Санкт-Петербургский государственный университет

Группа 23.Б12-мм

Проект по анализу графов

Самохин Павел Константинович

Студенческий проект

Преподаватель:
ассистент кафедры теоретической и прикладной механики, Г.А.Нестерчук

Санкт-Петербург
2024

Оглавление

1. Введение

1.1. История теории графов

Основную идею графов впервые представил Леонгард Эйлер, одним из выдающихся математиков 18 века. Он активно работал над знаменитой «проблемой семи мостов Кенигсберга», и эти исследования считаются началом теории графов.

В 18 века прусский город Кенигсберг располагался по обе стороны реки Прегель и находился на двух больших островах — Кнайпхоф и Ломзе. Эти острова и материковая часть города соединялись семью мостами. Такая география города вдохновила математиков поставить перед собой такую задачу: придумать такой маршрут по городу, который пересекал бы каждый из этих мостов один и только один раз. Чтобы решить эту задачу, Эйлер нарисовал первое визуальное представление современного графа. Посмотрим на рисунок ниже:

Именно так и выглядит современный граф:

- Это набор точек — их называют вершинами или узлами
- Эти точки соединены линиями — их называют ребрами

Любой граф — это абстракция от практической задачи. Это видно на примере той же задачи с мостами. Найти путь по городу через семь мостов — это практическая задача. Чтобы ее решить, нужно свести решение к абстракции: извлечь из условия данные о вершинах и ребрах, а потом построить граф по ним.

В ходе своего исследования Эйлер доказал, что эта конкретная задача не имеет решения. В процессе он столкнулся с трудностью: на тот

момент в математике не было инструментов, которые могли бы подтвердить нерешаемость задачи с математической точностью. Так появилась теория графов, но почти сразу она затихла на десятилетия. В наше время она снова стала актуальна, и наконец-то ее начали активно применять.

В конечном счете теория графов изучает взаимосвязи. Учитывая набор узлов и связей, которые могут абстрагировать что угодно, от планировки города до компьютерных данных.

Именно поэтому теория графов стала полезным инструментом количественной оценки и упрощения динамических систем. С ее помощью можно решить многие проблемы с логистикой, упростить создание и оптимизацию сетей, прозрачнее согласовывать и эксплуатировать разные системы. При этом инструменты из теории графов применяются в информатике, физике, биологии, социальных науках и инженерии ПО.

1.2. Описание выбранной темы

Устойчивость графов – это важное понятие в теории графов, которое позволяет определить, насколько стабильны и надежны графы в различных ситуациях. Устойчивые графы имеют определенные свойства и критерии, которые позволяют оценить их устойчивость.

Устойчивость графов является важным понятием в теории графов и имеет множество применений в различных областях, включая компьютерные науки, транспортную логистику, социальные сети и другие. Устойчивые графы обладают определенными свойствами, которые позволяют им успешно справляться с изменениями и сохранять свою структуру и функциональность. Критерии устойчивости графов позволяют определить, насколько граф устойчив и какие изменения он может выдержать без потери своих основных свойств. Применение устойчивых графов в реальной жизни помогает оптимизировать процессы, повышать эффективность и надежность систем, а также предсказывать и анализировать различные сценарии развития. В итоге, понимание устойчивости графов является важным инструментом для разра-

ботки и оптимизации различных систем и процессов.

Внутренняя устойчивость графа

Подмножество графа является множеством внутренней устойчивости, если никакие две вершины множества не являются смежными.

Число внутренней устойчивости – число равное наибольшей из мощностей множеств внутренней устойчивости.

Внешняя устойчивость графа

Подмножество графа называется множеством внешней устойчивости, если вершины графа:

- либо являются вершинами подмножества;
- либо дуги исходящие из вершины, должны заходить в какие-то вершины, принадлежащие подмножеству.

Число внешней устойчивости – число равное наименьшей из мощностей множеств внешней устойчивости.

Ядро графа

Множество вершин, которые одновременно являются внешне и внутренне устойчивыми называются ядром графа.

В графе может быть несколько ядер (возможно, разного размера), может не быть совсем.

1.3. Постановка задачи

Цель проекта - разработать программу, которая находит числа внутренней и внешней устойчивости, ядро графа, а также множества внутренней и внешней устойчивости графа.

Программа должна:

- принимать на вход данные о графе на выбор: матрицу смежности, список ребер, список связности и матрицу инцедентности;
- находить множества вершин внутренней и внешней устойчивости графа, также максимальные по размеру внутренне устойчивые множества и минимальные по размеру внешне устойчивые множества графа;
- находить числа внутренней и внешней устойчивости графа;
- определять ядро графа;
- выводить результаты работы в удобочитаемом формате, предоставляя найденные числа и множества вершин.

2. Работа над проектом

2.1. Выбор инструментов

Для реализации своего алгоритма я использовал язык программирования C++. Для создания графического окна я использовал библиотеку Qt.

2.2. Архитектура проекта

В проекте реализованы следующие классы:

- основной класс для хранения и обработки данных графа - CGraph;
- "вспомогательный" класс для реализации некоторых методов CGraph, хранения множеств внутренней, внешней устойчивости и ядра графа - LinkedList;

3 класса для графических окон:

- основное окно - MainWindow;
- окно для ввода графа через клавиатуру - CreateGraphWindow;
- окно для визуализации графа - GraphWidget.

2 класса для визуализации графа в графическом окне:

- визуализация ребер графа - Edge;
- визуализация вершин графа - NodeVisual.

3. Документация по коду

3.1. LinkedList

Класс LinkedList - односвязный список, элементы которого являются структурой Node. Он хранит в себе числа типа int. Важно, что если далее употребляются слова "внутренний список", список "внутри" списка и т.п., то имеется в виду список вида 1230230340, где 123, 23, 34 - внутренние списки списка 1230230340 ("0" является разделителем).

```
struct Node {
    int data;
    Node* next;
    Node(int data, Node* next = nullptr) : data(data), next(next) {}
    Node(const Node& node) : data(node.data), next(nullptr) {}
    ~Node() { data = 0; next = nullptr; }
    friend std::ostream& operator<<(std::ostream& stream, Node*& node);
};

class LinkedList {
public:
    friend class CGraph;
    LinkedList() : head(nullptr) {}
    LinkedList(const LinkedList& that) : head(nullptr)
    ~LinkedList() { dispose(); }
    void pushhead(int data);
    void pushtail(int data);
    void pushTailList(LinkedList& list);
    void addList(LinkedList& add);
    void sort();
    int pophead();
    int poptail();
    int length();
    bool isEmpty() { return head == nullptr; }
    friend std::ostream& operator<<(std::ostream& stream,
                                    const LinkedList& list);
    LinkedList& operator=(LinkedList other);
private:
    Node* head;
```

```
    void dispose();
    void insert(int index, int data);
    void ExtractFirst(int d);
    void ExtractList(LinkedList& list);
    void swap(int ia, int ib);
    void getSubset(LinkedList& other, int n);
    LinkedList getSubsetByLength(int length);
    void absorption(LinkedList& other);
    bool indexValid(int index);
    bool inList(int n);
    bool ListInOtherList(LinkedList& other, bool fullList = 1);
    int getElem(int index);
    int Extract(int data);
    int countElem(int data);
};
```

оператор вывода для Node

```
friend std::ostream& operator<<(std::ostream& stream, Node*& node)
{
    stream << node->data;
    return stream;
}
```

оператор вывода для LinkedList

```
friend std::ostream& operator<<(std::ostream& stream,
                                const LinkedList& list)
{
    Node* tmp = list.head;
    while (tmp != nullptr)
    {
        stream << tmp << '' '';
        tmp = tmp->next;
    }
    return stream;
}
```

оператор присваивания для LinkedList

```
\begin{figure}
    \centering
    \includegraphics[width=0.5\linewidth]{mainWinSave.png}
    \caption{Enter Caption}
    \label{fig:enter-label}
\end{figure}
LinkedList& operator=(LinkedList other)
{
    if (other.head == nullptr)
    {
        return *this;
    }
    if (head != nullptr)
    {
        dispose();
    }
    Node* tmp = other.head;
    while (tmp != nullptr)
    {
        pushtail(tmp->data);
        tmp = tmp->next;
    }
    return *this;
}
```

конструктор копирования LinkedList

```
LinkedList(const LinkedList& that) : head(nullptr)
{
    if (that.head == nullptr)
    {
        return;
    }
    Node* tmp = that.head;
    while (tmp != nullptr)
    {
        pushtail(tmp->data);
        tmp = tmp->next;
    }
}
```

Далее разберем методы класса LinkedList:

---

pophead() удаляет первый элемент списка, возвращает его значение

```
int LinkedList::pophead()
{
    if (isEmpty())
    {
        return -1;
    }
    Node* tmp = head;
    head = head->next;
    int res = tmp->data;
    delete tmp;
    return res;
}
```

---

dispose() удаляет список

```
void LinkedList::dispose()
{
    while (!isEmpty())
    {
        pophead();
    }
}
```

---

getElem(int index) возвращает число с индексом index в списке

```
int LinkedList::getElem(int index)
{
    if (isEmpty() || !indexValid(index))
    {
        return -1;
    }
    Node* tmp = head;
    for (int i = 0; i < index; ++i)
    {
        tmp = tmp->next;
```

```
    }
    return tmp->data;
}
```

---

poptail() удаляет последний элемент списка

```
int LinkedList::poptail()
{
    if (isEmpty())
    {
        return -1;
    }
    if (head->next == nullptr)
    {
        return pophead();
    }
    Node* tmp = head;
    while (tmp->next->next != nullptr)
    {
        tmp = tmp->next;
    }
    int res = tmp->next->data;
    delete tmp->next;
    tmp->next = nullptr;
    return res;
}
```

Extract(int index) возвращает число с индексом index в списке и удаляет его из списка

```
int LinkedList::Extract(int index)
{
    if (index == 0)
    {
        return pophead();
    }
    if (index == length() - 1)
    {
```

```
        return poptail();
    }
    if (!indexValid(index))
    {
        return -1;
    }
    Node* tmp = head;
    for (int i = 0; i < index - 1; ++i)
    {
        tmp = tmp->next;
    }
    Node* nres = tmp->next;
    tmp->next = tmp->next->next;
    int res = nres->data;
    delete nres;
    return res;
}
```

ExtractFirst(int d) удаляет первое найденное число d в списке, начиная с начала списка

```
void LinkedList::ExtractFirst(int d)
{
    if (isEmpty())
    {
        return;
    }
    Node* tmp = head;
    int i = 0;
    while (tmp != nullptr)
    {
        if (tmp->data == d)
        {
            break;
        }
        tmp = tmp->next;
        i++;
    }
    if (tmp == nullptr)
```

```
    {
        return;
    }
    Extract(i);
}
```

ExtractList(LinkedList& list) удаляет из текущего списка список list

```
void LinkedList::ExtractList(LinkedList& list)
{
    if (list.isEmpty() || isEmpty())
    {
        return;
    }
    Node* tmp = list.head;
    while (tmp != nullptr)
    {
        if (inList(tmp->data))
        {
            ExtractFirst(tmp->data);
        }
        tmp = tmp->next;
    }
}
```

---

sort() сортирует список Пузырьковой сортировкой

```
void LinkedList::sort()
{
    if (isEmpty() || length() == 1)
    {
        return;
    }
    for (int i = 0; i < length(); ++i)
    {
        for (int j = 0; j < length() - 1 - i; ++j)
        {
            if (getElem(j) > getElem(j + 1))
            {
```

```
                swap(j, j + 1);
            }
        }
    }
}
```

---

swap(int ia, int ib) меняет местами элементы с индексами ia и ib

```
void LinkedList::swap(int ia, int ib)
{
    if (isEmpty() || !indexValid(ia) || !indexValid(ib))
    {
        return;
    }
    if (ia > ib)
    {
        return swap(ib, ia);
    }
    int data_b = Extract(ib);
    int data_a = Extract(ia);
    insert(ia, data_b);
    insert(ib, data_a);
}
```

---

addList(LinkedList& add) добавляет элементы списка add, которых нет в текущем списке, в конец текущего списка

```
void LinkedList::addList(LinkedList& add)
{
    if (add.isEmpty())
    {
        return;
    }
    Node* tmp = add.head;
    while (tmp != nullptr)
    {
        if (!inList(tmp->data))
        {
```

```
            pushtail(tmp->data);
        }
        tmp = tmp->next;
    }
}
```

---

pushhead(int data) добавляет число data в начало списка

```
void LinkedList::pushhead(int data)
{
^^INode* newHead = new Node(data, head);
^^Ihead = newHead;
}
```

---

length() возвращает длину списка

```
int LinkedList::length()
{
    Node* tmp = head;
    int len = 0;
    while (tmp != nullptr)
    {
        tmp = tmp->next;
        ++len;
    }
    return len;
}
```

ListInOtherList(LinkedList& other, bool fullList) если fullList == true, то возвращает true, когда текущий список совпадает с любым списком внутри other, иначе false. Если fullList == false, то возвращает true, когда текущий список есть внутри любого списка внутри other (список other должен быть вида 12304302340, где 123, 43, 234 - внутренние списки списка other)

```
bool LinkedList::ListInOtherList(LinkedList& other, bool fullList)
{
```

```
    if (isEmpty())
    {
        return true;
    }
    if (other.isEmpty())
    {
        return false;
    }
    Node* main = other.head;
begin:
    Node* tmp = head;
    int tmpindex = 0;
    while (tmp != nullptr && main != nullptr)
    {
        if (main->data == 0)
        {
            main = main->next;
            goto begin;
        }
        else if (tmp->data == main->data)
        {
            tmp = tmp->next;
        }
        tmpindex++;
        main = main->next;
    }
    if (tmp == nullptr)
    {
        if (fullList == 1)
        {
            if (main != nullptr)
            {
                if (main->data == 0 && length()
                        == tmpindex)
                {
                    return true;
                }
                else
                {
                    goto begin;
```

```
                }
            }
        }
        else
        {
            return true;
        }
    }
    return false;
}
```

---

absorption(LinkedList& other) абсорбирует внутренние списки текущего списка списком other. К примеру, если текущий 1230230, other 23, то текущий станет равным 230

```
void LinkedList::absorption(LinkedList& other)
{
    if (isEmpty() || other.isEmpty())
    {
        return;
    }
    int index = 0;
    Node* main = head;
    while (other.ListInOtherList(*this, 0) && main != nullptr)
    {
        Node* tmp = other.head;
        int tmpindex = 0;
        while (tmp != nullptr && main != nullptr)
        {
            if (main->data == 0)
            {
                break;
            }
            if (tmp->data == main->data)
            {
                tmp = tmp->next;
            }
            main = main->next;
            index++;
```

```
            tmpindex++;
        }
        if (main != nullptr)
        {
            if (main->data == 0 && other.length()
                    == tmpindex)
            {
                index++;
                tmpindex++;
                main = main->next;
                continue;
            }
            while (main->data != 0)
            {
                main = main->next;
                index++;
                tmpindex++;
                if (main == nullptr)
                {
                    break;
                }
            }
            index++;
            tmpindex++;
            main = main->next;
        }
        if (tmp == nullptr)
        {
            int c = tmpindex;
            while (c != 0)
            {
                Extract(index - tmpindex);
                c--;
            }
            index -= tmpindex;
        }
    }
}
```

inList(int n) возвращает true, если число n есть в списке

```
bool LinkedList::inList(int n)
{
    if (isEmpty())
    {
        return false;
    }
    Node* tmp = head;
    while (tmp != nullptr)
    {
        if (tmp->data == n)
        {
            return true;
        }
        tmp = tmp->next;
    }
    return false;
}
```

---

indexValid(int index) возвращает true, если index не выходит за пределы списка

```
bool LinkedList::indexValid(int index)
{
    return (0 <= index && index < length());
}
```

---

pushtail(int data) добавляет число data в конец списка

```
void LinkedList::pushtail(int data)
{
    if (isEmpty())
    {
        return pushhead(data);
    }
    Node* tmp = head;
    while (tmp->next != nullptr)
    {
```

```
        tmp = tmp->next;
    }
    tmp->next = new Node(data);
}
```

---

insert(int index, int data) вставка числа data в место с индексом index

```
void LinkedList::insert(int index, int data)
{
    if (index == 0)
    {
        return pushhead(data);
    }
    if (index == length())
    {
        return pushtail(data);
    }
    if (indexValid(index))
    {
        Node* tmp = head;
        for (int i = 0; i < index - 1; ++i)
        {
            tmp = tmp->next;
        }
        tmp->next = new Node(data, tmp->next);
    }
}
```

---

countElem(int data) возвращает количество чисел data в списке

```
int LinkedList::countElem(int data)
{
    if (isEmpty())
    {
        return 0;
    }
    int count = 0;
    Node* tmp = head;
    while (tmp != nullptr)
```

```
    {
        if (tmp->data == data)
        {
            count++;
        }
        tmp = tmp->next;
    }
    return count;
}
```

---

pushTailList(LinkedList& list) добавляет список list в конец текущего списка

```
void LinkedList::pushTailList(LinkedList& list)
{
    if (list.isEmpty())
    {
        return;
    }
    Node* tmp = list.head;
    while (tmp != nullptr)
    {
        pushtail(tmp->data);
        tmp = tmp->next;
    }
}
```

---

getSubset(LinkedList& other, int n) присваивает текущему списку внутренний список списка other, который стоит на n-том месте (например, если other 1230230, n == 2, текущий станет равным 23)

```
void LinkedList::getSubset(LinkedList& other, int n)
{
    if (other.isEmpty() || n < 1 || n > other.countElem(0))
    {
        return;
    }
    if (!isEmpty())
```

```
    {
        dispose();
    }
    Node* tmp = other.head;
    while (n != 1)
    {
        if (tmp != nullptr)
        {
            while (tmp->data != 0)
            {
                tmp = tmp->next;
                if (tmp == nullptr)
                {
                    break;
                }
            }
        }
        if (tmp != nullptr)
        {
            tmp = tmp->next;
        }
        n--;
    }
    if (tmp != nullptr)
    {
        while (tmp->data != 0)
        {
            pushtail(tmp->data);
            tmp = tmp->next;
            if (tmp == nullptr)
            {
                return;
            }
        }
    }
}
```

---

getSubsetByLength(int length) возвращает внутренний список текущего списка, длина которого равна length (например, если текущий спи-

сок 1230230, length == 2, вернется список 23)

```
LinkedList LinkedList::getSubsetByLength(int length)
{
    if (isEmpty())
    {
        return LinkedList();
    }
    Node* tmp = head;
    LinkedList outputList;
    while (tmp != nullptr)
    {
        LinkedList tmpList;
        for (int i = 0; i < length; ++i)
        {
            if (tmp == nullptr)
            {
                break;
            }
            tmpList.pushtail(tmp->data);
            if (tmp->data == 0)
            {
                tmpList.dispose();
                i = -1;
            }
            tmp = tmp->next;
        }
        if (tmp != nullptr)
        {
            if (tmp->data == 0)
            {
                outputList.pushTailList(tmpList);
            }
            else
            {
                while (tmp->data != 0)
                {
                    tmp = tmp->next;
                    if (tmp == nullptr)
                    {
```

```
                    break;
                }
            }
        }
    }
    else
    {
        outputList.pushTailList(tmpList);
    }
}
return outputList;
}
```

3.2. CGraph

Класс CGraph хранит в себе информацию о графе: число вершин "_vertexes", матрицу смежности "_matrix", множества внутренней и внешней устойчивости "_IntStabList и _ExtStabList", множество ядер графа "_KernelList".

```
class CGraph {
public:
    friend class LinkedList;
    CGraph() : _vertexes(0), _matrix(nullptr), _IntStabList(),
                            _ExtStabList(), _KernelList() {}
    CGraph(int vertexes) : _vertexes(vertexes), _IntStabList(),
                                _ExtStabList(), _KernelList()
    {
        initMatrix();
    }
    CGraph(const CGraph& that);
    ~CGraph()
    {
        dispose();
    }
    CGraph& operator=(CGraph& that);
    std::string ReadMatrix(std::string& inputStr);
    std::string ReadEdges(std::string &inputStr);
    std::string ReadAdjacences(std::string &inputStr);
```

```
    std::string ReadIncidenceMatrix(std::string &inputStr);
    std::string GetEdgesList();
    std::string getIntStabilityList(bool word = 0);
    std::string getExtStabilityList(bool word = 0);
    std::string getKernelList(bool word = 0);
    std::string getMaxIntStList(bool word = 0);
    std::string getMinExtStList(bool word = 0);
    bool vertexIsValid(int v);
    bool indexIsValid(int i);
    int getISN();
    int getESN();
    int getEdgesCount();
    bool isEmpty();
    bool isConnected();
    void initStabilityLists();

private:
    void initMatrix();
    void initISList();
    void initESList();
    void initKernelList();
    void disposeMatrix();
    void dispose();
    void doSymmetric();
    void doReflexive(bool t);
    void weightUnimportant();
    LinkedList NeiboursList(int v);
    LinkedList InputVertexesList(int v);
    void ISList(int v, LinkedList& currentList, LinkedList list,
                                                LinkedList tmp);
    void ESList(int v, LinkedList& currentList, LinkedList list,
                                                LinkedList tmp);

    int _vertexes;
    int** _matrix;
    LinkedList _IntStabList;
    LinkedList _ExtStabList;
    LinkedList _KernelList;
};
```

конструктор копирования

```
CGraph(const CGraph& that) : _vertexes(that._vertexes)
{
    if (that._matrix != nullptr)
    {
        _matrix = new int* [_vertexes];
        for (int i = 0; i < _vertexes; ++i)
        {
            _matrix[i] = new int[_vertexes] { 0 };
        }
        for (int i = 0; i < _vertexes; ++i)
        {
            for (int j = 0; j < _vertexes; ++j)
            {
                _matrix[i][j] = that._matrix[i][j];
            }
        }
    }
    else
    {
        initMatrix();
    }
    if (that._IntStabList.head != nullptr)
    {
        _IntStabList.dispose();
        LinkedList tmpList(that._IntStabList);
        for (int i = 0; i < tmpList.length(); ++i)
        {
            _IntStabList.pushtail(tmpList.getElem(i));
        }
    }
    else
    {
        LinkedList _IntStabList;
    }
    if (that._ExtStabList.head != nullptr)
    {
        _ExtStabList.dispose();
```

```
        LinkedList tmpList(that._ExtStabList);
        for (int i = 0; i < tmpList.length(); ++i)
        {
            _ExtStabList.pushtail(tmpList.getElem(i));
        }
    }
    else
    {
        LinkedList _ExtStabList;
    }
    if (that._KernelList.head != nullptr)
    {
        _KernelList.dispose();
        LinkedList tmpList(that._KernelList);
        for (int i = 0; i < tmpList.length(); ++i)
        {
            _KernelList.pushtail(tmpList.getElem(i));
        }
    }
    else
    {
        LinkedList _KernelList;
    }
}
```

оператор присваивания

```
CGraph& CGraph::operator=(CGraph& that)
{
    if (that._matrix != nullptr)
    {
        disposeMatrix();
        _vertexes = that._vertexes;
        _matrix = new int* [_vertexes];
        for (int i = 0; i < _vertexes; ++i)
        {
            _matrix[i] = new int[_vertexes] { 0 };
        }
        for (int i = 0; i < _vertexes; ++i)
```

```
        {
            for (int j = 0; j < _vertexes; ++j)
            {
                _matrix[i][j] = that._matrix[i][j];
            }
        }
    }
    else
    {
        initMatrix();
    }
    return *this;
}
```

Далее разберем методы класса CGraph:

---

initMatrix() создает матрицу смежности из нулей размера _vertexes

```
void CGraph::initMatrix()
{
    if (_vertexes == 0)
    {
        return;
    }
    _matrix = new int* [_vertexes];
    for (int i = 0; i < _vertexes; ++i)
    {
        _matrix[i] = new int[_vertexes] { 0 };
    }
}
```

---

dispose() удаляет матрицу смежности и обнуляет число вершин

```
void CGraph::dispose()
{
    disposeMatrix();
    _vertexes = 0;
}
```

---

disposeMatrix() удаляет матрицу смежности

```
void CGraph::disposeMatrix()
{
    if (_matrix != nullptr)
    {
        for (int i = 0; i < _vertexes; ++i)
        {
            delete[] _matrix[i];
        }
        delete[] _matrix;
        _matrix = nullptr;
    }
}
```

GetEdgesList() возвращает список ребер в std::string

```
std::string CGraph::GetEdgesList()
{
    if (_matrix == nullptr)
    {
        return ''Graph empty'';
    }
    std::ostringstream SS;
    SS << _vertexes << '' '' << getEdgesCount() << '\n';
    for (int i = 0; i < _vertexes; ++i)
    {
        for (int j = 0; j < _vertexes; ++j)
        {
            if (_matrix[i][j] != 0)
            {
                SS << i + 1 << '' '' << j + 1 << '\n';
            }
        }
    }
    std::string outputString;
    outputString = SS.str();
    return outputString;
}
```

---

doReflexive(bool t) если t == true, делает граф рефлексивным, иначе делает граф антирефлексивным

```
void CGraph::doReflexive(bool t)
{
    if (_matrix == nullptr)
    {
        return;
    }
    for (int i = 0; i < _vertexes; ++i)
    {
        for (int j = 0; j < _vertexes; ++j)
        {
            if (i == j)
            {
                _matrix[i][j] = t;
            }
        }
    }
}
```

---

weightUnimportant() если вес ребра не равен 0, делает его равным 1 для каждого ребра

```
void CGraph::weightUnimportant()
{
    if (isEmpty())
    {
        return;
    }
    for (int i = 0; i < _vertexes; ++i)
    {
        for (int j = 0; j < _vertexes; ++j)
        {
            if (_matrix[i][j] != 0)
            {
                _matrix[i][j] = 1;
            }
```

```
        }
    }
}
```

---

doSymmetric() делает граф симметричным

```
void CGraph::doSymmetric()
{
    if (_matrix == nullptr)
    {
        return;
    }
    for (int i = 0; i < _vertexes; ++i)
    {
        for (int j = 0; j < _vertexes; ++j)
        {
            if (_matrix[i][j] > 0)
            {
                _matrix[j][i] = _matrix[i][j];
            }
        }
    }
}
```

---

getISN() возвращает число внутренней устойчивости

```
int CGraph::getISN()
{
    if (_IntStabList.isEmpty())
    {
        return 0;
    }
    int count = 0;
    int tmpCount = 0;
    Node* tmp = _IntStabList.head;
    while (tmp != nullptr)
    {
        if (tmp->data == 0)
        {
```

```
            count = (tmpCount > count ? tmpCount : count);
            tmpCount = 0;
        }
        else
        {
            tmpCount++;
        }
        tmp = tmp->next;
    }
    return count;
}
```

---

getESN() возвращает число внешней устойчивости

```
int CGraph::getESN()
{
    if (_ExtStabList.isEmpty())
    {
        return 0;
    }
    int count = 0;
    int tmpCount = 0;
    Node* tmp = _ExtStabList.head;
    while (tmp != nullptr)
    {
        if (tmp->data == 0)
        {
            if (count == 0)
            {
                count = tmpCount;
            }
            else
            {
                count = (tmpCount < count ? tmpCount : count);
            }
            tmpCount = 0;
        }
        else
        {
```

```
            tmpCount++;
        }
        tmp = tmp->next;
    }
    return count;
}
```

---

getEdgesCount() возвращает число дуг

```
int CGraph::getEdgesCount()
{
    if (_matrix == nullptr)
    {
        return 0;
    }
    int n = 0;
    for (int i = 0; i < _vertexes; ++i)
    {
        for (int j = 0; j < _vertexes; ++j)
        {
            if (_matrix[i][j] != 0)
            {
                n++;
            }
        }
    }
    return n;
}
```

---

initKernelList() создает и заполняет множество ядер

```
void CGraph::initKernelList()
{
    if (_IntStabList.isEmpty() || _ExtStabList.isEmpty())
    {
        return;
    }
    Node* tmp = _IntStabList.head;
    LinkedList tmpList;
```

```
    while (tmp != nullptr)
    {
        while (tmp->data != 0 && tmp->next != nullptr)
        {
            tmpList.pushtail(tmp->data);
            if (tmp->next != nullptr)
            {
                tmp = tmp->next;
            }
        }
        tmp = tmp->next;
        if (tmpList.ListInOtherList(_ExtStabList))
        {
            _KernelList.pushTailList(tmpList);
            _KernelList.pushtail(0);
        }
        tmpList.dispose();
    }
}
```

---

vertexIsValid() проверяет вершину на существование

```
bool CGraph::vertexIsValid(int v)
{
    return (v >= 1 && v <= _vertexes);
}
```

---

indexIsValid() проверяет индекс на существование

```
bool CGraph::indexIsValid(int i)
{
    return (i >= 0 && i < _vertexes);
}
```

---

clearString(std::string& str) убирает из строки ":", " ", ",". "-1" заменяет на "2" (сделано для считывания матриц инцедентности)

```
void clearString(std::string& str)
{
    while (str.find(’’,’’) != str.npos)
    {
        int i = 0;
        i = (unsigned)str.find(’’,’’);
        if (i >= 0 && i < (int)str.length())
        {
            str.replace(i, 1, ’’ ’’);
        }
    }
    while (str.find(’’:’’) != str.npos)
    {
        int i = 0;
        i = (unsigned)str.find(’’:’’);
        if (i >= 0 && i < (int)str.length())
        {
            str.replace(i, 1, ’’ ’’);
        }
    }
    while (str.find(’’-1’’) != str.npos)
    {
        int i = 0;
        i = (unsigned)str.find(’’-1’’);
        if (i >= 0 && i < (int)str.length())
        {
            str.replace(i, 2, ’’2’’);
        }
    }
    while (str.find(’’  ’’) != str.npos)
    {
        int i = 0;
        i = (unsigned)str.find(’’  ’’);
        if (i >= 0 && i < (int)str.length())
        {
            str.replace(i, 1, ’’’’);
        }
    }
}
```

ReadMatrix(std::string& inputStr) чтение матрицы смежности со строки inputStr. Возвращает результат операции.

```
std::string CGraph::ReadMatrix(std::string& inputStr)
{
    dispose();
    clearString(inputStr);
    int n = 0;
    std::istringstream SS(inputStr);
    if (inputStr.empty() || !(SS >> n))
    {
        return ''Ввод содержит символы или ввод пуст'';
    }
    if (n <= 0 || n > 100)
    {
        return ''Число вершин должно быть от 1 до 100'';
    }
    _vertexes = n;
    initMatrix();
    for (int i = 0; i < _vertexes; ++i)
    {
        for (int j = 0; j < _vertexes; ++j)
        {
            if (SS >> n)
            {
                _matrix[i][j] = n;
            }
            else
            {
                dispose();
                return ''Слишком мало элементов'';
            }
        }
    }
    if (SS >> n)
    {
        dispose();
        return ''Слишком много элементов'';
    }
```

```
    return ''Accepted'';
}
```

ReadEdges(std::string& inputStr) чтение списка ребер со строки inputStr. Возвращает результат операции.

```
std::string CGraph::ReadEdges(std::string& inputStr)
{
    dispose();
    clearString(inputStr);
    int vertexes = 0;
    int edges = 0;
    std::istringstream SS(inputStr);
    if (inputStr.empty() || !(SS >> vertexes))
    {
        return ''Ввод содержит символы или ввод пуст'';
    }
    SS >> edges;
    if (vertexes <= 0 || vertexes > 100 || edges < 0)
    {
        return ''Число вершин должно быть от 1 до 100, число дуг от 0'';
    }
    _vertexes = vertexes;
    initMatrix();
    int source = 0;
    int dest = 0;
    int counter = 0;
    while (SS >> source >> dest)
    {
        if (vertexIsValid(source) && vertexIsValid(dest))
        {
            _matrix[source - 1][dest - 1] = 1;
            counter++;
        }
        else
        {
            dispose();
            return ''Одна из введенных дуг не может
                                    существовать'';
```

```
        }
    }
    if (SS >> source || counter != edges)
    {
        dispose();
        return ''Число дуг не совпадает с объявленным числом'';
    }
    return ''Accepted'';
}
```

ReadAdjacences(std::string& inputStr) чтение списка смежности со строки inputStr. Возвращает результат операции.

```
std::string CGraph::ReadAdjacences(std::string& inputStr)
{
    dispose();
    clearString(inputStr);
    int vertexes = 0;
    std::istringstream SS(inputStr);
    if (inputStr.empty() || !(SS >> vertexes))
    {
        return ''Ввод содержит символы или ввод пуст'';
    }
    if (vertexes <= 0 || vertexes > 100)
    {
        return ''Число вершин должно быть от 1 до 100'';
    }
    _vertexes = vertexes;
    initMatrix();
    int source = 0;
    int dest = 0;
    for (int i = 0; i < _vertexes; ++i)
    {
        if (SS >> source)
        {
            if (vertexIsValid(source))
            {
                std::string tmpLine;
                std::getline(SS, tmpLine);
```

```
                for (auto i = tmpLine.begin(); i != tmpLine.end(); i++)
                {
                    if (*i >= 48 && *i <= 57)
                    {
                        dest = *i - 48;
                        if (vertexIsValid(dest))
                        {
                            _matrix[source - 1][dest - 1] = 1;
                        }
                        else
                        {
                            dispose();
                            return ''Одна из введенных дуг не может существовать'';
                        }
                    }
                }
            }
            else
            {
                dispose();
                return ''Одна из вершин введена неверно'';
            }
        }
        else
        {
            dispose();
            return ''Нужна информация про все вершины'';
        }
    }
    if (SS >> source)
    {
        dispose();
        return ''Слишком много элементов'';
    }
    return ''Accepted'';
}
```

ReadIncidenceMatrix(std::string& inputStr) чтение матрицы инцедентности со строки inputStr. Возвращает результат операции.

```
std::string CGraph::ReadIncidenceMatrix(std::string& inputStr)
{
    dispose();
    clearString(inputStr);
    int vertexes = 0;
    int edges = 0;
    std::istringstream SS(inputStr);
    if (inputStr.empty() || !(SS >> vertexes))
    {
        return ''Ввод содержит символы или ввод пуст'';
    }
    SS >> edges;
    if (vertexes <= 0 || vertexes > 100 || edges < 0)
    {
        return ''Число вершин должно быть от 1 до 100, число дуг от 0'';
    }
    _vertexes = vertexes;
    initMatrix();
    std::string string = '''';
    int n = 0;
    while (SS >> n)
    {
        string += (char)n + 48;
    }
    if ((int)string.length() != vertexes * edges)
    {
        dispose();
        return ''Число элементов матрицы не совпадает с объявленным'';
    }
    for (int i = 0; i < edges; ++i)
    {
        int k = 0;
        std::string tmp1 = ''0'';
        while (tmp1 == ''0'' && i + k < (int)string.length() && i + k >= 0)
        {
            tmp1 = string[i + k];
            k += edges;
        }
        int tmp1i = k / edges - 1;
        std::string tmp2 = ''0'';
```

```
            while (tmp2 == ''0'' && i + k < (int)string.length() && i + k >= 0)
            {
                tmp2 = string[i + k];
                k += edges;
            }
            int tmp2i = k / edges - 1;
            if (tmp1 < tmp2)
            {
                std::swap(tmp1i, tmp2i);
            }
            if (indexIsValid(tmp1i) && indexIsValid(tmp2i))
            {
                _matrix[tmp1i][tmp2i] = 1;
            }
        }
        return ''Accepted'';
}
```

---

isEmpty() проверяет граф на наличие вершин

```
bool CGraph::isEmpty()
{
    return _matrix == nullptr;
}
```

---

isConnected() проверяет граф на слабую связность

```
bool CGraph::isConnected()
{
    if (_matrix == nullptr)
    {
        return false;
    }
    int counter =  0;
    for (int i = 0; i < _vertexes; ++i)
    {
        for (int j = 0; j < _vertexes; ++j)
        {
            if (_matrix[i][j] != 0 || _matrix[j][i] != 0)
```

```
            {
                counter++;
                break;
            }
        }
    }
    return counter == _vertexes;
}
```

---

initStabilityLists() запускает процесс поиска множеств внутренней и внешней устойчивости

```
void CGraph::initStabilityLists()
{
    initISList();
    initESList();
    initKernelList();
}
```

NeiboursList(int v) возвращает список соседей вершины v

```
LinkedList CGraph::NeiboursList(int v)
{
    if (_matrix == nullptr || !vertexIsValid(v))
    {
        return LinkedList();
    }
    LinkedList list;
    for (int i = 0; i < _vertexes; ++i)
    {
        if (_matrix[v - 1][i] > 0 || _matrix[i][v - 1] > 0)
        {
            list.pushtail(i + 1);
        }
    }
    return list;
}
```

InputVertexesList(int v) возвращает список вершин, с дугами, входящими в вершину v

```
LinkedList CGraph::InputVertexesList(int v)
{
    if (_matrix == nullptr || _vertexes == 0 || !vertexIsValid(v))
    {
        return LinkedList();
    }
    LinkedList list;
    for (int i = 0; i < _vertexes; ++i)
    {
        if (_matrix[i][v - 1] > 0)
        {
            list.pushtail(i + 1);
        }
    }
    return list;
}
```

---

initISList() находит множества внутренней устойчивости, сохраняет их в класс графа, в список _IntStabList

```
void CGraph::initISList()
{
    if (_matrix == nullptr || _vertexes == 0 || !isConnected())
    {
        return;
    }
    LinkedList list;
    LinkedList tmp;
    LinkedList intList;
    CGraph graph(*this);
    graph.doReflexive(0);
    graph.doSymmetric();
    graph.weightUnimportant();
    for (int i = 1; i <= _vertexes; ++i)
    {
        list.pushtail(i);
```

```
    }
    for (int i = 1; i <= _vertexes; ++i)
    {
        graph.ISList(i, intList, list, tmp);
    }
    _IntStabList = intList;
}
```

---

initESList() находит множества внешней устойчивости, сохраняет их в класс графа, в список _ExtStabList

```
void CGraph::initESList()
{
    if (_matrix == nullptr || _vertexes == 0 || !isConnected())
    {
        return;
    }
    LinkedList list;
    LinkedList tmp;
    LinkedList extList;
    CGraph graph(*this);
    graph.doReflexive(1);
    graph.weightUnimportant();
    for (int i = 1; i <= _vertexes; ++i)
    {
        list.pushtail(i);
    }
    for (int i = 1; i <= _vertexes; ++i)
    {
        graph.ESList(i, extList, list, tmp);
    }
    list = extList;
    for (int i = 1; i <= list.countElem(0); ++i)
    {
        tmp.getSubset(list, i);
        extList.absorption(tmp);
    }
    _ExtStabList = extList;
}
```

ISList(int v, LinkedList& currentList, LinkedList list, LinkedList tmp) вызывает поиск в глубину для поиска вершин, которые являются внутренне устойчивыми, сохраняет найденные вершины в список currentList, разделяет найденные множества внутренней устойчивости цифрой 0

```
void CGraph::ISList(int v, LinkedList& currentList, LinkedList list,
                                                    LinkedList tmp)
{
    if (_matrix == nullptr || _vertexes == 0 || !vertexIsValid(v))
    {
        currentList.dispose();
        return;
    }
    list.ExtractFirst(v);
    tmp.pushtail(v);

    LinkedList neibours(NeiboursList(v));
    list.ExtractList(neibours);

    for (int i = 0; i < list.length(); ++i)
    {
        ISList(list.getElem(i), currentList, list, tmp);
    }
    if (list.isEmpty())
    {
        tmp.sort();
        if (!tmp.ListInOtherList(currentList))
        {
            currentList.pushTailList(tmp);
            currentList.pushtail(0);
        }
    }
    return;
}
```

ESList(int v, LinkedList& currentList, LinkedList list, LinkedList tmp) вызывает поиск в глубину для поиска вершин, которые являются внешне

устойчивыми, сохраняет найденные вершины в список currentList, разделяет найденные множества внешней устойчивости цифрой 0

```
void CGraph::ESList(int v, LinkedList& currentList, LinkedList list,
                                                    LinkedList tmp)
{
    if (_matrix == nullptr || _vertexes == 0 || !vertexIsValid(v))
    {
        currentList.dispose();
        return;
    }
    list.ExtractFirst(v);
    tmp.pushtail(v);

    LinkedList inputs(InputVertexesList(v));
    list.ExtractList(inputs);

    for (int i = 0; i < list.length(); ++i)
    {
        ESList(list.getElem(i), currentList, list, tmp);
    }
    if (list.isEmpty())
    {
        tmp.sort();
        if (!tmp.ListInOtherList(currentList))
        {
            currentList.pushTailList(tmp);
            currentList.pushtail(0);
        }
    }
    return;
}
```

---

getIntStabilityList(bool word) возвращает множества внутренней устойчивости в std::string. Если word == true, то вывод будет в виде букв английского алфавита, иначе в виде чисел

```
std::string CGraph::getIntStabilityList(bool word)
{
```

```
    if (_IntStabList.isEmpty())
    {
        return ''Множество внутренней устойчивости пусто'';
    }
    Node* tmp = _IntStabList.head;
    std::string outputStr = '''';
    while (tmp != nullptr)
    {
        if (tmp->data == 0)
        {
            outputStr += ''\n'';
            tmp = tmp->next;
        }
        else
        {
            if (word)
            {
                char c = tmp->data + 64;
                outputStr += c;
                outputStr += '' '';
                tmp = tmp->next;
            }
            else
            {
                outputStr += std::to_string(tmp->data) + '' '';
                tmp = tmp->next;
            }
        }
    }
    return outputStr;
}
```

---

getExtStabilityList(bool word) возвращает множества внешней устойчивости в std::string. Если word == true, то вывод будет в виде букв английского алфавита, иначе в виде чисел

```
std::string CGraph::getExtStabilityList(bool word)
{
    if (_ExtStabList.isEmpty())
```

```
    {
        return ''Множество внешней устойчивости пусто'';
    }
    Node* tmp = _ExtStabList.head;
    std::string outputStr = '''';
    while (tmp != nullptr)
    {
        if (tmp->data == 0)
        {
            outputStr += ''\n'';
            tmp = tmp->next;
        }
        else
        {
            if (word)
            {
                char c = tmp->data + 64;
                outputStr += c;
                outputStr += '' '';
                tmp = tmp->next;
            }
            else
            {
                outputStr += std::to_string(tmp->data) + '' '';
                tmp = tmp->next;
            }
        }
    }
    return outputStr;
}
```

---

getKernelList(bool word) возвращает ядра графа в std::string. Если word == true, то вывод будет в виде букв английского алфавита, иначе в виде чисел

```
std::string CGraph::getKernelList(bool word)
{
    if (_KernelList.isEmpty())
    {
```

```
        return ''Ядро пусто'';
    }
    Node* tmp = _KernelList.head;
    std::string outputStr = '''';
    while (tmp != nullptr)
    {
        if (tmp->data == 0)
        {
            outputStr += ''\n'';
            tmp = tmp->next;
        }
        else
        {
            if (word)
            {
                char c = tmp->data + 64;
                outputStr += c;
                outputStr += '' '';
                tmp = tmp->next;
            }
            else
            {
                outputStr += std::to_string(tmp->data) + '' '';
                tmp = tmp->next;
            }
        }
    }
    return outputStr;
}
```

---

getMaxIntStList(bool word) getIntStabilityList(bool word) возвращает максимальные по числу вершин множества внутренней устойчивости в std::string. Если word == true, то вывод будет в виде букв английского алфавита, иначе в виде чисел

```
std::string CGraph::getMaxIntStList(bool word)
{
    if (_IntStabList.isEmpty())
    {
```

```
        return ''Множество внутренней устойчивости пусто'';
    }
    LinkedList MaxIntStList;
    MaxIntStList = _IntStabList.getSubsetByLength(getISN());
    std::string outputStr = '''';
    Node* tmp = MaxIntStList.head;
    while (tmp != nullptr)
    {
        for (int i = 0; i < getISN(); ++i)
        {
            if (tmp == nullptr)
            {
                return outputStr;
            }
            if (word)
            {
                char c = tmp->data + 64;
                outputStr += c;
                outputStr += '' '';
            }
            else
            {
                outputStr += std::to_string(tmp->data) + '' '';
            }
            tmp = tmp->next;
        }
        outputStr += ''\n'';
    }
    return outputStr;
}
```

---

getMinExtStList(bool word) возвращает минимальные по числу вершин множества внешней устойчивости в std::string. Если word == true, то вывод будет в виде букв английского алфавита, иначе в виде чисел

```
std::string CGraph::getMinExtStList(bool word)
{
    if (_IntStabList.isEmpty())
    {
```

```
        return ''Множество внешней устойчивости пусто'';
    }
    LinkedList MinExtStList;
    MinExtStList = _ExtStabList.getSubsetByLength(getESN());
    std::string outputStr = '''';
    Node* tmp = MinExtStList.head;
    while (tmp != nullptr)
    {
        for (int i = 0; i < getESN(); ++i)
        {
            if (tmp == nullptr)
            {
                return outputStr;
            }
            if (word)
            {
                char c = tmp->data + 64;
                outputStr += c;
                outputStr += '' '';
            }
            else
            {
                outputStr += std::to_string(tmp->data) + '' '';
            }
            tmp = tmp->next;
        }
        outputStr += ''\n'';
    }
    return outputStr;
}
```

3.3. MainWindow

Класс MainWindow отвечает за главное окно рабочего приложения.

```
namespace Ui
{
    class MainWindow;
}
```

```
class MainWindow : public QMainWindow
{
    Q_OBJECT

    GraphWidget *graph;
    Ui::MainWindow *ui;

public:
    explicit MainWindow(QWidget *parent = 0);
    ~MainWindow();

    void writeSettings();//Для сохранения настроек
    void readSettings ();//Для восстановления настроек

signals:
    void changeOpenFile();

private:
    bool stop;
    QSettings  m_settings;//Для сохранения настроек
    QString InputPath =””;
    std::vector<int> shortPath;
    CreateGraphWindow* createWindow;

private slots:
    void deleteNode();// Удаляет вершины графа
    void DisconnectGraph();// Удалить все рёбра в графе
    void Open();  //Открыть форматированный файл с графом
    void save();  // Сохранить форматированный граф в файл
    void RemoveEdges();//Удалить рёбра delete
    void DeleteSelectedNodes();//Удалить выбранные вершины delete
    void CreateNode();//Создать вершину
    void ClearAll();//Очистить всё

    void on_action_Open_file_triggered();
    void on_action_Save_File_triggered();
    void on_action_createNode_triggered();
    void on_action_delEdge_triggered();
    void on_action_clear_all_triggered();
    void on_structButton_clicked();
```

```
    void on_createButton_clicked();
    void on_infoButton_clicked();

public slots:
    void slotForm(std::string EdgesList);
};
```

конструктор

```
MainWindow::MainWindow(QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::MainWindow)
  , m_settings(''VEB'',''MyProgram'')  //Для сохранения настроек
{
    ui->setupUi(this);
    readSettings();//Для восстановления настроек
    // Создаётся сигнал для управления удалением вершин
    ui->actionDeleteItem->setShortcut(tr(''Delete''));
    ui->actionDeleteItem->setStatusTip(tr(''Удалить выбранные вершины из графа''));
    connect(ui->actionDeleteItem, SIGNAL(triggered(bool)), this,
                                          SLOT(deleteNode()));

    createWindow = new CreateGraphWindow;
    connect(createWindow, &CreateGraphWindow::signalForm, this,
    &MainWindow::slotForm);

    ui->action_Open_file->setText(''&Open...'');
    ui->action_Open_file->setShortcut(QKeySequence(''CTRL+O''));
    ui->action_Open_file->setToolTip(''Open Document'');
    ui->action_Open_file->setStatusTip(''Open an existing file'');
    ui->action_Open_file->setWhatsThis(''Open an existing file'');
    ui->action_Open_file->setIcon(QPixmap('':/cars/fileopen.png''));

    ui->action_Save_File->setText(''&Save'');
    ui->action_Save_File->setShortcut(QKeySequence(''CTRL+S''));
    ui->action_Save_File->setToolTip(''Save Document'');
    ui->action_Save_File->setStatusTip(''Save the file to disk'');
    ui->action_Save_File->setWhatsThis(''Save the file to disk'');
    ui->action_Save_File->setIcon(QPixmap('':/cars/filesave.png''));
```

```
    ui->action_clear_all->setText(''Очистить всё'');
    ui->action_clear_all->setToolTip(''Очистить всё'');
    ui->action_clear_all->setStatusTip(''Очистить всё'');

    QToolBar* ptbFile = new QToolBar(''File Operations'');
    ptbFile->addAction(ui->action_Open_file);
    ptbFile->addAction(ui->action_Save_File);
    ptbFile->addAction(ui->action_clear_all);

    ui->actionMenuFileExit->setShortcuts(QKeySequence::Quit);
    ui->actionMenuFileExit->setStatusTip(tr(''Выйти из программы''));
    connect(ui->actionMenuFileExit, SIGNAL(triggered()), this, SLOT(close()));

    graph = ui->graphicsView;
    graph->setDragMode(QGraphicsView::RubberBandDrag);
    graph->setOptimizationFlags(QGraphicsView::DontSavePainterState);
    graph->setViewportUpdateMode(QGraphicsView::SmartViewportUpdate);
    graph->setTransformationAnchor(QGraphicsView::AnchorUnderMouse);
    graph->setInteractive(true);
    graph->setRenderHint(QPainter::Antialiasing);
}
```

деструктор

```
MainWindow::~MainWindow()
{
    delete ui;
    delete createWindow;
    writeSettings();
}
```

---

readSettings() чтение настроек (размер окна)

```
void MainWindow::readSettings()
{
    m_settings.beginGroup(''/Settings'');
    int     nWidth     = m_settings.value(''/width'', width()).toInt();
    int     nHeight    = m_settings.value(''/height'', height()).toInt();
```

```
    resize(nWidth, nHeight);
    m_settings.endGroup();
}
```

---

writeSettings() сохранение настроек (размер окна)

```
void MainWindow::writeSettings()
{
    m_settings.beginGroup(''/Settings'');
    m_settings.setValue(''/width'', width());
    m_settings.setValue(''/height'', height());
    m_settings.endGroup();
}
```

---

deleteNode() удаляет выбранные вершины графа

```
void MainWindow::deleteNode()
{
    graph->deleteSelectedItems();
    graph->changeIndecesOfAllVerteces();
}
```

DisconnectGraph() удаляет все рёбра в графе

```
void MainWindow::DisconnectGraph()
{
    foreach (NodeVisual *Node, graph->listOfNode)
        // Для каждой вершины -
    {
        foreach (Edge *edge, Node->edges())
            // - смотрим все, связанные с ней, рёбра
        {
            graph->scene->removeItem(edge);
            //Убрать ребро из окна

            edge->destNode()->removeEdge(edge);
            //Убрать ребро из списка рёбер первой вершины
            edge->sourceNode()->removeEdge(edge);
```

```
            //Убрать ребро из списка рёбер второй вершины

            delete edge;
        }
    }
}
```

Open() открывает файл "*.graph"

```
void MainWindow::Open()
{
    QString fileName = QFileDialog::getOpenFileName(this, ''Открыть граф'',
                                /*path*/InputPath, ''Graph file (*.graph)'');
    if (!fileName.isEmpty())
    {
        QFile file(fileName);
        if (file.open(QIODevice::ReadOnly))
        {
            QTextStream in(&file);
            ClearAll();
            graph->getEdges().clear();
            graph->listOfNode.clear();

            QFileInfo fi(file);

            QString fn4 = fi.absolutePath();
            InputPath = fn4;
            bool accept = true;
            int vertexes_count = 0;
            int edges_count = 0;
            in >> vertexes_count >> edges_count;
            while (vertexes_count != 0)
            {
                graph->addNode();
                vertexes_count--;
            }
            for (int i = 0; i < edges_count; i++)
            {
                accept = true;
```

```
            int uid = 0;
            int vid = 0;
            in >> uid >> vid;
            uid--;
            vid--;
            if (uid < 0 || vid < 0)
            {
                file.close();
                QMessageBox::critical(this, ''Файл неисправен'',
                ''Неверный формат записи:\nребра записаны неправильно'');
                graph->deleteAllItems();
                return;
            }
            NodeVisual *source = nullptr;
            NodeVisual *dest = nullptr;
            source = graph->findNode(uid);
            dest = graph->findNode(vid);
            if (source == nullptr || dest == nullptr)
            {
                file.close();
                QMessageBox::critical(this, ''Файл неисправен'',
                ''Неверный формат записи:\nколичество вершин неверно'');
                graph->deleteAllItems();
                return;
            }
            foreach (Edges *edge, graph->getEdges())
            {
                if (edge->source == uid && edge->dest == vid)
                    accept = false;
            }
            if (accept)
                graph->scene->addItem(new Edge(source, dest, 0));
        }
        graph->update();// Обновляем экран
        file.close();
        //пытаюсь открыть файл ещё раз с начала
        if(file.open(QIODevice::ReadOnly | QIODevice::Text))
        {
            QTextStream in(&file);
            ui->textBrowser->setText(in.readAll());
```

```
            }
            file.close();

        }
        else {
            QMessageBox::critical(this, ''Невозможно открыть файл'',
            file.errorString());
            return;
        }
    }
    else return;
}
```

---

save() сохраняет файл в "*.graph"

```
void MainWindow::save()
{
    if (graph->getListOfNodeSize() == 0)
    {
        QMessageBox::information(this, ''Невозможно сохранить файл'', ''Граф пуст'');
        return;
    }
    QString fileName = QFileDialog::getSaveFileName(this, ''Сохранить граф'',
    InputPath, ''Graph file (*.graph)'');
    if (!fileName.isEmpty())
    {
        QFile file(fileName);
        if (file.open(QIODevice::WriteOnly))
        {
            QTextStream out(&file);
            int i = 0;
            i = graph->getEdges().length();
            out << graph->getListOfNodeSize() << '' '' << i << ''\n'';
            foreach (Edges *item, graph->getEdges())
               out << item->source + 1 << '' '' << item->dest + 1 << ''\n'';
            file.close();
        }
        else {
            QMessageBox::critical(this, ''Невозможно открыть файл'',
```

```
            file.errorString());
            return;
      }
    } else return;
}
```

RemoveEdges() удаляет ребра

```
void MainWindow::RemoveEdges()
{
    DisconnectGraph();
}
```

DeleteSelectedNodes() удаляет вершины

```
void MainWindow::DeleteSelectedNodes()
{
   deleteNode();
}
```

CreateNode() создает вершину

```
void MainWindow::CreateNode()
{
    if (graph->getListOfNodeSize() > 119)
    {
        QMessageBox::warning(this, ''Предупреждение'', ''Слишком много вершин!'');
        return;
    }
    graph->addNode();
}
```

ClearAll() удаляет все элементы

```
void MainWindow::ClearAll()
{
    graph->deleteAllItems();
```

```
    ui->textBrowser->setText(''Двойной клик левой кнопкой мыши - добавить новую ве
                    ''Двойной клик правой кнопкой мыши - ''
                    ''соединить выбранные вершины с той, на которую кликал.'');
}
```

кнопка "Открыть"

```
void MainWindow::on_action_Open_file_triggered()
{
    Open();
}
```

кнопка "Сохранить"

```
void MainWindow::on_action_Save_File_triggered()
{
    save();
}
```

кнопка "Создать вершину"

```
void MainWindow::on_action_createNode_triggered()
{
    CreateNode();
}
```

кнопка "Удалить все ребра"

```
void MainWindow::on_action_delEdge_triggered()
{
    DisconnectGraph();
}
```

кнопка "Очистить все"

```
void MainWindow::on_action_clear_all_triggered()
{
    ClearAll();
}
```

кнопка ”Структурировать граф”

```
void MainWindow::on_structButton_clicked()
{
    graph->structurize();
}
```

кнопка ”Создать граф”

```
void MainWindow::on_createButton_clicked()
{
    createWindow->setModal(true);
    createWindow->setFixedSize(createWindow->size());
    createWindow->exec();
}
```

слот для передачи данных из окна CreateGraphWindow

```
void MainWindow::slotForm(std::string EdgesList)
{
    ClearAll();
    graph->getEdges().clear();
    graph->listOfNode.clear();
    bool accept = true;
    std::istringstream SS(EdgesList);
    int vertexes_count = 0;
    int edges_count = 0;
    SS >> vertexes_count >> edges_count;
    while (vertexes_count != 0)
    {
        graph->addNode();
        vertexes_count--;
    }
```

```
    for (int i = 0; i < edges_count; i++)
    {
        accept = true;
        int uid = 0;
        int vid = 0;
        SS >> uid >> vid;
        uid--;
        vid--;
        NodeVisual *source = nullptr;
        NodeVisual *dest = nullptr;
        source = graph->findNode(uid);
        dest = graph->findNode(vid);
        foreach (Edges *edge, graph->getEdges())
        {
            if (edge->source == uid && edge->dest == vid)
            {
                accept = false;
            }
        }
        if (accept)
        {
            graph->scene->addItem(new Edge(source, dest, 0));
        }
    }
    graph->update();// Обновляем экран
}
```

кнопка ”Информация о множествах устойчивости”

```
void MainWindow::on_infoButton_clicked()
{
    if (graph->getListOfNodeSize() == 0)
    {
        ui->textBrowser->setText(''Окно пусто'');
        return;
    }
    bool word = 0;
    word = ui->wordButton->isChecked();
    if (graph->getListOfNodeSize() > 26 && word == 1)
```

```
{
    ui->wordButton->setChecked(0);
    word = 0;
    QMessageBox::information(this, ''Внимание'', ''Вершин больше 26,\nбуквы буду
    изменены на цифры'');
}
std::string outputStr = '''';
outputStr += std::to_string(graph->getListOfNodeSize());
outputStr += '' '';
outputStr += std::to_string(graph->getEdges().length());
outputStr += ''\n'';
foreach (Edges *item, graph->getEdges())
{
    outputStr += std::to_string(item->source + 1);
    outputStr += '' '';
    outputStr += std::to_string(item->dest + 1);
    outputStr += ''\n'';
}
CGraph g;
g.ReadEdges(outputStr);
g.initStabilityLists();
if (g.getISN() == 0)
{
    ui->textBrowser->setText(QString::fromStdString(''Граф несвязный''));
    return;
}
outputStr = ''Число внутренней устойчивости: '' + std::to_string(g.getISN()) + '
outputStr += ''Число внешней устойчивости: '' + std::to_string(g.getESN()) + ''\
if (ui->fullButton->isChecked())
{
    outputStr += ''Множества внутренней устойчивости:\n'';
    outputStr += g.getIntStabilityList(word) + ''\n'';
    outputStr += ''Множества внешней устойчивости:\n'';
    outputStr += g.getExtStabilityList(word) + ''\n'';
}
outputStr += ''Наибольшие множества внутренней устойчивости:\n'';
outputStr += g.getMaxIntStList(word) + ''\n'';
outputStr += ''Наименьшие множества внешней устойчивости:\n'';
outputStr += g.getMinExtStList(word) + ''\n'';
outputStr += ''Ядра графа:\n'';
```

```
    outputStr += g.getKernelList(word) + ''\n'';
    ui->textBrowser->setText(QString::fromStdString(outputStr));
}
```

3.4. GraphWidget

GraphWidget это область экрана в главном окне, в которой можно создавать и изменять граф. Все методы нацелены на работу с этой областью экрана.

```
struct Edges // дуги графа
{
    Edge *edge;
    int source;
    int dest;
    int length;
};

class GraphWidget : public QGraphicsView // Виджет для рисования вершин и рёбер
{
    Q_OBJECT

public:
    QGraphicsScene *scene = new QGraphicsScene; // Окно, на котором рисуются
    все элементы
    QList<NodeVisual *> listOfNode; // Список вершин, отображающихся на окне

    NodeVisual* findNode(int val);//по номеру надо найти вершину

    explicit GraphWidget(QWidget *parent = 0);//Конструктор
    QPointF getPosOfNode(int index);// Возвращает местоположение вершины с
    номером index
    int getListOfNodeSize();//Возвращает количество вершин
    void changeIndecesOfAllVerteces();//Переименовывет вершины после удаления
    некоторых вершин
    QVector<Edges *> getEdges();

    void addNode();
    void deleteEdge(Edge *edge);
```

```
    NodeVisual* addNode1(QPointF position,int t);
    Edge *addEdge(NodeVisual *source, NodeVisual *dest, unsigned length);
    void addEdge(NodeVisual *source, NodeVisual *dest);
    void deleteAllItems();//Удаляет все вершины
    void structurize(); //Выставляет вершины по эллипсу

public slots:
    void deleteSelectedItems();//Удаляет выбранные вершины

protected:
    void drawBackground(QPainter *painter, const QRectF &rect);// Отрисовка фона
    void resizeEvent(QResizeEvent *event);//Вызывается при изменении размера
    окна
    void mouseDoubleClickEvent(QMouseEvent *event);// Отслеживание двойного
    клика мышкой
};
```

конструктор

```
GraphWidget::GraphWidget(QWidget *parent): QGraphicsView(parent)
{
    scene = new QGraphicsScene(this);
    setScene(scene);
}
```

---

resizeEvent(QResizeEvent *event) вызывается при изменении размера окна

```
void GraphWidget::resizeEvent(QResizeEvent *event)
{
    scene->setSceneRect(0, 0, width()-2, height()-2);
    foreach (NodeVisual *tmpNode, listOfNode)
    {
        tmpNode->returnToScene(tmpNode->pos());
    }
    QGraphicsView::resizeEvent(event);
}
```

drawBackground(QPainter *painter, const QRectF&) отрисовка фона

```
void GraphWidget::drawBackground(QPainter *painter, const QRectF &)
{
    QRectF sceneRect = this->sceneRect();
    painter->fillRect(sceneRect, QColor(255, 255, 255, 255));//Окрашиваем
    область в цвет градиента
}
```

addNode() добавление вершины на сцену

```
void GraphWidget::addNode()
{
    NodeVisual *node = new NodeVisual(this); // Создаём новую вершину
    scene->addItem(node); // Добавляем её на экран
    listOfNode << node; // Добавить эту вершину в список вершин
    changeIndecesOfAllVerteces(); // Поставить новый индекс вершине
    int i = 0;
    int distance = 100;
    double angle = 0;
begin: // далее идет задание позиции вершины (по эллипсу)
    angle = std::rand() / 6.28;
    int x = 0;
    x = (size().width() / 2) * (1 + cos(angle) * 2 / 3);
    int y = 0;
    y = (size().height() / 2) * (1 + sin(angle) * 2 / 3);
    foreach (NodeVisual *tmpNode, listOfNode)
    {
        if (pow((tmpNode->pos().x() - x), 2) + pow((tmpNode->pos().y() - y), 2)
        < pow(distance, 2) && i < listOfNode.length())
        {
            i++;
            goto begin;
        }
        else if (i >= listOfNode.length() && distance > 30)
        {
            distance--;
            i = 0;
            goto begin;
```

```
        }
    }
    node->setPos(QPointF(x, y));
    update(); // Обновляем экран
}
```

---

deleteEdge(Edge *edge) удаление вершины со сцены

```
void GraphWidget::deleteEdge(Edge *edge)
{
    scene->removeItem(edge);
    edge->destNode()->removeEdge(edge);
    edge->sourceNode()->removeEdge(edge);
    delete edge;
};
```

---

addNode1(QPointF position,int t) добавить вершину с заданной позицией position и номером t

```
NodeVisual* GraphWidget::addNode1(QPointF position,int t)
{
    NodeVisual *node = new NodeVisual(this); // Создаём новую вершину
    node->setIndex(t);
    scene->addItem(node); // Добавляем её на экран
    listOfNode << node; // Добавить эту вершину в список вершин
    node->setPos(position);
    update(); // Обновляем экран
    return node;
};
```

---

addEdge(NodeVisual *source,NodeVisual *dest, unsigned length) добавить дугу с весом

```
Edge * GraphWidget::addEdge(NodeVisual *source,NodeVisual *dest, unsigned length)
{
    Edge *edge = new Edge(source, dest, length);
    scene->addItem(edge);
    return edge;
}
```

addEdge(NodeVisual *source, NodeVisual *dest) добавить дугу

```
void GraphWidget::addEdge(NodeVisual *source, NodeVisual *dest)
{
    scene->addItem(new Edge(source, dest));
};
```

mouseDoubleClickEvent(QMouseEvent *event) отслеживание двойного клика мышкой

```
void GraphWidget::mouseDoubleClickEvent(QMouseEvent *event)
{
    //Если щёлкнули левой кнопкой мыши дважды - добавили вершину
    if (event->button() == Qt::LeftButton) // Если был произведён двойной клик
    левой кнопкой мыши
    {
        NodeVisual *node = new NodeVisual(this); // Создаём новую вершину
        scene->addItem(node); // Добавляем её на экран
        node->setPos(QPointF(event->pos().x(), event->pos().y())); // Перемещаем
        //к курсору
        listOfNode << node; // Добавить эту вершину в список вершин
        changeIndecesOfAllVerteces(); // Поставить новый индекс вершине
        update(); // Обновяем экран
    }

    //Если щёлкнули правой кнопкой мыши дважды - провели рёбра
    //от всех выбранных вершин к той, на которую кликнули
    else if (event->button() == Qt::RightButton)
    // Если был произведён двойной клик правой кнопкой мыши
    {
        NodeVisual *dest = static_cast<NodeVisual*>(itemAt(event->pos()));
        //Вершина, в которую надо провести все рёбра от выбранных вершин
        foreach (QGraphicsItem *item, scene->selectedItems())//Для всех
        //выбранных вершин
        {
            bool isFindSameEdge = false;
            NodeVisual *source = static_cast<NodeVisual*>(item);// Привести
            //текущую вершину к типу NodeVisual
```

```
            if (source == dest) continue;
            int lengthNum = 0;
            foreach (Edge *compareEdge, source->edges())
            {
                if ((compareEdge->sourceNode() == source &&
                     compareEdge->destNode() == dest))
                {
                    for (auto i : source->edges())
                        if (i->destNode() == dest)
                        {
                            i->setLength(static_cast<unsigned>(lengthNum));
                        }
                    isFindSameEdge = true;
                    break;
                }
            }
            if (isFindSameEdge) continue;
            scene->addItem(new Edge(source, dest,
            static_cast<unsigned>(lengthNum)));
        }
    }
}
```

---

getPosOfNode(int index) возвращает местоположение вершины с номером index

```
QPointF GraphWidget::getPosOfNode(int index)
{
    return listOfNode.at(index)->pos();
}

int GraphWidget::getListOfNodeSize()
//Возвращает количество вершин
{
    return listOfNode.size();
}

void GraphWidget::changeIndecesOfAllVerteces()
//Переименовывет вершины после удаления некоторых вершин
```

```
{
    foreach (NodeVisual *itemNode, listOfNode)
    {
        itemNode->setIndex(listOfNode.indexOf(itemNode));
        itemNode->update();
    }
}
```

---

getEdges() возвращает список всех рёбер графа

```
QVector<Edges *> GraphWidget::getEdges()
{
    QVector<Edges *> EdgesList;
    foreach (NodeVisual *Node, listOfNode)
    {
        foreach (Edge *edge, Node->edges())
        {
            if (Node->getIndex() == edge->destNode()->getIndex()) continue;
            Edges *curr = new Edges;
            curr->source = Node->getIndex(); // В каждую запись включает первую
            //вершину ребра
            curr->dest = edge->destNode()->getIndex(); // Вторую вершину ребра
            curr->length = (int)edge->getLength();      // Длину ребра
            curr->edge = edge; // Ссылку на само ребро
            EdgesList << curr;
        }
    }
    return EdgesList;
}
```

---

deleteSelectedItems() удаляет выбранные вершины

```
void GraphWidget::deleteSelectedItems()
{
    foreach (QGraphicsItem *itemNode, scene->selectedItems())//Для каждой
    выбранной вершины
    {
        scene->removeItem(itemNode);//Убрать её из окна
        //Убрать из списка вершин
```

```
        listOfNode.removeAt(listOfNode.indexOf((NodeVisual *)itemNode));

        //Для каждого ребра, связанного с данной вершиной
        foreach (Edge *itemEdge, ((NodeVisual *)itemNode)->edges())
        {
            // Удалить со сцены аним. объект
            if(!itemEdge->it->pixmap().isNull())
            {
                scene->removeItem(itemEdge->it);
            }
            scene->removeItem(itemEdge);//Убрать ребро из окна
            itemEdge->destNode()->removeEdge(itemEdge);//Убрать ребро из списка
            //рёбер первой вершины
            itemEdge->sourceNode()->removeEdge(itemEdge);//Убрать ребро из списка
            //рёбер второй вершины
            delete itemEdge;//Удалить само ребро
        }
        delete itemNode;//Удалить саму вершину
    }
}
```

---

deleteAllItems() удаляет все вершины

```
void GraphWidget::deleteAllItems()
{
   foreach (NodeVisual *itemNode, listOfNode)
    {
        scene->removeItem(itemNode);//Убрать её из окна

        foreach (Edge *itemEdge, itemNode->edges())
            //Для каждого ребра, связанного с данной вершиной
        {
            scene->removeItem(itemEdge);//Убрать ребро из окна
            itemEdge->destNode()->removeEdge(itemEdge);
            //Убрать ребро из списка рёбер первой вершины
            itemEdge->sourceNode()->removeEdge(itemEdge);
            //Убрать ребро из списка рёбер второй вершины
            delete itemEdge;//Удалить само ребро
        }
```

```
        itemNode->edges().clear();
        delete itemNode;//Удалить саму вершину
    }
   listOfNode.clear();
   scene->clear();
}
```

---

findNode(int val/*,QList<NodeVisual *> listOfNode*/) по номеру находит и возвращает вершину

```
NodeVisual* GraphWidget::findNode(int val/*,QList<NodeVisual *> listOfNode*/)
//по номеру надо найти вершину
{
    NodeVisual *a;
    //по номеру надо найти вершину
    foreach (NodeVisual *itemNode, listOfNode)
    {
        if (itemNode->getIndex() == val)
        {
            // Привести текущую вершину к типу Node
            a = static_cast<NodeVisual*>(itemNode);
            return a;
        }
    }
    return nullptr;
}
```

---

structurize() выставляет вершины по эллипсу

```
void GraphWidget::structurize()
{
    foreach (NodeVisual *tmpNode, listOfNode)
    {
        int i = 0;
        int distance = 100;
        double angle = 0;
    begin: // ищем подходящий угол
        angle = std::rand() / 6.28;
```

```
            int x = 0;
            x = (size().width() / 2) * (1 + cos(angle) * 2 / 3);
            int y = 0;
            y = (size().height() / 2) * (1 + sin(angle) * 2 / 3);
            foreach (NodeVisual *tmpNode, listOfNode)
            {
                if (pow((tmpNode->pos().x() - x), 2) +
                pow((tmpNode->pos().y() - y), 2) < pow(distance, 2) && i < listOfNode
                {
                    i++;
                    goto begin;
                }
                else if (i >= listOfNode.length() && distance > 30)
                {
                    distance--;
                    i = 0;
                    goto begin;
                }
            }
            tmpNode->setPos(QPointF(x, y));
            update();
        }
}
```

3.5. NodeVisual

NodeVisual - класс для работы с вершинами, изображенными на экране

```
class NodeVisual : public QGraphicsItem // Вершина графа
{
private:
    int index; // Номер вершины
    QSet<Edge *> edgeList; // Список ссылок рёбер, с которыми соединена вершина
    QPointF newPos; // Координаты вершины на экране
    GraphWidget *graph; // Ссылка на виждет, на котором должна отображаться
    вершина

public:
    NodeVisual(GraphWidget *graphWidget); // Конструктор
```

```
    void addEdge(Edge *edge); // Добавить к этой вершине ребро
    QSet<Edge *> edges(); // Вернуть список рёбер, к которыми соединена эта
    вершина
    void removeEdge(Edge *deletedEdge); // Удалить ребро от этой вершины
    int getIndex(); // Возвращает номер вершины
    void setIndex(int index); // Нумерует вершину
    void returnToScene(QPointF position); //Возвращает вершину на видимую
    область экрана

protected:
    QRectF boundingRect() const; // Просчитывает нужные координаты
    прямоугольника, внутри которого будет вершина
    QPainterPath shape() const; // Делает кликабельную область, в которой
    находится вершина, кругом, а не прямоугольником
    void paint(QPainter *painter, const QStyleOptionGraphicsItem *option,
    QWidget *widget); // Рисует вершину

    //Регистрирует изменение положения вершины
    QVariant itemChange(GraphicsItemChange change, const QVariant &value);
    // Регистрирует нажатие на вершину мышкой
    void mousePressEvent(QGraphicsSceneMouseEvent *event);
    // Регистрирует отпускание вершины мышкой
    void mouseReleaseEvent(QGraphicsSceneMouseEvent *event);
};
```

конструктор

```
NodeVisual::NodeVisual(GraphWidget *graphWidget): graph(graphWidget)
{
    setFlags(ItemIsSelectable | ItemIsMovable);
    setFlag(ItemSendsGeometryChanges);
    setCacheMode(DeviceCoordinateCache);
    setZValue(+1);
}
```

---

addEdge(Edge *edge) добавляет дугу к вершине

```
void NodeVisual::addEdge(Edge *edge)
{
    edgeList << edge;
    edge->adjust();
}
```

---

removeEdge(Edge *deletedEdge) удаляет дуги, связанные с вершиной

```
void NodeVisual::removeEdge(Edge *deletedEdge)
{
    edgeList.remove(deletedEdge);
}
```

---

edges() возвращает список ребер для вершины

```
QSet<Edge *> NodeVisual::edges()
{
    return edgeList;
}
```

---

getIndex() возвращает индекс вершины

```
int NodeVisual::getIndex()
{
    return index;
}
```

---

setIndex(int index) устанавливает индекс вершине

```
void NodeVisual::setIndex(int index)
{
    this->index = index;
}
```

---

boundingRect() добавляет hitbox вершине

```
QRectF NodeVisual::boundingRect() const
{
    qreal adjust = 20;
    return QRectF( -15 - adjust, -15 - adjust, 33 + adjust, 33 + adjust);
}
```

---

shape() добаляет круг вершине

```
QPainterPath NodeVisual::shape() const
{
    QPainterPath path;
    path.addEllipse(-15, -15, 30, 30);
    return path;
}
```

---

paint(...) разукрашивает вершину

```
void NodeVisual::paint(QPainter *painter, const QStyleOptionGraphicsItem *option,
                                                                    QWidget *)
{
    // Fill
    if (option->state & QStyle::State_Sunken)//затонувший
    {
        painter->setBrush(QColor(253, 158, 255));
    }
    else if (option->state & QStyle::State_Selected)//выделение вершины
    {
        painter->setBrush(QColor(253, 158, 255));
    }
    else
    {
        painter->setBrush(QColor(163, 175, 255));
    }

    painter->setPen(QPen(QColor(61, 86, 255), 3));
    painter->drawEllipse(-15, -15, 30, 30);
    painter->setPen(QPen(Qt::black));
    painter->setFont(QFont(''Roboto'', 9));
    setZValue(1300);
```

```
    QString text = QString::number(index + 1);
    if (index < 10)
    {
        painter->drawText(-4, 5, text);
    }
    else
    {
        painter->drawText(-8, 5, text);
    }
}
```

---

itemChange(...) обновляет экран после взаимодействия

```
QVariant NodeVisual::itemChange(GraphicsItemChange change, const QVariant &value)
{
    switch (change) {
    case ItemPositionHasChanged:
        foreach (Edge *edge, edgeList)
            edge->adjust();
        break;
    default:
        break;
    }

    return QGraphicsItem::itemChange(change, value);
}
```

---

returnToScene(QPointF position) возвращат вершину в видимую область экрана

```
void NodeVisual::returnToScene(QPointF position)
{
    if (position.x() < 16)
    {
        setX(16);
    }
    if (position.x() > scene()->width() - 16)
    {
```

```
        setX(scene()->width() - 16);
    }
    if (position.y() < 16)
    {
        setY(16);
    }
    if (position.y() > scene()->height() - 16)
    {
        setY(scene()->height() - 16);
    }
}
```

3.6. Edge

Edge - класс для дуг, изображенных на экране

```
class Edge : public QGraphicsItem
{
    Q_INTERFACES(QGraphicsItem)
private:
    NodeVisual *source, *dest;

    QPointF sourcePoint;
    QPointF destPoint;
    qreal arrowSize;
    QBrush color;
    Qt::PenStyle^^Istyle;
    QGraphicsItemAnimation^^I*posAnim;
    QTimeLine^^I^^I^^I^^I*timer;
    QGraphicsScene* scene;
    unsigned length;
    int flow;
public:
    Edge(NodeVisual *sourceNode, NodeVisual *destNode);
    Edge(NodeVisual *sourceNode, NodeVisual *destNode, unsigned length);
    QRectF boundingRect() const;
    NodeVisual *sourceNode() const;
    NodeVisual *destNode() const;

    void adjust();
```

```
    void setColor(QBrush newColor);
    void setStyle(Qt::PenStyle st);
    qreal getLength(); //Возвращает длину ребра
    int getFlow();
    void setLength(unsigned value);
    void setFlow(int value);
    void addFlow(int value);
    QGraphicsPixmapItem^^I*it; //объект на ребре
    void animate(QGraphicsScene *scene, bool auto1);
protected:
    void paint(QPainter *painter, const QStyleOptionGraphicsItem *option, QWidget
    *widget);
};
```

конструктор

```
Edge::Edge(NodeVisual *sourceNode, NodeVisual *destNode): arrowSize(15)
{
    setCacheMode(DeviceCoordinateCache);
    source = sourceNode;
    dest = destNode;
    source->addEdge(this);
    dest->addEdge(this);
    adjust();
    length = 100;
    flow = 0;
    color = Qt::black;

    timer = new QTimeLine; //создание таймера
    timer->setFrameRange(0,500);
    timer->setLoopCount(1);

    posAnim= new QGraphicsItemAnimation; //создание анимации
    posAnim->setTimeLine(timer);
    it = new QGraphicsPixmapItem;
}
```

конструктор для дуги с длиной length

```
Edge::Edge(NodeVisual *sourceNode, NodeVisual *destNode, unsigned length)
: Edge(sourceNode, destNode)
{
    setLength(length);
}
```

---

setColor(QBrush newColor) устанавливает цвет дуги

```
void Edge::setColor(QBrush newColor)
{
    color = newColor;
    setZValue(999);
}
```

---

setStyle(Qt::PenStyle st) устанавливает стиль дуги

```
void Edge::setStyle(Qt::PenStyle st)
{
    style = st;
    update();
}
```

---

sourceNode() возвращает вершину в начале дуги

```
 NodeVisual *Edge::sourceNode() const
{
    return source;
}
```

---

destNode() возвращает вершину в конце дуги

```
NodeVisual *Edge::destNode() const
{
    return dest;
}
```

---

adjust() регулирует дугу в процессе взаимодействия

```
void Edge::adjust()
{
    if (!source || !dest) return;

    QLineF line(mapFromItem(source, 0, 0), mapFromItem(dest, 0, 0));
    qreal length = line.length();

    prepareGeometryChange();
    QPointF edgeOffset((line.dx() * 15) / length, (line.dy() * 15) / length);

    //    Рисование простого ребра
    if (length > qreal(30.))
    {
        QPointF edgeOffset((line.dx() * 15) / length, (line.dy() * 15) / length);
        sourcePoint = line.p1() + edgeOffset;
        destPoint = line.p2() - edgeOffset;
    }
    else
    {
        sourcePoint = destPoint = line.p1();
    }
}
```

---

getLength() возвращает длину дуги

```
qreal Edge::getLength()
{
    return length;
}
```

---

getFlow() возвращает поток дуги

```
int Edge::getFlow()
{
    return flow;
}
```

---

setLength(unsigned value) устанавливает длину дуги

```
void Edge::setLength(unsigned value)
{
    length = value;
    update();
}
```

---

setFlow(int value) устанавливает поток дуги

```
void Edge::setFlow(int value)
{
    flow = value;
    update();
}
```

---

addFlow(int value) добавляет поток для дуги

```
void Edge::addFlow(int value)
{
    flow += value;
    update();
}
```

---

boundingRect() добавляет hitbox дуги

```
QRectF Edge::boundingRect() const
{
    if (!source || !dest) return QRectF();

    qreal penWidth = 2;
    qreal extra = (penWidth + arrowSize)+60;

    return QRectF(sourcePoint, QSizeF(destPoint.x() - sourcePoint.x(),
                                      destPoint.y() - sourcePoint.y()))
        .normalized()
        .adjusted(-extra, -extra, extra, extra);
}
```

---

paint(...) разукрашивает дугу

```
void Edge::paint(QPainter *painter, const QStyleOptionGraphicsItem *, QWidget *)
{
    if (!source || !dest) return;
    painter->setRenderHint(QPainter::Antialiasing);

    QLineF line(sourcePoint, destPoint);
    if (qFuzzyCompare(line.length(), qreal(0.))) return;

    // Draw the line
    painter->setPen(QPen(color, 2, Qt::SolidLine, Qt::RoundCap, Qt::RoundJoin));
    painter->drawLine(line);
    QFont font;
    font.setPixelSize(15);
    painter->setFont(font);

    // Draw the arrows
    double angle = ::acos(line.dx() / line.length());
    if (line.dy() >= 0)
        angle = TwoPi - angle;

    QPointF destArrowP1 = destPoint + QPointF(sin(angle - Pi / 3) * arrowSize,
                                              cos(angle - Pi / 3) * arrowSize);
    QPointF destArrowP2 = destPoint + QPointF(sin(angle - Pi + Pi / 3) * arrowSize
                                              cos(angle - Pi + Pi / 3) * arrowSize)
    QString output;
    if (flow != 0)
        output += '' F:'' + QString::number(flow);
    if (flow < 0)
    {
        QPointF bottom = {(sourcePoint.rx()+destPoint.rx())/2,
        (sourcePoint.ry()+destPoint.ry())/2-20};
         painter->drawText(bottom, output);
    }
    else
        painter->drawText((sourcePoint+destPoint)/2, output);
    painter->setBrush(Qt::black);
    painter->drawPolygon(QPolygonF() << line.p2() << destArrowP1 << destArrowP2);
}
```

animate(...) анимирует дугу в процессе взаимодействия

```
void Edge::animate(QGraphicsScene *scene, bool auto1)
{
    if(sourcePoint == destPoint)
        return;

    QLineF line(sourcePoint, destPoint);

/**/    double angle = ::acos(line.dx() / line.length());
    if (line.dy() >= 0)
        angle = TwoPi - angle;
//Пусть объект движется по прямой, а не по дуге
    QPointF c = 0.5*line.p1() + 0.5*line.p2();

    QPainterPath a;
    a.moveTo(sourcePoint);
    a.quadTo(c,destPoint);
    timer->setDuration(static_cast<int>(a.length()/0.1));
    if (auto1)
    {
        QPixmap p(''':/cars/1.png''');
        it->setPixmap(p); //установить изображение на объект
        it->setOffset(-p.size().width()/2.0,-p.size().height()/2.0); //установка
        //центра
\begin{figure}
            \centering
            \includegraphics[width=0.5\linewidth]{mainWinOpen.png}
            \caption{}
            \label{fig:enter-label}
        \end{figure}
                it->setZValue(0.5);
        it->setPos(sourceNode()->scenePos());
        it->setTransformationMode(Qt::SmoothTransformation);
        posAnim->setItem(it);
    }
/**/
// эта часть кода обеспечивает движение
    bool ce = true;
    for(int i = 0; i <= 10; i++)
```

```
    {
        qreal p = i/10.0;
        qreal p_pred =0;
        if(i>0)
            p_pred =(i-1)/10.0;
        if(a.angleAtPercent(p) > a.angleAtPercent(p_pred))
            ce = false;
        if(i==0)
            posAnim->setPosAt(p,sourceNode()->scenePos());
         else if(i==10)
            posAnim->setPosAt(p,destNode()->scenePos());
        else
            posAnim->setPosAt(p,a.pointAtPercent(p));

/**/        posAnim->setRotationAt(p,
                                   ce
                                   ? -a.angleAtPercent(p)
                                     : 360 - a.angleAtPercent(p)
                                   );/**/
    }/**/
    if (!auto1)
    {
        QGraphicsEllipseItem* a1 = scene->addEllipse(0, 0, 10, 10);
        QColor col = Qt::black;
        a1->setParentItem(this);
        a1->setPos(sourcePoint);
        a1->setPen(QPen(col, 0));
        a1->setBrush(QBrush(col));
        a1->update();
        posAnim->setItem(a1);

    }
    scene->addItem(it);
    timer->start();
}
```

3.7. CreateGraphWindow

CreateGraphWindow - окно для ввода графа через клавиатуру (открывается по кнопке в главном окне)

```
namespace Ui
{
    class CreateGraphWindow;
}

class CreateGraphWindow : public QDialog
{
    Q_OBJECT

public:
    explicit CreateGraphWindow(QWidget *parent = nullptr);
    ~CreateGraphWindow();

private slots:
    void on_acceptButton_clicked();
    void on_cancelButton_clicked();

private:
    Ui::CreateGraphWindow* ui;

signals:
    void signalForm(std::string EdgesList);
};
```

конструктор

```
CreateGraphWindow::CreateGraphWindow(QWidget *parent)
    : QDialog(parent)
    , ui(new Ui::CreateGraphWindow)
{
    ui->setupUi(this);
    ui->readMatrixButton->setChecked(true);
}
```

деструктор

```
CreateGraphWindow::~CreateGraphWindow()
{
    delete ui;
}
```

действие по кнопке ”Ок”

```
void CreateGraphWindow::on_acceptButton_clicked()
{
    CGraph g;
    std::string inputStr;
    inputStr = ui->textEdit->toPlainText().toStdString();
    std::string outputStr;
    if (ui->readMatrixButton->isChecked())
    {
        outputStr = g.ReadMatrix(inputStr);
    }
    else if (ui->readEdgesButton->isChecked())
    {
        outputStr = g.ReadEdges(inputStr);
    }
    else if (ui->readAdjacencesButton->isChecked())
    {
        outputStr = g.ReadAdjacences(inputStr);
    }
    else if (ui->readIncidenceMatrixButton->isChecked())
    {
        outputStr = g.ReadIncidenceMatrix(inputStr);
    }
    if (g.isEmpty())
    {
        QMessageBox::information(this, ''Ошибка ввода'',
        QString::fromStdString(outputStr));
        return;
    }
    emit signalForm(g.GetEdgesList());
    ui->textEdit->clear();
```

```
    window()->close();
}
```

действие по кнопке ”Отмена”

```
void CreateGraphWindow::on_cancelButton_clicked()
{
    window()->close();
}
```

3.8. main.cpp

```
#include ”MainWindow.h”
#include <QApplication>
#include <QMainWindow>

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    MainWindow w;
    w.show();
    return a.exec();
}
```

---

4. Демонстрация работы программы

4.1. Главное окно

В главном окне 5 кнопок и 2 флажка. Кнопка ”Структурировать граф” выставляет вершины по эллипсу оптимальным образом. Кнопка ”Создать граф” открывает окно ввода графа. Кнопка ”Информация о множествах устойчивости” выводит информацию о наибольших множествах внутренней устойчивости, наименьших множествах внешней устойчивости и о ядрах графа. При включенном флажке ”Буквенная форма” информация выводится буквами английского алфавита, иначе в виде чисел. При включенном флажке ”Вывод полных множеств” отдельно выводятся все множества внутренней и внешней устойчивости. На Рис. 1 представлено главное окно программы. На рис. 2 представлен один из примеров работы программы. Нажав на кнопку ”Файл” вылезает окно с кнопками для загрузки и сохранения файла с графом (Рис. 3). Нажав на кнопку ”Правка” вылезает окно с кнопками для быстрого редактирования графа в окне (Рис. 4).

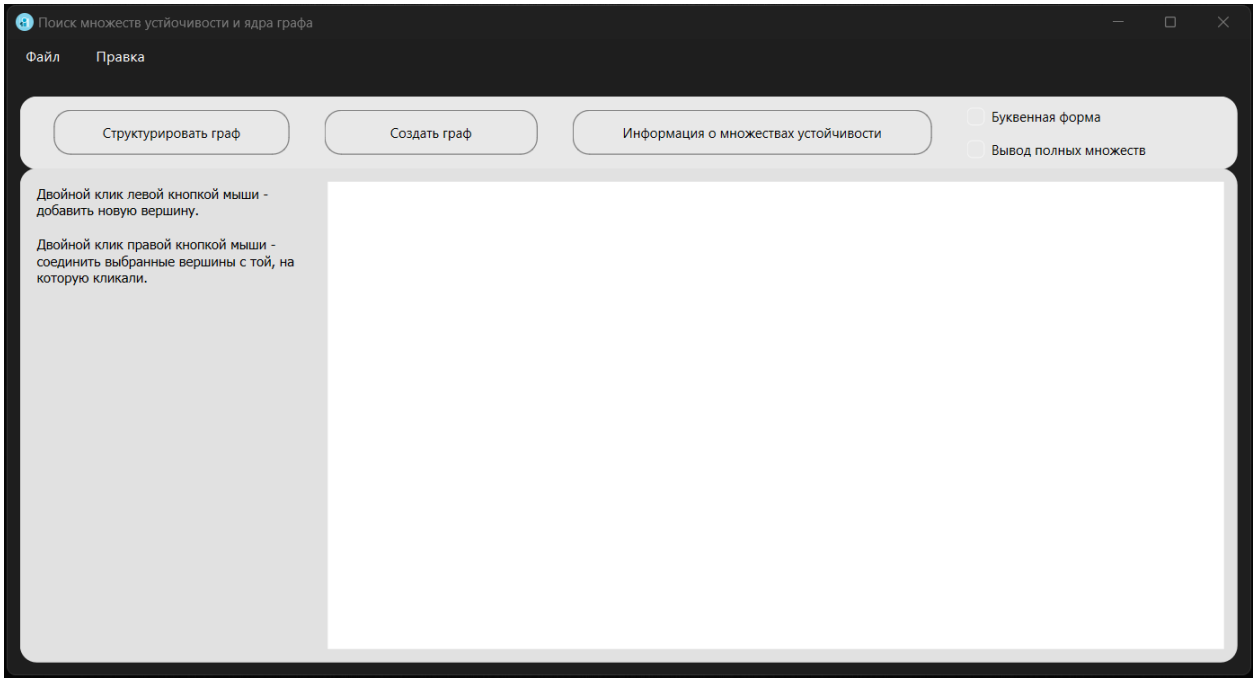


Рис. 1: Главное окно

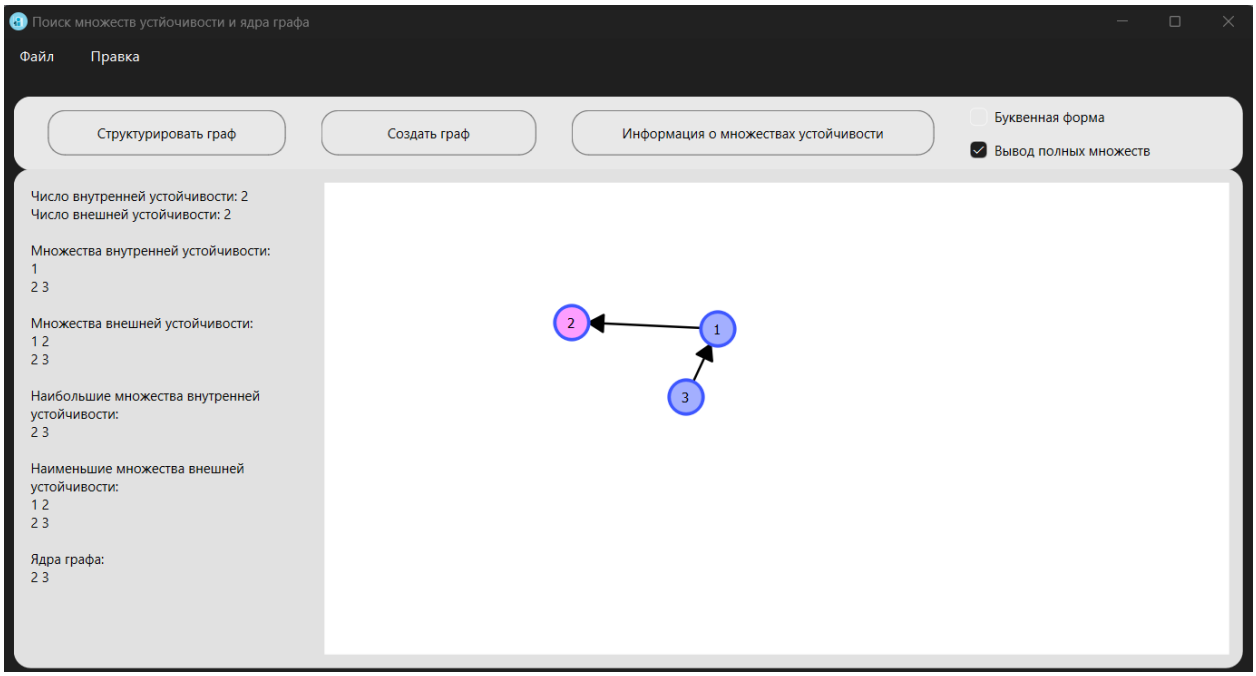


Рис. 2: Пример

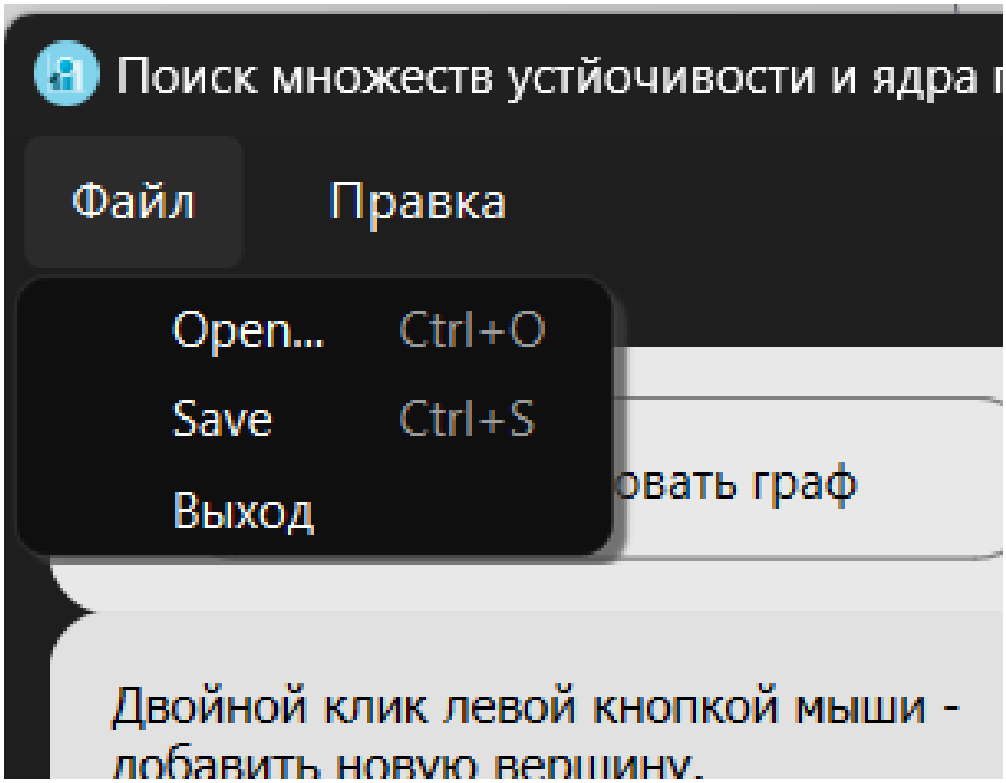


Рис. 3: Кнопка ”Файл”

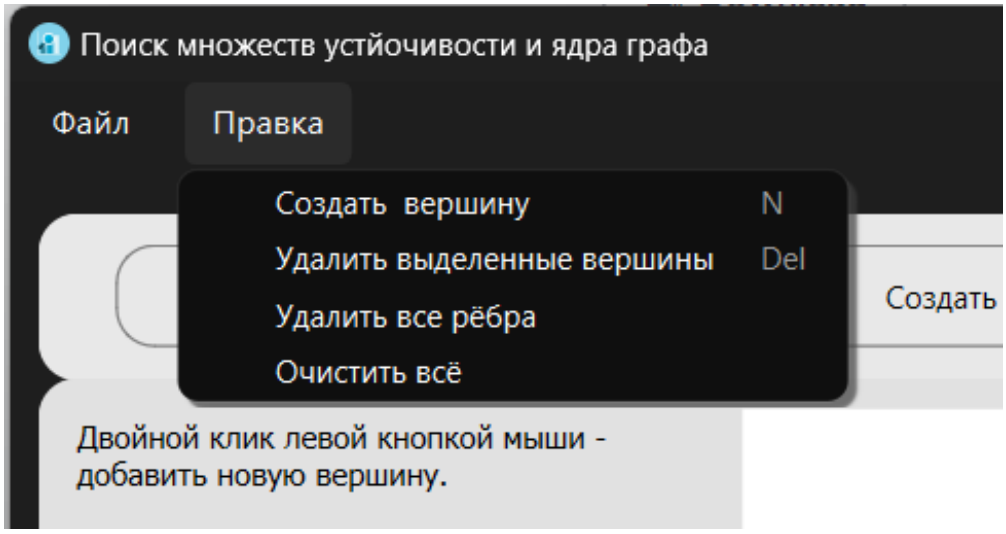


Рис. 4: Кнопка "Правка"

4.2. Окно ввода

В окне для ввода графа можно выбрать один из методов ввода графа: матрица смежности, список ребер, список связности или матрица инцидентности (Рис. 5). Примеры ввода представлены на Рис. 6 - Рис. 9

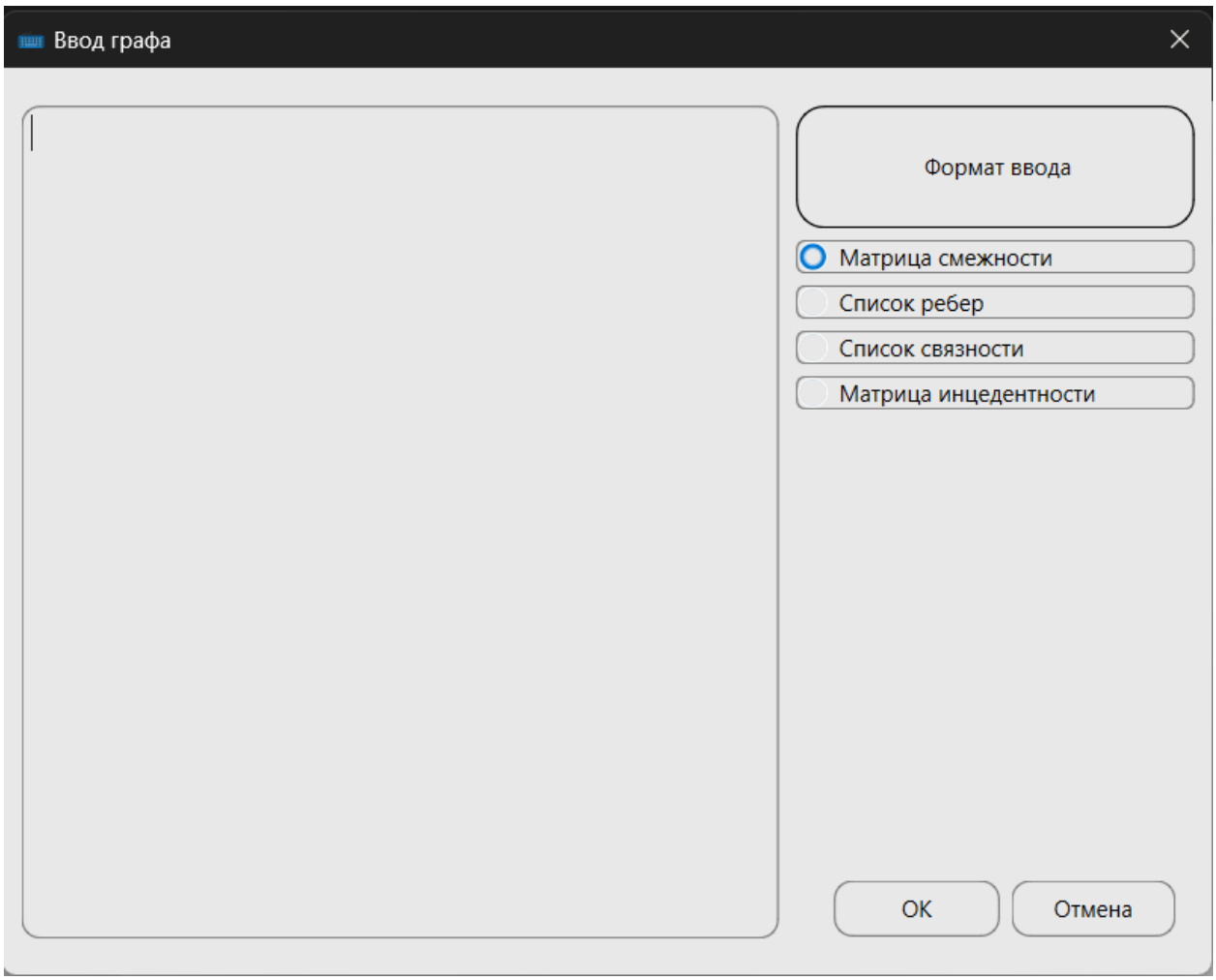
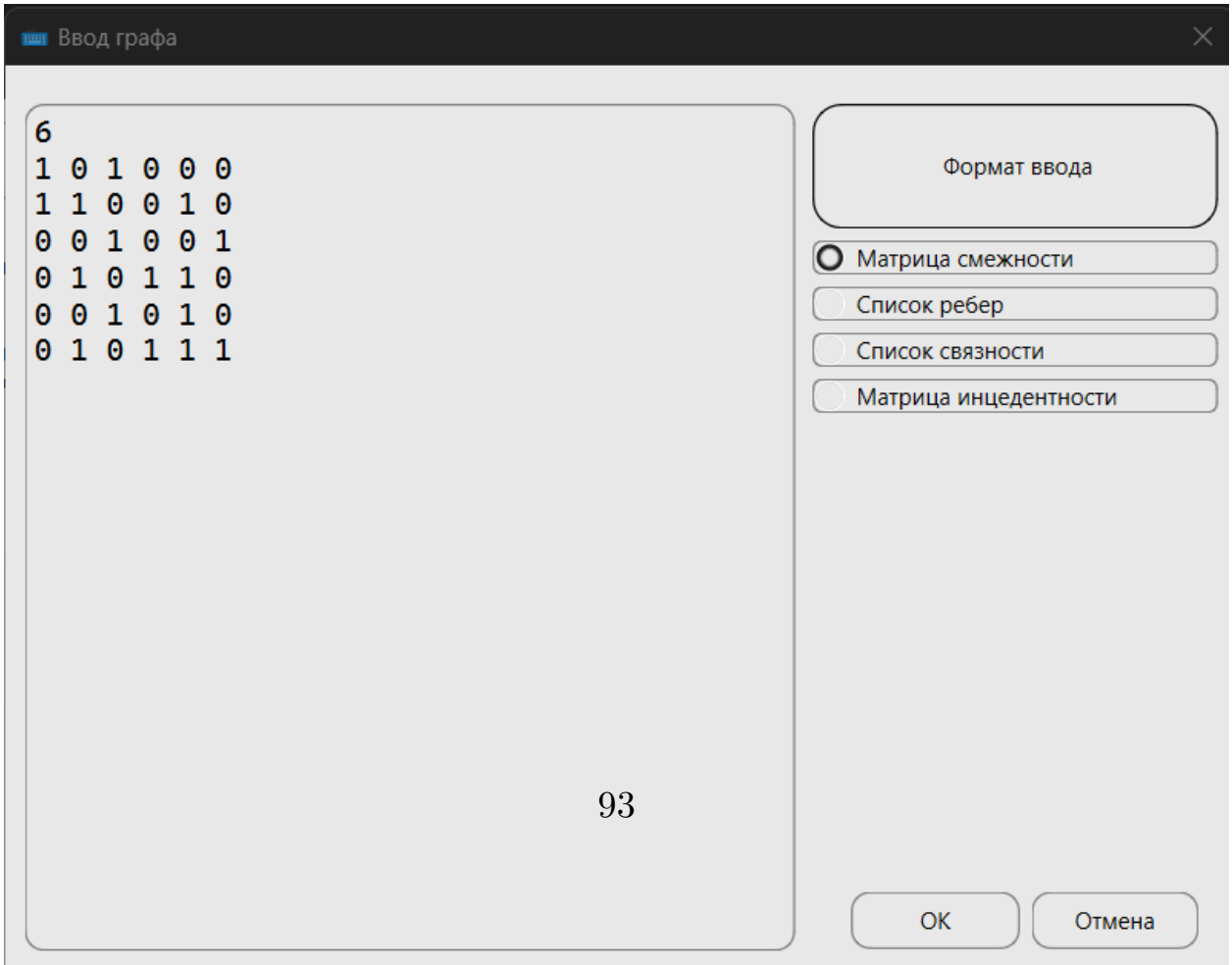


Рис. 5: Окно для ввода графа



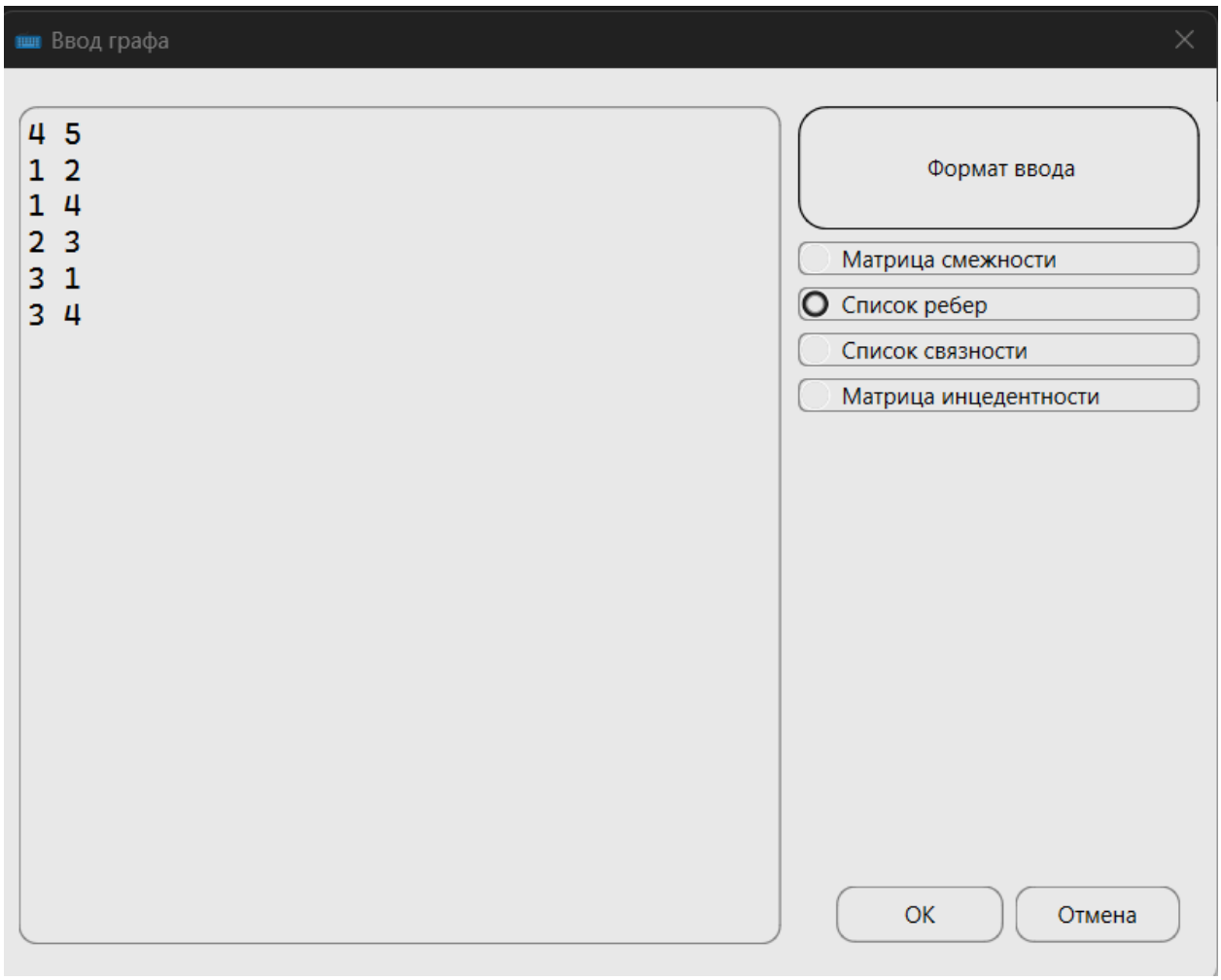


Рис. 7: Пример 2

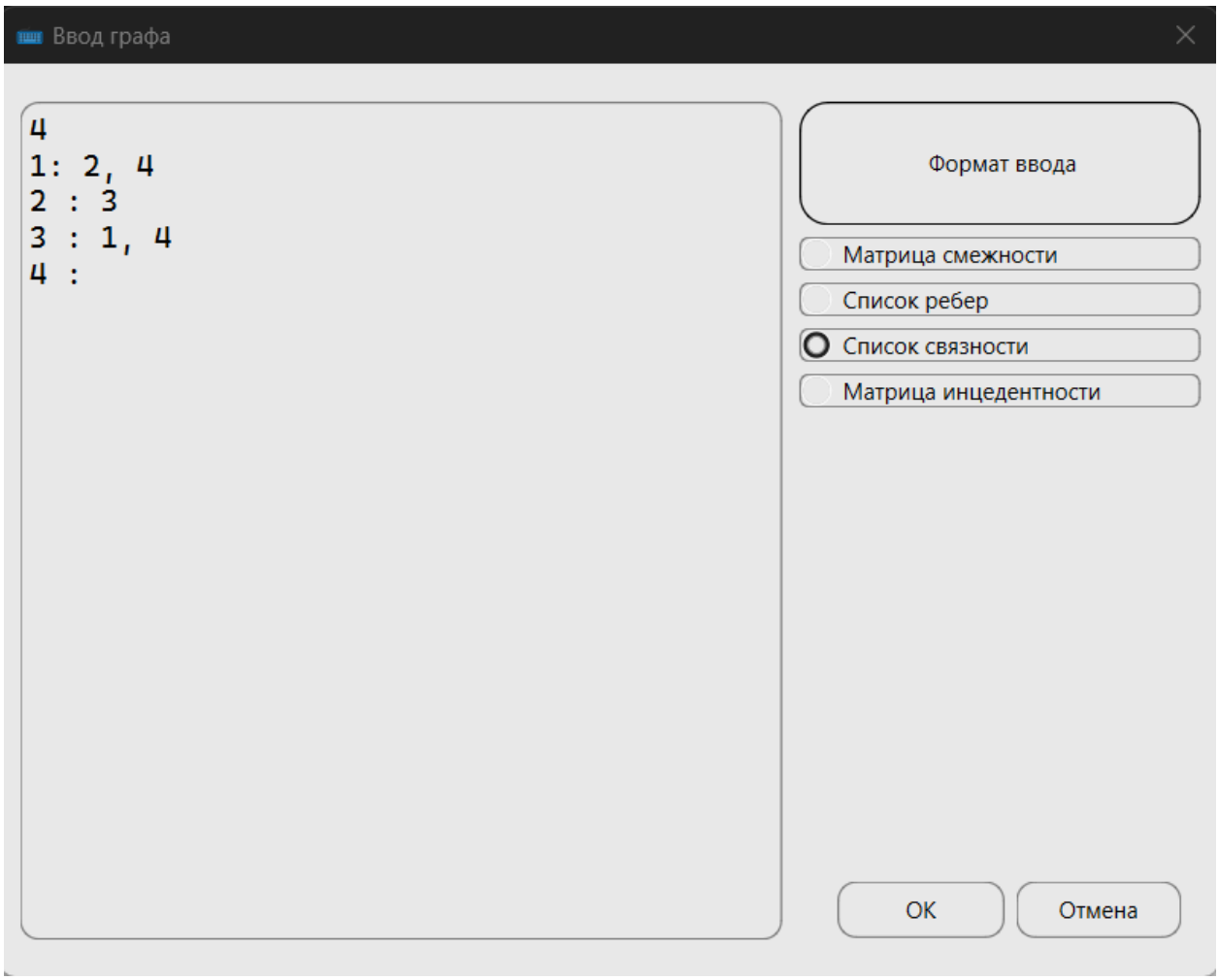


Рис. 8: Пример 3

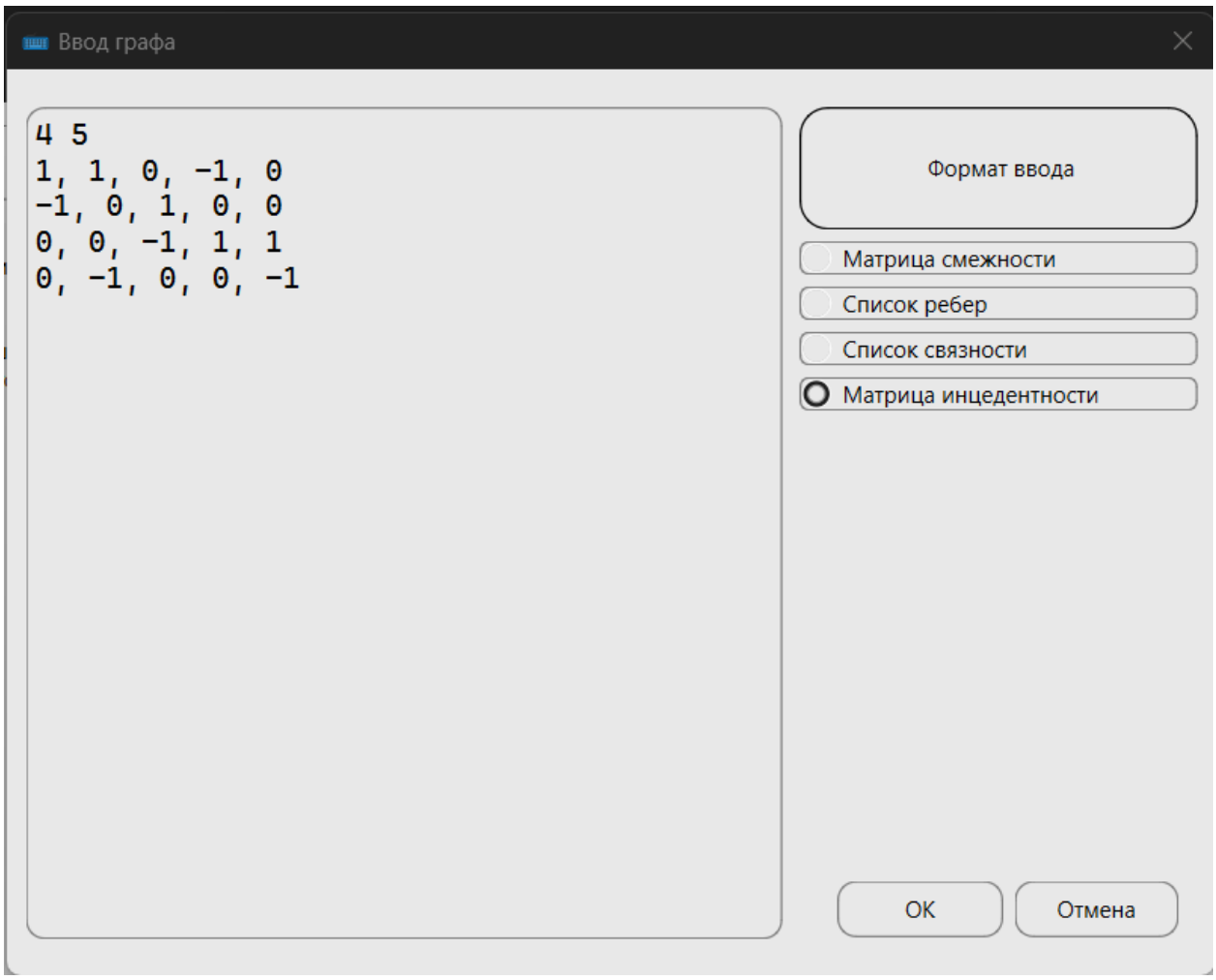


Рис. 9: Пример 4

5. Вывод

Поставленная цель была достигнута, программа может выводить множества внутренней, внешней устойчивости, ядро графа, а также имеет множество функций (например, сохранение графа в файл). В процессе разработки возникло множество трудностей, которые необходимо было решить. После выполнения работы остались идеи доработки проекта, однако я считаю, что проект можно считать завершенным.