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 №4.4

по дисциплине: Дискретная математика
тема: «Кратчайшие пути во взвешенном орграфе»

Выполнил: ст. группы ПВ-223
Пахомов Владислав Андреевич

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

Белгород 2023 г.

Лабораторная работа №4.4
Кратчайшие пути во взвешенном орграфе
Вариант 10

Цель работы: изучить алгоритм Дейкстры нахождения кратчайших путей между вершинами взвешенного орграфа, научиться рационально использовать его при решении различных задач.

1. Изучить алгоритм Дейкстры нахождения кратчайших путей между вершинами взвешенного орграфа.

```
template <typename N, typename V>
struct ShortestWayTree {
    // Массив V
    std::vector<bool> reachableNodes;
    // Массив D
    std::vector<V> distances;
    // Массив T
    std::vector<int> prevNodesIndices;
    std::vector<N> nodes;
    int rootNodeIndex;
    int endNodeIndex;
};
```

```
ShortestWayTree<N, EdgeValueType> getShortestWay(N* start, N* target) {
    int root = -1;
    int end = -1;
    for (int i = 0; i < this->nodes.size() && (root == -1 || end == -1); i++) {
        if (root == -1 && start->equals(*this->nodes[i])) root = i;
        if (end == -1 && target->equals(*this->nodes[i])) end = i;
    }

    if (root == -1 || end == -1) throw std::invalid_argument("Node doesn't belong to graph");

    ShortestWayTree<N, EdgeValueType> result;
    result.rootNodeIndex = root;
    result.endNodeIndex = end;

    // Шаг 1
    for (int i = 0; i < this->nodes.size(); i++) {
        result.reachableNodes.push_back(i == root);
        result.distances.push_back(i == root ? 0 : std::numeric_limits<EdgeValueType>::max());
        result.prevNodesIndices.push_back(i == root ? root : -1);
        result.nodes.push_back(*this->nodes[i]);
    }

    // Шаг 4
    while (root != end) {
        // Шаг 2
```

```
        for (int i = 0; i < this->nodes.size(); i++) {
            if (this->edges[root][i] == nullptr || result.reachableNodes[i]) continue;

            for (auto& edge : *this->edges[root][i]) {
                EdgeValueType dist = result.distances[root] + edge->current.name;
                if (dist > result.distances[i]) continue;

                result.distances[i] = dist;
                result.prevNodesIndices[i] = root;
            }
        }

        // Шаг 3
        int minLenNode = -1;
        for (int i = 0; i < this->nodes.size(); i++) {
            if (result.reachableNodes[i]) continue;

            if (minLenNode == -1) {
                minLenNode = i;
                continue;
            }

            if (result.distances[minLenNode] > result.distances[i])
                minLenNode = i;
        }

        if (minLenNode == -1 || result.distances[minLenNode] == std::numeric_limits<EdgeValueType>::max()) break;

        result.reachableNodes[minLenNode] = true;

        root = minLenNode;
    }

    return result;
}
```

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

   Определить, существуют ли во взвешенном орграфе две вершины, до которых длины кратчайших путей от заданной вершины одинаковы. Если существуют, то вывести кратчайшие пути от заданной вершины до найденных.

```
template<typename N, typename EV>
void printPath( ShortestWayTree<N, EV> shortestWayTree,
int index, bool isEnd) {
    if (index != shortestWayTree.rootNodeIndex)
        printPath(shortestWayTree, shortestWayTree.prevNodesIndices[index], false);

    std::cout << "(" << shortestWayTree.nodes[index].name << ")";
```

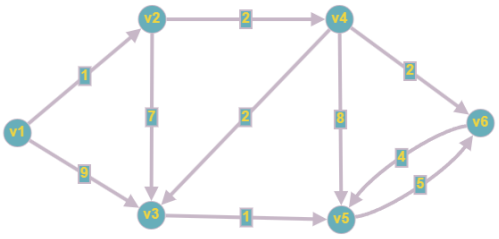
```
    if (!isEnd)
        std::cout << " ----> ";
}

template<typename T, typename N>
bool analyzeTree(N* start, T& g, std::string graphName) {
    std::map<typename T::EdgeValueType, std::pair<int, ShortestWayTree<N, typename T::EdgeValueType>>> results;

    for (int i = 0; i < g.nodes.size(); i++) {
        if (start->equals(*g.nodes[i])) continue;
        auto res = g.getShortestWay(start, g.nodes[i]);

        if (res.distances[i] == std::numeric_limits<unsigned long long>::max()) continue;
        if (results.find(res.distances[i]) == results.end()) {
            results[res.distances[i]] = {i, res};
            continue;
        }

        std::cout << "Found shortest paths with weight of " << res.distances[i] << " in graph '" << graphName <<
        ↪  "':\n";
        printPath(results[res.distances[i]].second, results[res.distances[i]].first, true);
        std::cout << "\n";
        printPath(res, i, true);
        std::cout << "\n" << std::endl;

        return true;
    }

    std::cout << "No shortest paths with same weight in graph '" << graphName << "':\n" << std::endl;
    return false;
}
```

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

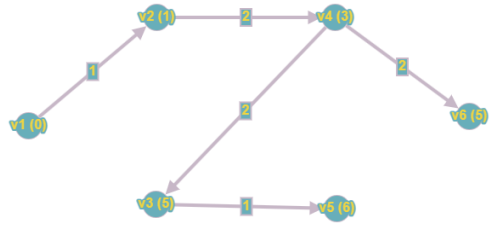
```
Found shortest paths with weight of 5 in graph 'Graph 1':
(v1) ----> (v2) ----> (v4) ----> (v3)
(v1) ----> (v2) ----> (v4) ----> (v6)

No shortest paths with same weight in graph 'Graph 2':

Found shortest paths with weight of 4 in graph 'Graph 3':
(1) ----> (2) ----> (5) ----> (6) ----> (3)
(1) ----> (4)

No shortest paths with same weight in graph 'Graph 4':

No shortest paths with same weight in graph 'Graph 5':

Found shortest paths with weight of 8 in graph 'Graph 6':
(1) ----> (4) ----> (2) ----> (3) ----> (5)
```

```
(1) ----> (4) ----> (2) ----> (6)
```

Граф 1



Полное дерево кратчайших путей с началом в вершине v0 будет выглядеть следующим образом.

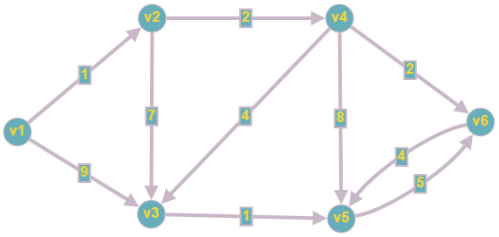


Вершины v6 и v3 будут иметь одинаковую длину 5.

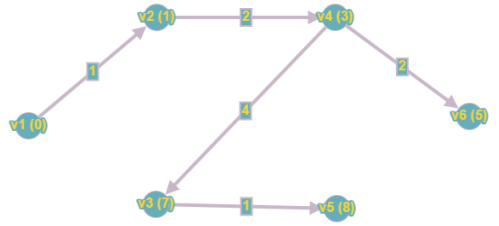
Маршрут для v6: v1 -> v2 -> v4 -> v6
Маршрут для v3: v1 -> v2 -> v4 -> v3

Результат выполнения программы совпал с предположениями.

Граф 2

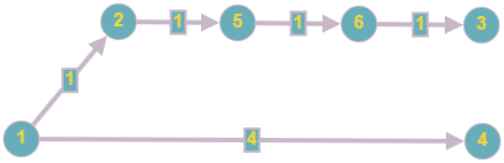


Полное дерево кратчайших путей с началом в вершине v0 будет выглядеть следующим образом.

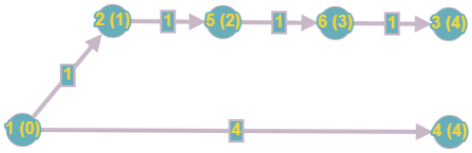


Вершин, удовлетворяющих условию, не будет.
Результат выполнения программы совпал с предположениями.

Граф 3



Полное дерево кратчайших путей с началом в вершине 1 будет выглядеть следующим образом.

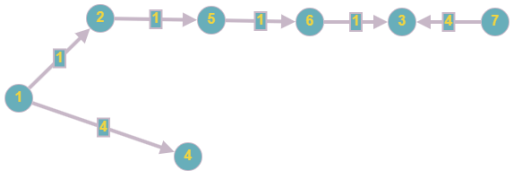


Вершины 3 и 4 будут иметь одинаковую длину 4.

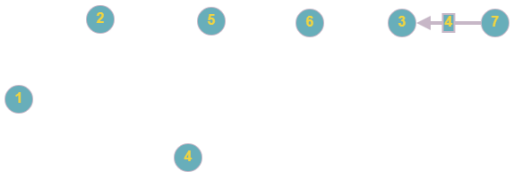
Маршрут для 3: 1 -> 2 -> 5 -> 6 -> 3
Маршрут для 4: 1 -> 4

Результат выполнения программы совпал с предположениями.

Граф 4

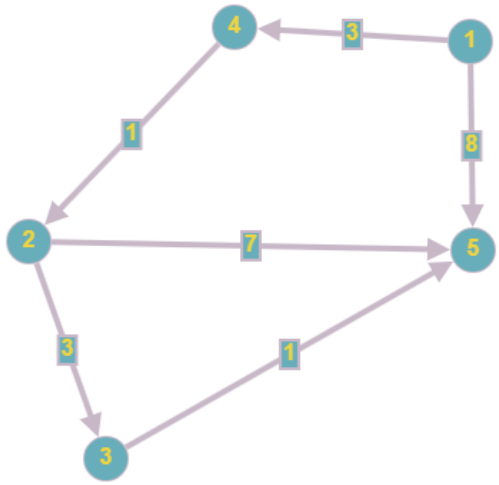


Полное дерево кратчайших путей с началом в вершине 7 будет выглядеть следующим образом.

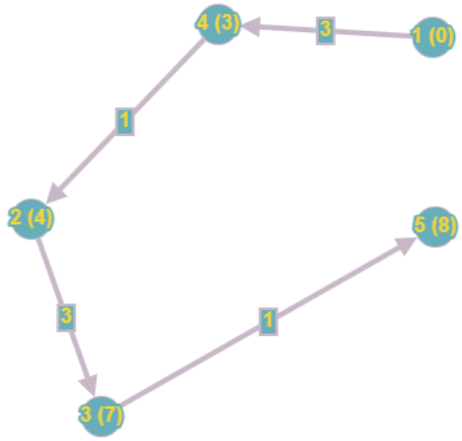


Вершин, удовлетворяющих условию, не будет.
Результат выполнения программы совпал с предположениями.

Граф 5

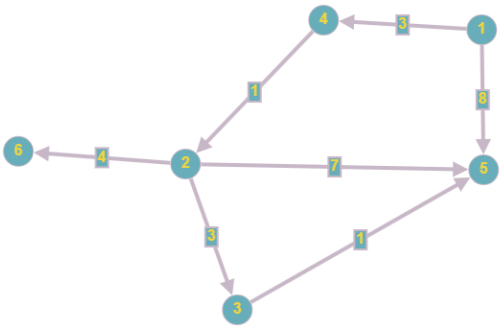


Полное дерево кратчайших путей с началом в вершине 1 будет выглядеть следующим образом.

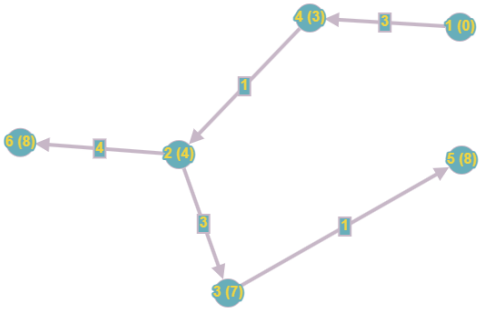


Вершин, удовлетворяющих условию, не будет.
Результат выполнения программы совпал с предположениями.

Граф 6



Полное дерево кратчайших путей с началом в вершине 1 будет выглядеть следующим образом.



Вершины 6 и 5 будут иметь одинаковую длину 8.

Маршрут для 6: 1 -> 4 -> 2 -> 6
Маршрут для 5: 1 -> 4 -> 2 -> 3 -> 5

Результат выполнения программы совпал с предположениями.

Программа:

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

#include <map>

template<typename N, typename EV>
void printPath( ShortestWayTree<N, EV> shortestWayTree,
int index, bool isEnd) {
    if (index != shortestWayTree.rootNodeIndex)
        printPath(shortestWayTree, shortestWayTree.prevNodesIndices[index], false);

    std::cout << "(" << shortestWayTree.nodes[index].name << ")";
    if (!isEnd)
        std::cout << " ----> ";
}

template<typename T, typename N>
bool analyzeTree(N* start, T& g, std::string graphName) {
    std::map<typename T::EdgeValueType, std::pair<int, ShortestWayTree<N, typename T::EdgeValueType>>> results;

    for (int i = 0; i < g.nodes.size(); i++) {
        if (start->equals(*g.nodes[i])) continue;
        auto res = g.getShortestWay(start, g.nodes[i]);

        if (res.distances[i] == std::numeric_limits<unsigned long long>::max()) continue;
        if (results.find(res.distances[i]) == results.end()) {
            results[res.distances[i]] = {i, res};
            continue;
        }

        std::cout << "Found shortest paths with weight of " << res.distances[i] << " in graph '" << graphName <<
        ↪  "':\n";
        printPath(results[res.distances[i]].second, results[res.distances[i]].first, true);
        std::cout << "\n";
        printPath(res, i, true);
```

```
        std::cout << "\n" << std::endl;

        return true;
    }

    std::cout << "No shortest paths with same weight in graph '" << graphName << "':\n" << std::endl;
    return false;
}

bool testTree1(std::string graphName) {
    AdjacencyMatrixGraph<NamedEdge<Node<std::string>, unsigned long long>> g;

    Node<std::string> N1("v1");
    Node<std::string> N2("v2");
    Node<std::string> N3("v3");
    Node<std::string> N4("v4");
    Node<std::string> N5("v5");
    Node<std::string> N6("v6");

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

    g.addEdge({{&N1, &N2}, 1, true});
    g.addEdge({{&N1, &N3}, 9, true});
    g.addEdge({{&N2, &N3}, 7, true});
    g.addEdge({{&N2, &N4}, 2, true});
    g.addEdge({{&N4, &N3}, 2, true});
    g.addEdge({{&N3, &N5}, 1, true});
    g.addEdge({{&N4, &N5}, 8, true});
    g.addEdge({{&N4, &N6}, 2, true});
    g.addEdge({{&N5, &N6}, 5, true});
    g.addEdge({{&N6, &N5}, 4, true});

    return analyzeTree(&N1, g, graphName);
}

bool testTree2(std::string graphName) {
    AdjacencyMatrixGraph<NamedEdge<Node<std::string>, unsigned long long>> g;

    Node<std::string> N1("v1");
    Node<std::string> N2("v2");
    Node<std::string> N3("v3");
    Node<std::string> N4("v4");
    Node<std::string> N5("v5");
    Node<std::string> N6("v6");

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

```
    g.addNode(N5);
    g.addNode(N6);

    g.addEdge({{&N1, &N2}, 1, true});
    g.addEdge({{&N1, &N3}, 9, true});
    g.addEdge({{&N2, &N3}, 7, true});
    g.addEdge({{&N2, &N4}, 2, true});
    g.addEdge({{&N4, &N3}, 4, true});
    g.addEdge({{&N3, &N5}, 1, true});
    g.addEdge({{&N4, &N5}, 8, true});
    g.addEdge({{&N4, &N6}, 2, true});
    g.addEdge({{&N5, &N6}, 5, true});
    g.addEdge({{&N6, &N5}, 4, true});

    return analyzeTree(&N1, g, graphName);
}

bool testTree3(std::string graphName) {
    AdjacencyMatrixGraph<NamedEdge<Node<int>, unsigned long long>> g;

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

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

    g.addEdge({{&N1, &N2}, 1, true});
    g.addEdge({{&N2, &N5}, 1, true});
    g.addEdge({{&N5, &N6}, 1, true});
    g.addEdge({{&N6, &N3}, 1, true});
    g.addEdge({{&N1, &N4}, 4, true});

    return analyzeTree(&N1, g, graphName);
}

bool testTree4(std::string graphName) {
    AdjacencyMatrixGraph<NamedEdge<Node<int>, unsigned long long>> g;

    Node<int> N1(1);
    Node<int> N2(2);
    Node<int> N3(3);
    Node<int> N4(4);
    Node<int> N5(5);
    Node<int> N6(6);
    Node<int> N7(7);
```

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

    g.addEdge({{&N1, &N2}, 1, true});
    g.addEdge({{&N2, &N5}, 1, true});
    g.addEdge({{&N5, &N6}, 1, true});
    g.addEdge({{&N6, &N3}, 1, true});
    g.addEdge({{&N1, &N4}, 4, true});
    g.addEdge({{&N7, &N3}, 4, true});

    return analyzeTree(&N7, g, graphName);
}

bool testTree5(std::string graphName) {
    AdjacencyMatrixGraph<NamedEdge<Node<int>, unsigned long long>> g;

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

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

    g.addEdge({{&N1, &N4}, 3, true});
    g.addEdge({{&N4, &N2}, 1, true});
    g.addEdge({{&N2, &N3}, 3, true});
    g.addEdge({{&N2, &N5}, 7, true});
    g.addEdge({{&N3, &N5}, 1, true});
    g.addEdge({{&N1, &N5}, 8, true});

    return analyzeTree(&N1, g, graphName);
}

bool testTree6(std::string graphName) {
    AdjacencyMatrixGraph<NamedEdge<Node<int>, unsigned long long>> g;

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

    g.addNode(N1);
```

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

    g.addEdge({{&N1, &N4}, 3, true});
    g.addEdge({{&N4, &N2}, 1, true});
    g.addEdge({{&N2, &N3}, 3, true});
    g.addEdge({{&N2, &N5}, 7, true});
    g.addEdge({{&N3, &N5}, 1, true});
    g.addEdge({{&N1, &N5}, 8, true});
    g.addEdge({{&N2, &N6}, 4, true});

    return analyzeTree(&N1, g, graphName);
}

void test() {
    assert(testTree1("Graph 1"));
    assert(!testTree2("Graph 2"));
    assert(testTree3("Graph 3"));
    assert(!testTree4("Graph 4"));
    assert(!testTree5("Graph 5"));
    assert(testTree6("Graph 6"));
}

int main() {
    test();
}
```

Вывод: в ходе лабораторной работы изучили алгоритм Дейкстры нахождения кратчайших путей между вершинами взвешенного орграфа, научились рационально использовать его при решении различных задач.