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 №4.3

по дисциплине: Дискретная математика
тема: «Связность»

Выполнил: ст. группы ПВ-223
Пахомов Владислав Андреевич

Проверили:
ст. пр. Рязанов Юрий Дмитриевич
ст. пр. Бондаренко Татьяна Владимировна

Белгород 2023 г.

Лабораторная работа №4.1
Связность
Вариант 10

Цель работы: изучить алгоритм Краскала построения покрывающего леса, научиться использовать его при решении различных задач.

1. Реализовать алгоритм Краскала построения покрывающего леса.

```
template <typename E, typename N = typename E::NodeType>
struct Forest {
    // Граф, содержащий получившийся остовный лес
    Graph<E, N>* trees;
    // Букеты
    std::vector<std::set<N*>> bouquets;
};
```

```
#include "../lab11/graph.tpp"

template <typename E, typename N>
Graph<E>* AdjacencyMatrixGraph<E, N>::clone() {
    AdjacencyMatrixGraph<E, N>* g = new AdjacencyMatrixGraph<E, N>();

    std::set<EdgeContainer*> obtained;
    for (int i = 0; i < this->nodes.size(); i++) {
        g->addNode(*this->nodes[i]);
    }

    for (int i = 0; i < this->nodes.size(); i++) {
        for (int j = 0; j < this->nodes.size(); j++) {
            if (this->edges[i][j] == nullptr) continue;

            for (auto &edgeContainer : *this->edges[i][j]) {
                if (obtained.find(edgeContainer) != obtained.end()) continue;

                obtained.insert(edgeContainer);
                if (edgeContainer->linked != nullptr)
                    obtained.insert(edgeContainer->linked);

                g->addEdge(edgeContainer->current);
            }
        }
    }

    return g;
}

template <typename E, typename N>
Forest<E, N> AdjacencyMatrixGraph<E, N>::getSpanningForest() {
```

```
    AdjacencyMatrixGraph<E> *trees = new AdjacencyMatrixGraph<E>();
    std::vector<int> bouquet(this->nodes.size());
    for (int i = 0; i < this->nodes.size(); i++) {
        bouquet[i] = i;
        trees->addNode(*this->nodes[i]);
    }

    for (int i = 0; i < this->nodes.size(); i++) {
        for (int j = 0; j < this->nodes.size(); j++) {
            // Ребра между i и j нет, пропускаем.
            if (this->edges[i][j] == nullptr) continue;

            int bI = bouquet[i];
            int bJ = bouquet[j];

            // Обе концевые вершины принадлежат одному букету, ребро не добавляем.
            if (bI == bJ) continue;

            // Концевые вершины принадлежат разным букетам, поэтому нужно их объединить в один.
            for (int k = 0; k < bouquet.size(); k++)
                if (bouquet[k] == bI)
                    bouquet[k] = bJ;

            for (auto& edge : *this->edges[i][j])
                trees->addEdge((*edge).current);
        }
    }

    Forest<E, N> result;
    result.trees = trees;
    for (int i = 0; i < this->nodes.size(); i++) {
        std::set<N*> rBouquet;

        for (int j = 0; j < this->nodes.size(); j++)
            if (bouquet[j] == i) rBouquet.insert(trees->nodes[j]);

        if (!rBouquet.empty())
            result.bouquets.push_back(rBouquet);
    }

    return result;
}
```

2. Используя алгоритм Краскала, разработать и реализовать алгоритм решения задачи.

   Найти минимальное множество ребер, удаление которых из связного графа разбивает его на три связные компоненты.

   Если граф несвязный, то после просмотра графа при помощи алгоритма Краскала будет сформирован лес из k покрывающих деревьев, где k - количество связных

компонентов в графе.
Для данной задачи можно использовать жадный алгоритм, удаляя все сочетания из 1, 2, 3.. рёбер и проверяя количество полученных деревьев, полученных при помощи метода Краскала.

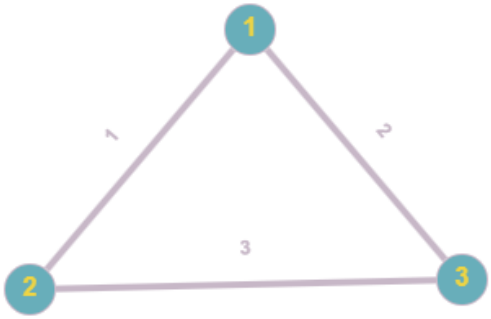
```
#include "../alg.h"

template <typename E>
std::pair<Forest<E>, std::vector<E>> getEdgesToDeleteToDivideGraphInNLinkedComponents(Graph<E> *g, std::vector<E>
↪  edges, int linkedComponentsAmount) {
    for (int deleteAmount = 0; deleteAmount <= edges.size(); deleteAmount++) {
        for (auto &comb : getCombinations<E>(edges, deleteAmount)){
            auto originClone = g->clone();

            for (auto &edge : comb)
                originClone->deleteEdge(edge);

            auto forest = originClone->getSpanningForest();
            delete originClone;

            if (forest.bouquets.size() == linkedComponentsAmount)
                return {forest, comb};

            delete forest.trees;
        }
    }

    return {{nullptr, {}}, std::vector<E>(0)};
}
```

Функция getEdgesToDeleteToDivideGraphInNLinkedComponents возвращает пару - полученный лес и массив рёбер, которые можно удалить.

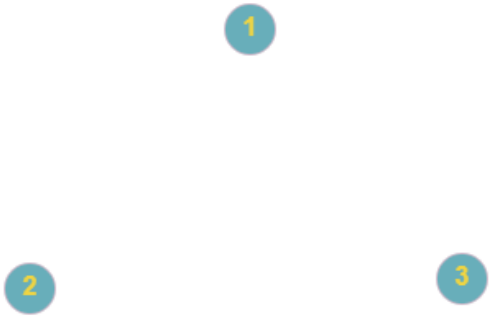
3. Подобрать тестовые данные. Результат представить в виде диаграммы графа. Вывод в консоль:

```
Edges to delete in graph "Three points": 1 2 3
Edges to delete in graph "Home": 4 5 6
Edges to delete in graph "Full with 5 nodes": 3 4 6 7 8 9 10
Edges to delete in graph "Full with 6 nodes": 4 5 8 9 11 12 13 14 15
Edges to delete in graph "Three triangles": 4 8
Edges to delete in graph "Three triangles strongly linked": 9 10 11
```

Треугольник

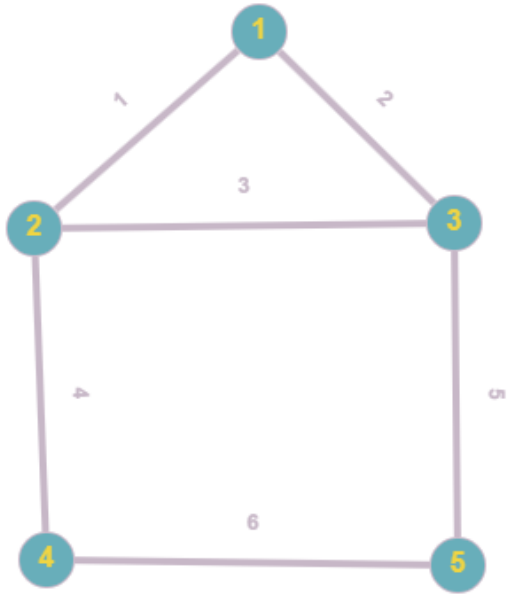


В данном графе логично удалить все три ребра.

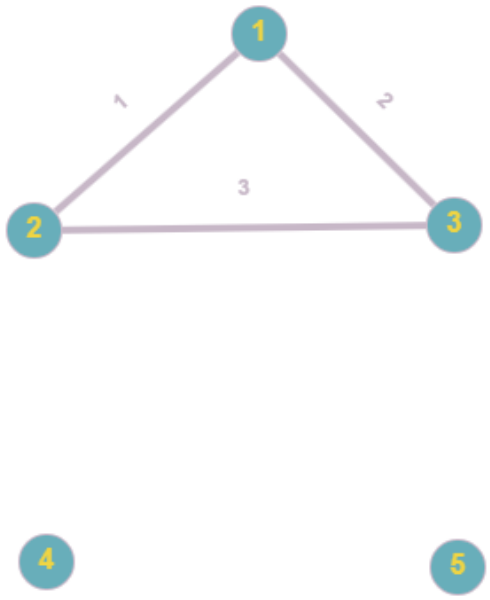


Результат выполнения программы совпал с предположениями.

Домик

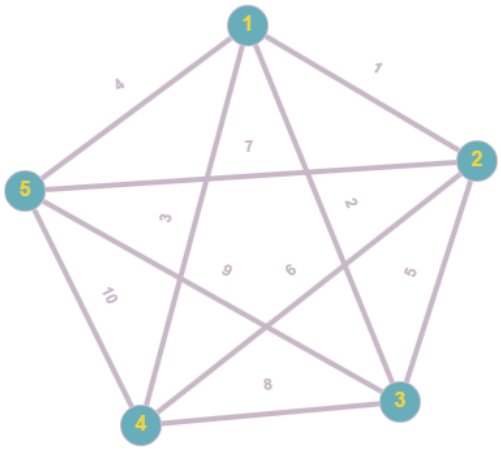


В данном графе можно удалить рёбра 4, 5, 6.

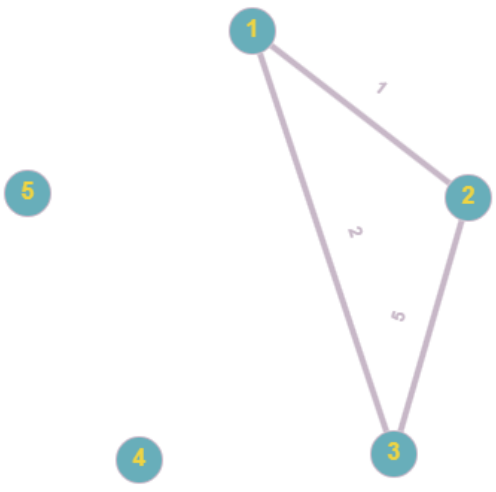


Результат выполнения программы совпал с предположениями.

Полный граф из 5 вершин

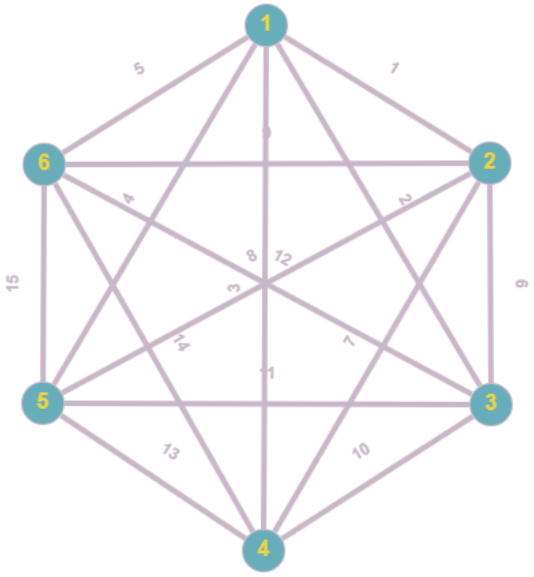


Из данного графа можно получить граф следующего вида, для этого можно удалить 7 рёбер.

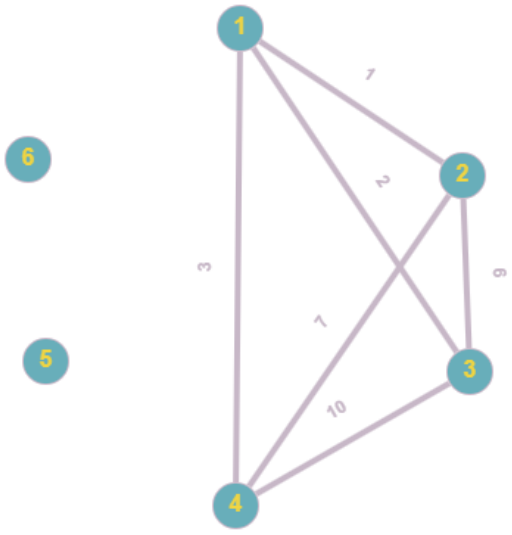


Результат выполнения программы совпал с предположениями.

Полный граф из 6 вершин

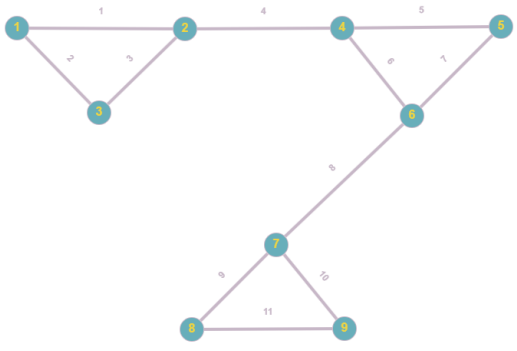


Из данного графа можно получить граф следующего вида, для этого можно удалить 9 рёбер.

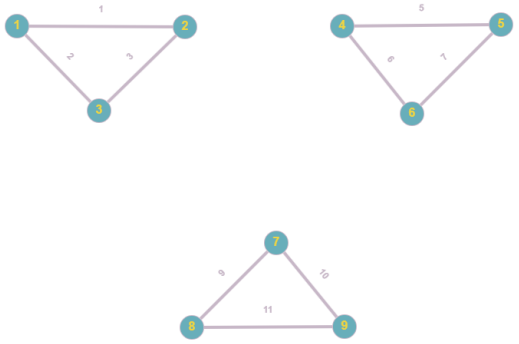


Результат выполнения программы совпал с предположениями.

Три треугольника

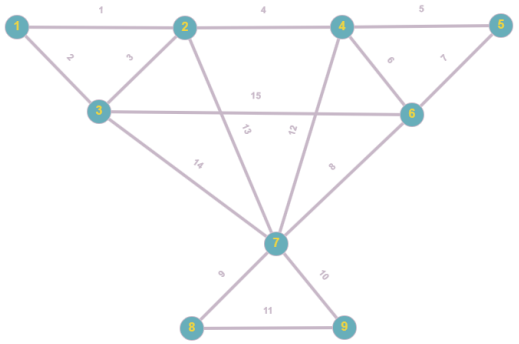


Довольно очевидно, что из графа достаточно удалить рёбра 4 и 8

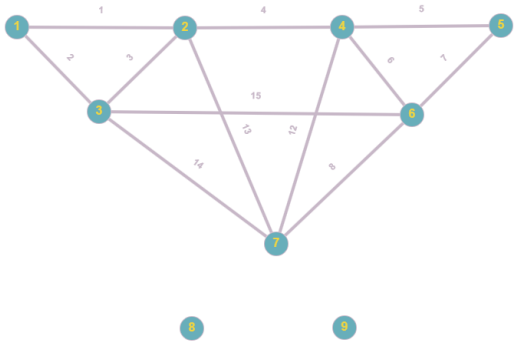


Результат выполнения программы совпал с предположениями.

Три треугольника сильно связанные



Удалять рёбра из центра графа больше не выгодно, в данном случае выгодно удалить три ребра 9, 10, 11



Результат выполнения программы совпал с предположениями.

Программа:

```
#include "../../libs/alg/alg.h"

template <typename E>
void testGraph(std::string graphName, Graph<E>* g, std::vector<E> edges) {
    auto res = getEdgesToDeleteToDivideGraphInNLinkedComponents(g, edges, 3);
    assert(res.first.trees != nullptr);

    // Проверка на наличие 3 связных компонентов
    assert(res.first.bouquets.size() == 3);
    std::cout << "Edges to delete in graph \"" << graphName << "\": ";

    // Удаление границ из графа
    for (auto& edge : res.second) {
```

```
        g->deleteEdge(edge);
        std::cout << edge.name << " ";
    }

    std::cout << std::endl;

    std::set<Node<int>*> bNodes;

    // Проверка на то, что вершины из разных букетов не будут иметь цепь в обновлённом графе
    // А из одного букета - будут.
    for (int i = 0; i < res.first.bouquets.size(); i++)
        for (auto &nodeA : res.first.bouquets[i]) {
            bNodes.insert(nodeA);

            for (int j = 0; j < res.first.bouquets.size(); j++)
                for (auto &nodeB : res.first.bouquets[j]) {
                    if (i == j) assert(g->isChainBetweenNodesExist(nodeA, nodeB));
                    else assert(!g->isChainBetweenNodesExist(nodeA, nodeB));
                }
        }

    // Проверка на то, что в букетах содержатся все вершины
    assert(bNodes.size() == g->getNodesSize());

    delete res.first.trees;
}

void testThreeNodes() {
    AdjacencyMatrixGraph<NamedEdge<Node<int>, int>> g;

    Node N1(1);
    Node N2(2);
    Node N3(3);

    g.addNode(N1);
    g.addNode(N2);
    g.addNode(N3);

    NamedEdge E1({&N1, &N2}, 1, false);
    NamedEdge E2({&N1, &N3}, 2, false);
    NamedEdge E3({&N2, &N3}, 3, false);
    std::vector<NamedEdge<Node<int>, int>> edges = {E1, E2, E3};

    g.addEdge(E1);
    g.addEdge(E2);
    g.addEdge(E3);

    testGraph("Three points", &g, edges);
}

void testFigureHome() {
    AdjacencyMatrixGraph<NamedEdge<Node<int>, int>> g;

    Node N1(1);
```

```
    Node N2(2);
    Node N3(3);
    Node N4(4);
    Node N5(5);

    g.addNode(N1);
    g.addNode(N2);
    g.addNode(N3);
    g.addNode(N4);
    g.addNode(N5);

    NamedEdge E1({&N1, &N2}, 1, false);
    NamedEdge E2({&N1, &N3}, 2, false);
    NamedEdge E3({&N2, &N3}, 3, false);
    NamedEdge E4({&N4, &N2}, 4, false);
    NamedEdge E5({&N3, &N5}, 5, false);
    NamedEdge E6({&N4, &N5}, 6, false);
    std::vector<NamedEdge<Node<int>, int>> edges = {E1, E2, E3, E4, E5, E6};

    g.addEdge(E1);
    g.addEdge(E2);
    g.addEdge(E3);
    g.addEdge(E4);
    g.addEdge(E5);
    g.addEdge(E6);

    testGraph("Home", &g, edges);
}

void testFull5() {
    AdjacencyMatrixGraph<NamedEdge<Node<int>, int>> g;

    Node N1(1);
    Node N2(2);
    Node N3(3);
    Node N4(4);
    Node N5(5);

    g.addNode(N1);
    g.addNode(N2);
    g.addNode(N3);
    g.addNode(N4);
    g.addNode(N5);

    NamedEdge E1({&N1, &N2}, 1, false);
    NamedEdge E2({&N1, &N3}, 2, false);
    NamedEdge E3({&N1, &N4}, 3, false);
    NamedEdge E4({&N1, &N5}, 4, false);
    NamedEdge E5({&N2, &N3}, 5, false);
    NamedEdge E6({&N2, &N4}, 6, false);
    NamedEdge E7({&N2, &N5}, 7, false);
    NamedEdge E8({&N3, &N4}, 8, false);
    NamedEdge E9({&N3, &N5}, 9, false);
    NamedEdge E10({&N4, &N5}, 10, false);
```

```
    std::vector<NamedEdge<Node<int>, int>> edges = {E1, E2, E3, E4, E5, E6, E7, E8, E9, E10};

    g.addEdge(E1);
    g.addEdge(E2);
    g.addEdge(E3);
    g.addEdge(E4);
    g.addEdge(E5);
    g.addEdge(E6);
    g.addEdge(E7);
    g.addEdge(E8);
    g.addEdge(E9);
    g.addEdge(E10);

    testGraph("Full with 5 nodes", &g, edges);
}

void testFull6() {
    AdjacencyMatrixGraph<NamedEdge<Node<int>, int>> g;

    Node N1(1);
    Node N2(2);
    Node N3(3);
    Node N4(4);
    Node N5(5);
    Node N6(6);

    g.addNode(N1);
    g.addNode(N2);
    g.addNode(N3);
    g.addNode(N4);
    g.addNode(N5);
    g.addNode(N6);

    NamedEdge E1({&N1, &N2}, 1, false);
    NamedEdge E2({&N1, &N3}, 2, false);
    NamedEdge E3({&N1, &N4}, 3, false);
    NamedEdge E4({&N1, &N5}, 4, false);
    NamedEdge E5({&N1, &N6}, 5, false);
    NamedEdge E6({&N2, &N3}, 6, false);
    NamedEdge E7({&N2, &N4}, 7, false);
    NamedEdge E8({&N2, &N5}, 8, false);
    NamedEdge E9({&N2, &N6}, 9, false);
    NamedEdge E10({&N3, &N4}, 10, false);
    NamedEdge E11({&N3, &N5}, 11, false);
    NamedEdge E12({&N3, &N6}, 12, false);
    NamedEdge E13({&N4, &N5}, 13, false);
    NamedEdge E14({&N4, &N6}, 14, false);
    NamedEdge E15({&N5, &N6}, 15, false);
    std::vector<NamedEdge<Node<int>, int>> edges = {E1, E2, E3, E4, E5, E6, E7, E8, E9, E10, E11, E12, E13, E14,
    ↪  E15};

    g.addEdge(E1);
    g.addEdge(E2);
    g.addEdge(E3);
```

```
    g.addEdge(E4);
    g.addEdge(E5);
    g.addEdge(E6);
    g.addEdge(E7);
    g.addEdge(E8);
    g.addEdge(E9);
    g.addEdge(E10);
    g.addEdge(E11);
    g.addEdge(E12);
    g.addEdge(E13);
    g.addEdge(E14);
    g.addEdge(E15);

    testGraph("Full with 6 nodes", &g, edges);
}

void testThreeTriangles() {
    AdjacencyMatrixGraph<NamedEdge<Node<int>, int>> g;

    Node N1(1);
    Node N2(2);
    Node N3(3);
    Node N4(4);
    Node N5(5);
    Node N6(6);
    Node N7(7);
    Node N8(8);
    Node N9(9);

    g.addNode(N1);
    g.addNode(N2);
    g.addNode(N3);
    g.addNode(N4);
    g.addNode(N5);
    g.addNode(N6);
    g.addNode(N7);
    g.addNode(N8);
    g.addNode(N9);

    NamedEdge E1({&N1, &N2}, 1, false);
    NamedEdge E2({&N1, &N3}, 2, false);
    NamedEdge E3({&N2, &N3}, 3, false);
    NamedEdge E4({&N2, &N4}, 4, false);
    NamedEdge E5({&N4, &N5}, 5, false);
    NamedEdge E6({&N4, &N6}, 6, false);
    NamedEdge E7({&N6, &N5}, 7, false);
    NamedEdge E8({&N6, &N7}, 8, false);
    NamedEdge E9({&N8, &N7}, 9, false);
    NamedEdge E10({&N7, &N9}, 10, false);
    NamedEdge E11({&N8, &N9}, 11, false);
    std::vector<NamedEdge<Node<int>, int>> edges = {E1, E2, E3, E4, E5, E6, E7, E8, E9, E10, E11};

    g.addEdge(E1);
    g.addEdge(E2);
```

```
    g.addEdge(E3);
    g.addEdge(E4);
    g.addEdge(E5);
    g.addEdge(E6);
    g.addEdge(E7);
    g.addEdge(E8);
    g.addEdge(E9);
    g.addEdge(E10);
    g.addEdge(E11);

    testGraph("Three triangles", &g, edges);
}

void testThreeTrianglesStronglyLinked() {
    AdjacencyMatrixGraph<NamedEdge<Node<int>, int>> g;

    Node N1(1);
    Node N2(2);
    Node N3(3);
    Node N4(4);
    Node N5(5);
    Node N6(6);
    Node N7(7);
    Node N8(8);
    Node N9(9);

    g.addNode(N1);
    g.addNode(N2);
    g.addNode(N3);
    g.addNode(N4);
    g.addNode(N5);
    g.addNode(N6);
    g.addNode(N7);
    g.addNode(N8);
    g.addNode(N9);

    NamedEdge E1({&N1, &N2}, 1, false);
    NamedEdge E2({&N1, &N3}, 2, false);
    NamedEdge E3({&N2, &N3}, 3, false);
    NamedEdge E4({&N2, &N4}, 4, false);
    NamedEdge E5({&N4, &N5}, 5, false);
    NamedEdge E6({&N4, &N6}, 6, false);
    NamedEdge E7({&N6, &N5}, 7, false);
    NamedEdge E8({&N6, &N7}, 8, false);
    NamedEdge E9({&N8, &N7}, 9, false);
    NamedEdge E10({&N7, &N9}, 10, false);
    NamedEdge E11({&N8, &N9}, 11, false);
    NamedEdge E12({&N4, &N7}, 12, false);
    NamedEdge E13({&N2, &N7}, 13, false);
    NamedEdge E14({&N3, &N7}, 14, false);
    NamedEdge E15({&N3, &N6}, 15, false);
    std::vector<NamedEdge<Node<int>, int>> edges = {E1, E2, E3, E4, E5, E6, E7, E8, E9, E10, E11, E12, E13, E14,
    ↪  E15};
```

```
    g.addEdge(E1);
    g.addEdge(E2);
    g.addEdge(E3);
    g.addEdge(E4);
    g.addEdge(E5);
    g.addEdge(E6);
    g.addEdge(E7);
    g.addEdge(E8);
    g.addEdge(E9);
    g.addEdge(E10);
    g.addEdge(E11);
    g.addEdge(E12);
    g.addEdge(E13);
    g.addEdge(E14);
    g.addEdge(E15);

    testGraph("Three triangles strongly linked", &g, edges);
}

void test() {
    testThreeNodes();
    testFigureHome();
    testFull5();
    testFull6();
    testThreeTriangles();
    testThreeTrianglesStronglyLinked();
}

int main() {
    test();

    return 0;
}
```

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