

Федеральное государственное автономное образовательное учреждение
высшего образования «Пермский национальный исследовательский
политехнический университет»

Лабораторная работа №2
«Введение в теорию графов. Алгоритмы Дейкстры и Флойда»
Вариант №22

Выполнил:

студент первого курса

ЭТФ группы РИС-23-36

Акбашева Софья Руслановна

Проверила:

Доцент кафедры ИТАС О. А. Полякова

Пермь, 2024

Введение в теорию графов. Алгоритмы Дейкстры и Флойда

Цель работы

Получить практические навыки работы с графами

Постановка задачи

1) Реализовать алгоритмы для собственного двунаправленного графа, имеющего не менее 6 вершин. Алгоритмы:

1. Обход в ширину.
2. Обход в глубину.
3. Алгоритм Дейкстры.

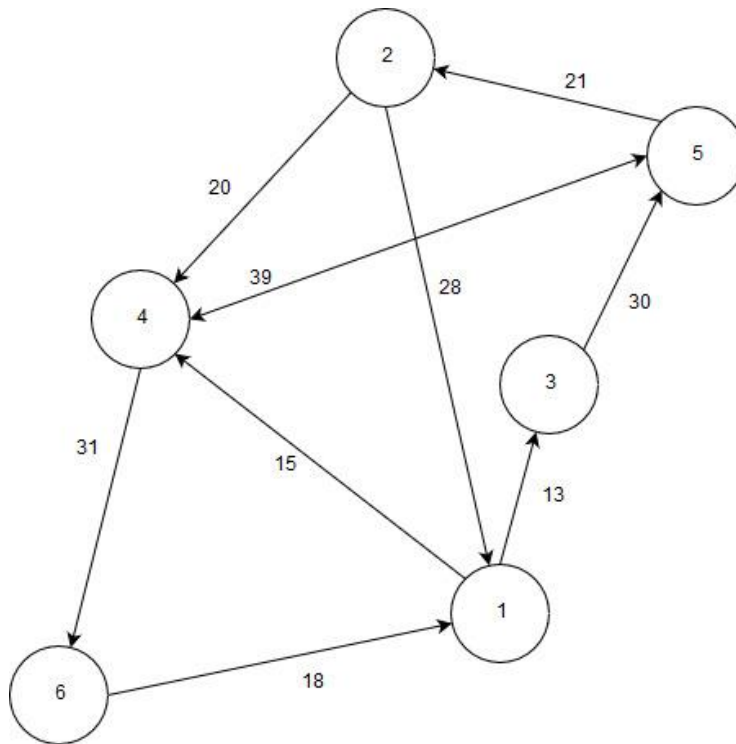
4. Выполнить задание своего варианта из методички:

Laby_Chast_3.docx (лаб работа по графам)

- 2) Разработать пользовательский интерфейс
- 3) Визуализация графа с использованием любой доступной графической библиотеки (SFML, SDL, OpenGL и подобных)
- 4) Необходимо реализовать функции для редактирования графа:
 - Создание новой вершины.
 - Удаление вершины.
 - Добавление и удаление ребра.
 - Редактирование весов ребер.
 - Редактирование матрицы смежности (или инцидентности в зависимости от реализации).
 - Реализовать вывод графа.

Вариант задания

Реализовать граф, а также Флойда, построив все необходимые матрицы. Выполнение начать с вершины 5.



Анализ задачи

1) Для визуализации графа и интерфейса будет использована библиотека SFML.

```
#include <SFML/Graphics.hpp>
```

2) Класс Graph описывает графы. Класс содержит вектор вершин, матрицу смежности и матрицы для работы алгоритма Флойда. В классе описаны методы редактирования графа (добавления графа и ребра, удаление графа и ребра), обходы графа (обходы в глубину и ширину), алгоритм Флойда и Дейкстеры, функции отрисовки графа.

3) В методе Draw происходит отрисовка графа на окне, переданном через параметр window. Код определяет размеры окна, центрирует граф и устанавливает радиус и угол для расположения вершин. Затем он проходит по списку вершин и рисует каждую из них, используя метод `calculating_node_coordinates`, который возвращает координаты вершины. После этого код рисует все рёбра графа и расстояния между вершинами, используя матрицу смежности `adjMatrix`.

4) Методы DFS и BFS реализуют обход графа в глубину и ширину соответственно. Методы принимают начальную вершину, вектор посещенных

вершин и вектор для хранения результатов обхода. В методе DFS обход начинается с заданной вершины, и все соседи этой вершины, которые еще не посещены, также посещаются рекурсивно. В методе BFS используется очередь для хранения вершин, которые должны быть посещены. Вершины добавляются в очередь, если они являются соседями текущей вершины и еще не посещены.

5) Метод `Dijkstra_3` класса `Graph` реализует алгоритм Дейкстры для нахождения кратчайших путей от одной вершины до всех остальных в графе. Метод принимает начальную вершину и возвращает вектор пар, где каждая пара содержит вершину и длину кратчайшего пути до этой вершины. В методе создается матрица путей, заполняется вектор посещенных вершин, вычисляются минимальные пути и строится вектор путей.

6) Метод `Floyd` класса `Graph` реализует алгоритм Флойда-Уоршелла для нахождения кратчайших путей между всеми парами вершин в графе. Алгоритм проходит по всем вершинам и строит все возможные маршруты из каждой вершины, используя очередь и структуру `Route`. Затем он находит кратчайшие маршруты и обновляет соответствующие матрицы.

7) Функция `calculating_node_coordinates` вычисляет координаты третьей точки, исходя из координат двух других точек и угла относительно первой точки. Используются тригонометрические функции `std::cos` и `std::sin`.

8) Функция `beam_length` вычисляет длину отрезка между двумя точками, используя теорему Пифагора и функцию `sqrt` для извлечения квадратного корня.

9) Функция `point_on_the_node_boundary` определяет координаты точки на границе узла, исходя из координат двух других точек и расстояния от одной из них. Используются функции `atan2` для определения угла и `sqrt` для извлечения квадратного корня.

10) `triangleArea` вычисляет площадь треугольника, заданного тремя точками. Используется формула площади треугольника через координаты его вершин.

11) `sideLength` вычисляет длину стороны треугольника, заданного двумя точками. Используется формула длины отрезка через координаты его концов.

12) `find_angle` ищет угол треугольника по координатам его вершин. Сначала вычисляется площадь треугольника, затем длины его сторон, и после этого используется формула для нахождения угла между двумя сторонами через площадь и длины сторон.

13) `string_to_int` преобразует строку в целое число. Используется потоковый ввод для преобразования строки в число.

14) `string_to_int_bool` проверяет, можно ли преобразовать строку в целое число. Используется тот же подход, что и в `string_to_int`, но возвращается булево значение, указывающее на успешность преобразования.

15) Для получения значения узла, который надо добавить или удалить, необходимо текстовое поле, куда пользователь будет вводить данные. Класс `TextBox` в этом коде представляет собой текстовый редактор, который может быть использован в приложениях, созданных с помощью библиотеки SFML. Он предоставляет различные методы для настройки размера, положения, текста и других параметров. Также класс имеет внутренние структуры для рисования рамки и мигающего курсора. Методы класса включают:

- `draw`: для отрисовки текстового поля и его содержимого.
- `handleEvent`: для обработки событий ввода, таких как нажатия клавиш.
- `getCurrentText`: для получения текущего текста в текстовом поле.

Кроме того, класс `TextBox` содержит вложенный класс `Text`, который управляет отображением текста внутри текстового поля. Этот класс предоставляет методы для установки текста, его позиции и размера.

16) Класс `Button` представляет собой абстрактный класс, который определяет базовые методы для всех типов кнопок. Подклассы должны реализовать эти методы для конкретной реализации кнопки.

17) Класс `RectButton` наследуется от `Button` и реализует конкретные методы для прямоугольной кнопки. Он также содержит дополнительные члены данных, такие как `sf::RectangleShape button`, которые используются для рисования самой кнопки. Конструктор `RectButton` принимает параметры для определения размеров и положения кнопки. Деструктор не делает ничего особенного, так как он пустой. Метод `getButtonStatus` обрабатывает события мыши и обновляет состояние кнопки (`isHover`, `isPressed`). Метод `draw` отвечает за отрисовку кнопки на экране. Метод `setButtonLabel` устанавливает надпись на кнопке.

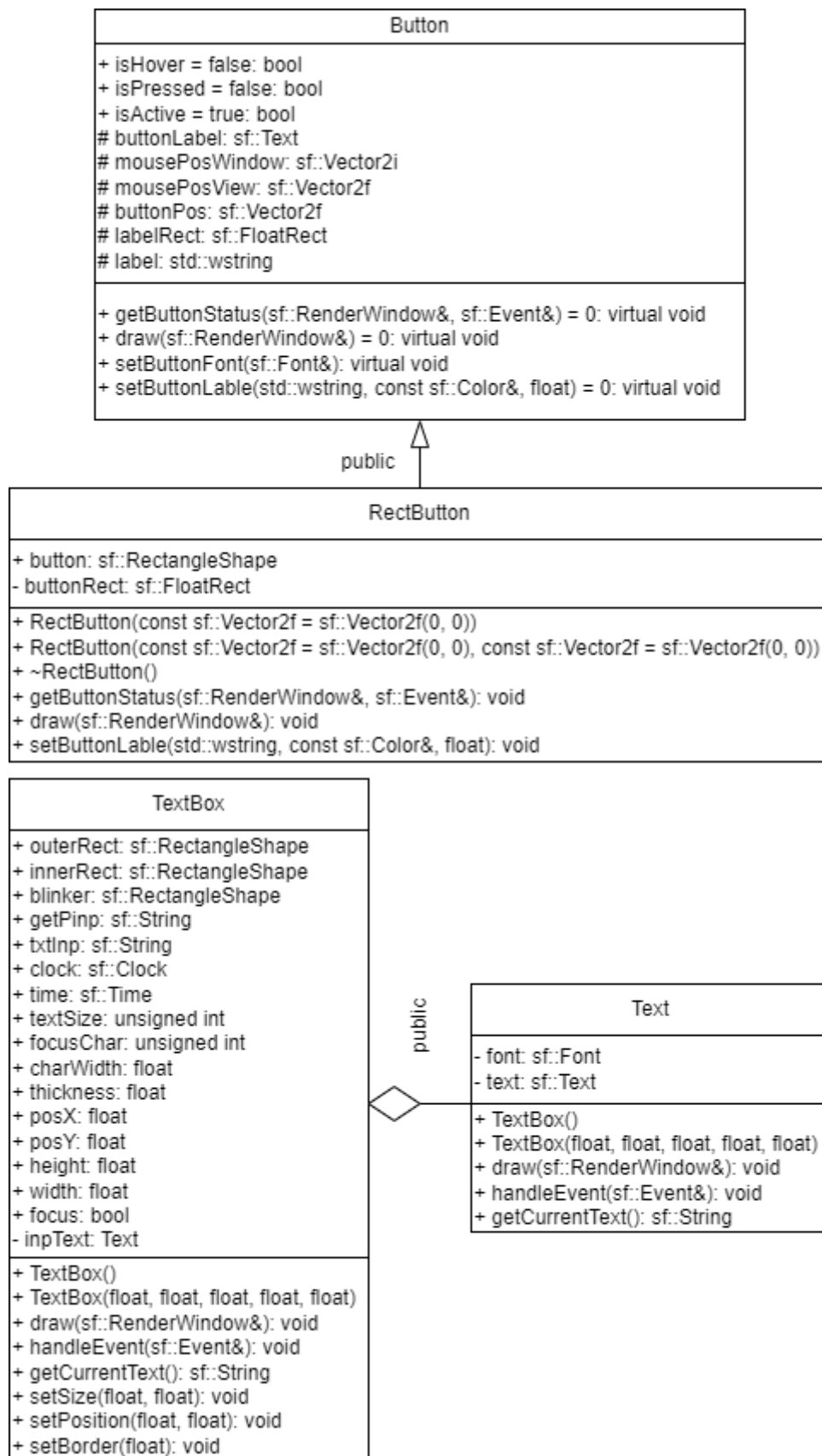
18) Функция `enter_the_data` создает окно с заголовком "Ведите..." и содержит кнопку "Продолжить". Когда пользователь нажимает кнопку, программа закрывается. В окне также есть текстовое поле, где пользователь может ввести информацию. После закрытия окна программа возвращает введенную информацию с форматом `string`. Функции `enter_the_three_data` и `enter_the_two_data` работают аналогично и используются для введения трех и двух данных соответственно.

19) Функция `error_or_success_message` создает окно с заголовком `title` и отображает сообщение `message` в центре окна. Она также добавляет кнопку "Ok" внизу окна. При нажатии на кнопку "Ok" окно закрывается. Пользовательский ввод обрабатывается через событие мыши.

20) Для использования какого-либо шрифта в интерфейсе необходимо загрузить в папку с проектом файл с расширением `.ttf`, в котором хранится необходимый шрифт.

21) В главной функции `main` реализуется меню, с помощью которого пользователь взаимодействует с гарфом.

UML диаграмма



Graph
- vertex_list: std::vector<T> - adjMatrix: std::vector<std::vector<int>> - shortest_paths_matrix: std::vector<std::vector<int>> - second_matrix: std::vector<std::vector<int>>
+ Graph(const int& size = 0) + ~Graph() {} + is_empty(): bool + insert_vertex(const T& vert): void + erase_vertex(const T& vert): void + get_vert_pos(const T& data): int + get_amount_verts(): int + get_weight(const T& vert_1, const T& vert_2): int + get_neighbors(const T& data): std::vector<T> + insert_edge_orient(const T& vert_1, const T& vert_2, int weight = 1): void + erase_edge_orient(const T& vert_1, const T& vert_2): void + get_amount_edge_orient(): int + DFS(T& start_verts, std::vector<bool>& visited_verts, std::vector<T>& vect): std::vector<T> + BFS(T& start_verts, std::vector<bool>& visited_verts, std::vector<T>& vect): std::vector<T> + Dijkstra_3(T& startVertex): std::vector<std::pair<T, int>> + Floyd(): void + reading_the_length_vector_from_Floyd(): std::vector<std::tuple<T, T, std::vector<T>>> + Draw(sf::RenderWindow& window): void + Draw_node(sf::RenderWindow& window, std::map< T, sf::Vector2f>& positions, T vertex): void + Draw_edge(sf::RenderWindow& window, std::map< T, sf::Vector2f>& positions, T vertex_1, T vertex_2): void + Draw_distance(sf::RenderWindow& window, std::map< T, sf::Vector2f>& positions, T vertex_1, T vertex_2): void

Код

Файл Graph.h

```
#pragma once
#include "other_functions.h"
#include <SFML/Graphics.hpp>
#include <vector>
#include <queue>
#include <iomanip>
#include <set>

template <class T>
class Graph {
private:
    std::vector<T> vertex_list; //вектор вершин
    std::vector<std::vector<int>> adjMatrix; //матрица смежности
    std::vector<std::vector<int>> shortest_paths_matrix; //матрица кратчайших
    std::vector<std::vector<int>> second_matrix; //вторая матрица для флойда
public:
    Graph(const int& size = 0); //конструктор с размером графа
    ~Graph() {} //деструктор
    bool is_empty(); //граф пуст?
    void insert_vertex(const T& vert); //вставка вершины
    void erase_vertex(const T& vert); //удаление вершины
    int get_vert_pos(const T& data); //ИНДЕКС вершины с переданными данными
    int get_amount_verts(); //количество существующих вершин
    int get_weight(const T& vert_1, const T& vert_2); //вес пути между вершинами
    std::vector<T> get_neighbors(const T& data); //вектор соседей элемента с
    переданными данными
    void insert_edge_orient(const T& vert_1, const T& vert_2, int weight = 1);
    //вставка ребра между двумя узлами - ОРИЕНТИРОВАННЫЙ граф
    void erase_edge_orient(const T& vert_1, const T& vert_2); //удаление ребра
    между двумя узлами
    int get_amount_edge_orient(); //количество ребер - ОРИЕНТИРОВАННЫЙ граф
```



```

        std::vector<T> DFS(T& start_verts, std::vector<bool>& visited_verts,
std::vector<T>& vect); //обход графа в ГЛУБИНУ
        std::vector<T> BFS(T& start_verts, std::vector<bool>& visited_verts,
std::vector<T>& vect); //обход графа в ШИРИНУ

        std::vector<std::pair<T, int>> Dijkstra_3(T& startVertex); //алгоритм
дейкстры

        void Floyd(); //алгоритм флойда
        std::vector<std::tuple<T, T, std::vector<T>>>
reading_the_length_vector_from_Floyd();

        void Draw(sf::RenderWindow& window); //отрисовываю граф полностью
        void Draw_node(sf::RenderWindow& window, std::map< T, sf::Vector2f>&
positions, T vertex); //отрисовываю вершину
        void Draw_edge(sf::RenderWindow& window, std::map< T, sf::Vector2f>&
positions, T vertex_1, T vertex_2); //отрисовываю ребро
        void Draw_distance(sf::RenderWindow& window, std::map< T, sf::Vector2f>&
positions, T vertex_1, T vertex_2); //отрисовываю расстояние
    };

template <class T>
Graph<T>::Graph(const int& size) { //конструктор с размером графа
    this->adjMatrix = std::vector<std::vector<T>>(size,
std::vector<T>(size)); //устанавливаю матрицу смежности
    for (int i = 0; i < size; i++) { //иду по строкам
        for (int j = 0; j < size; j++) { //иду по столбцам
            this->adjMatrix[i][j] = 0;
        }
    }
}

template <class T>
bool Graph<T>::is_empty() { //граф пуст?
    return this->vertex_list.size() == 0;
}

template <class T>
void Graph<T>::insert_vertex(const T& data) { //вставка вершины
    this->vertex_list.push_back(data); //добавляю новый узел в вектор всех узлов
    std::vector<int> tmp_1(vertex_list.size(), 0); //вектор с 0 для добавленного
узла
    std::vector<int> tmp_2(vertex_list.size(), 0);

    std::vector<int> tmp_3(vertex_list.size(), 10000);
    tmp_3[vertex_list.size() - 1] = 0;
    this->adjMatrix.push_back(tmp_1); //добавляю в матрицу новую строку
    this->second_matrix.push_back(tmp_2);
    this->shortest_paths_matrix.push_back(tmp_3);
    for (int i = 0; i < vertex_list.size() - 1; i++) {
        this->adjMatrix[i].push_back(0); //добавляю новый столбец для нового
узла
        this->second_matrix[i].push_back(0);
        this->shortest_paths_matrix[i].push_back(10000);
    }
}

template <class T>
void Graph<T>::erase_vertex(const T& data) { //удаление вершины
    int index_vert = this->get_vert_pos(data);
    if (index_vert != -1) { //если такая вершина существует
        this->vertex_list.erase(this->vertex_list.begin() +
index_vert); //удаляю вершину из вектора узлов

        this->adjMatrix[index_vert].erase(this-
>adjMatrix[index_vert].begin(), this->adjMatrix[index_vert].end());
    }
}

```

```

        this->adjMatrix.erase(this->adjMatrix.begin() + index_vert);

        this->shortest_paths_matrix[index_vert].erase(this-
>shortest_paths_matrix[index_vert].begin(), this-
>shortest_paths_matrix[index_vert].end());
        this->shortest_paths_matrix.erase(this->shortest_paths_matrix.begin()
+ index_vert);

        this->second_matrix[index_vert].erase(this-
>second_matrix[index_vert].begin(), this->second_matrix[index_vert].end());
        this->second_matrix.erase(this->second_matrix.begin() + index_vert);

        for (int i = 0; i < vertex_list.size(); i++) {
            this->adjMatrix[i].erase(this->adjMatrix[i].begin() +
index_vert);
            this->shortest_paths_matrix[i].erase(this-
>shortest_paths_matrix[i].begin() + index_vert);
            this->second_matrix[i].erase(this->second_matrix[i].begin() +
index_vert);
        }
    }

template <class T>
int Graph<T>::get_vert_pos(const T& data) { //ИНДЕКС вершины с переданными данными
    for (int i = 0; i < this->vertex_list.size(); i++) { //иду по всем вершинам
        if (this->vertex_list[i] == data) { //если вершина найдена
            return i; //возвращаю индекс
        }
    }
    return -1; //если такой вершины нет
}

template <class T>
int Graph<T>::get_amount_verts() { //количество существующих вершин
    return this->vertex_list.size(); //размер вектора вершин
}

template <class T>
int Graph<T>::get_weight(const T& vert_1, const T& vert_2) { //вес пути между
вершинами
    if (this->is_empty()) { //если нет вершин
        return 0;
    }
    int position_1 = this->get_vert_pos(vert_1); //индекс узла
    int position_2 = this->get_vert_pos(vert_2); //индекс узла
    if (position_1 == -1 || position_2 == -1) { //если к-л узла нет
        return 0;
    }
    return this->adjMatrix[position_1][position_2];
}

template <class T>
std::vector<T> Graph<T>::get_neighbors(const T& data) { //вектор соседей элемента с
переданными данными, только те соседи куда можно перейти
    std::vector<T> nbrs_list; //вектор соседей
    int pos = this->get_vert_pos(data); //индекс узла в матрице смежности
    if (pos != -1) { //если узел существует
        for (int i = 0; i < this->vertex_list.size(); i++) { //прохожу по всем
узелам
            if (this->adjMatrix[pos][i] != 0 && this->adjMatrix[pos][i] !=
10000) { //если есть путь между необходимым узлом и к-л другим
                nbrs_list.push_back(this->vertex_list[i]);
            }
        }
    }
    return nbrs_list;
}

```

```

}

template <class T>
void Graph<T>::insert_edge_orient(const T& vert_1, const T& vert_2, int weight) {
    //вставка ребра между двумя узлами
    if (this->get_vert_pos(vert_1) == -1 || this->get_vert_pos(vert_2) == -1)
    {
        //если вершин не существует
        return;
    }
    else {
        int position_1 = this->get_vert_pos(vert_1); //индекс узла
        int position_2 = this->get_vert_pos(vert_2); //индекс узла
        if (this->adjMatrix[position_2][position_1] != 0 && this->adjMatrix[position_2][position_1] != 10000) {
            this->adjMatrix[position_1][position_2] = weight;
            this->adjMatrix[position_2][position_1] = weight;
        }
        else {
            this->adjMatrix[position_1][position_2] = weight;
        }

        this->second_matrix[position_1][position_2] = vert_2;
    }
}

template <class T>
void Graph<T>::erase_edge_orient(const T& vert_1, const T& vert_2) {
    if (this->get_vert_pos(vert_1) == -1 || this->get_vert_pos(vert_2) == -1)
    {
        //если вершин не существует
        return;
    }
    else {
        int position_1 = this->get_vert_pos(vert_1); //индекс узла
        int position_2 = this->get_vert_pos(vert_2); //индекс узла
        this->adjMatrix[position_1][position_2] = 0;
        this->shortest_paths_matrix[position_1][position_2] = 10000;
    }
}

template <class T>
int Graph<T>::get_amount_edge_orient() { //количество ребер - ОРИЕНТИРОВАННЫЙ граф
    int amount = 0;
    if (!this->is_empty()) {
        for (int i = 0; i < this->vertex_list.size(); i++) { //иду по строкам
            for (int j = 0; j < this->vertex_list.size(); j++) { //иду по
                столбцам
                    if (this->adjMatrix[i][j] != 0) {
                        if (!(this->adjMatrix[i][j] != 0 && this->adjMatrix[j][i] != 10000 && this->adjMatrix[j][i] != 0 && i > j)) {
                            amount++;
                        }
                    }
                }
            }
        }
    }
    return amount;
}

template <class T>
void Graph<T>::Draw_distance(sf::RenderWindow& window, std::map < T,
sf::Vector2f>& positions, T vertex_1, T vertex_2) {
    if (!(this->adjMatrix[this->get_vert_pos(vertex_2)][this->get_vert_pos(vertex_1)] == this->adjMatrix[this->get_vert_pos(vertex_1)][this->get_vert_pos(vertex_2)] &&

```

```

        this->get_vert_pos(vertex_1) > this->get_vert_pos(vertex_2))) {
//чтоб дубликатов не было
        sf::Color arrow_color(100, 100, 100); //циферки
        sf::Vector2f positions_1 = positions[vertex_1];
        sf::Vector2f positions_2 = positions[vertex_2];

        positions_1 = point_on_the_node_boundary(positions_2, positions_1,
22);
        positions_2 = point_on_the_node_boundary(positions_1, positions_2,
22);
        positions_1 = point_on_the_node_boundary(positions_2, positions_1,
        sideLength(positions_1, positions_2) / 3 * 2);

        sf::Text text_1;
        sf::Font font;
        font.loadFromFile("ofont.ru_Desyatiy.ttf"); //загружаю шрифт
        text_1.setFont(font);

        text_1.setString(std::to_string(this->adjMatrix[this-
>get_vert_pos(vertex_1)][this->get_vert_pos(vertex_2)])); //настраиваю текст
        text_1.setFillColor(text_color);
        text_1.setCharacterSize(25);

        sf::FloatRect textRect = text_1.getLocalBounds(); //центрирую текст
        text_1.setOrigin(textRect.left + textRect.width / 2.0f, textRect.top
+ textRect.height / 2.0f);
        text_1.setPosition(positions_1);

        window.draw(text_1);
    }
}

template <class T>
void Graph<T>::Draw(sf::RenderWindow& window) {
    sf::Vector2u size_window = window.getSize();
    unsigned int width = size_window.x;
    unsigned int height = size_window.y;

    unsigned int zero_x = width / 2 + 170; //условный центр графа по x
    unsigned int zero_y = height / 2; //условный центр графа по y

    int default_radius = 80 + this->vertex_list.size() * 15;
    float default_angle = 360 / this->vertex_list.size();

    std::map<T, sf::Vector2f> Positions;
    int x = 50, y = 50;
    for (int i = 0; i < this->vertex_list.size(); i++) { //иду по всем вершинам
        Positions[this->vertex_list[i]] =
        calculating_node_coordinates(sf::Vector2f(zero_x, zero_y - default_radius),
        sf::Vector2f(zero_x, zero_y), default_angle * i);
        Draw_node(window, Positions, this->vertex_list[i]);
    }

    for (int i = 0; i < this->adjMatrix.size(); i++) { //прохожу по матрице
    смежности
        for (int j = 0; j < this->adjMatrix.size(); j++) { //прохожу по
        матрице смежности
            if (this->adjMatrix[i][j] != 0 && this->adjMatrix[i][j] !=
10000) {
                Draw_edge(window, Positions, vertex_list[i],
vertex_list[j]); //рисую стрелку
            }
        }
    }
    for (int i = 0; i < this->adjMatrix.size(); i++) { //прохожу по матрице
    смежности

```

```

        for (int j = 0; j < this->adjMatrix.size(); j++) { //прохожу по
матрице смежности
            if (this->adjMatrix[i][j] != 0 && this->adjMatrix[i][j] !=
10000) {
                Draw_distance(window, Positions, vertex_list[i],
vertex_list[j]); //пишу расстояние
            }
        }
    }
}

template <class T>
void Graph<T>::Draw_node(sf::RenderWindow& window, std::map < T, sf::Vector2f>&
positions, T vertex) {
    sf::Vector2f position = positions[vertex]; //позиция узла
    int radiys = 20;
    sf::CircleShape circle_1(radiys); //генерирую круг
    circle_1.setFillColor(node_color); //цвет внутри круга
    circle_1.setOutlineColor(text_color); //цвет снаружи круга
    circle_1.setOutlineThickness(2); //толщина внешнего контура
    circle_1.setPosition(position.x - radiys, position.y - radiys); //позиция

    sf::Text text_1;
    sf::Font font;
    font.loadFromFile("ofont.ru_Expressway.ttf"); //загружаю шрифт
    text_1.setFont(font);

    text_1.setString(std::to_string(vertex)); //настраиваю текст
    text_1.setFillColor(text_color);
    text_1.setCharacterSize(radiys);

    sf::FloatRect textRect = text_1.getLocalBounds(); //центрирую текст
    text_1.setOrigin(textRect.left + textRect.width / 2.0f, textRect.top +
textRect.height / 2.0f);
    text_1.setPosition(sf::Vector2f(position.x, position.y));

    window.draw(circle_1); //рисую круг
    window.draw(text_1); //рисую текст
}

template <class T>
void Graph<T>::Draw_edge(sf::RenderWindow& window, std::map < T, sf::Vector2f>&
positions, T vertex_1, T vertex_2) { //рисую ребро

    sf::Color arrow_color(117, 90, 87); //стрелка

    sf::Vector2f positions_1 = positions[vertex_1];
    sf::Vector2f positions_2 = positions[vertex_2];

    positions_1 = point_on_the_node_boundary(positions_2, positions_1, 22);
    positions_2 = point_on_the_node_boundary(positions_1, positions_2, 22);

    sf::VertexArray triangleStrip(sf::TriangleStrip, 4);
    if ((positions_1.x < positions_2.x && positions_1.y < positions_2.y) ||
        (positions_1.x > positions_2.x && positions_1.y > positions_2.y)) {

        triangleStrip[0].position = sf::Vector2f(positions_1.x + 1,
positions_1.y - 1);
        triangleStrip[1].position = sf::Vector2f(positions_1.x - 1,
positions_1.y + 1);
        triangleStrip[2].position = sf::Vector2f(positions_2.x + 1,
positions_2.y - 1);
        triangleStrip[3].position = sf::Vector2f(positions_2.x - 1,
positions_2.y + 1);
    }
    else {

```

```

        triangleStrip[0].position = sf::Vector2f(positions_1.x - 1,
positions_1.y - 1);
        triangleStrip[1].position = sf::Vector2f(positions_1.x + 1,
positions_1.y + 1);
        triangleStrip[2].position = sf::Vector2f(positions_2.x - 1,
positions_2.y - 1);
        triangleStrip[3].position = sf::Vector2f(positions_2.x + 1,
positions_2.y + 1);
    }

    triangleStrip[0].color = arrow_color;
    triangleStrip[1].color = arrow_color;
    triangleStrip[2].color = arrow_color;
    triangleStrip[3].color = arrow_color;

    sf::VertexArray myTriangles(sf::Triangles, 3); //сама стрелка
    double arrow_angle = find_angle(sf::Vector2f(positions_2.x, positions_2.y),
sf::Vector2f(positions_1.x, positions_1.y), sf::Vector2f(positions_2.x,
positions_2.y + 20));
    if (positions_1.x > positions_2.x) {
        arrow_angle *= -1;
    }

    sf::Vector2f point_2 =
calculating_node_coordinates(sf::Vector2f(positions_2.x - 10, positions_2.y + 20),
positions_2, arrow_angle);
    sf::Vector2f point_3 =
calculating_node_coordinates(sf::Vector2f(positions_2.x + 10, positions_2.y + 20),
positions_2, arrow_angle);

    myTriangles[0].position = sf::Vector2f(positions_2.x, positions_2.y);
    myTriangles[1].position = point_2;
    myTriangles[2].position = point_3;

    myTriangles[0].color = arrow_color;
    myTriangles[1].color = arrow_color;
    myTriangles[2].color = arrow_color;

    window.draw(myTriangles);
    window.draw(triangleStrip);
}

template <class T>
std::vector<T> Graph<T>::DFS(T& start_verts, std::vector<bool>& visited_verts,
std::vector<T>& vect) { //обход графа в глубину
    vect.push_back(start_verts);
    visited_verts[this->get_vert_pos(start_verts)] = true; //отмечаю, что
вершина посещена
    std::vector<T> neighbors = this->get_neighbors(start_verts); //соседи данной
вершины
    for (int i = 0; i < neighbors.size(); ++i) {
        if (!visited_verts[this->get_vert_pos(neighbors[i])]) { //если узел еще
не посещен
            this->DFS(neighbors[i], visited_verts, vect); //посещаю узел
        }
    }
    return vect;
}

template <class T>
std::vector<T> Graph<T>::BFS(T& start_verts, std::vector<bool>& visited_verts,
std::vector<T>& vect) {
    std::queue<T> q; // Используем очередь для хранения вершин
    q.push(start_verts); // Начинаем обход с начальной вершины
    visited_verts[get_vert_pos(start_verts)] = true; // Помечаем начальную
вершину как посещенную

```

```

        while (!q.empty()) { //пока очередь не опустеет
            T current = q.front(); //первый элемент в очереди
            q.pop(); //удаляю первый элемент
            vect.push_back(current);
            for (int i = 0; i < adjMatrix[get_vert_pos(current)].size(); ++i)
            { //пока не пройду все элементы в строке матрицы графов
                if (adjMatrix[get_vert_pos(current)][i] > 0 &&
!visited_verts[i]) { //если между вершинами есть дорога и вершина еще не посещена
                    q.push(vertex_list[i]); //добавляю вершину в очередь
ожидания посещения
                    visited_verts[i] = true; //отмечаю, что вершина посещена
                }
            }
        }
        return vect;
    }

template<class T>
std::vector<std::pair<T, int>> Graph<T>::Dijkstra_3(T& start_vertex){
    int number_of_vertices = this->get_amount_verts(); //количество вершин
    std::vector<std::vector<int>> the_path_matrix; //матрицы путей (хранит вес
ребер), где несуществующие ребра имеют бесконечный вес
    the_path_matrix.resize(number_of_vertices);
    int start_vertex_index = this->get_vert_pos(start_vertex); //индекс вершины
    for (int i = 0; i < number_of_vertices; i++)
        the_path_matrix[i].resize(number_of_vertices);

    for (int i = 0; i < number_of_vertices; i++) { //заполняю матрицу путей
        for (int j = 0; j < number_of_vertices; j++) {
            if (this->adjMatrix[i][j] != 0) { //если есть ребро
                the_path_matrix[i][j] = this->adjMatrix[i][j];
            }
            else {
                the_path_matrix[i][j] = 10000;
            }
        }
    }

    std::vector<bool> visited(number_of_vertices, false); //посещенные вершины,
изначально посещенных нет
    visited[start_vertex_index] = true; //текущая вершина посещена
    std::vector<int> minimum_paths(number_of_vertices);
    for (int i = 0; i < number_of_vertices; i++){
        minimum_paths[i] = the_path_matrix[start_vertex_index][i];
    }
    minimum_paths[start_vertex_index] = 0;
    int index_1 = 0, index_2 = 0;
    for (int i = 0; i < number_of_vertices; i++){
        int min_way = 10000;
        for (int j = 0; j < number_of_vertices; j++){
            if (!visited[j] && minimum_paths[j] < min_way){
                min_way = minimum_paths[j];
                index_1 = j;
            }
        }
        index_2 = index_1; //запоминаю индекс
        visited[index_2] = true; //вершина посещена
        for (int j = 0; j < number_of_vertices; j++){ //пока не пройду все
вершины
            if (!visited[j] && the_path_matrix[index_2][j] != 10000 &&
minimum_paths[index_2] != 10000 && (minimum_paths[index_2] +
the_path_matrix[index_2][j] < minimum_paths[j])){
                minimum_paths[j] = minimum_paths[index_2] +
the_path_matrix[index_2][j];
            }
        }
    }
}

```

```

        std::vector<std::pair<T, int>> vect_of_way;//вектор путей

        for (int i = 0; i < number_of_vertices; i++){
            if (minimum_paths[i] != 10000){
                vect_of_way.push_back(std::pair<T, int>(this->vertex_list[i],
minimum_paths[i]));
            }
            else {
                vect_of_way.push_back(std::pair<T, int>(this->vertex_list[i], -1));
            }
        }
        return vect_of_way;
    }

template<class T>
void Graph<T>::Floyd() {
    struct Route {
        std::vector<T> Verts; // Вершины маршрута
        int weight{ 0 }; // Вес маршрута
        bool isVertexExists(const T vertex) const { //проверка: маршрут
проходит через эту вершину или нет?
            return std::find(Verts.cbegin(), Verts.cend(), vertex) !=
Verts.cend();
        }
    };

    for (int i = 0, size = this->vertex_list.size(); i < size; ++i) { //прохожу
по всем вершинам
        std::vector<Route> routes; // Вектор всевозможных маршрутов из текущей
вершины

        {
            std::queue<Route> routesToWatch; // Очередь вершин, которые
нужно посмотреть

            {
                Route route; // Создание нового маршрута
                route.Verts.push_back(this->vertex_list[i]); // Добавление
в маршрут текущей вершины
                routesToWatch.push(route); // Добавление маршрута в
очередь
            }

            while (!routesToWatch.empty()) { // Цикл работает, пока очередь
не пуста
                Route currentRoute = routesToWatch.front(); //Запоминание
маршрута, находящегося в голове очереди
                routesToWatch.pop(); // Удаление маршрута из головы
очереди
                routes.push_back(currentRoute); // Добавление нового
маршрута в вектор маршрутов
                const T lastRouteVertex =
currentRoute.Verts.back(); //Создание и инициализация последней вершины маршрута
                for (const T& neighbor : this-
>get_neighbors(lastRouteVertex)) { //Цикл работает для всех соседей последней
вершины текущего рассматриваемого маршрута
                    if (!currentRoute.isVertexExists(neighbor))
                    { //Если в текущем маршруте нет вершины, соседней с последней вершиной маршрута, то
создаем новый маршрут(копия)
                        Route routeToWatch = currentRoute;

                        routeToWatch.Verts.push_back(neighbor); //Затем в маршрут-копию помещается
сосед последней вершины currentRoute
                        routeToWatch.weight += this->adjMatrix[this-
>get_vert_pos(lastRouteVertex)][this->get_vert_pos(neighbor)]; //У нового маршрута
увеличивается вес (так как в него только что добавили еще одну вершину)
                        routesToWatch.push(routeToWatch); // В
очередь заносится этот маршрут-копия
                    }
                }
            }
        }
    }
}

```



```

        routes.erase(routes.begin()); //Удаление первого маршрута из
вектора маршрутов
    }
    // Массив ассоциаций:
    std::map<T, std::pair<T, int>> shortestRoutes;
    {
        for (const Route& route : routes) { // В цикле будут выявлены
кратчайшие маршруты
            const T endVertex = route.Verts.back(); //Создание и
инициализация последней вершины маршрута
            const T stepVertex = route.Verts[1]; //Создание и
инициализация промежуточной вершины, в которую надо делать шаг
            /* нужно посмотреть shortestRoutes:
            если там до endVertex находится более короткий путь,
            то его не трогать; если его там нет для
            endVertex, то добавить пару; если он есть,
            но длиннее, то его изменить */
            if (shortestRoutes.find(endVertex) ==
shortestRoutes.end()) {
                shortestRoutes.insert(std::make_pair(endVertex,
std::make_pair(stepVertex, route.weight)));
            }
            else {
                const int minimWeight =
shortestRoutes[endVertex].second;
                if (minimWeight > route.weight) {
                    shortestRoutes[endVertex] =
std::make_pair(stepVertex, route.weight);
                }
            }
        }
        for (const std::pair<const T, std::pair<T, int>>& shortestRoute :
shortestRoutes) { // Цикл заполнения матриц данными
            const T endVertex = shortestRoute.first; //Извлечение конечной
вершины текущего рассматриваемого кратчайшего пути
            const T stepVertex = shortestRoute.second.first; // Извлечение
промежуточной вершины
            const int minWeight = shortestRoute.second.second; // Извлечение
веса кратчайшего маршрута
            this->shortest_paths_matrix[i][this->get_vert_pos(endVertex)] =
minWeight; // Заполнение первой матрицы
            this->second_matrix[i][this->get_vert_pos(endVertex)] =
stepVertex; // Заполнение второй матрицы
        }
    }

template<class T>
std::vector<std::tuple<T, T, std::vector<T>>>
Graph<T>::reading_the_length_vector_from_Floyd() {
    int cur = 0, col = 0;
    std::vector<std::tuple<T, T, std::vector<T>>> vector_of_dists;
    for (int i = 0, size = this->vertex_list.size(); i < size; ++i) {
        for (int j = 0; j < size; ++j) {
            if (this->second_matrix[i][j] != 0) { // Проверка, что есть
следующая (промежуточная) вершина
                col = j; // Запоминаем конечную вершину (в которую идем)
                cur = this->second_matrix[i][j]; // Присвоение в cur
промежуточной вершины

                std::vector<T> vect_of_way;
                vect_of_way.push_back(this->vertex_list[i]);
                while (cur != 0) { //Цикл, который идет по второй матрице;
в нем изменяется промежуточная вершина
                    vect_of_way.push_back(cur);

```

```

        if (this->second_matrix[this -
>get_vert_pos(cur)][col] != 0) { // Проверка, что есть следующая промежуточная
вершина
                                cur = this->second_matrix[this -
>get_vert_pos(cur)][col]; // Если есть, то она присваивается в cur
                                }
                                else { // Если нет, то cur обнуляется, цикл
завершится
                                cur = 0;
                                }
                                }
                                std::tuple < T, T, std::vector<T>> tuple_1(this-
>vertex_list[i], this->vertex_list[j], vect_of_way);
                                vector_of_dists.push_back(tuple_1);
                                }
                                }
                                return vector_of_dists;
                                }
}

template <class T>
void Graph_traversal(Graph<T> Graf_1, T vertex_0) { //вывод всех обходов дерева
    std::vector<std::vector<T>> vector_graf(2);
    std::vector<bool> v_bool_1(Graf_1.get_amount_verts(), false);
    std::vector<bool> v_bool_2(Graf_1.get_amount_verts(), false);
    std::vector<std::string> vect_wstring;

    Graf_1.DFS(vertex_0, v_bool_1, vector_graf[0]); //обход в глубину
    Graf_1.BFS(vertex_0, v_bool_2, vector_graf[1]); //обход в ширину

    for (int i = 0; i < 2; i++) { //иду по всем обходам
        std::string all_str = ""; //собираю строку
        for (int j = 0; j < vector_graf[i].size(); j++) {
            std::ostringstream buffet; //обрабатываю число с .
            buffet << std::fixed << std::setprecision(0) <<
vector_graf[i][j];
            all_str = all_str + buffet.str() + " "; //собираю строку
        }
        vect_wstring.push_back(all_str);
    }

    sf::RenderWindow window(sf::VideoMode(500 + Graf_1.get_amount_verts() * 10,
380), L"Обходы бинарного дерева");

    sf::Font font;
    font.loadFromFile("ofont.ru_Expressway.ttf"); //загружаю шрифт

    sf::Text obxod_binary_tree;
    obxod_binary_tree.setFont(font);
    obxod_binary_tree.setString(L"Обходы графа с вершины " +
std::to_wstring(vertex_0));
    obxod_binary_tree.setFillColor(text_color);
    obxod_binary_tree.setCharacterSize(50);
    obxod_binary_tree.setPosition(30, 10);

    sf::Text obxod_1_name;
    obxod_1_name.setFont(font);
    obxod_1_name.setString(L"Обход в ширину");
    obxod_1_name.setFillColor(text_color);
    obxod_1_name.setCharacterSize(40);
    obxod_1_name.setPosition(30, 80);

    sf::Text obxod_1_value;
    obxod_1_value.setFont(font);
    obxod_1_value.setString(vect_wstring[0]);
    obxod_1_value.setFillColor(text_color);
    obxod_1_value.setCharacterSize(40);

```

```

obxod_1_value.setPosition(30, 130);

sf::Text obxod_2_name;
obxod_2_name.setFont(font);
obxod_2_name.setString(L"Обход в глубину");
obxod_2_name.setFillColor(text_color);
obxod_2_name.setCharacterSize(40);
obxod_2_name.setPosition(30, 190);

sf::Text obxod_2_value;
obxod_2_value.setFont(font);
obxod_2_value.setString(vect_wstring[1]);
obxod_2_value.setFillColor(text_color);
obxod_2_value.setCharacterSize(40);
obxod_2_value.setPosition(30, 240);

RectButton button1(sf::Vector2f(150, 60), sf::Vector2f(window.getSize().x -
180, 300));
button1.setButtonFont(font);
button1.setButtonLabel(L"Ok", text_color, 30);

while (window.isOpen()) {
    В ОКНЕ
    sf::Vector2i mousePoz = sf::Mouse::getPosition(window); //позиция мыши

    sf::Event event;
    button1.getButtonStatus(window, event);
    while (window.pollEvent(event))
    {
        if (event.type == sf::Event::Closed)
            window.close();
        if (event.type == sf::Event::MouseButtonPressed) {
            if (event.key.code == sf::Mouse::Left) {
                if (button1.isPressed) {
                    window.close();
                }
            }
        }
        window.clear(background_color);
        button1.draw(window);
        window.draw(obxod_binary_tree);
        window.draw(obxod_1_name);
        window.draw(obxod_1_value);
        window.draw(obxod_2_name);
        window.draw(obxod_2_value);
        window.display();
    }
}

template <class T>
void all_actions_to_bypass(Graph<T>& Graf_1) { //функция для обходов дерева
    std::string vertex = enter_the_data(L"Введите вершину, с которой
реализовать обходы");
    if (string_to_int_bool(vertex)) {
        int vert_int = string_to_int(vertex); //вершина
        int index_vert = Graf_1.get_vert_pos(vert_int);
        if (index_vert != -1) {
            Graph_traversal(Graf_1, vert_int);
        }
        else {
            error_or_success_message(L"Такой вершины нет!", L"Ошибка");
        }
    }
    else {
        error_or_success_message(L"Такой вершины нет!", L"Ай-ай-ай");
    }
}

```

```

template <class T>
void running_Dijkstra_algorithm(Graph<T>& Graf_1) {
    std::string vertex = enter_the_data(L"Введите вершину, с которой
реализовать алгоритм Дейкстры");
    if (string_to_int_bool(vertex)) {
        int vert_int = string_to_int(vertex); //вершина
        if (Graf_1.get_vert_pos(vert_int) != -1) { //если вершина существует
            std::vector<std::pair<int, int>> vect_3 =
Graf_1.Dijkstra_3(vert_int);

            std::wstring first_str = L"Длина пути от вершины " +
std::to_wstring(vert_int) + L" до остальных";
            sf::RenderWindow window(sf::VideoMode(650, 200 + vect_3.size()
* 40), L"Алгоритм Дейкстры");

            sf::Font font;
            font.loadFromFile("ofont.ru_Expressway.ttf"); //загружаю шрифт

            sf::Text mes;
            mes.setFont(font);
            mes.setString(L"Алгоритм Дейкстры");
            mes.setFillColor(sf::Color::Black);
            mes.setCharacterSize(40);
            mes.setPosition(window.getSize().x / 2 - 150, 15);

            sf::Text text_1;
            text_1.setFont(font);
            text_1.setFillColor(sf::Color::Black);
            text_1.setCharacterSize(40);

            sf::Text text_2;
            text_2.setFont(font);
            text_2.setString(first_str);
            text_2.setFillColor(sf::Color::Black);
            text_2.setCharacterSize(40);
            text_2.setPosition(30, 70);

            RectButton button1(sf::Vector2f(150, 60),
sf::Vector2f(window.getSize().x / 2 - 75, window.getSize().y - 80)); //Вертикальная
печать дерева

            button1.setButtonFont(font);
            button1.setButtonLabel(L"Ok", sf::Color::Black, 30);

            while (window.isOpen()) {
                sf::Vector2i mousePoz =
sf::Mouse::getPosition(window); //позиция мыши в окне
                sf::Event event;
                button1.getButtonStatus(window, event);

                while (window.pollEvent(event))
                {
                    if (event.type == sf::Event::Closed)
                        window.close();
                    if (event.type == sf::Event::MouseButtonPressed) {
                        if (event.key.code == sf::Mouse::Left) {
                            if (button1.isPressed) {
                                window.close();
                            }
                        }
                    }
                }
                window.clear(background_color);

                button1.draw(window);
                window.draw(mes);
                window.draw(text_2);
            }
        }
    }
}

```

```

        int def_pos_y = 80;

        for (int i = 0; i < vect_3.size(); i++) {
            int cur_ver = vect_3[i].first;
            if (cur_ver != vert_int) {
                std::wstring second_str;
                if (vect_3[i].second != -1) {
                    second_str = std::to_wstring(vert_int)
+ L" -> " + std::to_wstring(vect_3[i].first) + L" = " +
std::to_wstring(vect_3[i].second);
                }
                else {
                    second_str = std::to_wstring(vert_int)
+ L" -> " + std::to_wstring(vect_3[i].first) + L" = нет пути";
                }

                text_1.setString(second_str);
                text_1.setPosition(30, def_pos_y += 40);
                window.draw(text_1);
            }
        }
        window.display();
    }
    else {
        error_or_success_message(L"Такой вершины нет!", L"Ошибка");
    }
}
else {
    error_or_success_message(L"Такой вершины нет!", L"Ай-ай-ай");
}
}

template <class T>
void running_Floyd_algorithm(Graph<T> Graf_1) { //запуск алгоритма флойда
    std::string vertex = enter_the_data(L"Введите вершину, с которой
реализовать алгоритм Флойда");
    if (string_to_int_bool(vertex)) {
        int vert_int = string_to_int(vertex); //вершина
        int index_vert = Graf_1.get_vert_pos(vert_int);
        if (index_vert != -1) {
            std::vector<T> vector_graf;
            std::vector<bool> v_bool_1(Graf_1.get_amount_verts(), false);
            Graf_1.DFS(vert_int, v_bool_1, vector_graf); //обход в глубину
            if (vector_graf.size() != 1) {
                Graf_1.Floyd();
                std::vector<std::tuple<T, T, std::vector<T>>>
vect_of_way = Graf_1.reading_the_length_vector_from_Floyd();
                bool flag = true;
                int index_t, count = 0;
                for (int i = 0; i < vect_of_way.size(); i++) {
                    if (std::get<0>(vect_of_way[i]) == vert_int) {
                        if (flag) {
                            index_t = i;
                        }
                        flag = false;
                        count++;
                    }
                }

                std::wstring first_str = L"Кратчайший путь от вершины " +
std::to_wstring(vert_int);
                sf::RenderWindow window(sf::VideoMode(600, 270 + count *
40), L"Алгоритм Флойда");

                sf::Font font;

```

шрифт

```
font.loadFromFile("ofont.ru_Expressway.ttf");//загружаю

sf::Text mes;
mes.setFont(font);
mes.setString(L"Алгоритм Флойда");
mes.setFillColor(sf::Color::Black);
mes.setCharacterSize(40);
mes.setPosition(30, 15);

sf::Text text_1;
text_1.setFont(font);
text_1.setFillColor(sf::Color::Black);
text_1.setCharacterSize(40);

sf::Text text_2;
text_2.setFont(font);
text_2.setString(first_str);
text_2.setFillColor(sf::Color::Black);
text_2.setCharacterSize(40);
text_2.setPosition(30, 70);

RectButton button1(sf::Vector2f(150, 60),
sf::Vector2f(window.getSize().x / 2 - 75, window.getSize().y - 80)); //Вертикальная
печать дерева

button1.setButtonFont(font);
button1.setButtonLabel(L"Ok", sf::Color::Black, 30);

while (window.isOpen())
{
    sf::Vector2i mousePoz =
sf::Mouse::getPosition(window); //позиция мыши в окне
    sf::Event event;
    button1.getButtonStatus(window, event);

    while (window.pollEvent(event))
    {
        if (event.type == sf::Event::Closed)
            window.close();
        if (event.type ==
sf::Event::MouseButtonPressed) {
            if (event.key.code == sf::Mouse::Left)
            {
                if (button1.isPressed) {
                    window.close();
                }
            }
        }
        window.clear(background_color);

        button1.draw(window);
        window.draw(mes);
        window.draw(text_2);

        int def_pos_y = 80;

        for (int i = index_t; i < vect_of_way.size(); i++)
        {
            int cur_ver = std::get<0>(vect_of_way[i]);
            if (cur_ver == vert_int) {
                std::wstring second_str = L" к
вершине " + std::to_wstring(std::get<1>(vect_of_way[i])) + L" : ";
                std::vector<T> vect_dist_way =
std::get<2>(vect_of_way[i]);
                for (int j = 0; j <
vect_dist_way.size(); j++) {
```

```

second_str = second_str +
std::to_wstring(vect_dist_way[j]) + L" ";
    }
    text_1.setString(second_str);
    text_1.setPosition(30, def_pos_y +=
40);
    window.draw(text_1);
    }
    }
    window.display();
    }
    }
    else {
        error_or_success_message(L"Вершина не ведет куда-либо",
L"...");
    }
    }
    else {
        error_or_success_message(L"Такой вершины нет!", L"Ошибка");
    }
    }
    else {
        error_or_success_message(L"Такой вершины нет!", L"Ай-ай-ай");
    }
    }
}

template <class T>
void add_a_vertex_completely(Graph<T>& Graf_1) { //добавляю вершину
    std::string str_vertex = enter_the_data(L"Введите название вершины, которую
хотите добавить (int)");
    if (string_to_int_bool(str_vertex)) {
        int vert_int = string_to_int(str_vertex); //вершина
        if (Graf_1.get_vert_pos(vert_int) == -1) {
            Graf_1.insert_vertex(vert_int);
            error_or_success_message(L"Вершина добавлена", L"Успех");
        }
        else {
            error_or_success_message(L"Такая вершина уже есть", L"Ошибка");
        }
    }
    else {
        error_or_success_message(L"Это не число", L"Ай-ай-ай");
    }
}

template <class T>
void delete_a_vertex_completely(Graph<T>& Graf_1) { //удаляю вершину
    std::string str_vertex = enter_the_data(L"Введите имя вершины, которую
хотите удалить (int)");
    if (string_to_int_bool(str_vertex)) {
        int vert_int = string_to_int(str_vertex); //вершина
        if (Graf_1.get_vert_pos(vert_int) != -1) {
            Graf_1.erase_vertex(vert_int);
            error_or_success_message(L"Вершина удалена", L"Успех");
        }
        else {
            error_or_success_message(L"Такой вершины нет", L"Ошибка");
        }
    }
    else {
        error_or_success_message(L"Такой вершины нет!", L"Ай-ай-ай");
    }
}

```

```

template <class T>
void add_an_edge_completely(Graph<T>& Graf_1) { //добавляю ребро
    std::string vertex_1, vertex_2, content;
    enter_the_three_data(L"Добавить/Изменить ребро...", L"Введите первую
    вершину", L"Введите вторую вершину", L"Введите расстояние между вершинами",
    vertex_1, vertex_2, content);
    if (string_to_int_bool(vertex_1) && string_to_int_bool(vertex_2))
    { //вершины-числа?
        if (string_to_int_bool(content) && vertex_1 != vertex_2)
        { //расстояние - число?
            int content_int = string_to_int(content);
            if (content_int > 0 && content_int < 10000) { //расстояние
            положительное?
                int vertex_1_int = string_to_int(vertex_1);
                int vertex_2_int = string_to_int(vertex_2);
                if (Graf_1.get_vert_pos(vertex_1_int) != -1 &&
                Graf_1.get_vert_pos(vertex_2_int) != -1) { //вершины есть в графе?
                    Graf_1.insert_edge_orient(vertex_1_int,
                    vertex_2_int, content_int);
                    error_or_success_message(L"Ребро добавлено",
                    L"Успех");
                }
                else {
                    error_or_success_message(L"Одной из вершин (или
                    обеих) не существует", L"Ошибка");
                }
            }
            else {
                error_or_success_message(L"Расстояние между вершинами не
                корректно", L"Ошибка");
            }
        }
        else {
            error_or_success_message(L"Расстояние между вершинами не
            корректно", L"Ай-ай-ай");
        }
    }
    else {
        error_or_success_message(L"Одной из вершин (или обеих) не
        существует", L"Ай-ай-ай");
    }
}

template <class T>
void delete_an_edge_completely(Graph<T>& Graf_1) { //удаление ребра
    std::string vertex_1, vertex_2;
    enter_the_two_data(L"Удалить ребро...", L"Введите первую вершину",
    L"Введите вторую вершину", vertex_1, vertex_2);
    if (string_to_int_bool(vertex_1) && string_to_int_bool(vertex_2))
    { //вершины-числа?
        int vertex_1_int = string_to_int(vertex_1);
        int vertex_2_int = string_to_int(vertex_2);
        if (Graf_1.get_vert_pos(vertex_1_int) != -1 &&
        Graf_1.get_vert_pos(vertex_2_int) != -1) { //вершины есть в графе?
            Graf_1.erase_edge_orient(vertex_1_int, vertex_2_int);
            error_or_success_message(L"Ребро удалено", L"Успех");
        }
        else {
            error_or_success_message(L"Одной из вершин (или обеих) не
            существует", L"Ошибка");
        }
    }
    else {
        error_or_success_message(L"Одной из вершин (или обеих) не
        существует", L"Ай-ай-ай");
    }
}

```



```
}
```

Файл textbox.hpp

```
#include<SFML/Graphics.hpp>
#ifndef SDX_TEXTBOX
#define SDX_TEXTBOX

namespace sdx {
    class TextBox {
    private:
        sf::RectangleShape outerRect;
        sf::RectangleShape innerRect;
        sf::RectangleShape blinker;
        sf::String getPinp;
        sf::String txtInp;
        sf::Clock clock;
        sf::Time time;

        unsigned int textSize;
        unsigned int focusChar;
        float charWidth;
        float thickness;
        float posX;
        float posY;
        float height;
        float width;
        bool focus;
    public:
        class Text {
        private:
            sf::Font font;
            sf::Text text;
        public:
            Text(sf::String, float, float); //конструктор, первый параметр -
            //текстовая строка, второй - позиция x, третий - позиция y
            sf::Text get(); //возвращает класс sf::Text, который можно отрисовать
            void setText(sf::String); //установка текста
            void setPosition(float, float); //позиция текста
            void setSize(unsigned int); //размер текста
        };
        TextBox(); //конструктор
        TextBox(float, float, float, float, float); //первые два параметра задают
        //размер, вторые два - положение, а последний - толщину
        void draw(sf::RenderWindow&); //отрисовка
        void handleEvent(sf::Event&); //обрабатывает ввод текста
        sf::String getCurrentText(); //получаю то, что в текстовом поле
    public:
        void setSize(float, float); //размер окна обновления - первый параметр
        //для x, второй для y
        void setPosition(float, float); //ставлю позицию (x,y)
        void setBorder(float); //ставлю толщину границы
    private:
        Text inpText;
    };
}
#endif
```

Файл sfml_button.hpp

```
#pragma once

#ifndef BUTTON_HPP_INCLUDED
#define BUTTON_HPP_INCLUDED
#include <SFML/Graphics.hpp>
#include <iostream>
```

```

#include <string>

const sf::Color defaultHovered = sf::Color(98, 167, 124); //цвет кнопки
const sf::Color defaultPressed = sf::Color(124, 195, 152); //цвет кнопки если она
нажата

class Button{
public:
    virtual void getButtonStatus(sf::RenderWindow&, sf::Event&) = 0; //статус
кнопки
    virtual void draw(sf::RenderWindow&) = 0; //отображение кнопки
    virtual void setButtonFont(sf::Font&); //шрифт текста на кнопке
    virtual void setButtonLabel(std::wstring, const sf::Color&, float) =
0; //установка надписи на кнопке

    bool isHover = false; //курсор наведен?
    bool isPressed = false; //нажата или нет
    bool isActive = true; //состояние кнопки

protected:
    sf::Text buttonLabel; //буквы на кнопке
    sf::Vector2i mousePosWindow; //позиция мыши
    sf::Vector2f mousePosView;
    sf::Vector2f buttonPos; //позиция кнопки
    sf::FloatRect labelRect;
    std::wstring label; //надпись
};

class RectButton : public Button{ //подкласс прямоугольных кнопок
public:
    RectButton(const sf::Vector2f = sf::Vector2f(0, 0)); //конструкторы
    RectButton(const sf::Vector2f = sf::Vector2f(0, 0), const sf::Vector2f =
sf::Vector2f(0, 0));
    ~RectButton(); //деструктор

    void getButtonStatus(sf::RenderWindow&, sf::Event&); //статус кнопки (нажата/не
нажата)
    void draw(sf::RenderWindow&); //отображение кнопки
    void setButtonLabel(std::wstring, const sf::Color&, float); //отображение
надписи
    sf::RectangleShape button;

private:
    sf::FloatRect buttonRect;
};
#endif

```

Файл main.cpp

```

#include <SFML/Graphics.hpp>
#include "Graph.h"
#include "other functions.h"

sf::Font jackInput;

int main() {
    system("chcp 1251 > Null");

    Graph<int> Graf_2; //создаю граф

    Graf_2.insert_vertex(1);
    Graf_2.insert_vertex(2);
    Graf_2.insert_vertex(3);
    Graf_2.insert_vertex(4);
    Graf_2.insert_vertex(5);
    Graf_2.insert_vertex(6);
}

```

```

Graf_2.insert_edge_orient(1, 3, 13);
Graf_2.insert_edge_orient(1, 4, 15);
Graf_2.insert_edge_orient(2, 4, 20);
Graf_2.insert_edge_orient(2, 1, 28);
Graf_2.insert_edge_orient(3, 5, 30);
Graf_2.insert_edge_orient(4, 6, 31);
Graf_2.insert_edge_orient(4, 5, 39);
Graf_2.insert_edge_orient(5, 4, 39);
Graf_2.insert_edge_orient(5, 2, 21);
Graf_2.insert_edge_orient(6, 1, 18);

sf::RenderWindow window(sf::VideoMode(1000, 570), "Graph - menu");//главное
ОКНО
jackInput.loadFromFile("ofont.ru_Nikoleta.ttf");
int tmp_size_y = window.getSize().y / 2 - 230;

sf::Text menu;
menu.setFont(jackInput);
menu.setString(L"Меню");
menu.setFillColor(text_color);
menu.setCharacterSize(40);
menu.setPosition(150, tmp_size_y);

RectButton button_1(sf::Vector2f(360, 40), sf::Vector2f(20, tmp_size_y+= 50));
button_1.setButtonFont(jackInput);
button_1.setButtonLabel(L"Обходы графа", text_color, 30);

RectButton button_2(sf::Vector2f(360, 40), sf::Vector2f(20, tmp_size_y +=
50));
button_2.setButtonFont(jackInput);
button_2.setButtonLabel(L"Алгоритм Дейкстеры", text_color, 30);

RectButton button_3(sf::Vector2f(360, 40), sf::Vector2f(20, tmp_size_y +=
50));
button_3.setButtonFont(jackInput);
button_3.setButtonLabel(L"Алгоритм Флойда", text_color, 30);

RectButton button_4(sf::Vector2f(360, 40), sf::Vector2f(20, tmp_size_y +=
50));
button_4.setButtonFont(jackInput);
button_4.setButtonLabel(L"Добавить вершину в граф", text_color, 30);

RectButton button_5(sf::Vector2f(360, 40), sf::Vector2f(20, tmp_size_y +=
50));
button_5.setButtonFont(jackInput);
button_5.setButtonLabel(L"Удалить вершину из графа", text_color, 30);

RectButton button_6(sf::Vector2f(360, 40), sf::Vector2f(20, tmp_size_y +=
50));
button_6.setButtonFont(jackInput);
button_6.setButtonLabel(L"Добавить (Изменить) ребро", text_color, 30);

RectButton button_7(sf::Vector2f(360, 40), sf::Vector2f(20, tmp_size_y +=
50));
button_7.setButtonFont(jackInput);
button_7.setButtonLabel(L"Удалить ребро из графа", text_color, 30);

RectButton button_exit(sf::Vector2f(360, 40), sf::Vector2f(20, tmp_size_y +=
80));
button_exit.setButtonFont(jackInput);
button_exit.setButtonLabel(L"Выход", text_color, 30);

while (window.isOpen()) {
    sf::Vector2i mousePoz = sf::Mouse::getPosition(window);//позиция мыши в
ОКНО
    sf::Event event;

```

```

button_1.getButtonStatus(window, event);
button_2.getButtonStatus(window, event);
button_3.getButtonStatus(window, event);
button_4.getButtonStatus(window, event);
button_5.getButtonStatus(window, event);
button_6.getButtonStatus(window, event);
button_7.getButtonStatus(window, event);
button_exit.getButtonStatus(window, event);

while (window.pollEvent(event)) {
    if (event.type == sf::Event::Closed)
        window.close();
    if (event.type == sf::Event::MouseButtonPressed) {
        if (event.key.code == sf::Mouse::Left) {
            if (button_exit.isPressed) {
                window.close();
            }
            else if (button_1.isPressed) { //обходы дерева
                all_actions_to_bypass(Graf_2);
            }
            else if (button_2.isPressed) { //алгоритм дейкстеры
                running_Dijkstra_algorithm(Graf_2);
            }
            else if (button_3.isPressed) {
                running_Floyd_algorithm(Graf_2);
            }
            else if (button_4.isPressed) { //Добавить вершину в граф
                add_a_vertex_completely(Graf_2);
            }
            else if (button_5.isPressed) { //удалить вершину
                delete_a_vertex_completely(Graf_2);
            }
            else if (button_6.isPressed) { //добавить ребро
                add_an_edge_completely(Graf_2);
            }
            else if (button_7.isPressed) { //удалить ребро
                delete_an_edge_completely(Graf_2);
            }
        }
    }

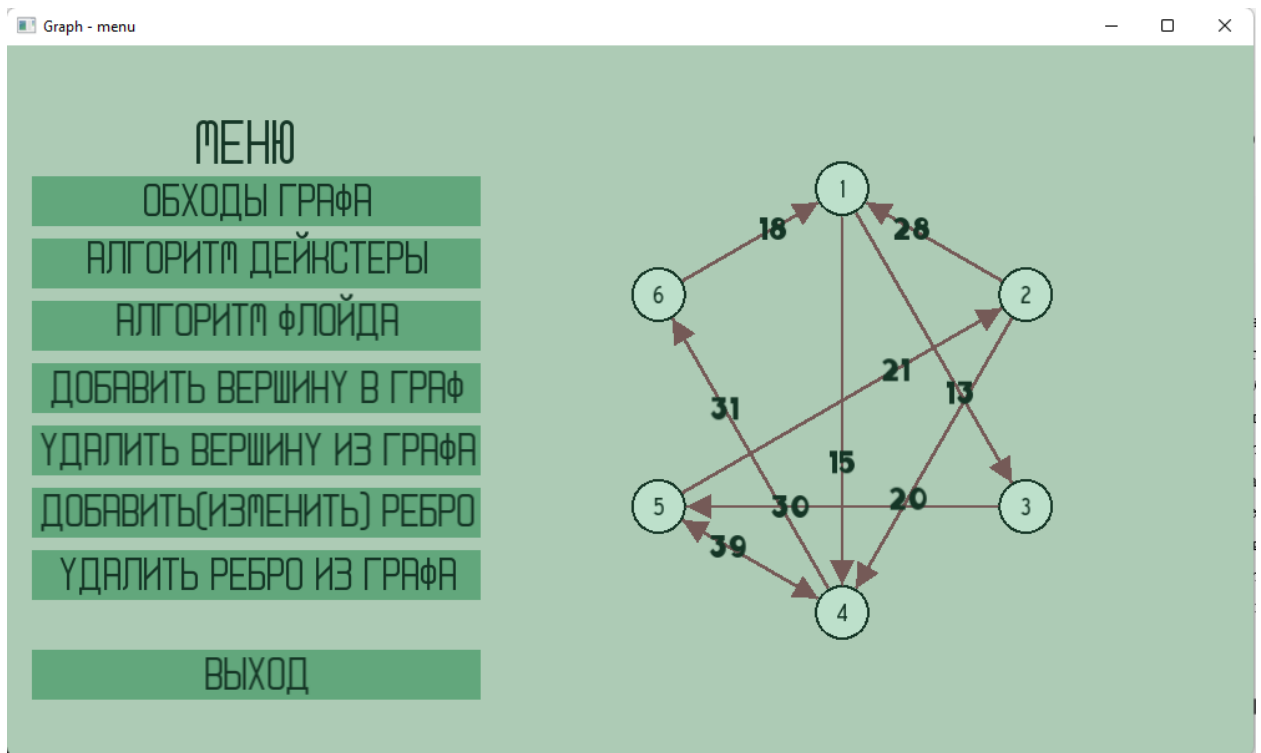
    }

    window.clear(background_color);
    if (!Graf_2.is_empty()) {
        Graf_2.Draw(window);
    }
    window.draw(menu);
    button_1.draw(window);
    button_2.draw(window);
    button_3.draw(window);
    button_4.draw(window);
    button_5.draw(window);
    button_6.draw(window);
    button_7.draw(window);
    button_exit.draw(window);

    window.display();
}
}

```

Результаты работы



Graph - menu

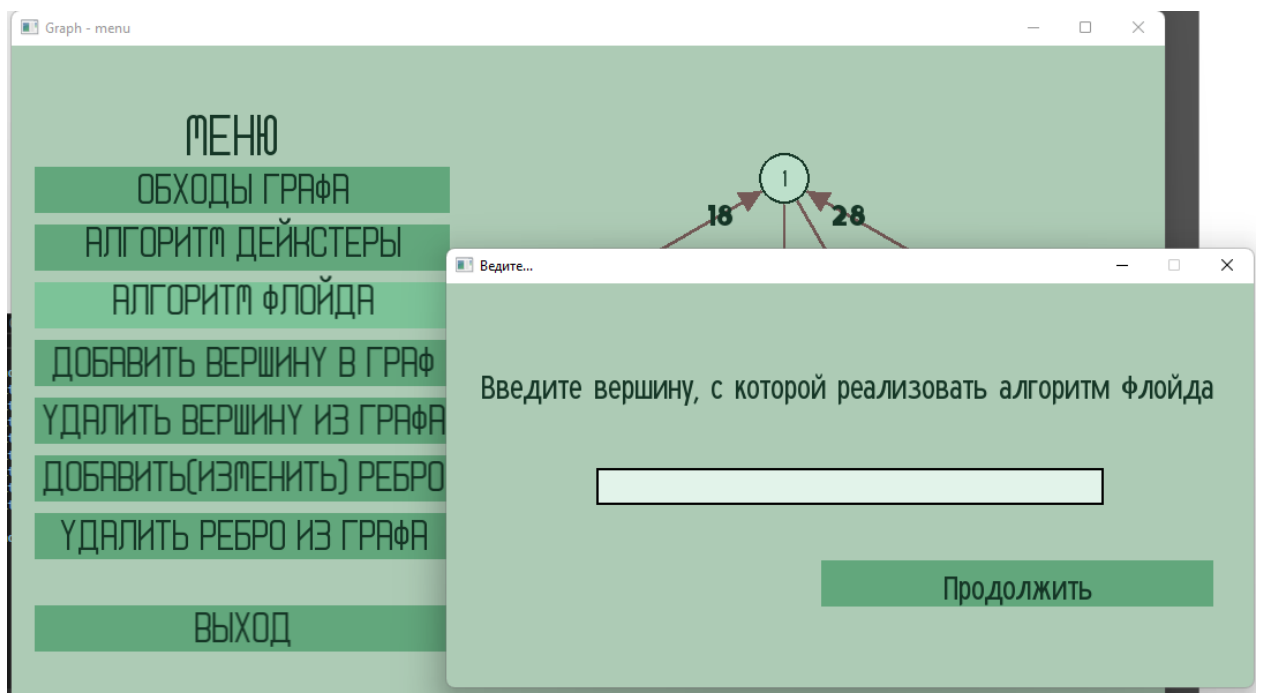
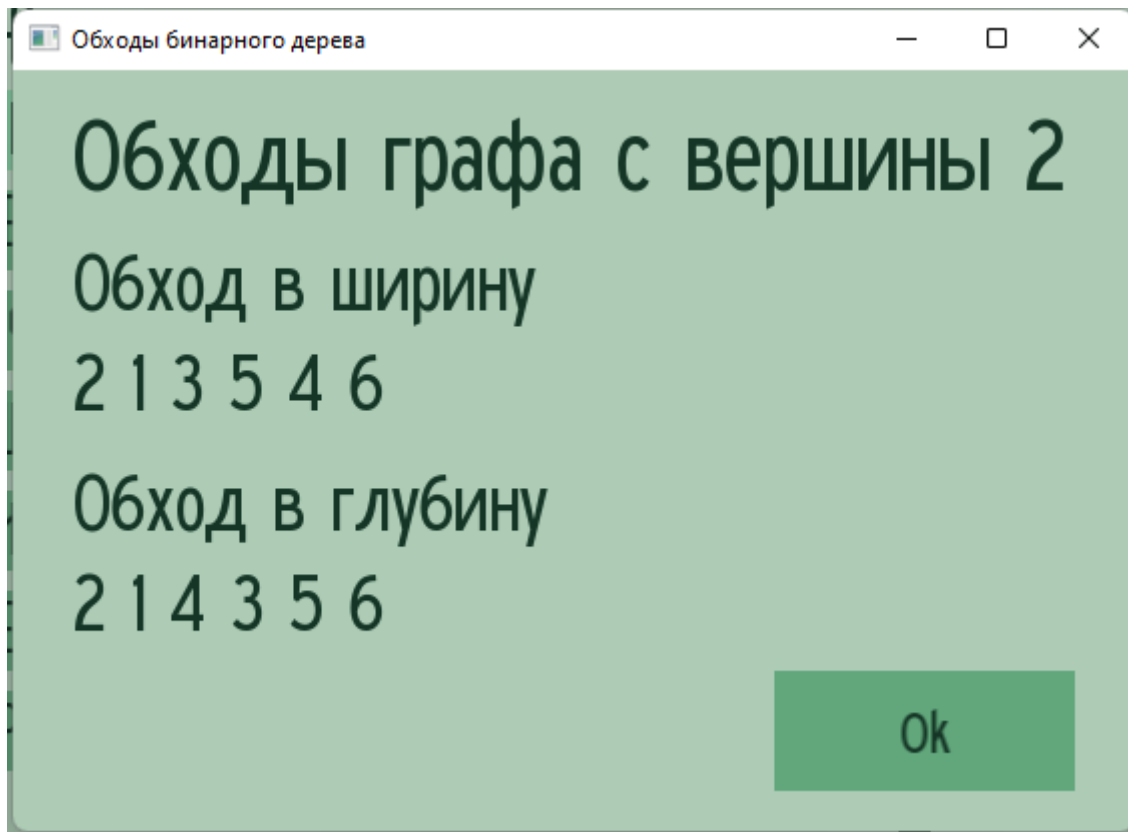
МЕНЮ

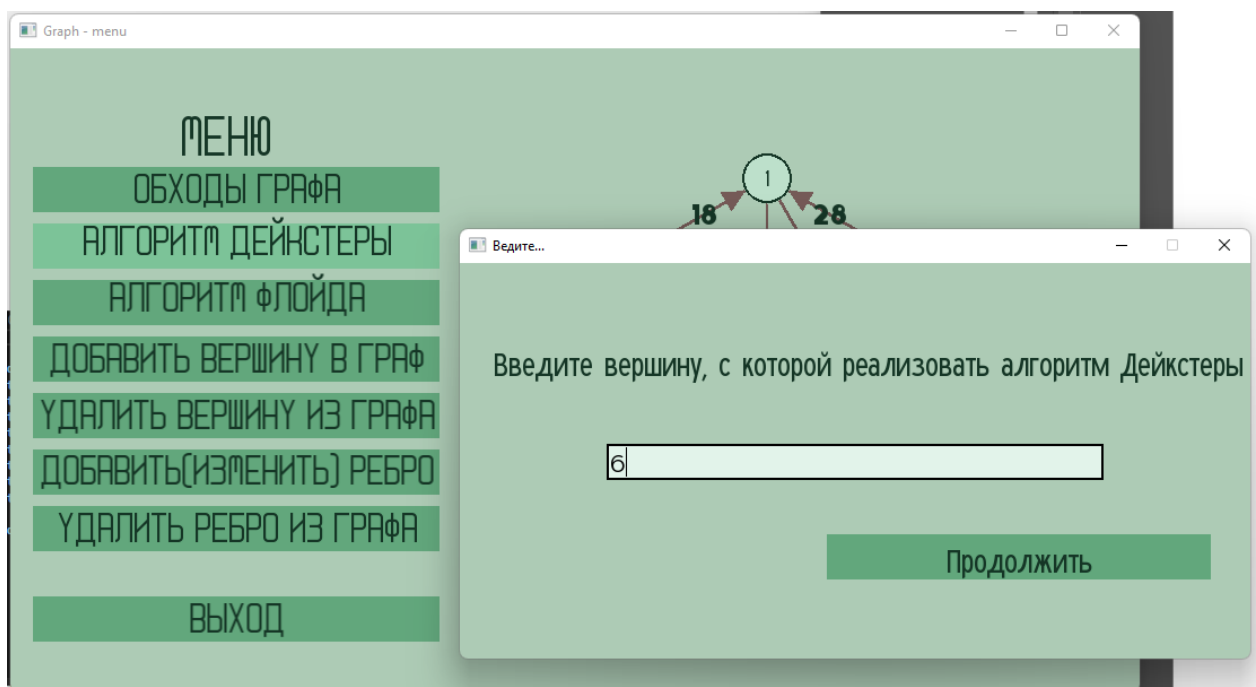
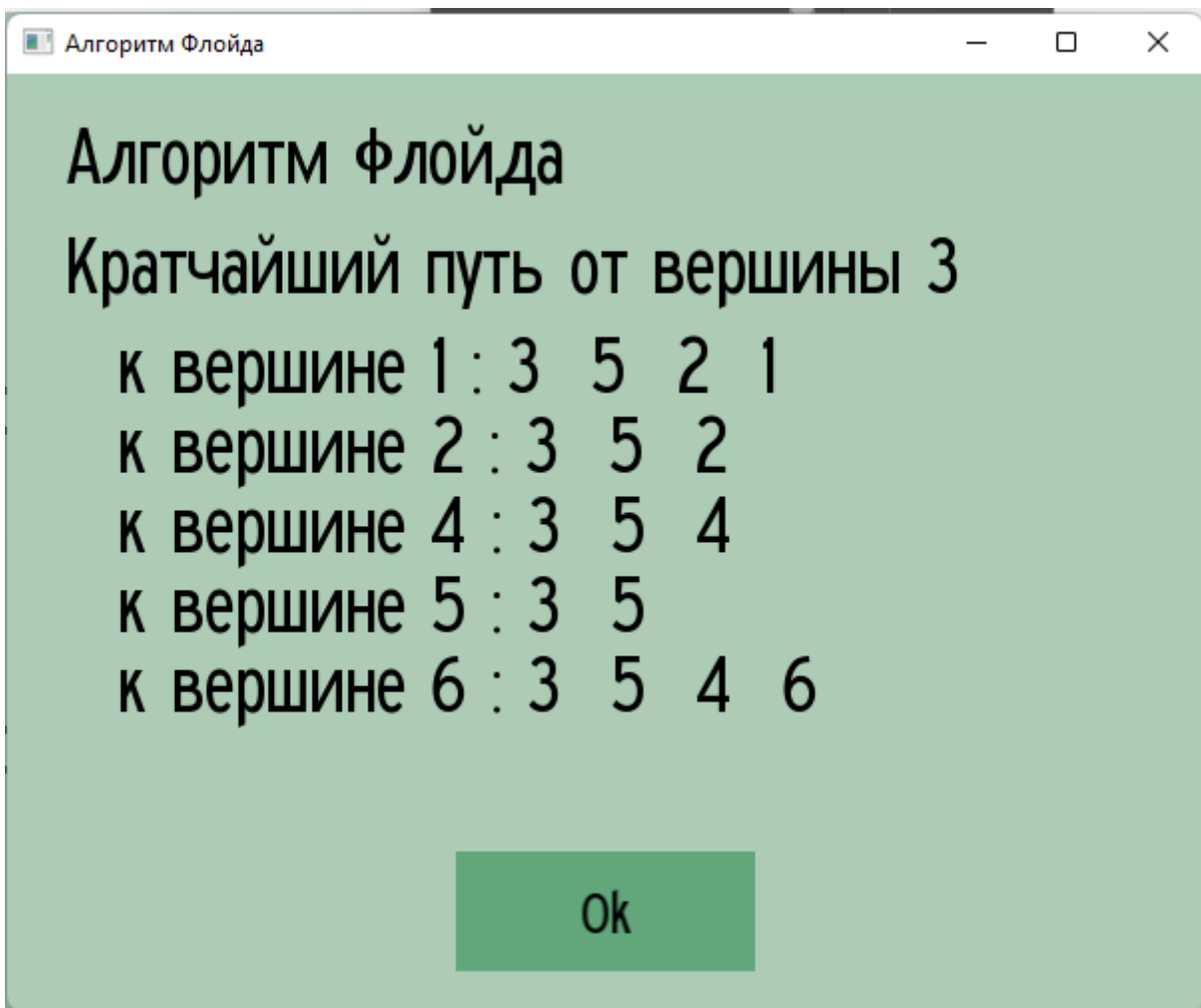
- ОБХОДЫ ГРАФА
- АЛГОРИТМ ДЕЙНСТЕРЫ
- АЛГОРИТМ ФЛОЙДА
- ДОБАВИТЬ ВЕРШИНУ В ГРАФ
- УДАЛИТЬ ВЕРШИНУ ИЗ ГРАФА
- ДОБАВИТЬ(ИЗМЕНИТЬ) РЕБРО
- УДАЛИТЬ РЕБРО ИЗ ГРАФА
- ВЫХОД

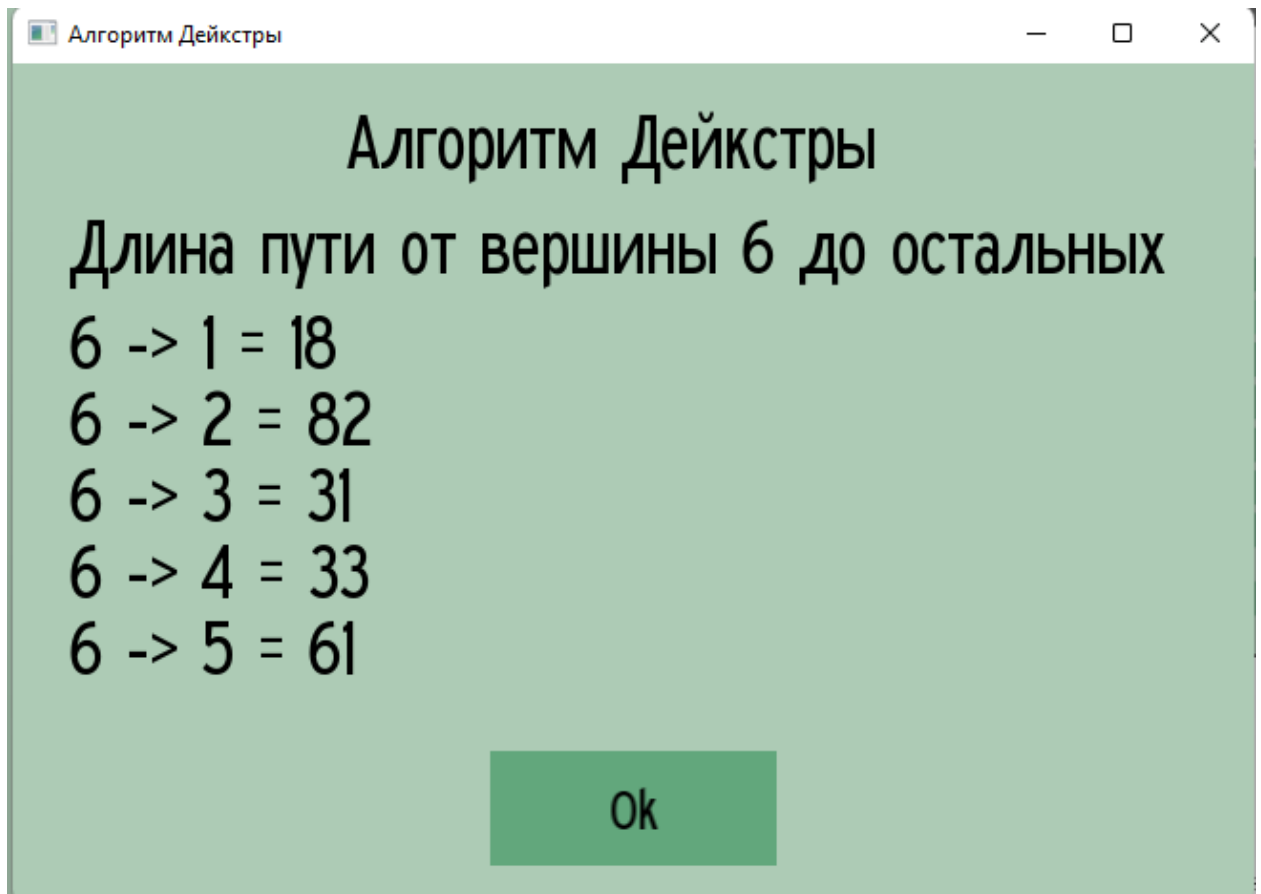
Введите...

Введите вершину, с которой реализовать обходы

Продолжить





















Вывод

В ходе работы я применила знания о работе с классами, и интерфейсами. По ходу работы был разработан граф, операции с которым выполняются посредством работы с интерфейсом, разработанным с помощью средств SFML. Были разработаны обходы графа в ширину и глубину, реализованы алгоритмы Дейкстера и Флойда, функции вывода графа. Также, были разработаны функции редактирования графа: добавление и удаление вершины, добавление и удаление ребра. В коде были реализованы особые классы, которые реализуют кнопки и текстовые боксы для упрощения реализации интерфейса. По итогу работы был реализован граф, с меню, которое позволяет управлять им.

GitHub

Ссылка: <https://github.com/SonyAkb/Laboratory-works-for-the-2-semester/tree/main/Graphs>

 SonyAkb Add files via upload

Name
 ..
 Graph.h
 README
 RectButton.cpp
 main.cpp
 monospace.ttf
 ofont.ru_Desyatiy.ttf
 ofont.ru_Expressway.ttf
 ofont.ru_Nikoleta.ttf
 other_functions.h
 sfml_button.cpp
 sfml_button.hpp
 textbox.cpp
 textbox.hpp