написать программу, определяющую количество маршрутов заданной длины между каждой парой вершин графа. Использовать программу для определения количества маршрутов заданной длины между каждой парой вершин в графах  $G_1$  и  $G_2$ . В данном задании использовался метод `countAllRoutesBetweenAllNodes`.

*main.cpp*

```
#include "../util.h"

int main() {
```

```

Graph<Edge<IntNode>> *G1 = constructGraph1();
Graph<Edge<IntNode>> *G2 = constructGraph2();

std::cout << "All routes between pairs in 1 step in G1: " << G1->countAllRoutesBetweenAllNodes(1) <<
↳ std::endl;
std::cout << "All routes between pairs in 3 steps in G1: " << G1->countAllRoutesBetweenAllNodes(3) <<
↳ std::endl;
std::cout << "All routes between pairs in 2 steps in G2: " << G2->countAllRoutesBetweenAllNodes(2) <<
↳ std::endl;

return 0;
}

```

Результат выполнения программы:

```

All routes between pairs in 1 step in G1: 20
All routes between pairs in 3 steps in G1: 206
All routes between pairs in 2 steps in G2: 74

```

6. Написать программу, определяющую все маршруты заданной длины между заданной парой вершин графа. Использовать программу для определения всех маршрутов заданной длины между заданной парой вершин в графах  $G_1$  и  $G_2$ .

В данном задании использовался метод `getAllRoutesBetweenTwoNodes`.

*main.cpp*

```

#include "../util.h"

int main() {
    Graph<Edge<IntNode>> *G1 = constructGraph1();
    Graph<Edge<IntNode>> *G2 = constructGraph2();

    std::cout << "All routes between adjacent elements 1 and 7 in 1 step in G1: \n";

    auto r = G1->getAllRoutesBetweenTwoNodes((*G1)[1], (*G1)[7], 1);
    for (auto &route : r) {
        for (int i = 0; i < route.route.size(); i++) {
            if (i == route.route.size() - 1) {
                std::cout << route.route[i]->name;
            } else {
                std::cout << route.route[i]->name << ", ";
            }
        }
    }

    std::cout << std::endl;

    std::cout << "All routes between non-adjacent elements 3 and 4 in 4 steps in G1: \n";
    r = G1->getAllRoutesBetweenTwoNodes((*G1)[3], (*G1)[7], 4);
    for (auto &route : r) {
        for (int i = 0; i < route.route.size(); i++) {

```

```

        if (i == route.route.size() - 1) {
            std::cout << route.route[i]->name;
        } else {
            std::cout << route.route[i]->name << ", ";
        }
    }

    std::cout << std::endl;
}

std::cout << "All routes between non-adjacent elements 7 and 5 in 4 steps in G2: \n";
r = G2->getAllRoutesBetweenTwoNodes((*G2)[7], (*G2)[5], 4);
for (auto &route : r) {
    for (int i = 0; i < route.route.size(); i++) {
        if (i == route.route.size() - 1) {
            std::cout << route.route[i]->name;
        } else {
            std::cout << route.route[i]->name << ", ";
        }
    }

    std::cout << std::endl;
}

return 0;
}

```

Результат выполнения программы:

```

All routes between adjacent elements 1 and 7 in 1 step in G1:
1, 7
All routes between non-adjacent elements 3 and 4 in 4 steps in G1:
3, 2, 1, 6, 7
3, 2, 5, 1, 7
3, 2, 6, 1, 7
3, 4, 1, 6, 7
All routes between non-adjacent elements 7 and 5 in 4 steps in G2:
7, 1, 2, 3, 5
7, 1, 4, 3, 5
7, 1, 6, 3, 5
7, 1, 7, 6, 5
7, 6, 1, 6, 5
7, 6, 3, 6, 5
7, 6, 5, 3, 5
7, 6, 5, 6, 5
7, 6, 7, 6, 5

```

- Написать программу, получающую все простые максимальные цепи, выходящие из заданной вершины графа. Использовать программу для получения всех простых максимальных цепей, выходящих из заданной вершины в графах  $G_1$  и  $G_2$ . В данном задании использовался метод `getAllSimpleMaximumChains`.

```
#include "../util.h"

int main() {
    Graph<Edge<IntNode>> *G1 = constructGraph1();
    Graph<Edge<IntNode>> *G2 = constructGraph2();

    std::cout << "Simple maximum chains, starting in 4 in G1:\n";
    auto r = G1->getAllSimpleMaximumChains((*G1)[4]);
    for (auto &route : r) {
        for (int i = 0; i < route.route.size(); i++) {
            if (i == route.route.size() - 1) {
                std::cout << route.route[i]->name;
            } else {
                std::cout << route.route[i]->name << ", ";
            }
        }

        std::cout << std::endl;
    }

    std::cout << "Simple maximum chains, starting in 2 in G2:\n";
    r = G1->getAllSimpleMaximumChains((*G1)[2]);
    for (auto &route : r) {
        for (int i = 0; i < route.route.size(); i++) {
            if (i == route.route.size() - 1) {
                std::cout << route.route[i]->name;
            } else {
                std::cout << route.route[i]->name << ", ";
            }
        }

        std::cout << std::endl;
    }

    return 0;
}
```

## Результат выполнения программы:

```
Simple maximum chains, starting in 4 in G1:
4, 1, 2, 3
4, 1, 2, 5
4, 1, 2, 6, 7
4, 1, 5, 2, 3
4, 1, 5, 2, 6, 7
4, 1, 6, 2, 3
4, 1, 6, 2, 5
4, 1, 6, 7
4, 1, 7, 6, 2, 3
4, 1, 7, 6, 2, 5
```

```
4, 3, 2, 1, 5
4, 3, 2, 1, 6, 7
4, 3, 2, 1, 7, 6
4, 3, 2, 5, 1, 6, 7
4, 3, 2, 5, 1, 7, 6
4, 3, 2, 6, 1, 5
4, 3, 2, 6, 1, 7
4, 3, 2, 6, 7, 1, 5
Simple maximum chains, starting in 2 in G2:
2, 1, 4, 3
2, 1, 5
2, 1, 6, 7
2, 1, 7, 6
2, 3, 4, 1, 5
2, 3, 4, 1, 6, 7
2, 3, 4, 1, 7, 6
2, 5, 1, 4, 3
2, 5, 1, 6, 7
2, 5, 1, 7, 6
2, 6, 1, 4, 3
2, 6, 1, 5
2, 6, 1, 7
2, 6, 7, 1, 4, 3
2, 6, 7, 1, 5
```

**Вывод:** в ходе лабораторной работы изучили основные понятия теории графов, способы задания графов, научиться программно реализовывать алгоритмы получения и анализа маршрутов в графах.