

МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

ОТЧЕТ
по учебной практике
Тема: Сильно связанные компоненты оргграфа

Студентка гр. 8383	_____	Аверина О.С.
Студентка гр. 8383	_____	Максимова А.А.
Студент гр. 8383	_____	Мирсков А.А.
Руководитель	_____	Фирсов М.А.

Санкт-Петербург
2020

ЗАДАНИЕ НА УЧЕБНУЮ ПРАКТИКУ

Студентка Аверина О.С. группы 8383

Студентка Максимова А.А. группы 8383

Студент Мирсков А.А. группы 8383

Тема практики: **Сильно связанные компоненты орграфа**

Задание на практику:

Командная итеративная разработка визуализатора алгоритма(ов) на Java с графическим интерфейсом.

Алгоритм: Косарайю.

Сроки прохождения практики: 29.06.2020 – 12.07.2020

Дата сдачи отчета: 11.07.2020

Дата защиты отчета: 00.07.2020

Студентка		Аверина О.С.
Студентка		Максимова А.А.
Студент		Мирсков А.А.
Руководитель		Фирсов М.А.

АННОТАЦИЯ

Целью данной практической работы является изучение и углубление теоретических знаний языка Java, закрепление материала при разработке собственного пошагового визуализатора алгоритма поиска сильно связанных компонент орграфа, обладающего удобным и понятным пользовательским интерфейсом и предусматривающего поведение пользователя, которое без обработки, может приводить к неопределенному поведению программы.

Данная практическая работа состоит из введения, в котором описана спецификация приложения, сопровождаемая макетом меню и диаграммой сценариев, требований, которые должны быть реализованы в программе, прототипа и промежуточных версий кода, плана разработки приложения и распределения ролей в бригаде, описания особенностей реализации, используемых в работе структур данных и разработанных методов, тестирования работы алгоритма и пользовательского интерфейса, а также заключения и списка источников, используемых при написании программы.

SUMMARY

The purpose of this practical work is to study and deepen the theoretical knowledge of the Java language, consolidate the material when developing your own step-by-step visualizer of the search algorithm for strongly connected components of a digraph, which has a convenient and intuitive user interface and provides user behavior that, without processing, can lead to undefined program behavior.

This practical work consists of an introduction, which describes the specification of the application, accompanied by a menu layout and a diagram of scenarios, requirements that must be implemented in the program, prototype and

intermediate versions of the code, an application development plan and role distribution in the team, description of the implementation features used in the work of data structures and developed methods, testing the operation of the algorithm and the user interface, as well as the conclusion and list of sources used when writing the program.

СОДЕРЖАНИЕ

	Введение	6
1.	Требования к программе	7
1.1.	Исходные требования к программе*	7
1.2.	Уточнение требований после сдачи прототипа	13
1.3.	Уточнение требований после сдачи 1-ой версии	13
1.4.	Уточнение требований после сдачи 2-ой версии	13
2.	План разработки и распределение ролей в бригаде	14
2.1.	План разработки	14
2.2.	Распределение ролей в бригаде	15
3.	Особенности реализации	16
3.1.	Описание алгоритма Косарайю	16
3.2.	Структуры данных	18
3.3.	Основные методы	22
4.	Тестирование	30
4.1.	План тестирования	30
4.2.	Тестирование работы алгоритма и ввода графа из файла.	32
4.3.	Тестирование пошаговой визуализации	51
	Заключение	63
	Список использованных источников	64
	Приложение А. Исходный код – только в электронном виде	65

ВВЕДЕНИЕ

Целью выполнения данной практической работы, является разработка программы, на базе высокоуровневого языка Java, реализующей пошаговую визуализацию алгоритма поиска сильно связанных компонент орграфа. Визуализатор алгоритма при этом должен обладать понятным и удобным пользовательским интерфейсом.

Реализация поиска сильно связанных компонент основана на алгоритме Косарайю, в котором ключевым аспектом является поиск в глубину с фиксированием времени выхода из каждой вершины орграфа.

Под сильно связными компонентами орграфа подразумевается его максимальные по включению сильно связанные подграфы. Сильно связанные подграф - это граф, содержащий некое подмножество вершин данного графа и все ребра, инцидентные данному подмножеству, в котором между любыми двумя вершинами, включенными в него, существуют ориентированные пути из s в t и из t в s , где s и t - две любые вершины подграфа.

1. ТРЕБОВАНИЯ К ПРОГРАММЕ

1.1. Исходные требования к программе

1.1.1. Входные данные

Входные данные, а именно орграф, описываемый вершинами и ориентированными невзвешенными ребрами, может быть задан как список ребер, где указывается “Количество ребер”, а затем строки формата “Начальная вершина, конечная вершина” в текстовом файле, импортируемого в программу при нажатии клавиши “Импорт файла”.

Пример:

```
4
1 3
4 2
3 4
3 2
```

Также может быть построен вручную с помощью использования клавиш, располагаемых на панели управления и нажатия на экран с помощью курсора мыши для выбора местоположения вершин или построения ориентированного ребра между ними. Так, чтобы построить граф нужно выполнить следующую последовательность шагов:

1. Нажатие клавиши “Добавление вершины” с последующим щелчком на место, куда будет добавлена вершина
2. Для добавления ориентированного ребра: выбор клавиши “Добавить ребро”, поочередное нажатие на вершины, между которыми будет построено ориентированное ребро (первая вершина начальная, вторая - конечная)
3. Для отмены добавления вершины или ребра необходимо выбрать одну из кнопок “Удаление ребра”, “Удаление вершины” и щелкнуть на объект

1.1.2. Визуализация

Графический интерфейс представляет собой меню, состоящее из панели управления - набора клавиш, предназначенных для вызовов команд, реализующих построение графа, окна, в котором отображается пошаговая визуализация алгоритма или выводится результат работы программы - раскрашенный орграф, в зависимости от выбора пользователя, а также имеется окно, используемое для вывода текстовых пояснений и промежуточной информации.

Пошаговая визуализация алгоритма:

- 1) Построение графа, введенного пользователем.
- 2) При визуализации поиска в глубину, вершины и ребра, при посещении будут помечаться цветом. Рядом с вершиной будет написано время выхода.
- 3) Затем запуск поиска компонент сильной связности, окрашивание каждой компоненты в свой цвет.

*При этом на каждом шаге будет выводиться описание действия.

Эскиз интерфейса, реализующего визуализацию алгоритма поиска сильно связанных компонент орграфа, а также предоставляющего возможность ввода входных данных одним из нескольких предоставленных способов, изображен на рисунке 1.

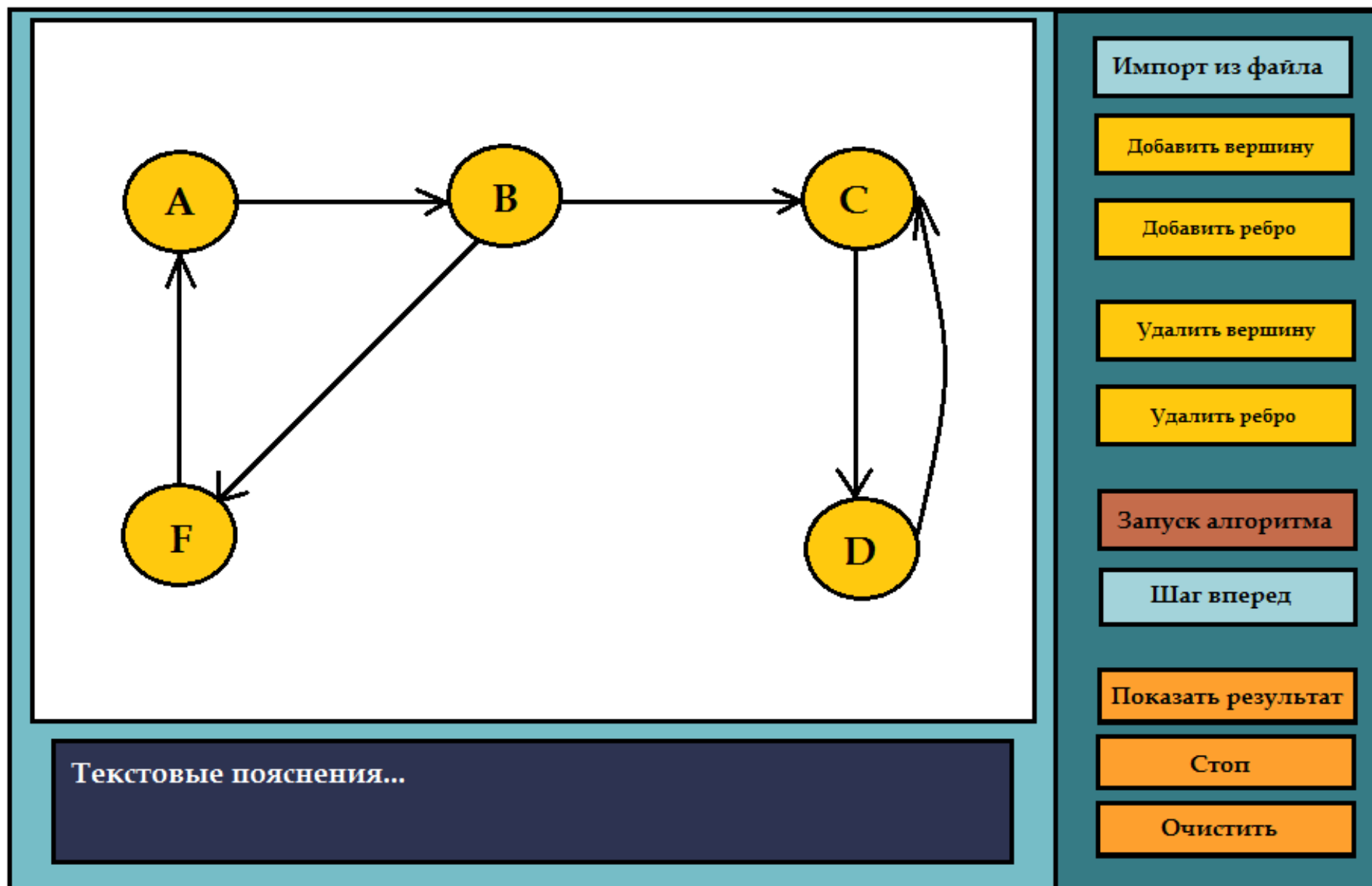


Рис. 1 - Макет меню

1.1.3. Взаимодействие пользователя с программой

Весь пользовательский интерфейс описан на панели управления меню - названия клавиш, являются одноименными методам, реализующим интерфейс. Назначение клавиш см. в таблице 1.

Клавиша	Назначение
“Импорт из файла”	Используется для ввода орграфа через файл.
“Добавить вершину”	Предназначена для добавления новой вершины.
“Добавить ребро”	Необходима для добавления нового ориентированного ребра.
“Удалить вершину”	Используется для удаления вершины.
“Удалить ребро”	Используется для удаления ребра.
“Запуск алгоритма”	После того, как пользователь ввел граф из файла или построил вручную, становится возможным нажатие данной клавиши, которая запускает работу самого алгоритма, сопровождаемую пошаговой визуализацией.
“Шаг вперед”	Переходит к следующему шагу алгоритма.
“Очистить”	Очищение экрана. Предоставляет возможность повторного использования программы без перезапуска, начиная с ввода исходного орграфа.
“Показать результат”	Используется для получения графа, после применения к нему алгоритма Косарайю, без пошаговой визуализации.
“Стоп”	При нажатии на эту клавишу выполнение алгоритма останавливается, введенный граф остается и может быть отредактирован перед повторным запуском алгоритма.

Таблица 1 - Клавиши панели управления

Возможные сценарии взаимодействия пользователя с интерфейсом разрабатываемой программы, представлены на рисунке 2.

Во время работы алгоритма в строке состояний снизу от окна графа будут поясняться действия алгоритма, производимые на соответствующем шаге, например: “Найдена компонента связности 1. Запуск поиска в глубину из вершины 5”. По возможности такие сообщения будут подробными, но, не

слишком нагроможденными и будут формулироваться из расчета, что пользователь ранее не был знаком с алгоритмом Косарайю.

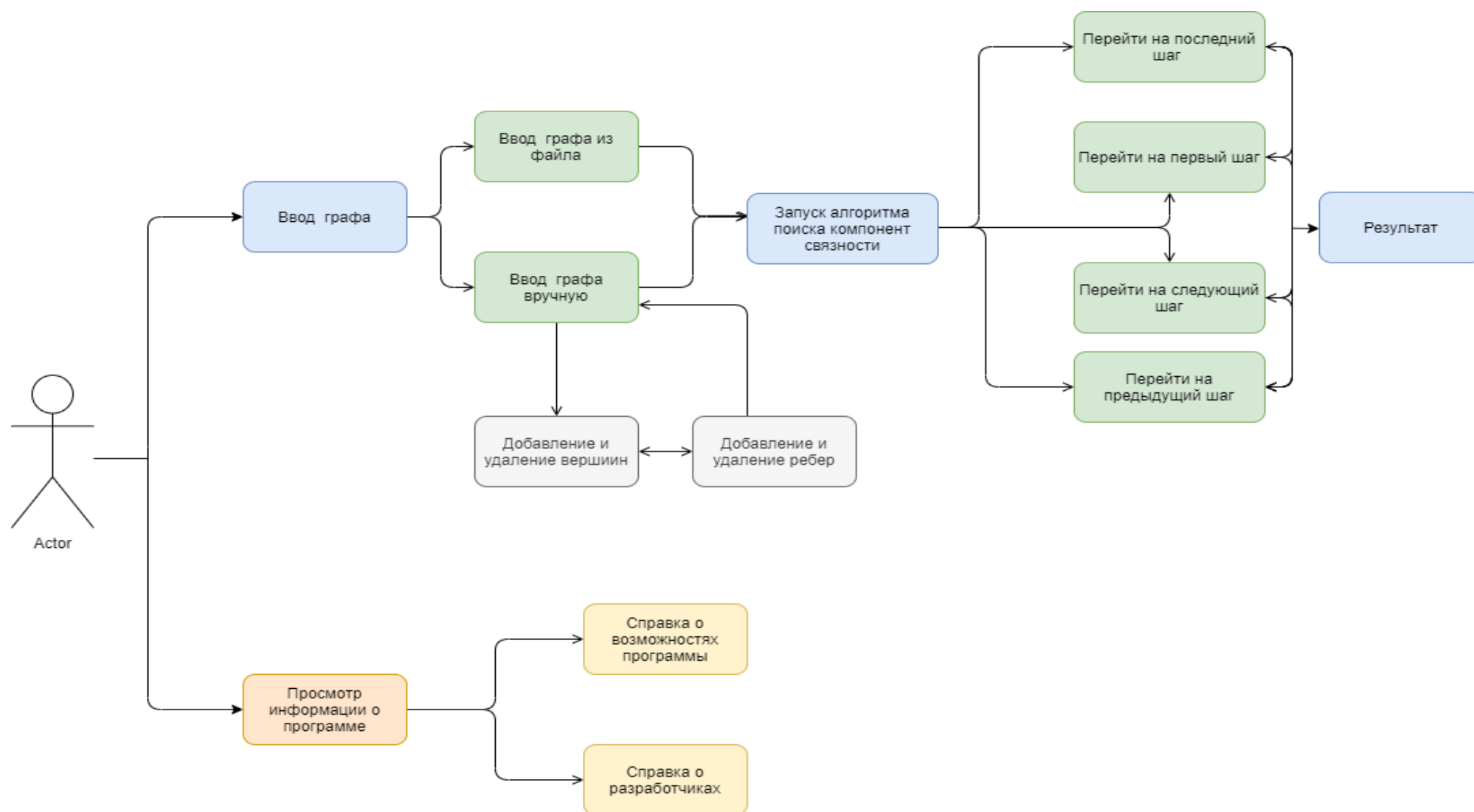


Рис. 2. Диаграмма сценариев

1.2. Уточнения требований в ходе разработки

1.2.1. Уточнения требований после сдачи прототипа

1. Текстовые пояснения должны выделяться и копироваться.
2. При изменении размера окна должен меняться размер области вывода графа.

1.2.2. Уточнение требований после сдачи 1-ой версии

1. Добавить импорт из файла
2. Добавить вывод некоторых текстовых пояснений

1.2.3. Уточнение требований после сдачи 2-ой версии

1. Выделять обработанные рёбра цветами (древесные - одним, не древесные - другим).
2. Выделять обработанное на шаге ребро отдельным цветом.
3. Область для ввода пояснений немного увеличить и сделать так, чтобы каждое новое пояснение добавлялось, а не заменяло предыдущее.
4. Добавить возможность перемещения вершин.
5. При редактировании графа текущая вершина должна выделяться (необходимо при добавлении и удалении ребер).
6. Добавить возможность удаления петель.
7. Добавить возможность редактировать граф после импорта из файла без шанса появления вершин с одинаковыми именами.

2. ПЛАН РАЗРАБОТКИ И РАСПРЕДЕЛЕНИЕ РОЛЕЙ В БРИГАДЕ

2.1. План разработки

Таблица 2 - План разработки программы

02.07	03.07 Показ прототипа	04.07	05.07	06.07 Показ 1 версии
<ul style="list-style-type: none"> + Сдать вводное задание. + Написать спецификацию. 	<ul style="list-style-type: none"> + Сдать спецификацию. + Создать окно интерфейса, добавить кнопки управления. + Сделать план тестирования + Сдать прототип программы. 	<ul style="list-style-type: none"> + Реализовать алгоритм + Сделать функцию добавления вершины на поле. + Сделать импорт графа из файла в виде списка ребер и преобразование в список смежности. 	<ul style="list-style-type: none"> + Реализовать добавление ориентированного ребра для двух вершин. + Создать класс вершины + Создать класс графа + Соединить алгоритм и визуализацию. 	<ul style="list-style-type: none"> + Добавить функцию окрашивания вершин, используемую в визуализации + Подключить клавиши: "Импорт из файла", "Добавление ребра", "Добавление вершины", "Запуск". + Провести тестирования + Исправить недочеты
07.07	08.07 Показ 2 версии	09.07	10.07	11.07 Показ финальной версии
<ul style="list-style-type: none"> + Добавить пометки времени выхода вершин при обходе в глубину. + Добавить удаление ребра и вершины. + Добавить функцию очистки поля и подключить кнопку "Очистить". + Сделать текстовые пояснения. 	<ul style="list-style-type: none"> + Реализовать пошаговую визуализацию алгоритма. + Подключить клавишу "Шаг вперед", "Показать результат". + Провести тестирования. + Исправить недочеты недочеты. 	<ul style="list-style-type: none"> + Подключить клавишу "Стоп". + Заполнить разделы "Структуры данных", "Основные методы", "Список используемой литературы" в отчете + Добавить выделение вершин в разных событиях + Добавить окраску обработанных ребер 	<ul style="list-style-type: none"> + Провести финальное тестирование, добавить в отчет тестирование пошаговой визуализации. + Исправить недочеты. 	<ul style="list-style-type: none"> + Сдать финальную версию.

2.2. Распределение ролей в бригаде

Распределение ролей в процессе разработки представлено в таблице 3.

Таблица 3 - Распределение ролей

Роль	Ответственный(е)
Лидер	Ольга Аверина
Алгоритмист	Анастасия Максимова
Фронтенд	Андрей Мирсков, Ольга Аверина
Тестировщик	Ольга Аверина, Анастасия Максимова
Документация	Анастасия Максимова

Пояснение:

- ☐ **Лидер** - имеет решающее право голоса, занимается управлением репозитория, помогает фронтенду
- ☐ **Алгоритмист** - реализует основной алгоритм программы (поиска компонент сильной связности), отвечает за его работу и обеспечивает его безопасность от пользователя (обрабатывает ошибки, которые возможно предвидеть заранее)
- ☐ **Фронтенд** - реализация пользовательского интерфейса, отвечает за его наполнение, реализацию, включая пошаговую визуализацию алгоритма
- ☐ **Тестировщик** - организует тестирование функционала, создает наборы тестовых данных, знает, что тестировалось, а что нет и по какой причине
- ☐ **Документация** - создание документации проекта в виде отчета, выбор формата комментариев, используемых в программе, следит за их написанием

3. ОСОБЕННОСТИ РЕАЛИЗАЦИИ

Для разработки графического интерфейса используется библиотека JavaFX.

3.1. Описание алгоритма Косарайю.

Для реализации алгоритма, выполняющего поиск сильно связанных компонент в ориентированном графе, были написаны три класса: `Event`, `Vertex`, `AlgorithmKosarayu`, описанные в разделах 3.2. Структуры данных и 3.3. Основные методы.

На вход алгоритм принимает список ребер, а на выходе возвращает список событий, используемый для пошаговой реализации алгоритма.

Список событий имеет следующий вид: `ArrayList <Event> events`, где `Event` - класс, описывающий в конкретный момент времени одно из девяти событий:

Событие 1: Запуск метода `dfs1`, выполняющего поиск в глубину, от вершины V_1 .

Событие 2: Конец `dfs1` - у текущей вершины V_1 не осталось соседних необработанных вершин и все рекурсивные вызовы метода уже выполнены.

Событие 3: Переход по ребру (во время выполнения `dfs1`) в графе из вершины V_1 в вершину V_2 .

Событие 4: У текущей вершины (во время выполнения `dfs1`) V_1 не остались необработанные соседние вершины. Возврат на вершину V_2 , из которой попали в V_1 .

Событие 5: Инвертирование графа - изменение направления ребер на противоположные, за исключением петель.

Событие 6: Запуск метода `dfs2`, выполняющего поиск в глубину, от вершины V_1 , которая принадлежит компоненте X .

Событие 7: Конец `dfs2` - у текущей вершины V_1 не осталось необработанных соседних вершин и все рекурсивные вызовы метода уже выполнены.

Событие 8: Переход по ребру (во время выполнения `dfs2`) в графе из вершины V_1 в вершину V_2 , принадлежащих компоненте X .

Событие 9: У текущей вершины (во время выполнения dfs2) V_1 не остались необработанные соседние вершины. Возврат на вершину V_2 , из которой попали в V_1 .

Событие 10: Сигнализирует о следующей, по порядку времени выхода, вершине.

Алгоритм можно представить в виде следующей последовательности шагов:

Шаг 1. Создание графа (массив вершин): список ребер преобразуется в списки смежности вершин.

Шаг 2. Запуск поиска в глубину (на исходном графе), из необработанных вершин, с фиксированием времени выхода из них (dfs1):

Шаг 2.1. Помечаем вершину, как обрабатываемую "0".

Шаг 2.2. Запоминаем время входа в вершину.

Шаг 2.3. Ищем смежные необработанные вершины, если нет, переход на Шаг 2.4.

Шаг 2.3.1. Если нашли, проверяем: петля?

Шаг 2.3.1. 1. Да, пропускаем вершину и возвращаемся к шагу 2.3.

Шаг 2.3.1. 2. Нет - переход на шаг 2.3.2.

Шаг 2.3.2. Выполняем рекурсивный вызов поиска в глубину от найденной вершины, запоминая, из какой вершины в нее попали.

Шаг 2.4. Помечаем вершину, как обработанную "1".

Шаг 2.5. Запоминаем время выхода из вершины.

Шаг 3. Инвертируем дуги исходного ориентированного графа:

Шаг 3.1. В исходном списке ребер меняем вершины местами, кроме, петель.

Шаг 3.2. Используем преобразованный список, для инициализации инвертированного графа.

Шаг 4. Запуск поиска в глубину (на инвертированном графе), из необработанных вершин, выбирая вершины в порядке уменьшения времени выхода из них, полученном на шаге 2:

Шаг 4.1. Помечаем вершину, как обрабатываемую "0".

Шаг 4.2. Ищем смежные необработанные вершины, если нет, переход на шаг 4.3.

Шаг 4.2.1. Если нашли, проверяем: петля?

Шаг 4.2.1.1. Да, пропускаем вершину и возвращаемся к шагу 4.2.

Шаг 4.2.1.2. Нет - переход на шаг 4.2.2.

Шаг 4.2.2. Фиксируем принадлежность к компоненте сильной связности.

Шаг 4.2.3. Выполняем рекурсивный вызов поиска в глубину от найденной вершины, запоминая, из какой вершины в нее попали.

Шаг 4.3. Помечаем вершину, как обработанную "1".

Шаг 5. Снова инвертируем граф - получаем исходный.

Примечание: вершины, имеющие одинаковые значения поля `component` образуют деревья, являющиеся сильно связными компонентами исходного графа.

3.2. Структуры данных

1. Классы, относящиеся к реализации алгоритма Косарайю:

- **Class Vertex:** используется для описания одной вершины.

Поля:

- `private final Integer NAME;` - хранит имя вершины типа `Integer`, так как имя меняться не должно, то используется `final`.
- `private Integer whence = -1;` - переменная типа `Integer`, используемая для хранения имени вершины, из которой перешли по ребру в текущую во время выполнения поиска в глубину.
- `private ArrayList <Vertex> adjacentVertex;` - список вершин, смежных к текущей.

- `private int color = -1;` - целочисленная переменная, используемая во время поиска в глубину, где "-1" обозначает, что вершина еще не обработана, "0" - обрабатывается, "1" - обработана.
- `private int component = -1;` - целочисленная переменная, используемая для хранения информации о принадлежности вершины сильной компоненте связности.
- `private int timeOut = -1;` - целочисленная переменная, используемая для хранения информации о времени выхода из конкретной вершины, при поиске в глубину.

- **Class Event:**

Поля:

- `private final Integer EVENT_NUMB;` - переменная типа `Integer`, хранящая номер события (от 1 до 10), изменять значение не предполагается, поэтому используется `final`.

Примечание: остальные поля используются в зависимости от события.

- `private Integer nameVertex = -1;` - хранит имя вершины, используется при событиях: 1, 2, 7, 10 по.
- `private Integer[] dataDFS2;` - используется для передачи тройки аргументов, используется при событии 8.
- `private Pair <Integer, Integer> transition;` - используется для хранения двух переменных типа `Integer`, при событиях: 3, 4, 6, 9.
- `private ArrayList <Pair<Integer, Integer>> inventedListEdge;` - список пар, используемый для хранения инвертированного списка ребер для события под номером 5.
- `private String textHints;` - строка, содержащая текстовые пояснения, выводимые в специальное окно, в соответствии с событием.

- **Class AlgorithmKosarayu :**

Поля:

- `private ArrayList <Vertex> graph;` - список вершин, образующих граф.
- `private ArrayList <Pair<Integer, Integer>> timeOut;` - список пар, состоящих из имени вершины и времени выхода, используемый для вызова поиска в глубину от вершин в порядке уменьшения времени выхода из них.
- `private ArrayList <Event> events;` - список событий, используемый для хранения событий, возвращается пользователю после выполнения алгоритма.
- `private ArrayList <Pair<Integer, Integer>> listEdge;` - список ребер, задаваемый пользователем.
- `private int dfsTimer = 1;` - целочисленная переменная, используемая в качестве счетчика времени.
- `private int component = 1;` - целочисленная переменная, используемая для нумерации сильных компонент связности.

2. Классы, относящиеся к реализации графического интерфейса:

- **Class Graph:**

Поля:

- `private final ArrayList <Vertex> vertexes` - список вершин в графе.
- `private ArrayList <Event> events` - список событий в ходе работы алгоритма.
- `private int orderIndex = 0` - индекс текущей вершины в массиве времени выхода.
- `private int eventIndex = 0` - индекс текущего события.

- **Class Vertex:**

Поля:

- `private final ArrayList<Vertex> adj` – массив смежных вершин с данной.
- `private double x, private double y` – координаты вершины на поле.
- `private final int number` – идентификационный номер вершины.
- `private boolean isHasLoop` – флаг, помечающий, есть ли у вершины петля.
- `private boolean isRinged` – флаг, помечающий, должна ли вершина выделяться красным.
- `private Color currentColor` – цвет окраски вершин.
- `public static int vertexCount` – счетчик вершин.
- `public static int RADIUS` – радиус вершины при отрисовке.
- `private int nextSelectedEdge` – номер вершины, в которой ведет ребро, по которому происходит переход в текущий шаг. Используется для выделения ребра красным цветом.
- `private final ArrayList<Pair<Integer, Color>> selectedEdges` – массив вершин, где ребра между каждой из них и текущей вершиной были просмотрены и должны выделяться определенным цветом.
- `public static Color[] colors` – массив цветов для раскраски.

- **Class Controller:**

- `private Canvas mainCanvas` – поле для отрисовки.
- `private TextArea mainTextArea` – элемент интерфейса для вывода текстовых пояснений.
- `private HBox mainLabel` – контейнер для массива вершин, отсортированных по времени выхода.

- `private Button importButton, addVButton, addEButton, delVButton, delEButton, runButton, stepButton, stepButtonBack, showButton, stopButton, clearButton` – кнопки интерфейса.
- `private Graph graph` – экземпляр класса `Graph`, с которым взаимодействует интерфейс.
- `private Vertex prevChosenVertex` – номер предыдущей выбранной вершины.
- `private Vertex draggedVertex` – номер вершины, для которой происходит перетаскивание.
- **Class ModalWindow** - класс окна ввода имени файла для импорта:
 - `public static String fileName` – введенное имя файла для импорта.
- **Class ImportManager:**
 - `private final ArrayList<Pair<Integer, Integer>> graph` – список ребер, считанный из файла.
 - `private boolean isOpenAndReadOK` – флаг успешного считывания графа из файла.

3.3. Основные методы

1. Классы, относящиеся к реализации алгоритма Косарайю:

- **Class Vertex:**
 - Содержит конструктор для инициализации полей: `private final Integer NAME;` и `private ArrayList <Vertex> adjacentVertex;`
 - Сеттеры для установки значений полей (например, `public void setColor(int color);` - для поля `color`).
 - Геттеры для получения значений полей (например, `public Integer getNAME();` - для поля `NAME`).

- **Class Event:**

- Содержит конструктор для инициализации полей `private final Integer EVENT_NUMB; private ArrayList <Pair<Integer, Integer>> inventedListEdge; private Integer[] dataDFS2;`
- Сеттеры для установки значений полей (например, `public void setNameVertex(Integer nameVertex);` - для поля `nameVertex`).
- Геттеры для получения значений полей (например, `public Integer getNameEvent();` - для поля `NAME_EVENT`).

- **Class AlgorithmKosarayu:**

- `public AlgorithmKosarayu();` - конструктор класса, используемый для инициализации полей: `private ArrayList <Pair<Integer, Integer>> timeOut; private ArrayList <Event> events;`
- `public ArrayList <Event> start(ArrayList <Pair<Integer, Integer>> listEdge);` - метод, принимающий на вход список ребер, где ребра - пары, типа `<Integer, Integer>`, задаваемый пользователем. Используется для вызова методов в порядке, определенным алгоритмом (создать граф, запустить поиск в глубину, инвертировать граф, запустить поиск в глубину). Метод возвращает массив событий, с помощью которого реализуется пошаговая визуализация алгоритма.
- `private void initGraph();` - метод, ничего не принимающий на вход, используется для инициализации графа - списка вершин, имена которых, а также смежных им вершин, берутся из списка ребер, подаваемого на вход программе. Метод ничего не возвращает.
- `private void dfs1(int index);` - метод, принимающий на вход индекс элемента в графе. Используется для поиска в глубину на ориентированном графе, с фиксированием порядка времени выхода из вершин. Метод ничего не возвращает.
- `private ArrayList <Pair<Integer, Integer>> inverting();` - метод, ничего не принимающий на вход, используется для

инвертирования списка ребер - изменения их направления на противоположное, за исключением петель. Метод возвращает список ребер.

- `private void dfs2(int index);` - метод, принимающий на вход индекс элемента в орграфе. Используется для реализации поиска в глубину на инвертированном графе, с фиксированием принадлежности вершины к конкретной сильной компоненте связности.
- `private int find(Integer name);` - метод, принимающий на вход имя вершины типа `Integer`. Необходим для определения индекса вершины в списке вершин (графа). Метод возвращает искомый индекс.

2. Классы, относящиеся к реализации графического интерфейса:

- **Class Main:**

- `public void start(Stage primaryStage)` - функция для запуска сцены интерфейса. Принимает объект `Stage`, определяющий главное окно программы. В ней происходит инициализация сцены и отрисовка поля по таймеру.

- **Class Graph:**

- `public void addVertex(Vertex vertex)` - функция для добавления переданной вершины в массив вершин графа. Принимает вершину типа `Vertex`.
- `public void removeVertex(Vertex vert)` - функция для удаления переданной вершины из массива вершин графа. Принимает вершину типа `Vertex`.
- `public void addEdge(Vertex vert1, Vertex vert2)` - функция для добавления ребра между двумя вершинами или петли. Принимает начальную и конечную вершины.

- `public void removeEdge(Vertex vert1, Vertex vert2)` - функция для удаления ребра или петли. Принимает начальную и конечную вершины.
- `public void drawAll(GraphicsContext gc)` - функция для отрисовки всех компонент графа. Принимает объект, содержащий контекст поля для отрисовки.
- `private void drawArrow(GraphicsContext gc, double node1X, double node1Y, double node2X, double node2Y, Color color, boolean isOnlyArrow)` - функция для рисования направленного ребра. Принимает контекст поля, начальные и конечные координаты ребра, цвет раскраски и флаг для отрисовки всего ребра или только стрелки.
- `private static void drawRing(GraphicsContext gc, double x, double y)` - функция для рисования окружности поверх вершины, для ее выделения на шаге алгоритма. Принимает контекст поля и координаты вершины.
- `public Vertex checkCollision(double dotX, double dotY, double diff)` - функция для определения, есть ли в радиусе с центром в полученных координатах уже какая-либо вершина. Принимает координаты вершины и дополнительный радиус. Возвращает найденную вершину, если она есть, и null в обратном случае.
- `public void runAlgorithm()` - функция для запуска алгоритма Косарайю. Ничего не принимает и не возвращает.
- `public void visualiseStep(TextArea textArea, HBox labelBox)` - функция для обработки шагов алгоритма. Обрабатывается массив событий, полученный от объекта event, и вызываются соответствующие функции для их обработки. Принимаются объекты текстового поля для записи пояснений и контейнер для массива времени выхода вершин.

- `public void visualiseStepBack(TextArea textArea, HBox labelBox)` - функция для шага назад. Принимаются объекты текстового поля для записи пояснений и контейнер для массива времени выхода вершин.
- `public void standardView(HBox labelBox)` - функция для отображения графа в стандартном виде без выделений и раскрасок. Принимает контейнер для массива времени выхода вершин для его очистки.
- `public void inputFileGraph(GraphicsContext gc, TextArea mainTextArea)` - функция для импорта графа из файла и генерации координат каждой его вершины. Принимает контекст поля и текстовое поле для вывода сообщений.
- `public Vertex findVertex(int number)` - функция для поиска вершины в графе. Принимает номер искомой вершины. Возвращает вершину, если она найдена, или `null`, если нет.
- `private void clearCanvas(GraphicsContext gc)` - функция для очистки контекста поля. Принимает объект контекста поля.
- `public boolean isLastEvent()` - булева функция, определяющая является ли текущий момент последним или нет.
- `public boolean isFirstEvent()` - булева функция, определяющая является ли текущий момент первым или нет.
- `public void endAlgorithm(TextArea textArea, HBox label)` - функция, обрабатывающая события до конца без пошаговой демонстрации.

- **Class Vertex:**

- `public Vertex(double x, double y, int number)` - конструктор, инициализирующий координаты и номер вершины. Принимает координаты и номер вершины.
- `public void draw(GraphicsContext gc)` - функция для рисования вершины. Принимает контекст поля.

- `private void drawLoop(GraphicsContext gc)` - функция для рисования петли вершины.
- `public boolean isDotInRadius(double dotX, double dotY, double diff)` - булева функция, определяющая, входит ли координата в радиус вершины.
- `public static Color computeColor(int numb)` - функция для определения цвета при помощи номера.

А также геттеры и сеттеры полей класса.

- **Class Controller:**

- `private void isBtnAddVertClicked(), isBtnAddRibClicked(), isBtnDelVertClicked(), isBtnDelRibClicked()` - функции для обработки нажатий на кнопки добавления вершины, добавления ребер, удаления вершин и удаления ребер соответственно.
- `private void canvasMousePressed(javaafx.scene.input.MouseEvent mouseEvent)` - функция для обработки нажатия мышкой на экран. Происходит поиск вершины, находящийся по координатам нажатия. Принимает объект события нажатия ПКМ.
- `private void canvasMouseReleased()` - функция для обработки окончания удерживания ПКМ.
- `private void canvasMouseDragged(javaafx.scene.input.MouseEvent mouseEvent)` - функция для обработки удерживания ПКМ. Используется для перемещения вершины путем изменения ее координат.
- `private void runAlgorithm()` - функция для обработки нажатия кнопки Запуск. Запускает алгоритм, добавляет на экран массив времени выхода вершин.
- `private void clearButton()` - функция для обработки кнопки Очистить. Создает новый экземпляр класса Graph.

- `private void stopButton()` - функция для обработки нажатия кнопки Стоп.
- `private void stepBack()` - функция для обработки нажатия кнопки Шаг назад.
- `private void showResult()` - функция для обработки нажатия Показать результат. Запускает алгоритм без пошаговой демонстрации.
- `private void importGraph()` - функция для обработки нажатия кнопки Импорт из файла. Выводит текстовое пояснение, создает новый экземпляр класса и вызывает функцию импорта.
- `public void resizeCanvas()` - функция для изменения размеров поля для отрисовки графа.

- **Class ModalWindow:**

- `public static void newWindow(String title)` - функция для создания окна для ввода имени файла для импорта файла. Принимает строку, записываемую в заголовке окна.
- `public static void closeFile(String str, Stage window)` - функция, которая сохраняет введенное имя файла и закрывает окно.

- **Class VisualSteps:**

- `public static void event1(ArrayList<Vertex> vertexes, int number)` - функция для обработки события начала очередного dfs1.
- `public static void event2(ArrayList<Vertex> vertexes, int number, HBox labelBox)` - функция для обработки события окончания очередного dfs1.
- `public static void event3(ArrayList<Vertex> vertexes, int fromNumber, int toNumber)` - функция для обработки события перехода по ребру в dfs1.
- `public static void event4(ArrayList<Vertex> vertexes, int fromNumber, int toNumber, HBox labelBox)` - функция для обработки события возврата в dfs1.

- `public static void event5(ArrayList<Vertex> vertexes)` - функция для обработки события инвертирования всех рёбер графа.
- `public static void event6(ArrayList<Vertex> vertexes, int number, int colorNumber)` - функция для обработки события начала очередного dfs2.
- `public static void event7(ArrayList<Vertex> vertexes, int number)` - функция для обработки события окончания очередного dfs2.
- `public static void event8(ArrayList<Vertex> vertexes, int fromNumber, int toNumber, int colorNumber)` - функция для обработки события перехода по ребру в dfs2
- `public static void event9(ArrayList<Vertex> vertexes, int fromNumber, int toNumber)` - функция для обработки события возврата в dfs2
- `public static void event10(ArrayList<Vertex> vertexes, int number)` - функция для обработки события просмотра нового кандидата для начала dfs2

- **Class ImportManager:**

- `public ImportManager(String name, TextArea mainTextArea)` - конструктор, в котором происходит импорта файла. Открывает файл с переданным именем и считывает список ребер графа. При работе с файлом производится отлавливание возможных исключений. Принимает имя файла и текстовое поле для записи сообщений.

4. ТЕСТИРОВАНИЕ

4.1. План тестирования

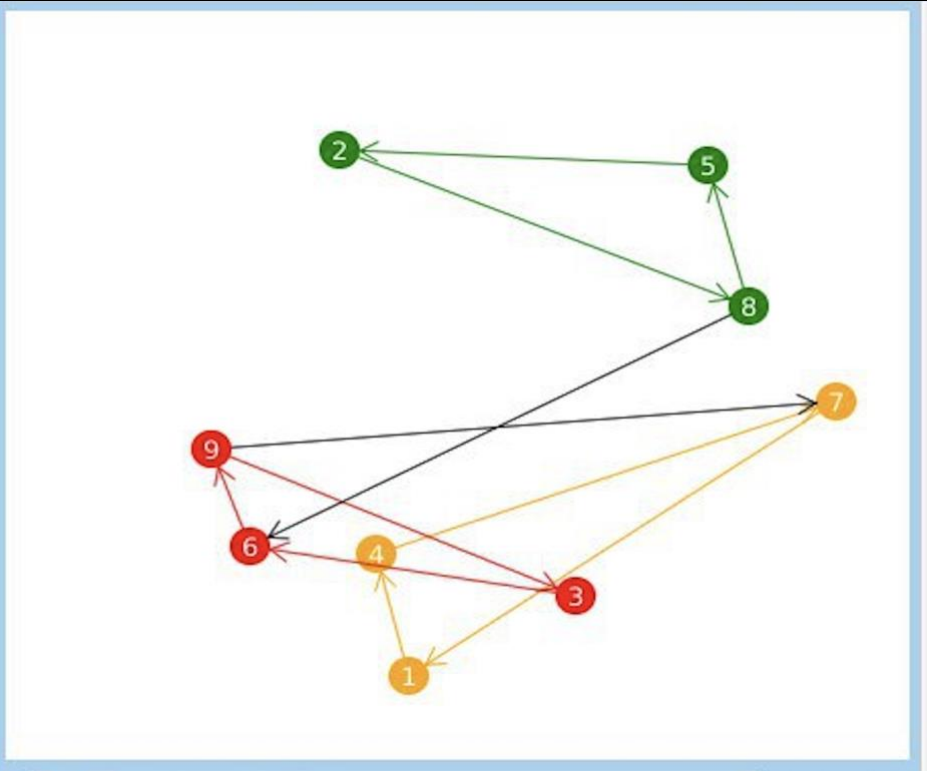
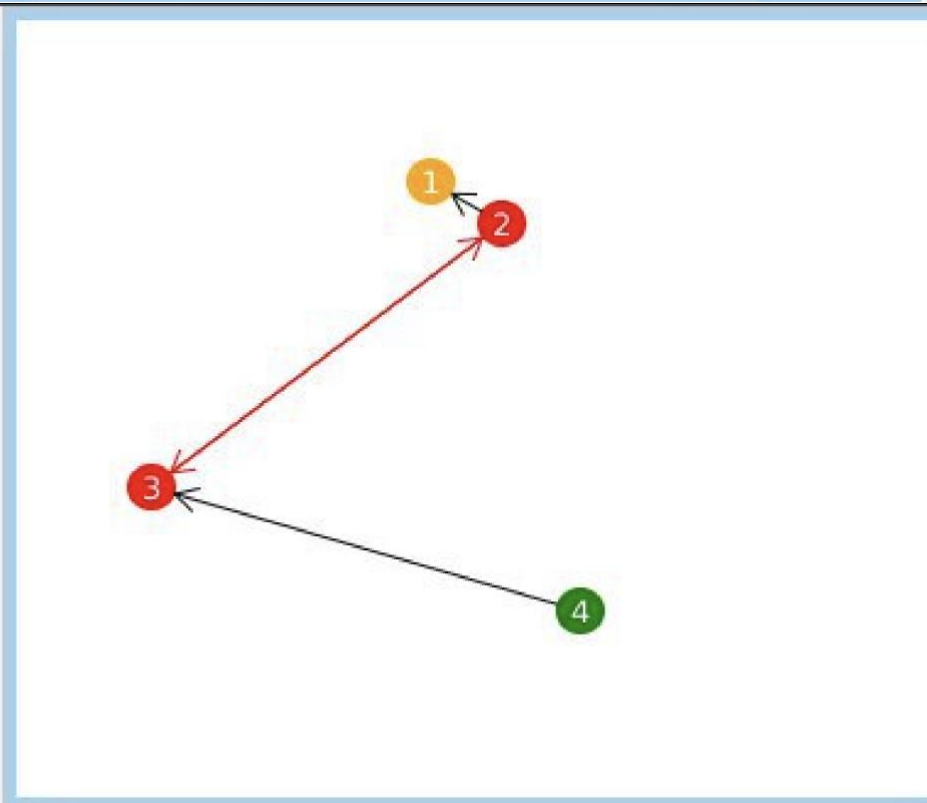
1. В процессе разработки предстоит протестировать следующее:
 - a. Ручной ввод графа.
 - b. Ввод графа из файла.
 - c. Корректность работы алгоритма
 - d. Пошаговая работа алгоритма и раскраска графа.
 - e. Работа кнопок.
2. Для этого будут использоваться следующие методы тестирования:
 - a. Ручной ввод графа.
 - i. Добавление вершин, ребер
 - ii. Удаление вершин, ребер
 - b. Ввод графа из файла.
 - i. Ввод в файл некорректных данных отслеживание реакции программы. Некорректные данные — это, например, ввод из файла с одним числом в файле, или когда в строке с “ребром” введен один номер вершины.
 - c. Корректность работы алгоритма
 - i. Проверка работы алгоритма на графе из одной вершины, из двух несмежных вершин.
 - ii. Проверка работы на входных данных и проверка корректности результата
 - d. Пошаговая работа алгоритма и раскраска графа.
 - i. Запуск и пошаговый “прогон” алгоритма с отслеживанием шагов и окраски соответствующих вершин.
 - e. Работа кнопок.
 - i. Попытка нажать на кнопки “Удалить вершины/ребра” при отсутствии на поле вершин/ребер, соответственно.

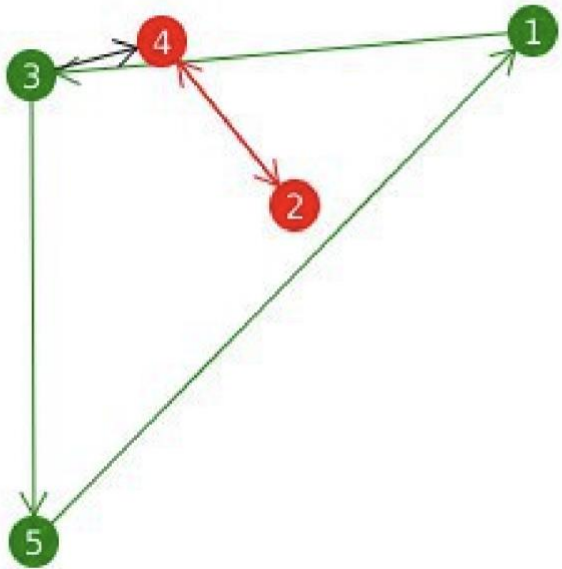
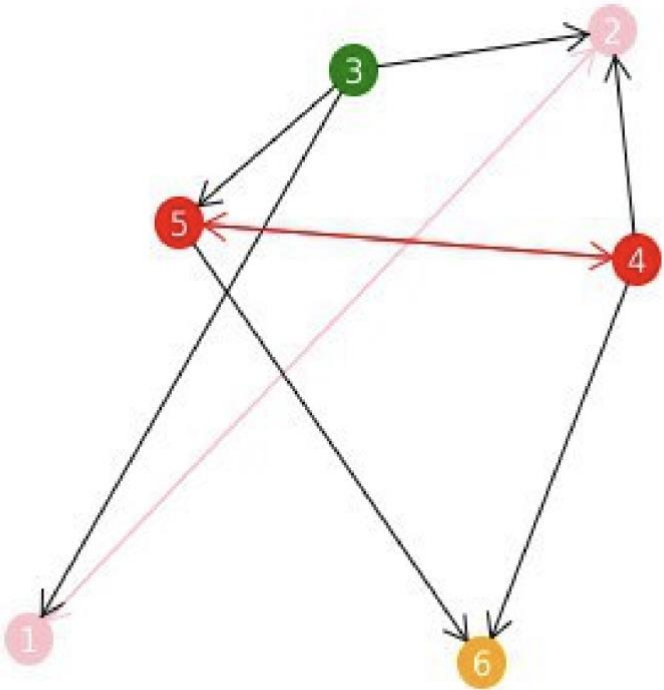
- ii. Попытка во время работы алгоритма нажать кнопки “Импорт из файла”, “Добавить вершины/ребра”, “Удалить вершины/ребра”, “Запуск”.

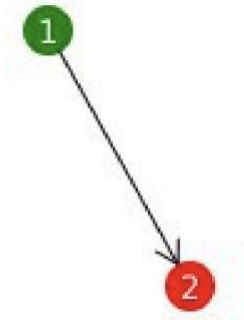
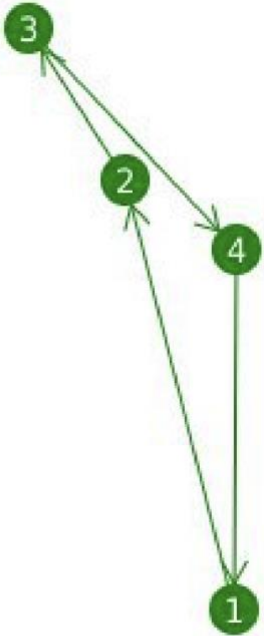
3. Тестирование проводит на следующих этапах разработки:

- a. Ручной ввод графа: между сдачей 1 и 2 версий программы.
- b. Ввод графа из файла: перед сдачей 1 версии.
- c. Корректность работы алгоритма: перед сдачей 1 версии.
- d. Пошаговая работа алгоритма и раскраска графа: перед сдачей 2 версии.
- e. Работа кнопок: перед сдачей финальной версии.


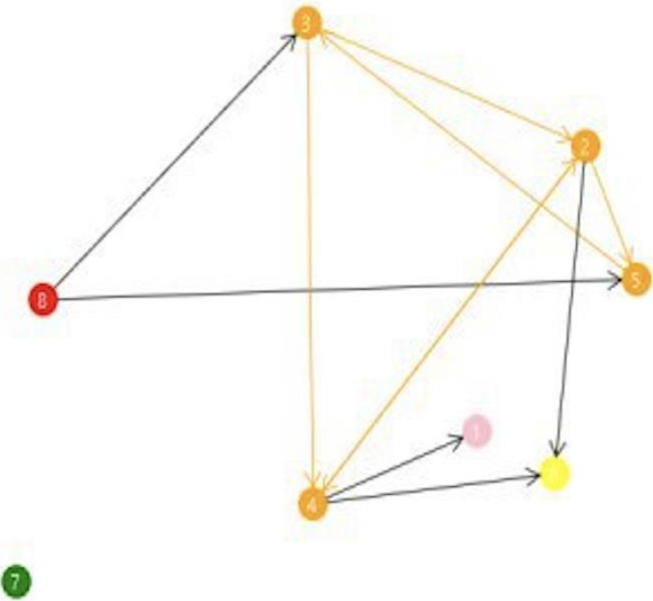
4.2. Тестирование работы алгоритма и ввода графа из файла.

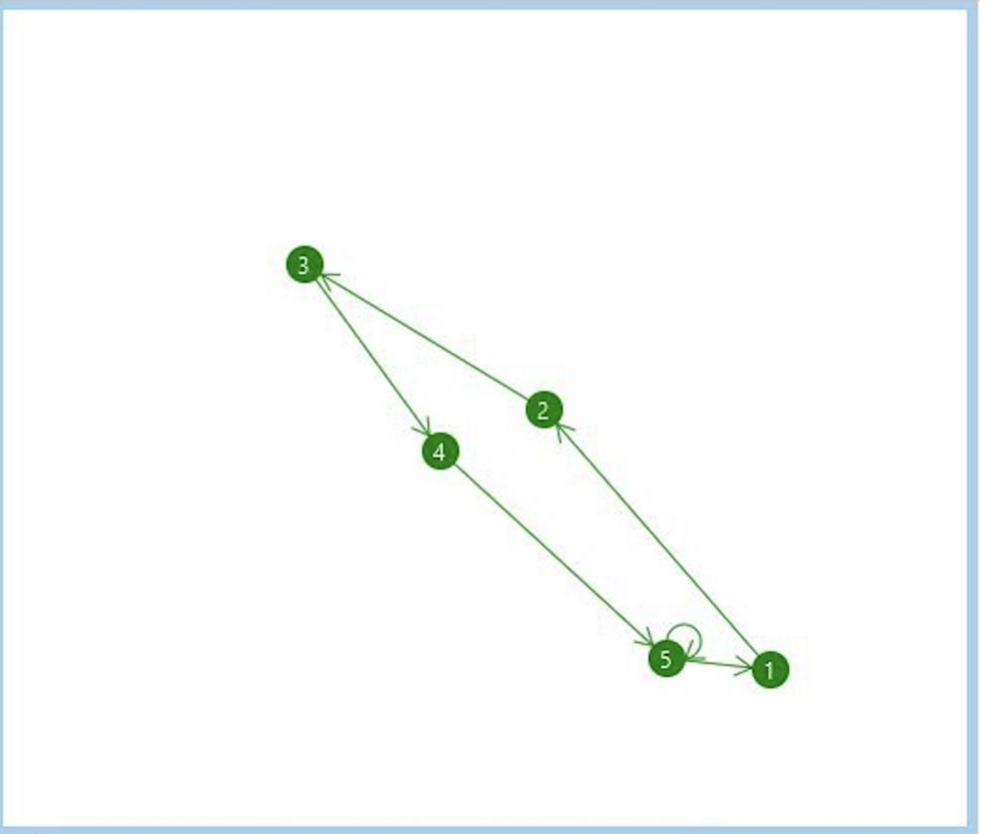
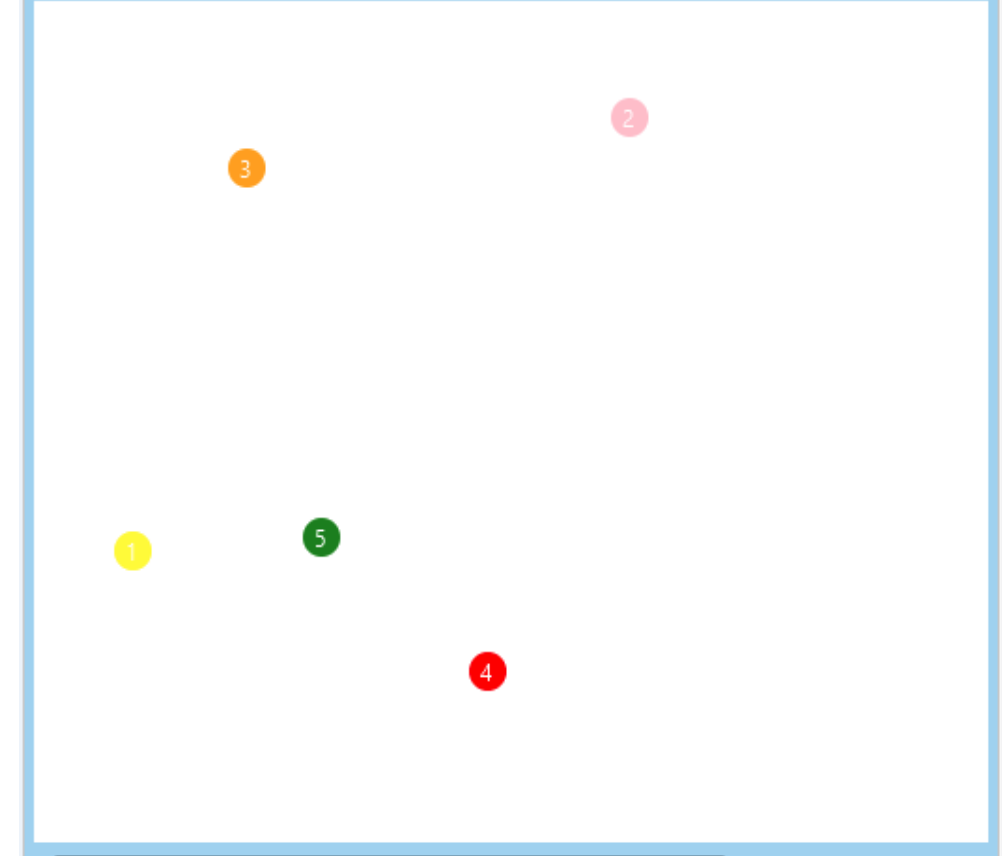
Входные данные	Выходные данные
<p>7 1</p> <p>4 7</p> <p>1 4</p> <p>9 7</p> <p>6 9</p> <p>3 6</p> <p>9 3</p> <p>8 6</p> <p>2 8</p> <p>5 2</p> <p>8 5</p>	
<p>2 1</p> <p>3 2</p> <p>2 3</p> <p>4 3</p>	

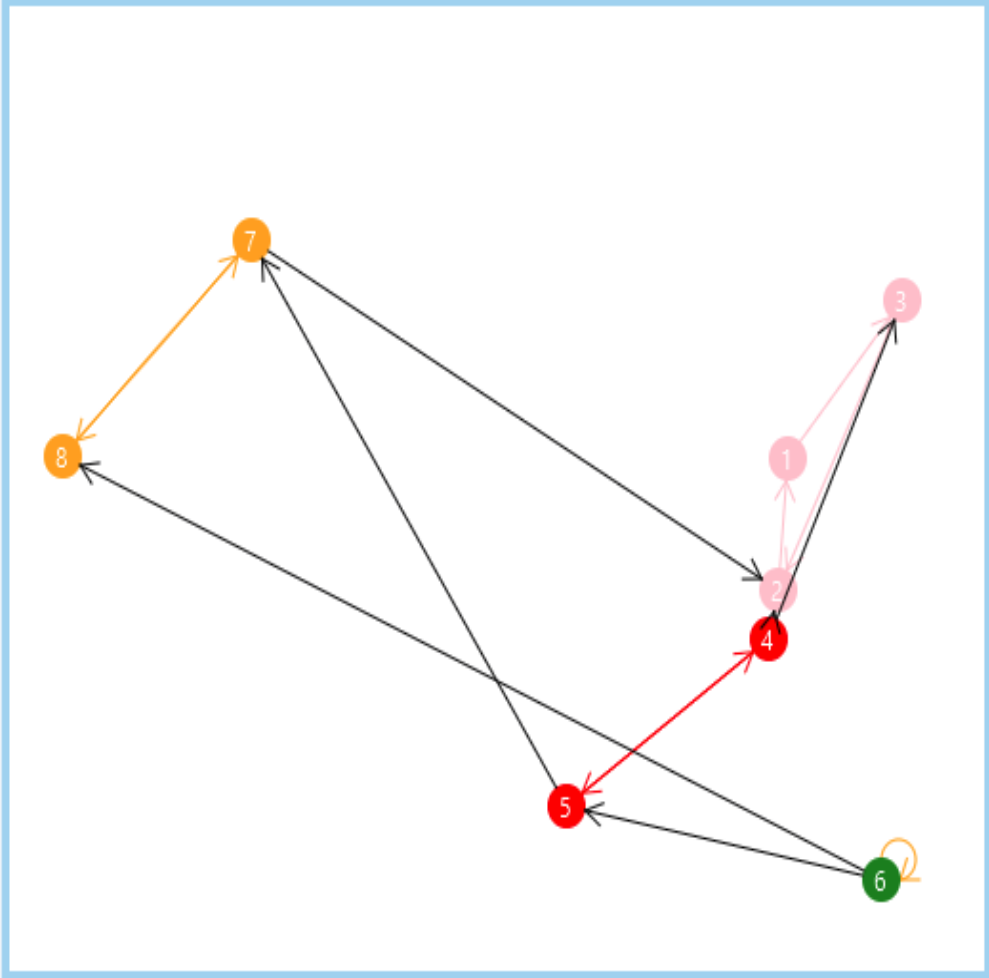

<p>1 3</p> <p>5 1</p> <p>3 5</p> <p>3 4</p> <p>4 2</p> <p>2 4</p>	 <pre> graph TD 3((3)) -- green --> 4((4)) 4((4)) -- green --> 3((3)) 4((4)) -- red --> 2((2)) 3((3)) -- green --> 5((5)) 5((5)) -- green --> 1((1)) </pre>
<p>2 1</p> <p>1 2</p> <p>3 1</p> <p>3 2</p> <p>4 2</p> <p>3 5</p> <p>5 4</p> <p>4 5</p> <p>4 6</p> <p>5 6</p>	 <pre> graph TD 3((3)) -- green --> 2((2)) 3((3)) -- green --> 5((5)) 3((3)) -- green --> 6((6)) 5((5)) -- red --> 4((4)) 4((4)) -- green --> 2((2)) 4((4)) -- green --> 6((6)) 1((1)) -- pink --> 2((2)) 1((1)) -- pink --> 6((6)) </pre>

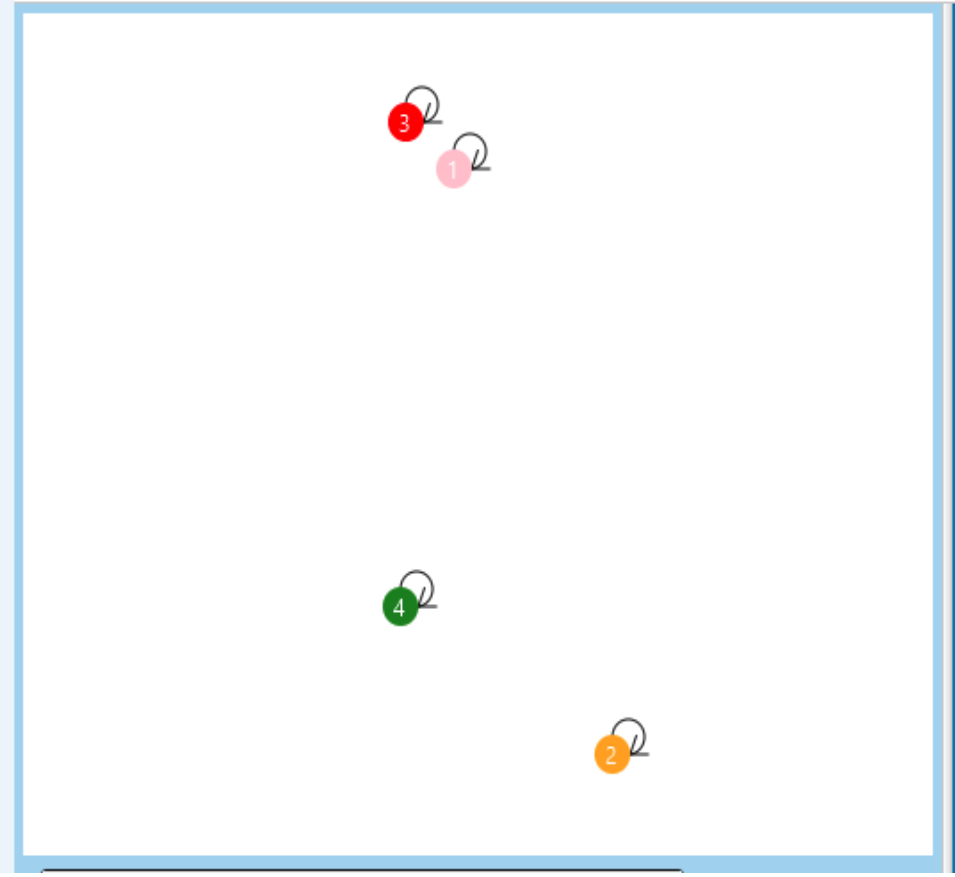
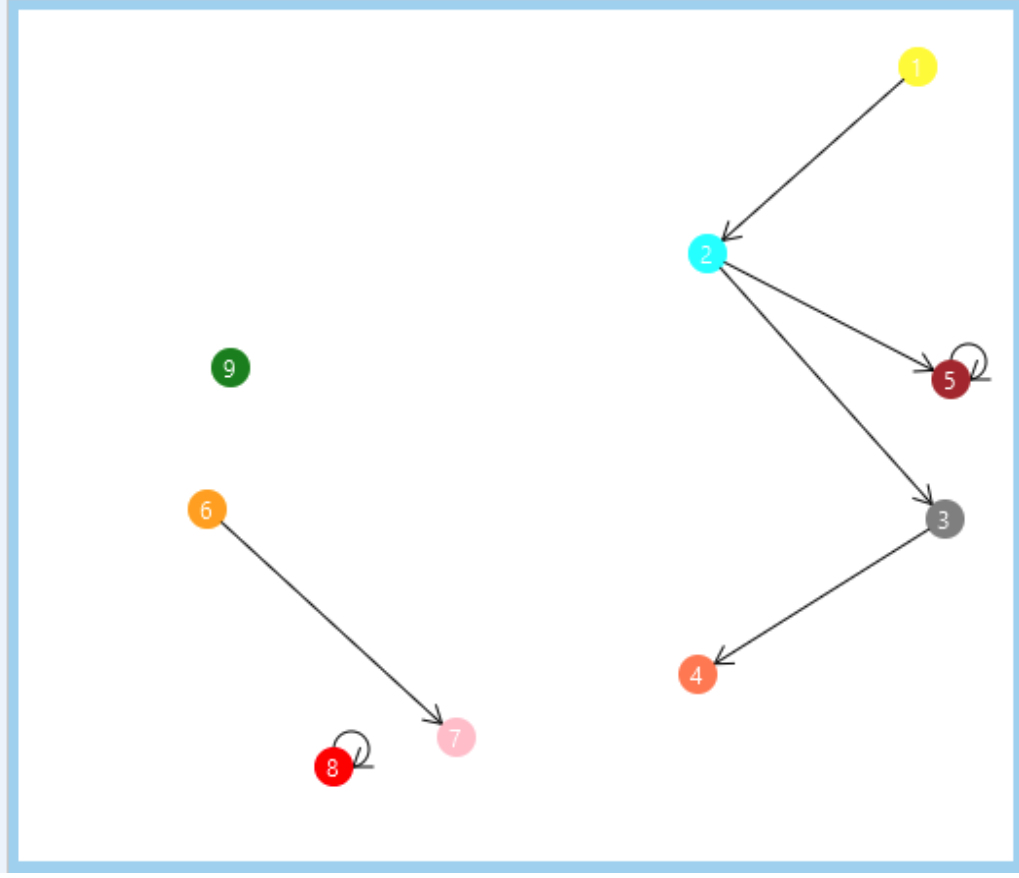
<p>1 2</p>	 <pre> graph TD 1((1)) --> 2((2)) </pre>
<p>1 2 4 1 3 4 2 3</p>	 <pre> graph TD 1((1)) --> 4((4)) 4 --> 2((2)) 2 --> 3((3)) 3 --> 1 </pre>

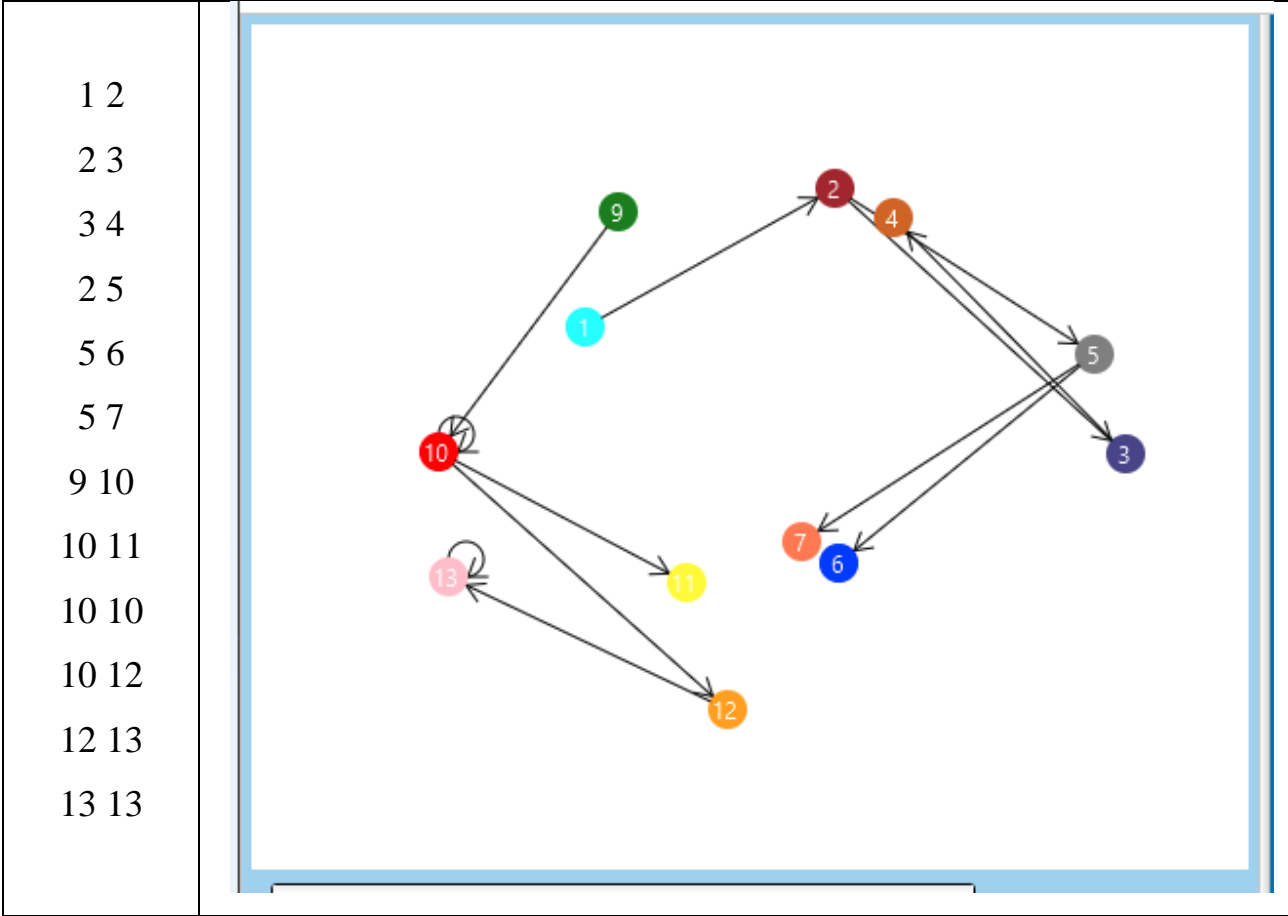
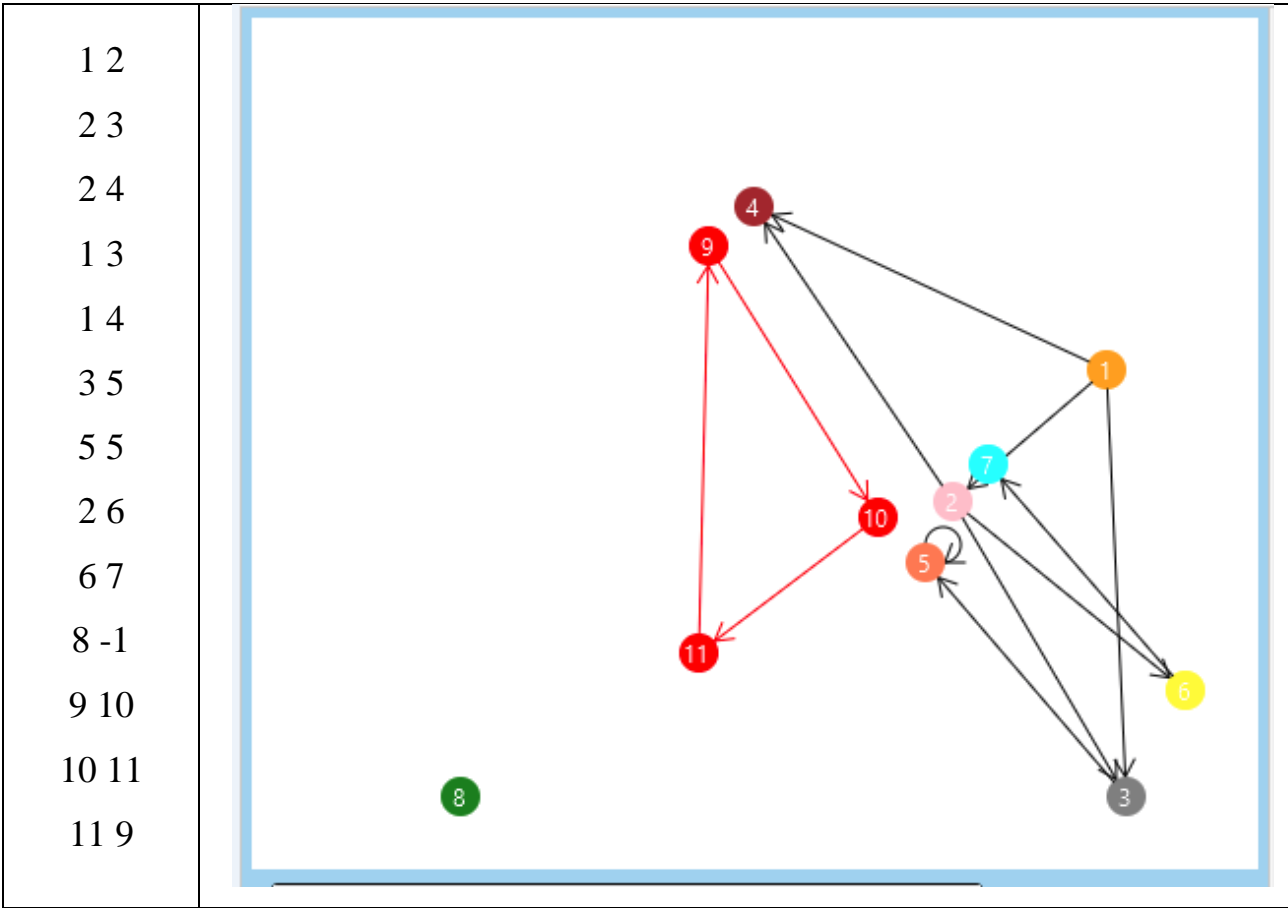
1 3	
3 2	
2 1	
4 3	
4 2	
7 2	
4 5	
5 4	
5 7	
7 8	
8 7	
6 5	
6 6	
6 8	
1 2	
3 1	
2 3	
2 4	
3 4	
4 5	
5 4	
6 5	
7 5	
8 6	
6 8	
8 7	
7 8	

1 -1	
7 -1 8 5 8 3 2 6 2 5 2 4 3 2 3 4 4 2 4 1 4 6 5 3	

<div data-bbox="331 302 386 347">1 2</div> <div data-bbox="331 369 386 414">2 3</div> <div data-bbox="331 436 386 481">3 4</div> <div data-bbox="331 504 386 548">4 5</div> <div data-bbox="331 571 386 616">5 1</div> <div data-bbox="331 638 386 683">5 5</div>	 <pre data-bbox="494 78 1500 909">graph TD; 1((1)) --> 2((2)); 2 --> 3((3)); 3 --> 4((4)); 4 --> 5((5)); 5 --> 1; 5 --> 5;</pre>
<div data-bbox="331 1187 386 1232">1 -1</div> <div data-bbox="331 1254 386 1299">2 -1</div> <div data-bbox="331 1321 386 1366">3 -1</div> <div data-bbox="331 1388 386 1433">4 -1</div> <div data-bbox="331 1456 386 1500">5 -1</div>	 <pre data-bbox="494 918 1500 1771">graph TD; 1((1)); 2((2)); 3((3)); 4((4)); 5((5));</pre>

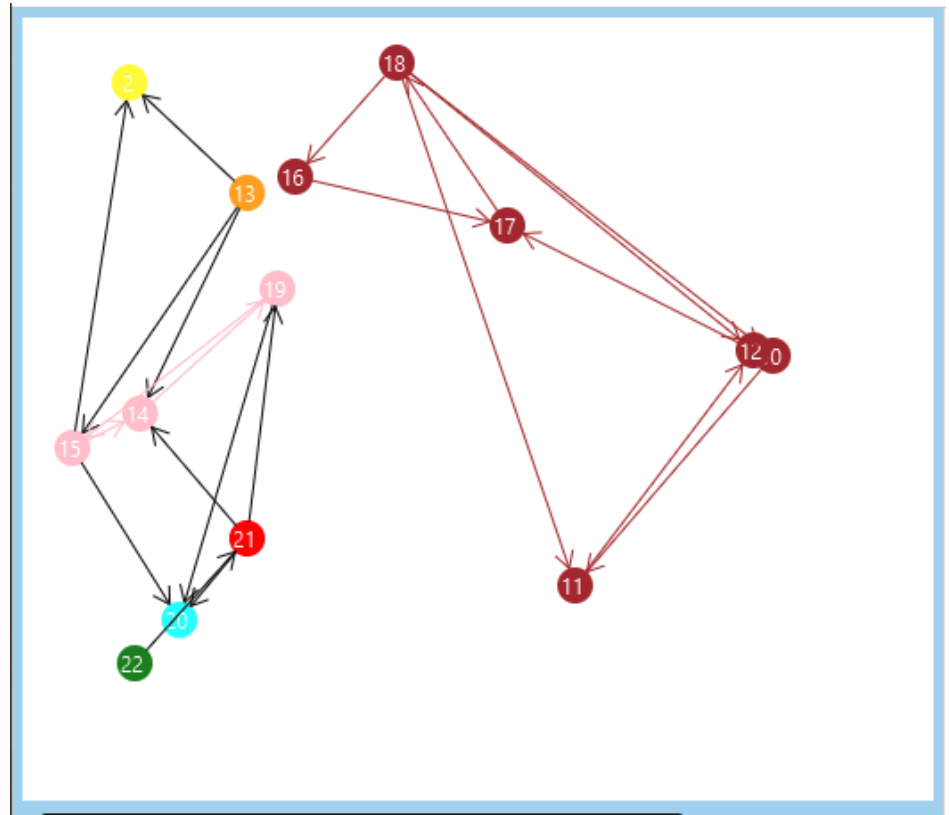
<p>1 3</p> <p>3 2</p> <p>2 1</p> <p>4 3</p> <p>4 2</p> <p>7 2</p> <p>4 5</p> <p>5 4</p> <p>5 7</p> <p>7 8</p> <p>8 7</p> <p>6 5</p> <p>6 6</p> <p>6 8</p>	
<p>1 -1</p> <p>1 1</p>	

<div data-bbox="311 403 359 638"><p>1 1</p><p>2 2</p><p>3 3</p><p>4 4</p></div>	<div data-bbox="502 89 1460 963"><p>A diagram showing four nodes, each with a self-loop. Node 1 is pink, node 2 is orange, node 3 is red, and node 4 is green. They are arranged in a square pattern within a light blue border.</p></div>
<div data-bbox="311 1176 359 1668"><p>1 2</p><p>2 3</p><p>2 5</p><p>5 5</p><p>3 4</p><p>6 7</p><p>8 8</p><p>9 -1</p></div>	<div data-bbox="478 985 1500 1859"><p>A diagram showing a directed graph with nodes 1 through 9. Node 1 (yellow) points to node 2 (cyan). Node 2 (cyan) points to nodes 3 (grey) and 5 (dark red). Node 3 (grey) points to node 4 (red). Node 6 (orange) points to node 7 (pink). Node 8 (red) has a self-loop. Node 9 (green) is isolated. The nodes are arranged in a light blue border.</p></div>



<div data-bbox="284 145 370 896"> <p>1 2</p> <p>2 3</p> <p>3 4</p> <p>4 5</p> <p>4 6</p> <p>3 7</p> <p>7 8</p> <p>7 9</p> <p>10 7</p> <p>9 9</p> <p>11 -1</p> <p>12 13</p> </div>	<div data-bbox="475 85 1481 958"> </div>
<div data-bbox="295 1064 359 1809"> <p>1 2</p> <p>2 6</p> <p>2 4</p> <p>3 5</p> <p>3 4</p> <p>4 7</p> <p>4 2</p> <p>5 9</p> <p>6 4</p> <p>6 7</p> <p>7 1</p> <p>9 8</p> </div>	<div data-bbox="475 992 1481 1888"> </div>

10 11
 11 12
 12 17
 13 14
 13 15
 13 2
 14 19
 15 2
 15 14
 15 20
 16 17
 17 18
 18 12
 18 16
 18 11
 18 10
 19 20
 19 15
 21 20
 21 19
 21 14
 22 21



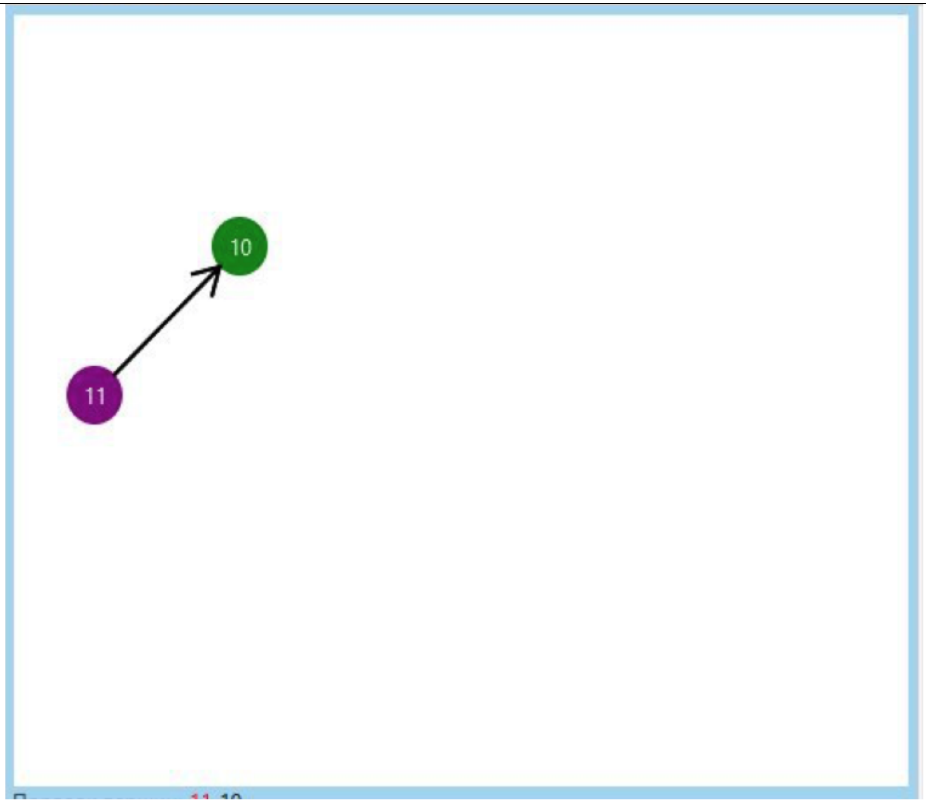
1 2

1 2

2 3

3 4

4 1



1 2

2 3

1 3

3 4

4 5

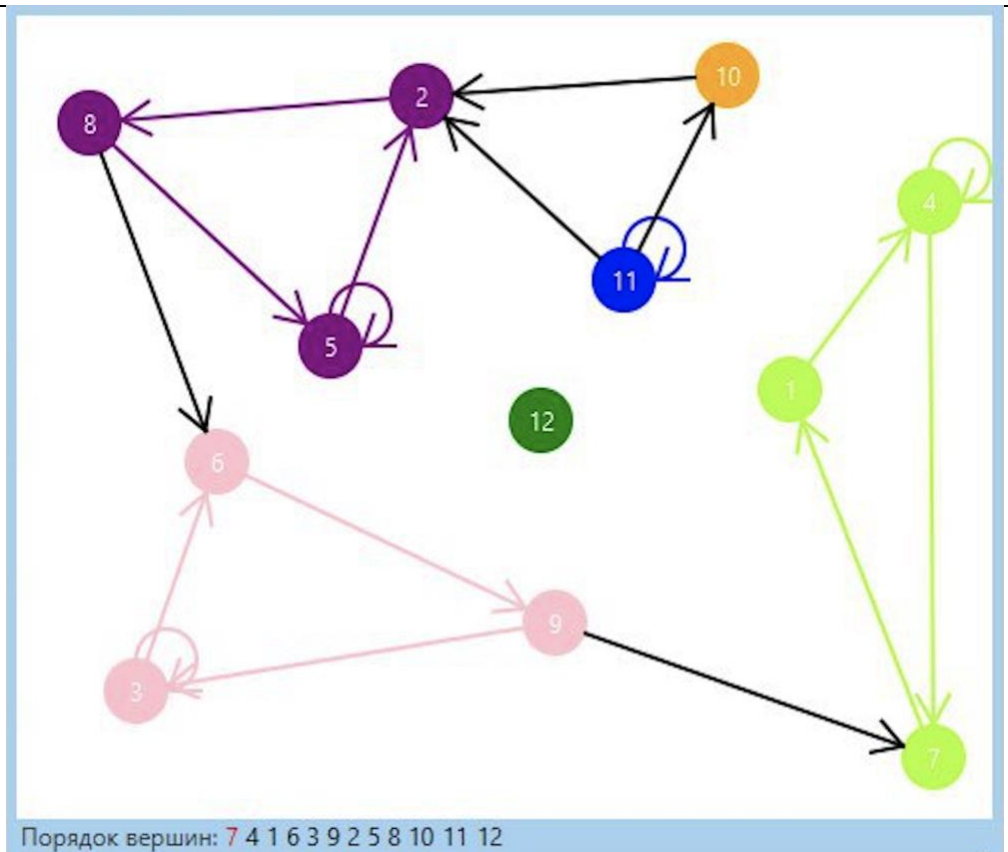
5 6

6 7

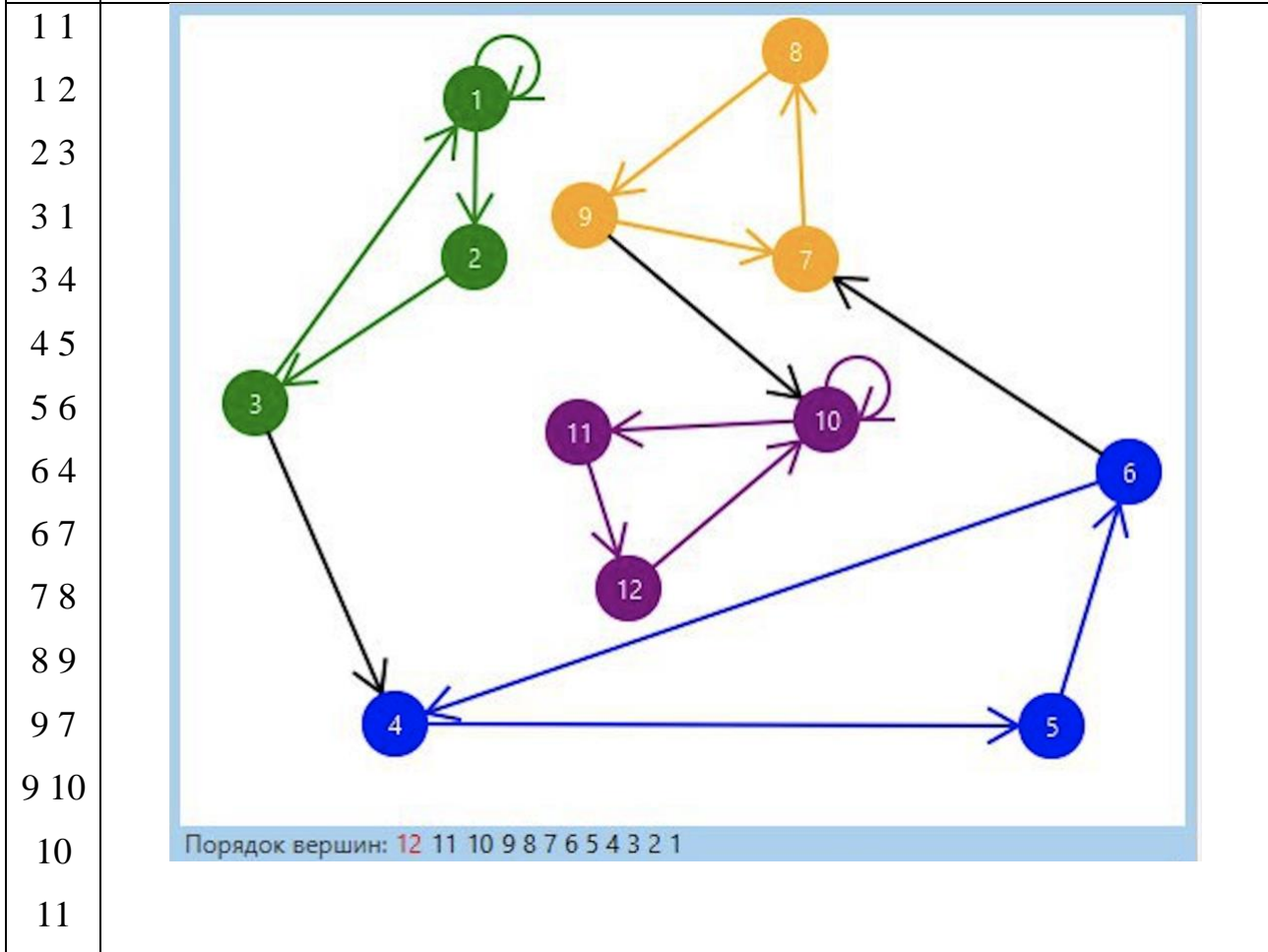
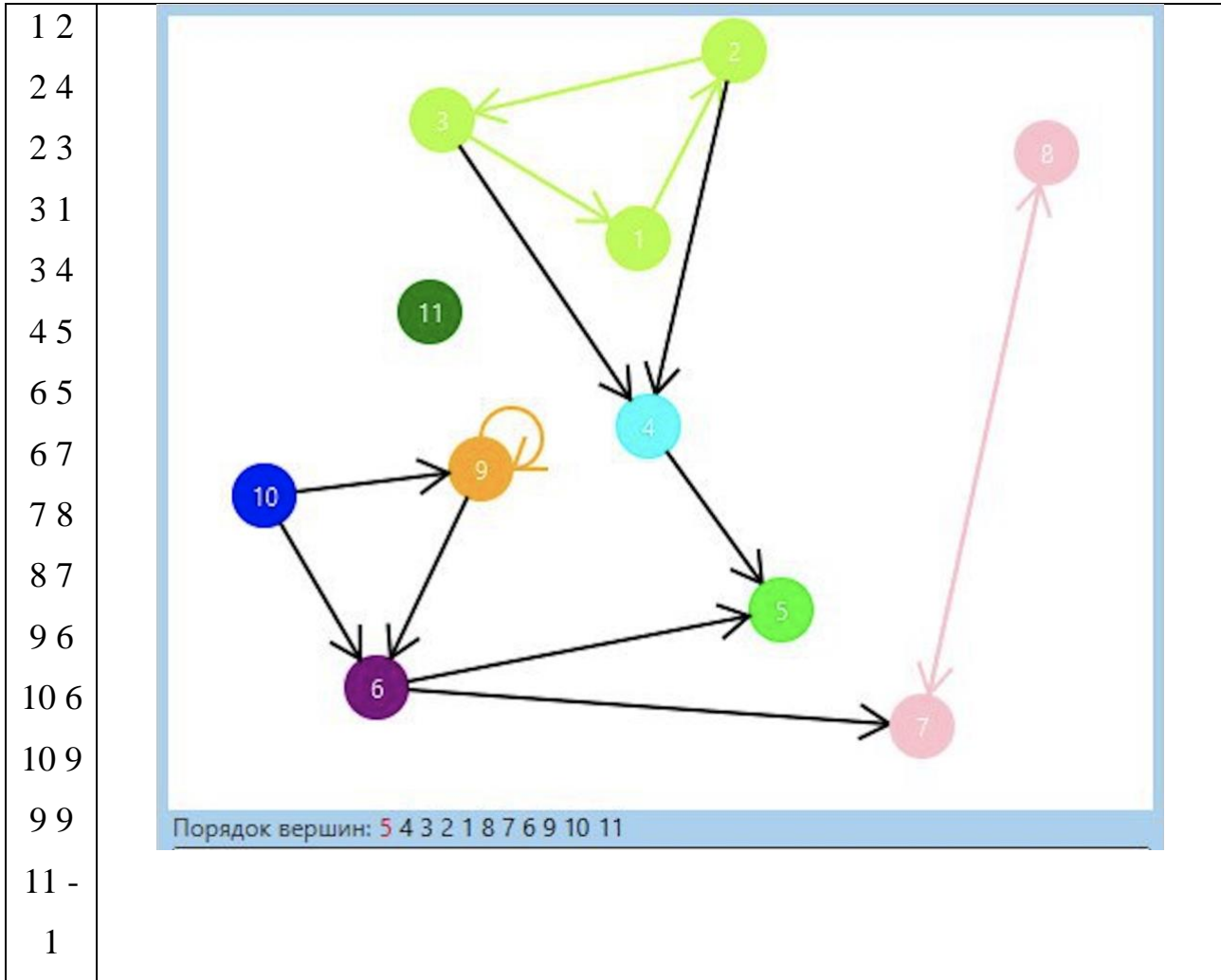
7 8

8 9

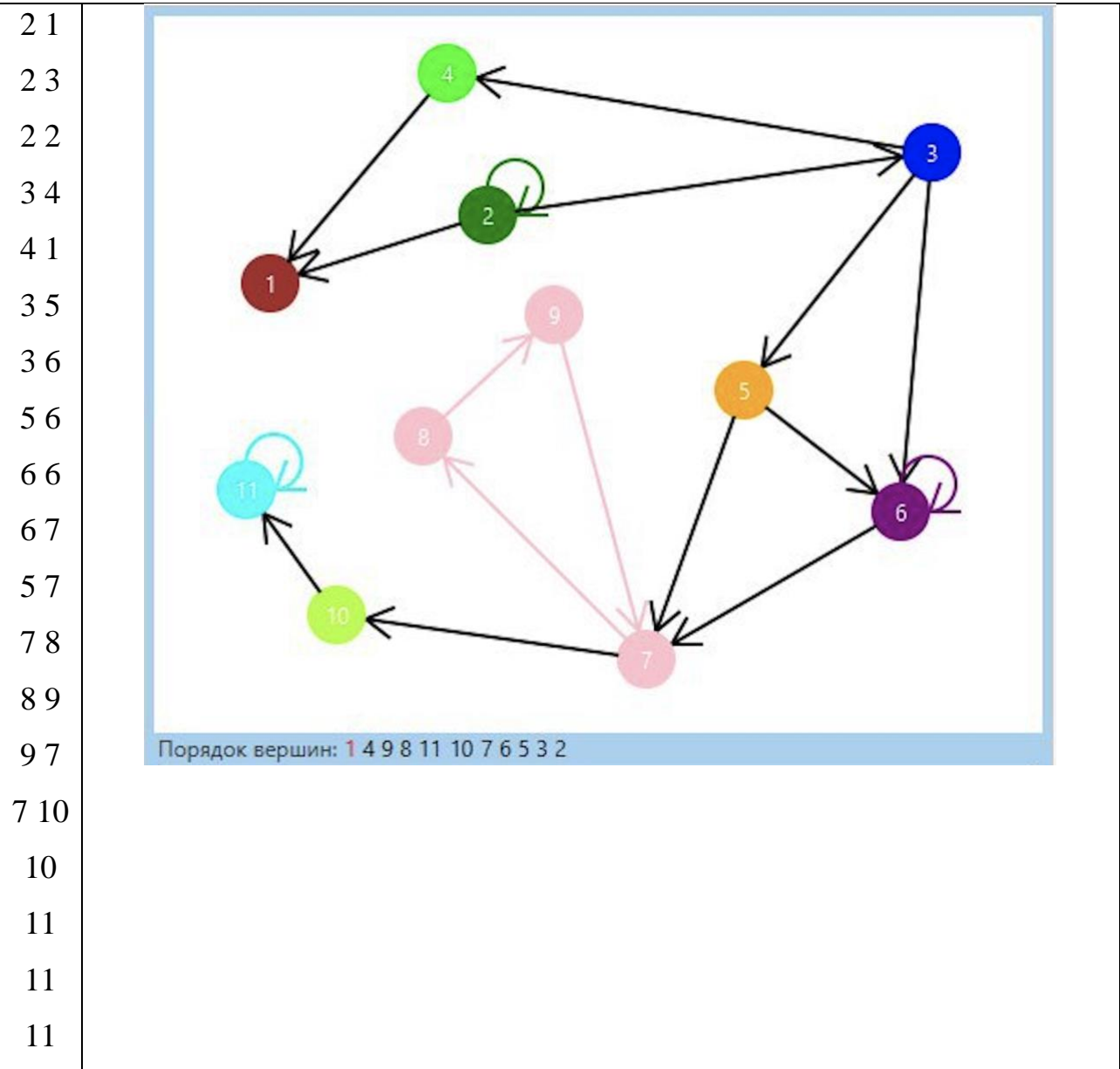
9 4

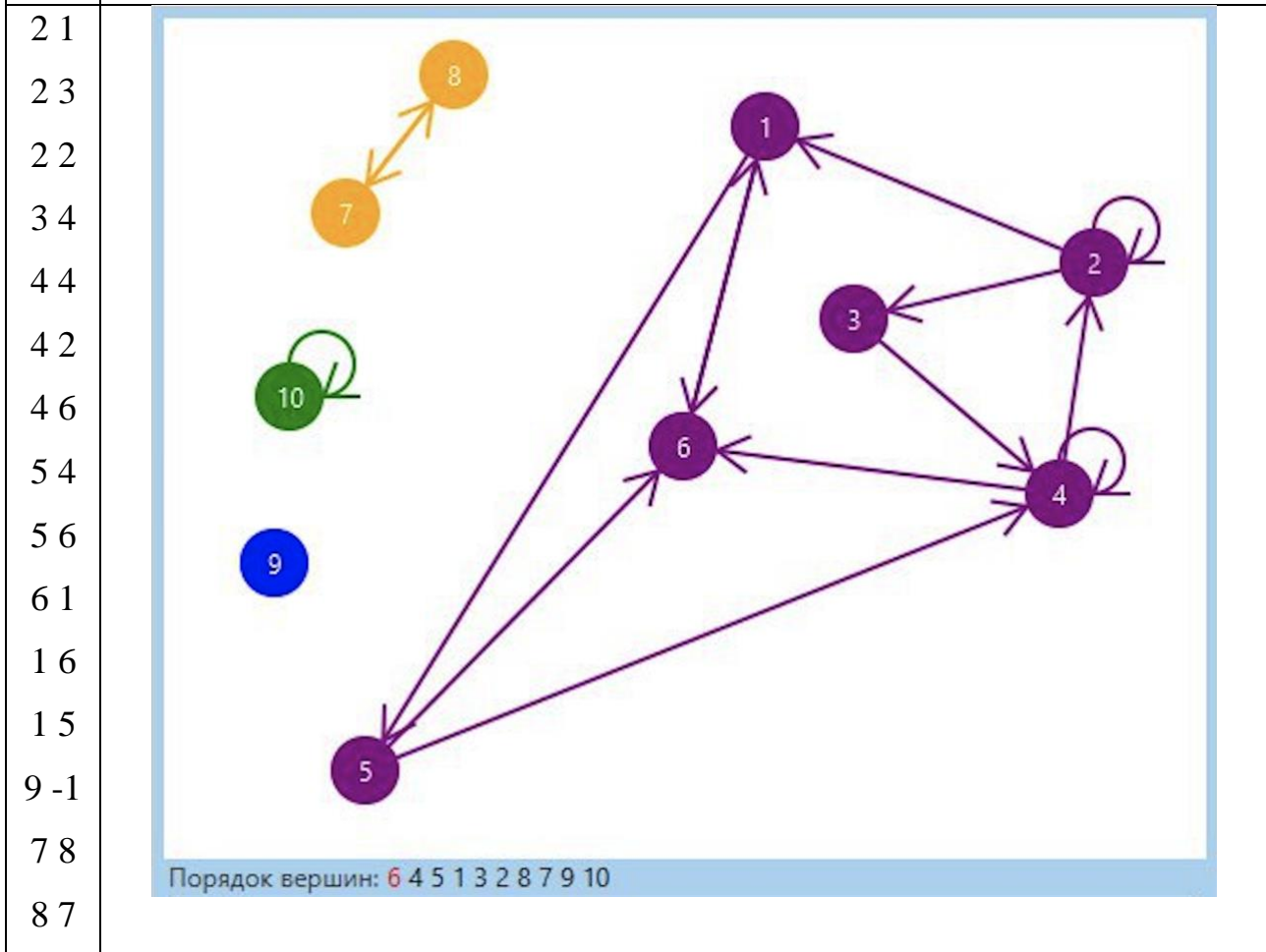
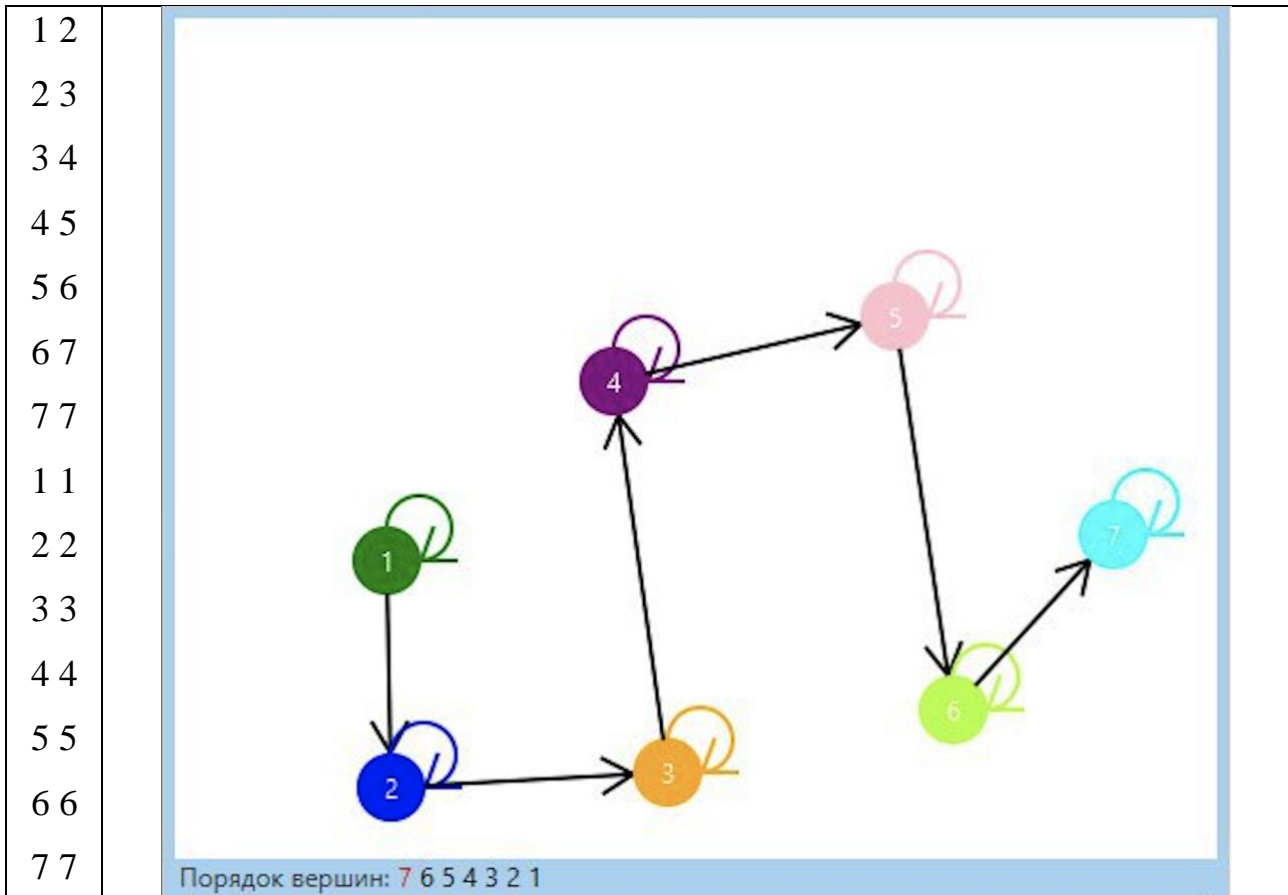


1 4	<p>Порядок вершин: 3 5 2 9 4 1 10 6 8 7 11</p>
2 1	
2 5	
4 3	
4 9	
5 2	
5 9	
6 5	
6 10	
7 8	
8 7	
8 3	
9 2	
10 6	
10 10	
7 7	<p>Порядок вершин: 9 8 7 6 5 4 3 2 1</p>
2 2	
11 -1	
1 2	
2 3	
1 3	
3 4	
4 5	
5 6	
6 7	
7 8	
8 9	
9 4	



11	
12	
12	
10	
10	
10	
1 2	<p>Порядок вершин: 3 2 1 6 7 5 4 10 9 8 12 11 13</p>
2 3	
3 1	
4 3	
4 5	
5 7	
7 6	
6 4	
7 4	
7 7	
3 3	
8 9	
9 10	
10 8	
11	
12	
12	
12	
13 -	
1	





10

10

11

12

13

14

15

16

17

18

81

82

83

84

85

86

87

88

21

22

23

24

25

26

27

28

31

32

33

34

35

36

37

38

41

42

43

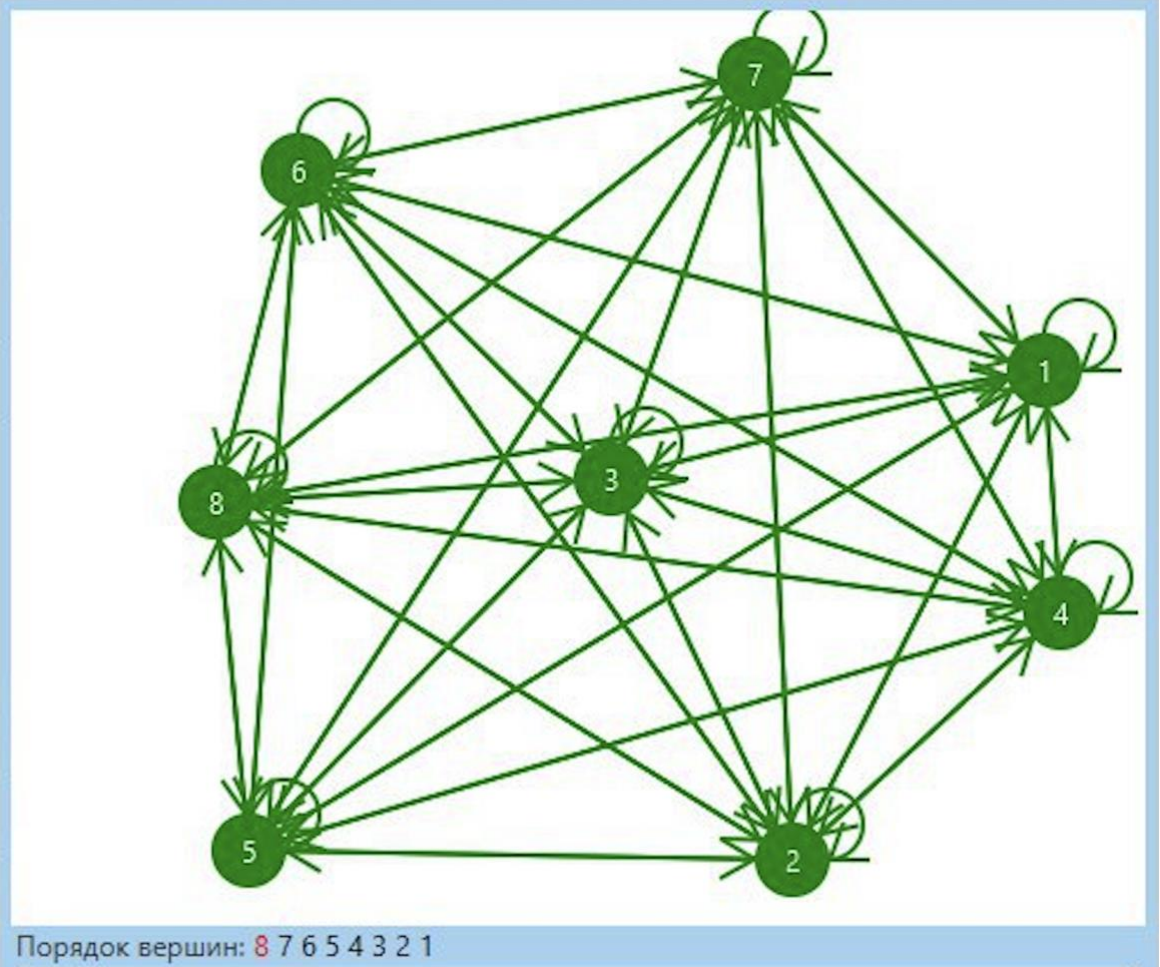
44

45

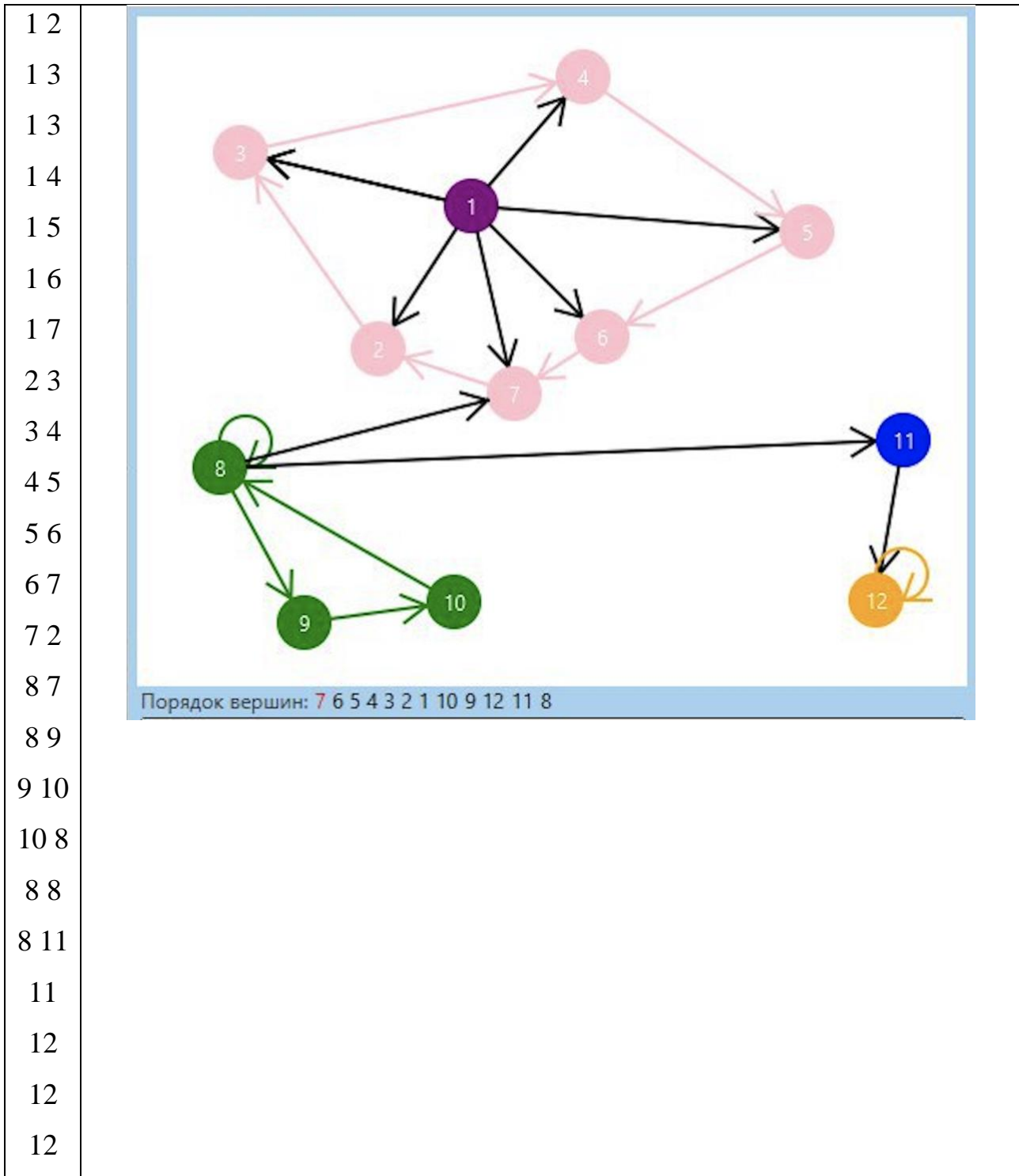
46

47

48



51	
52	
53	
54	
55	
56	
57	
58	
61	
62	
63	
64	
65	
66	
67	
68	
71	
72	
73	
74	
75	
76	
77	
78	



4.3. Пошаговое тестирование алгоритма.

Входные данные:

1 2

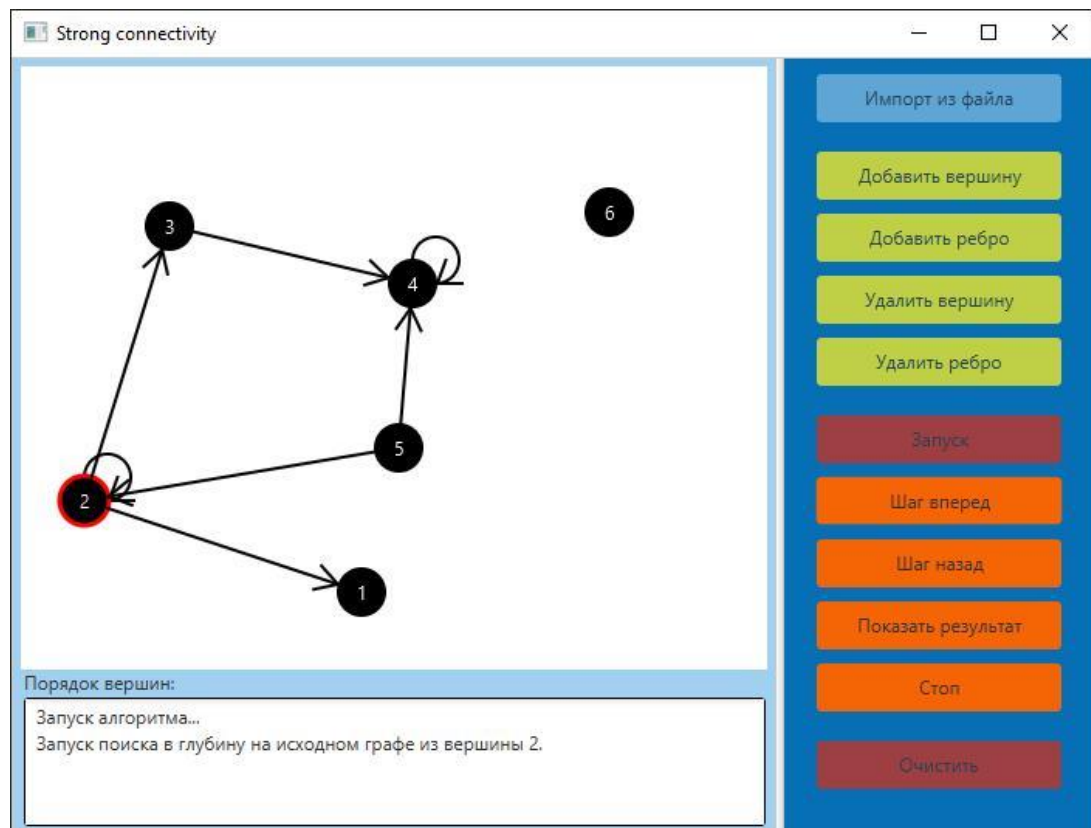
2 4

4 1

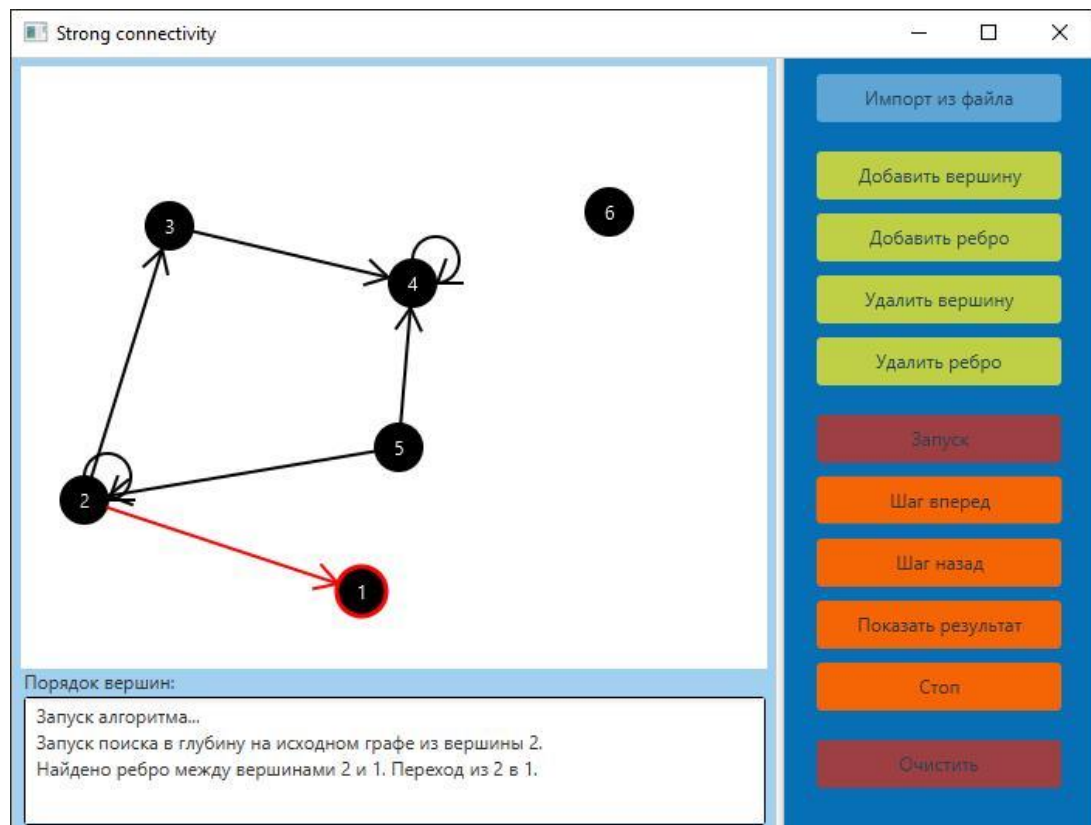
3 3

Выходные данные:

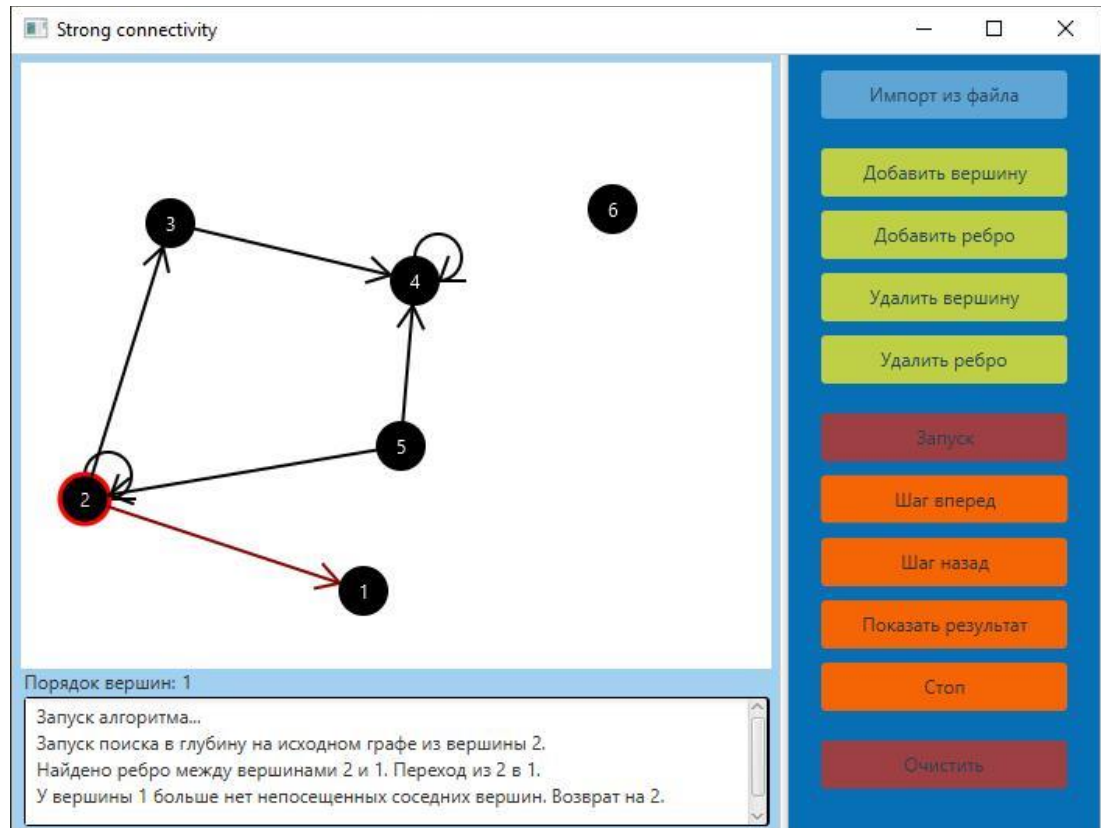
Шаг 1.



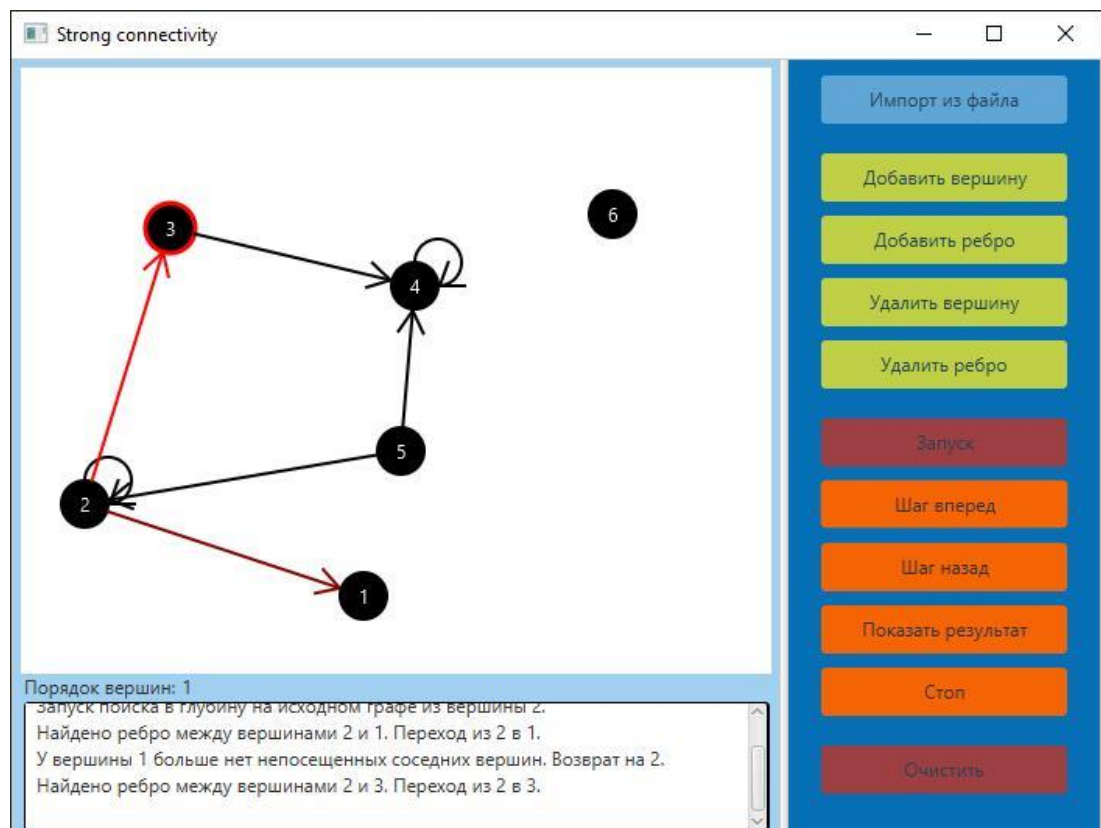
Шаг 2.



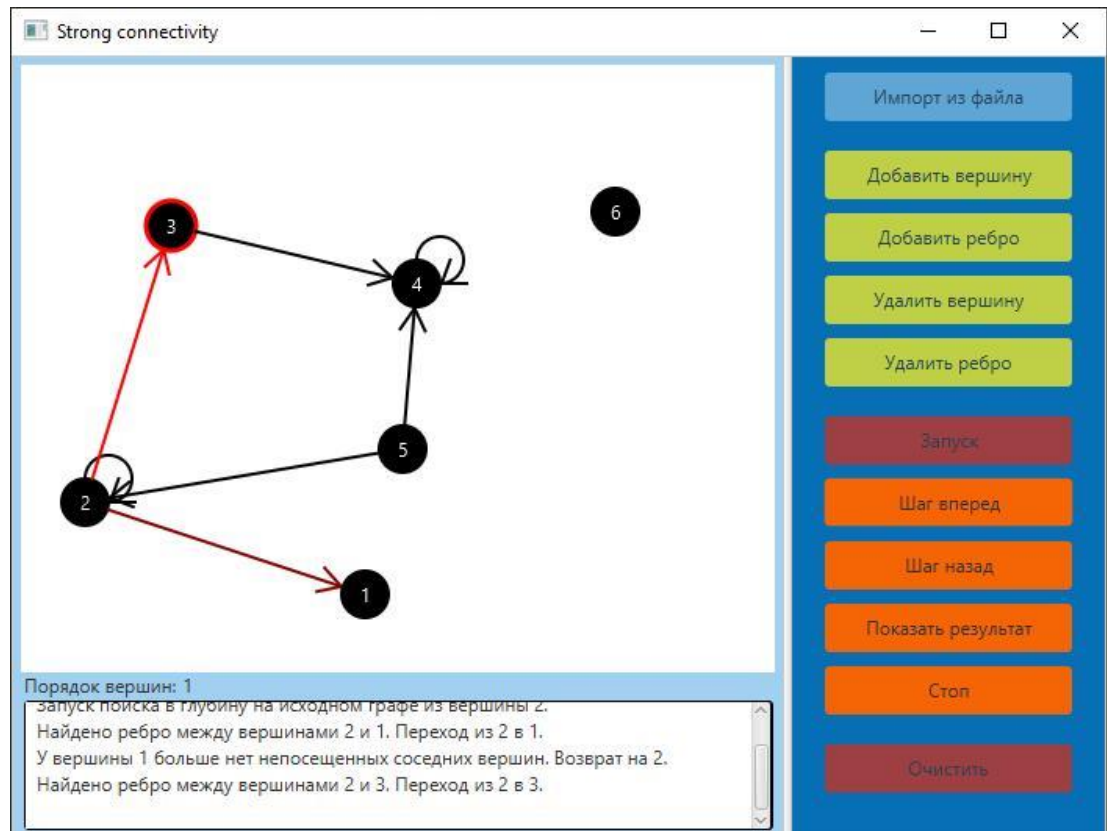
Шаг 3.



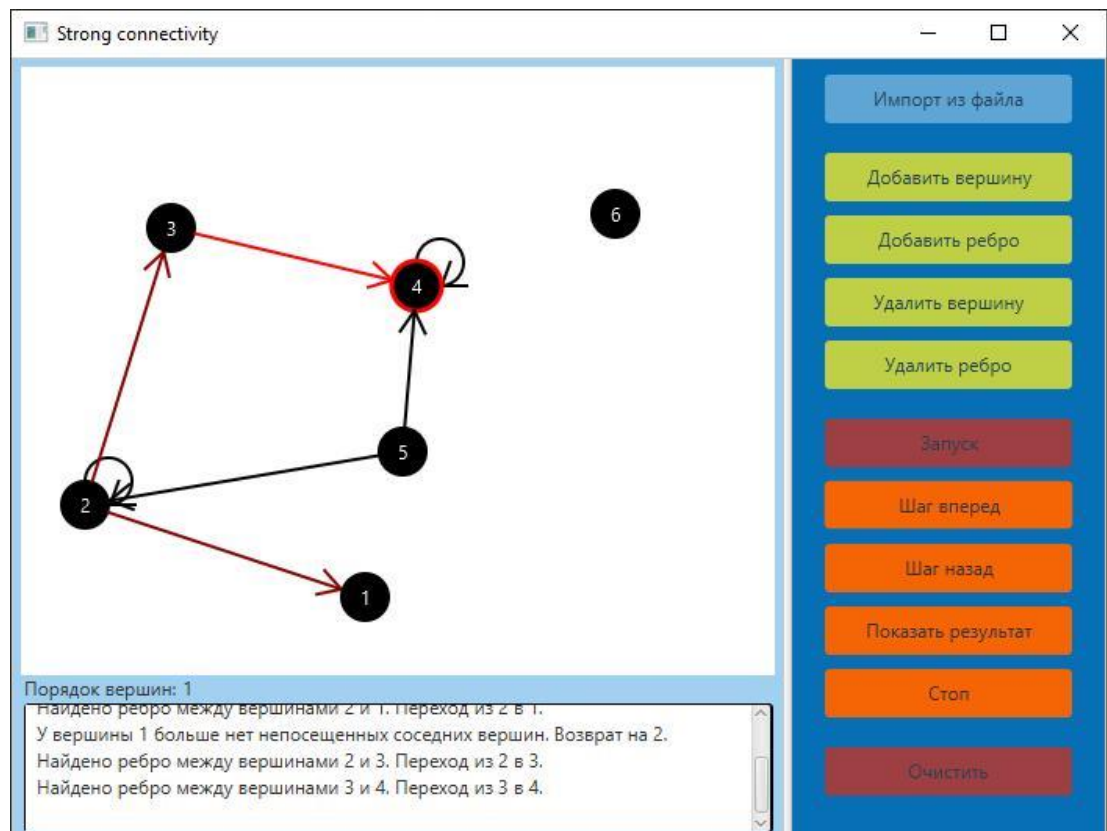
Шаг 4.



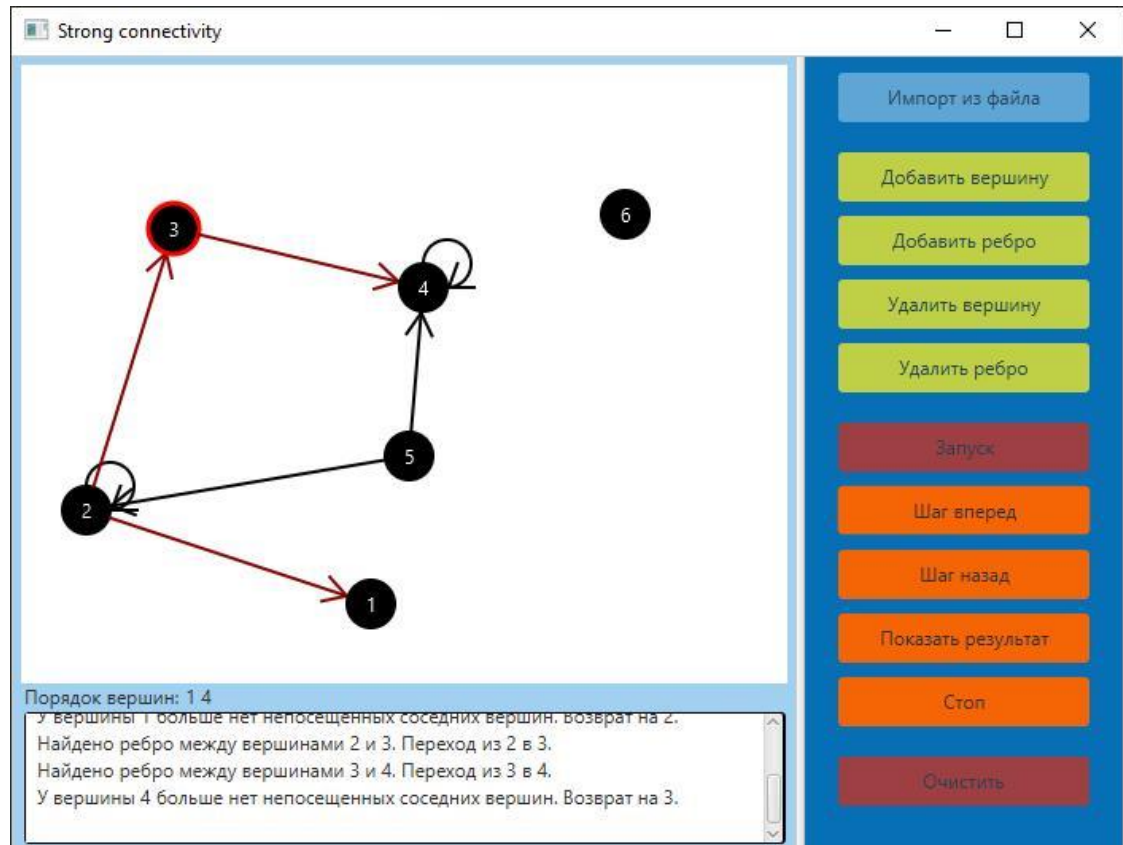
Шаг 5.



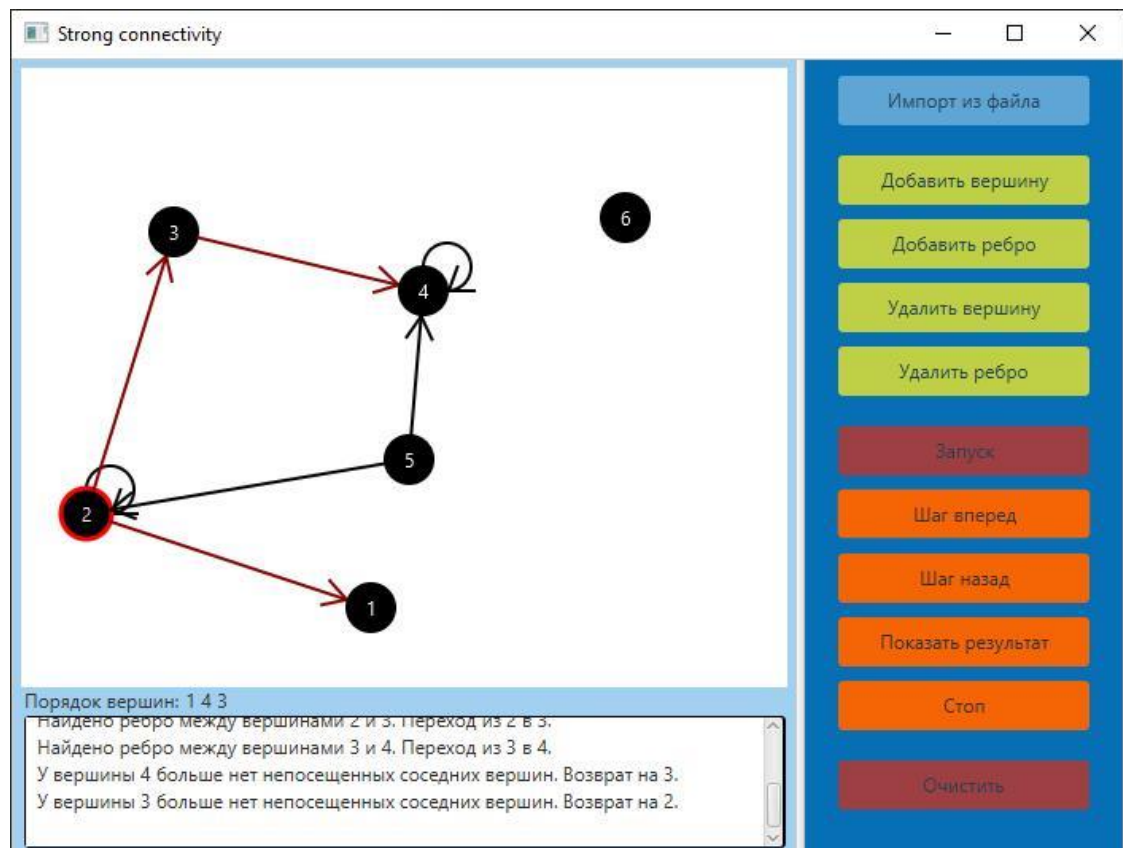
Шаг 6.



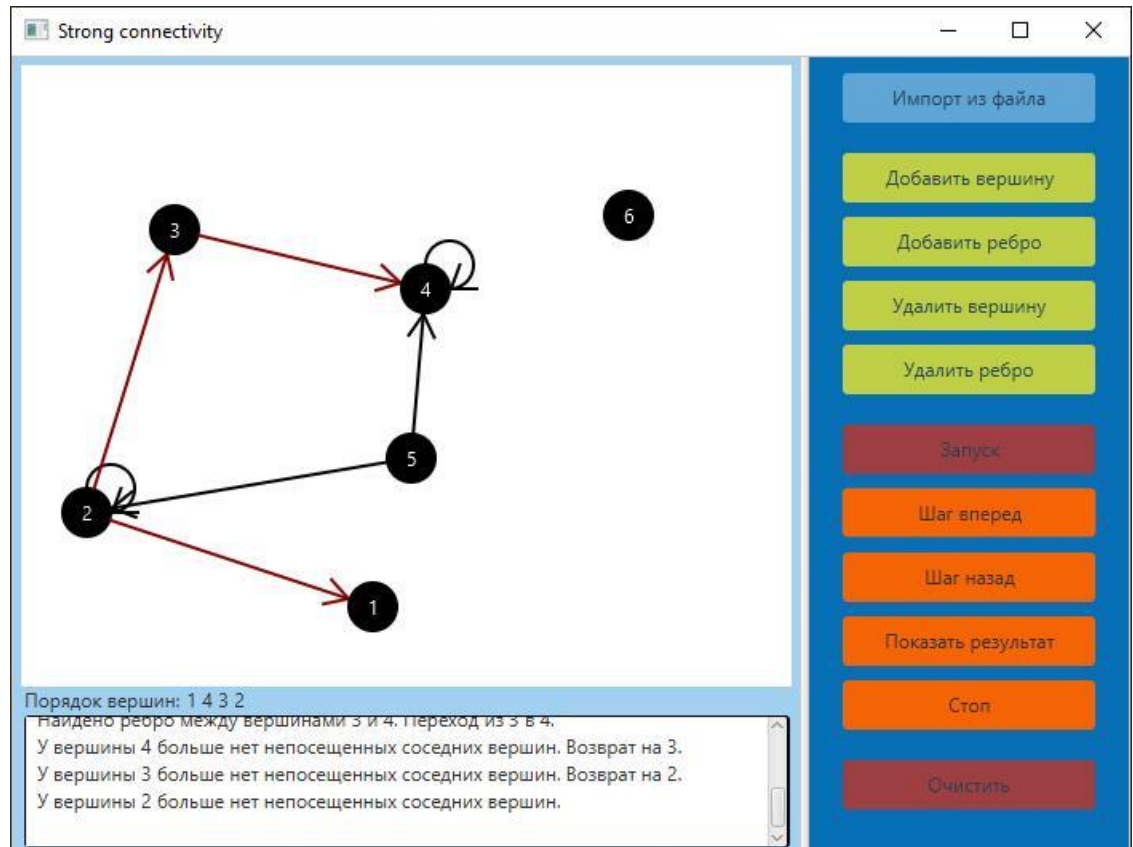
Шаг 7.



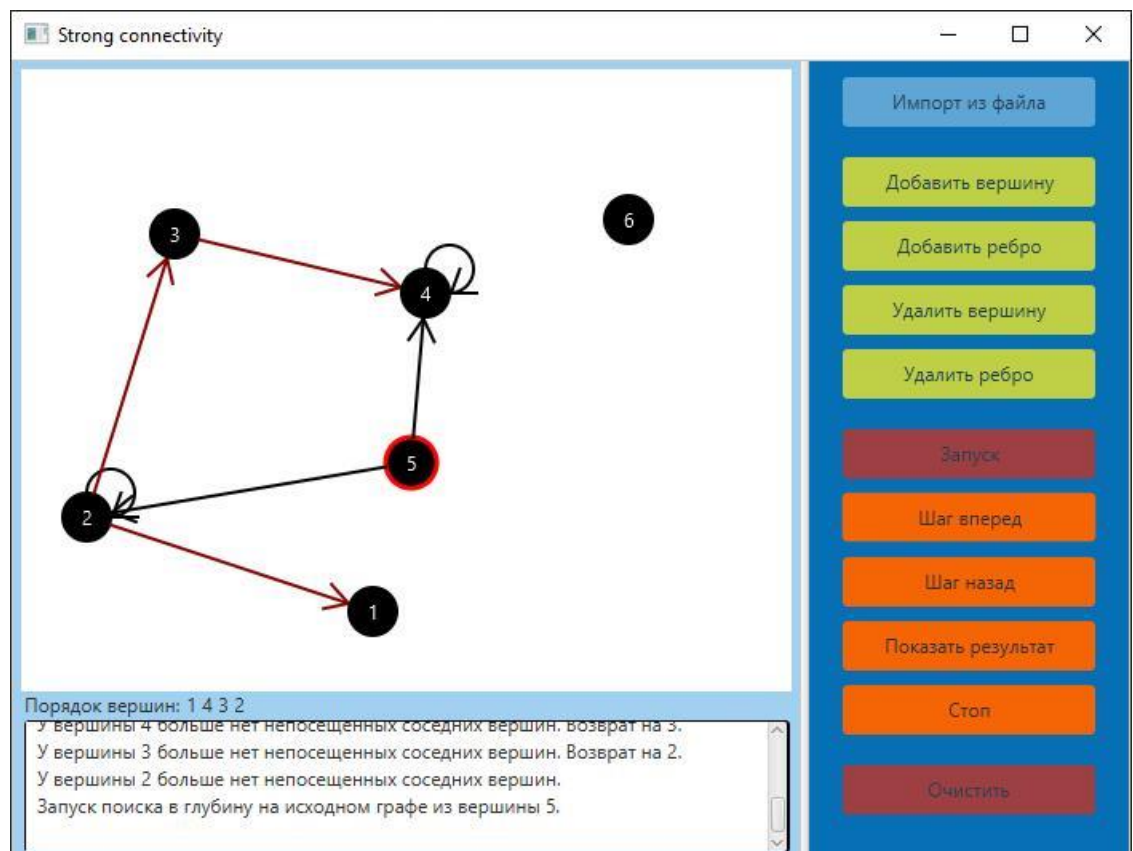
Шаг 8.



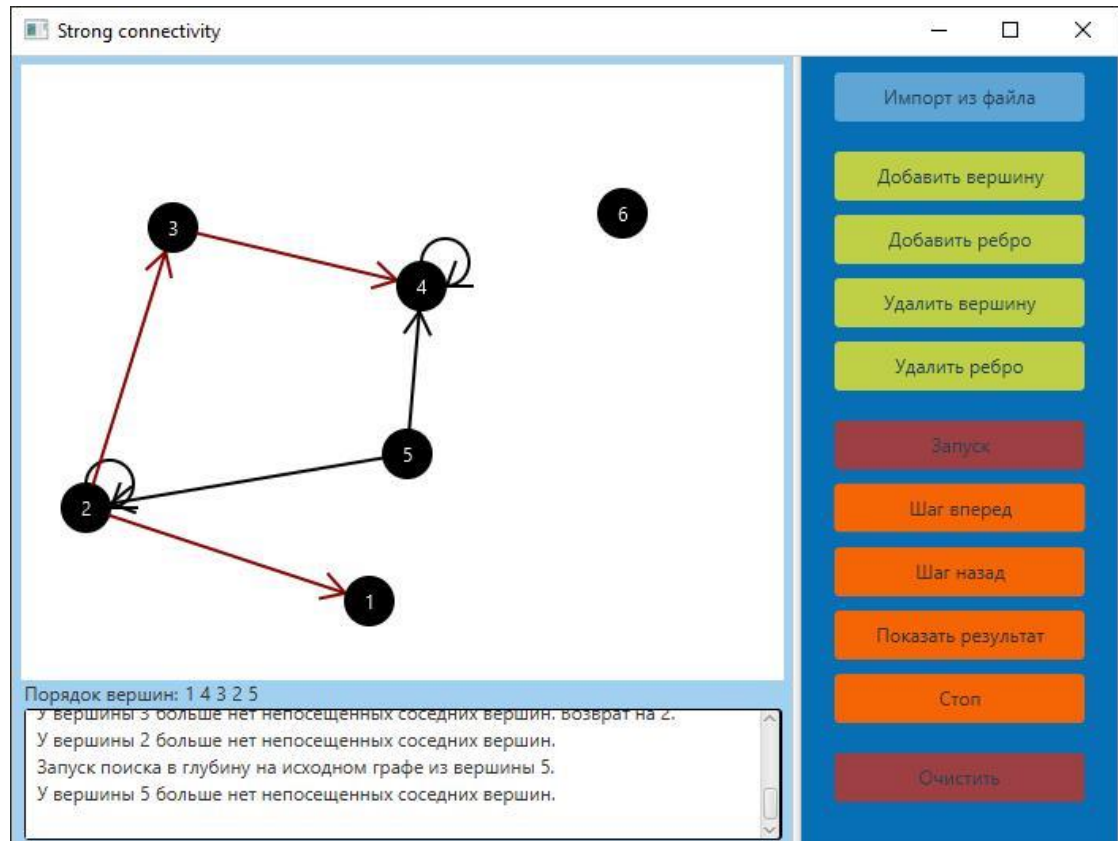
Шаг 9.



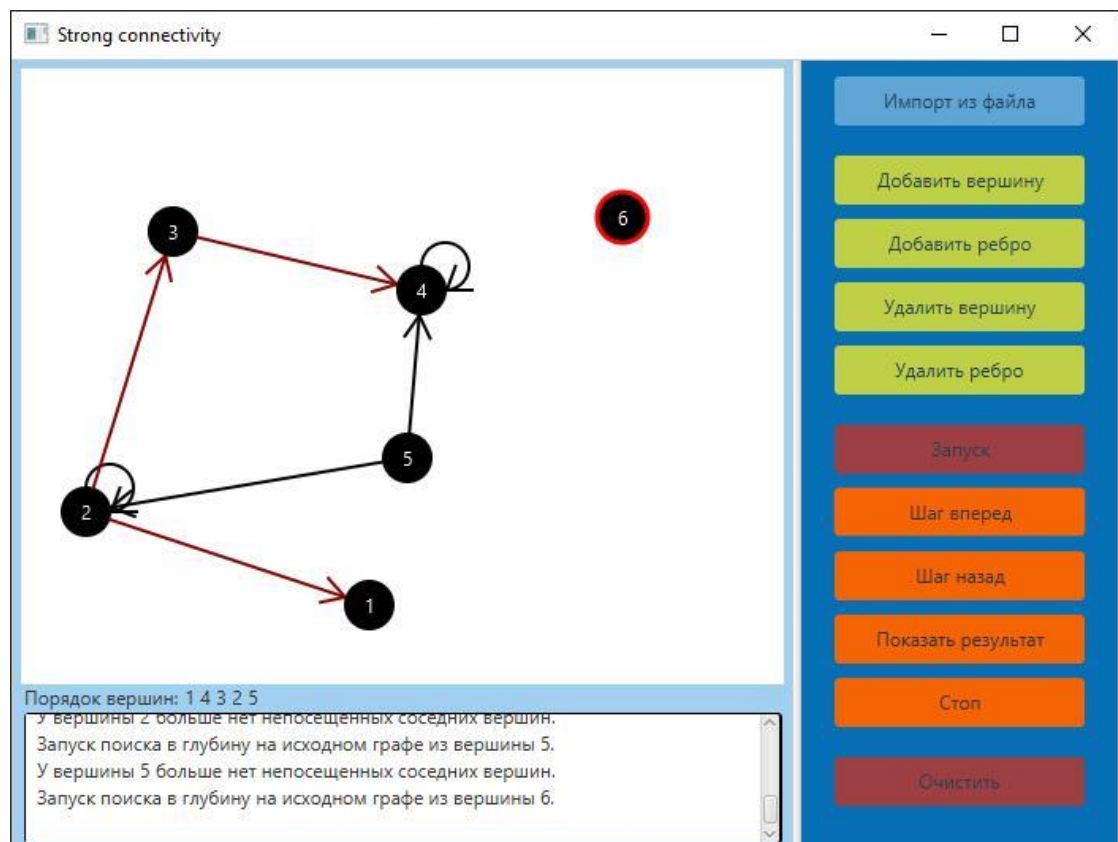
Шаг 10.



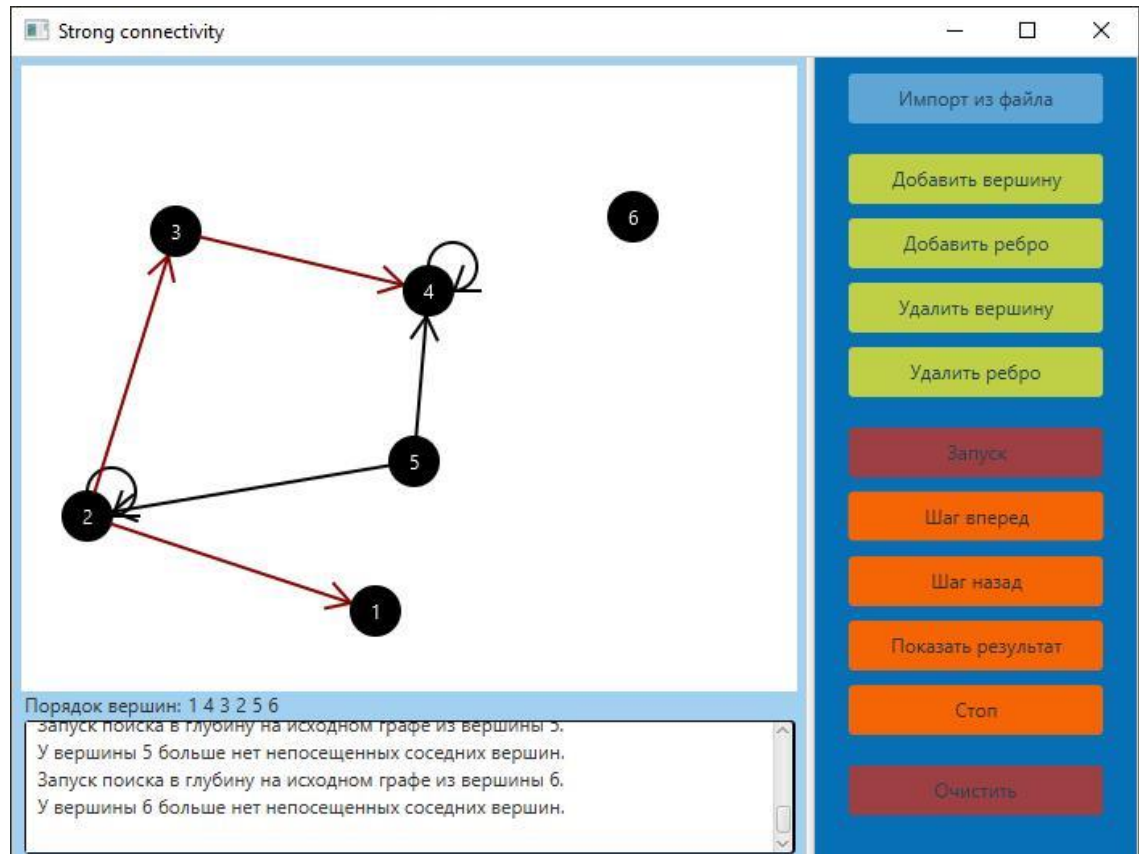
Шаг 11.



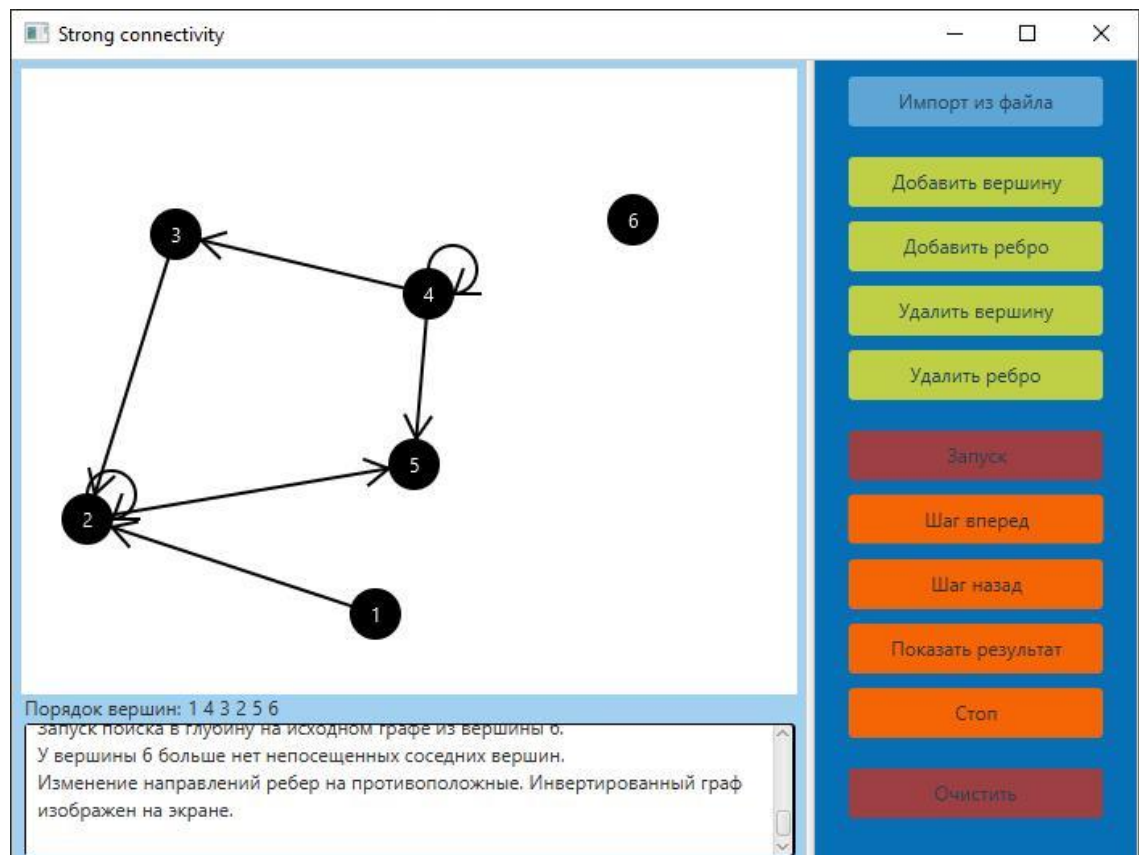
Шаг 12.



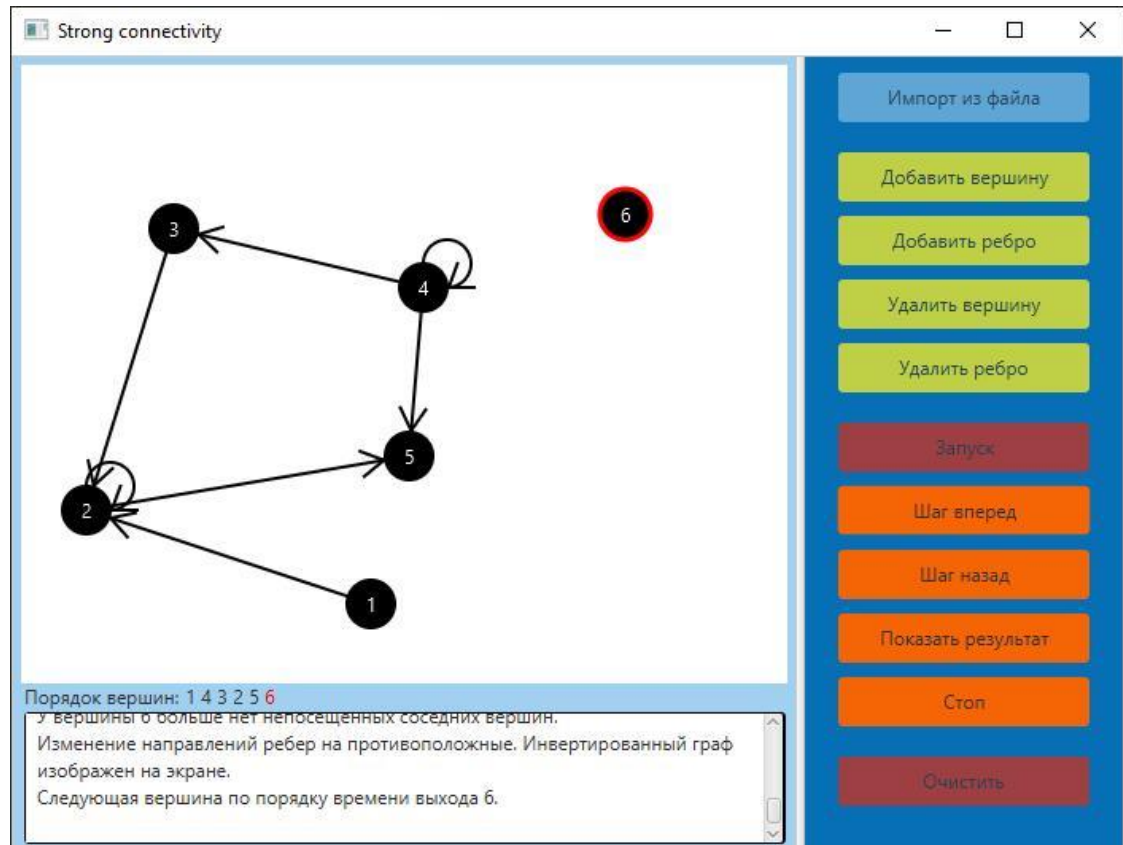
Шаг 13.



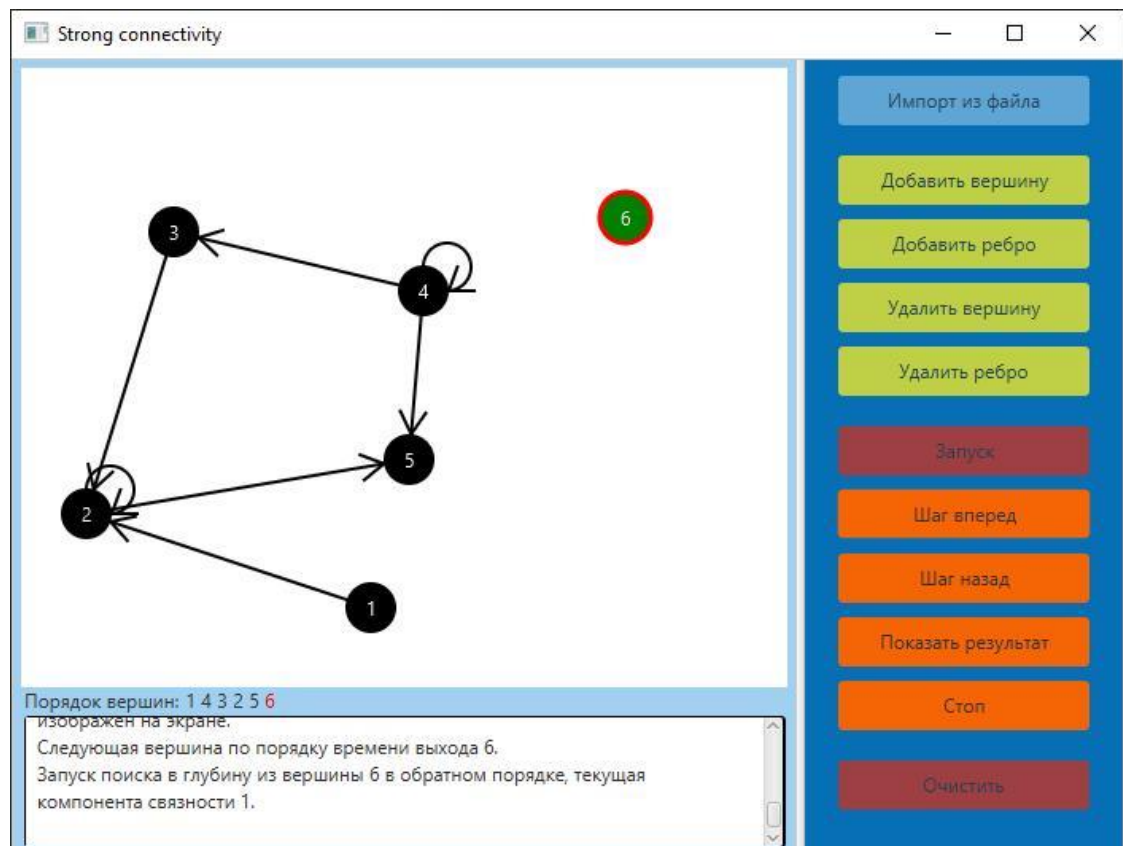
Шаг 14.



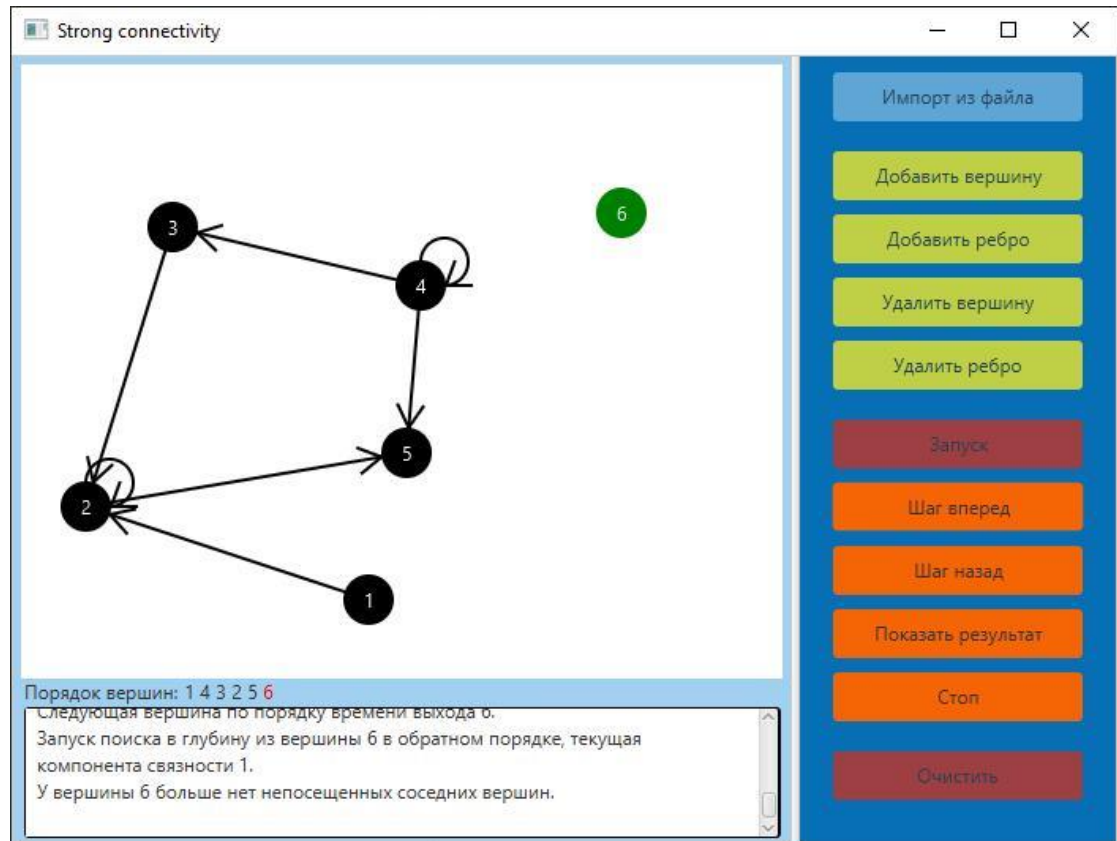
Шаг 15.



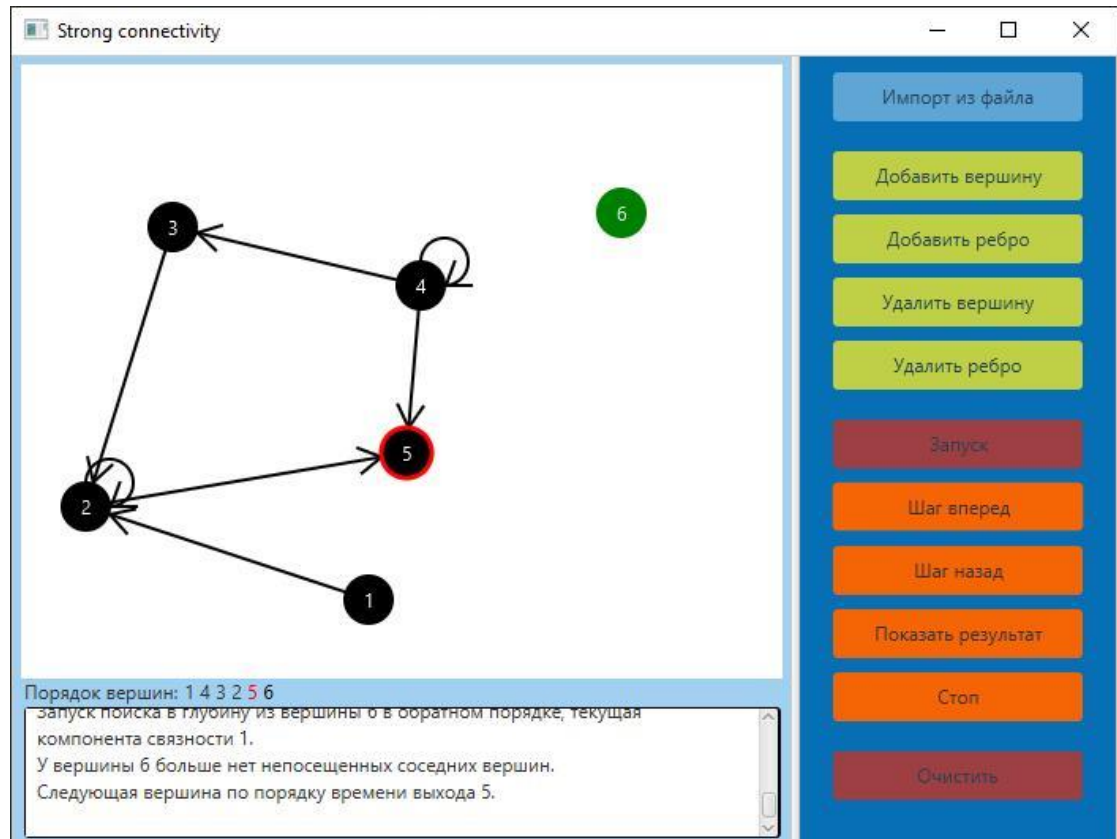
Шаг 16.



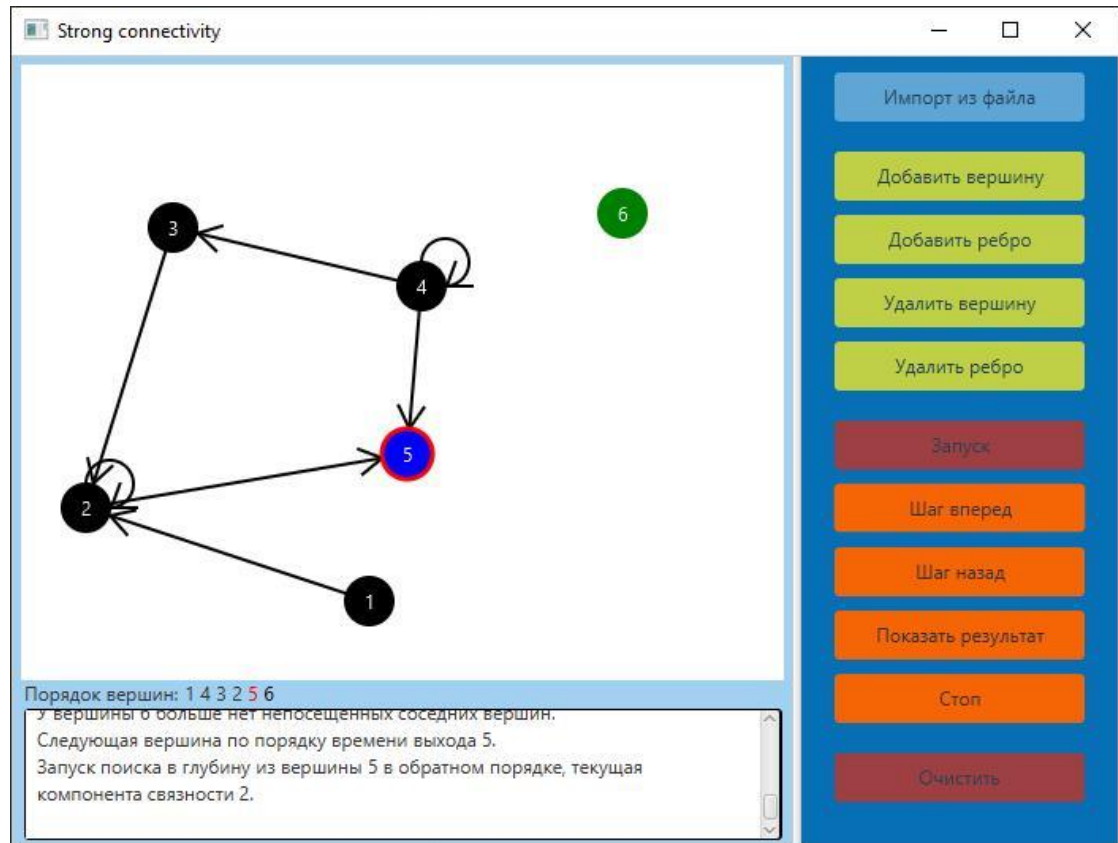
Шаг 17.



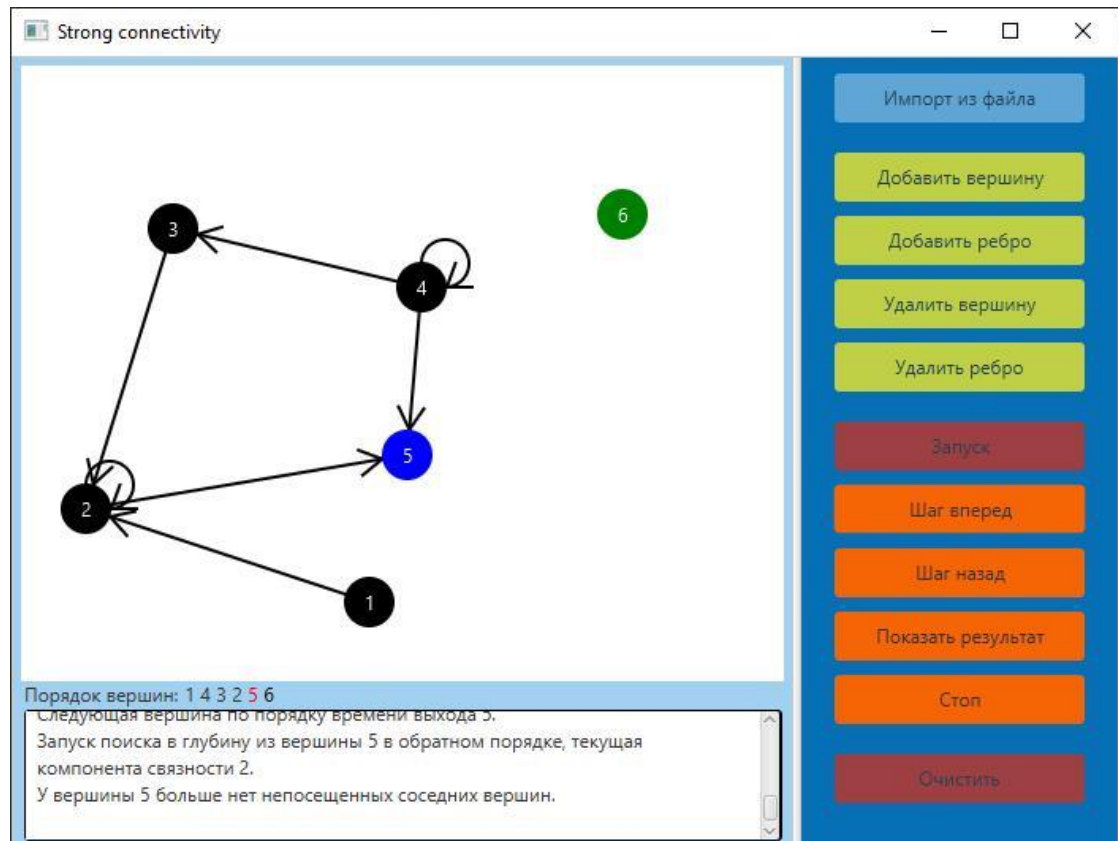
Шаг 18.



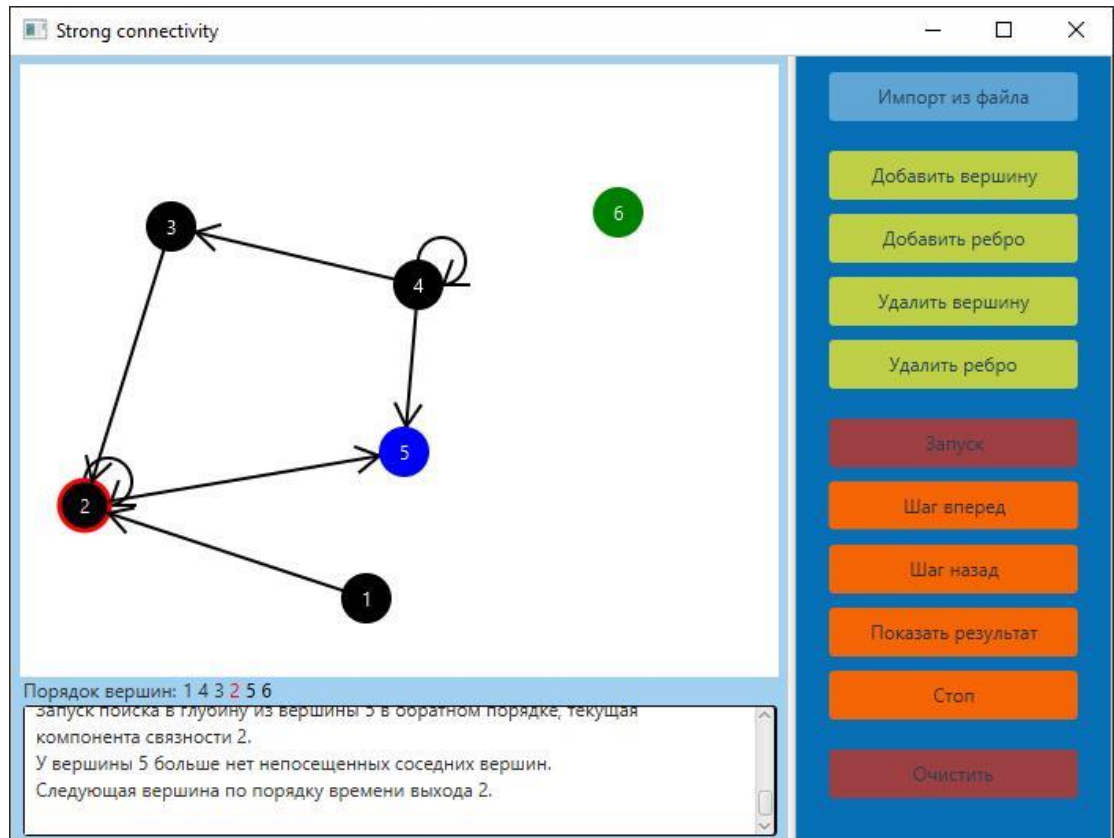
Шаг 19.



Шаг 20.



Шаг 21.



ЗАКЛЮЧЕНИЕ

В результате выполнения практической работы было реализовано приложение, на языке Java, предназначенное для визуализации алгоритма Косарайю - поиска компонент сильной связности, на любом ориентированном графе, вводимым пользователем вручную или посредством импорта из файла.

Программа позволяет либо сразу получить результат, либо запустить пошаговую демонстрацию применения алгоритма к орграфу. Кроме того, программа предоставляет набор дополнительных возможностей для пользователя: перемещение вершин по экрану, возможность остановки пошаговой демонстрации в любой момент работы алгоритма с последующим редактированием графа, наличие не только кнопки “шаг вперед”, но и “шаг назад”, возможность многократного использования приложения для разных графов без перезапуска программы.

Также визуализация алгоритма сопровождается текстовыми пояснениями для пользователя, в ходе пошаговой демонстрации выделяются обрабатываемые и обработанные ребра.

В процессе разработки проекта был получен опыт работы в команде, были улучшены навыки программирования на языке Java.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Эккель Брюс. Философия Java. Издательство Питер, 1168 с.
2. Герберт Шилдт. Java 8. Руководство для начинающих. Издательство Вильямс, 720 с.
3. Дасгупта С.6 Пападимитриу Х., Вазирани У. Алгоритмы. Издательство МЦНМО, 320 с.
4. Кен Арнольд, Джеймс Гослинг. Язык программирования Java. Издательство Питер, 304 с.
5. Машнин Т. JavaFX 2.0 Разработка RIA-приложений. Издательство БХВ-Путурбург, 320 с.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

```
//AlgorithmKosarayu.java
package com.Algorithm;
import java.util.ArrayList;
import javafx.util.Pair;

public class AlgorithmKosarayu{

    private ArrayList<Vertex> graph = new ArrayList<Vertex>();
//граф
    private ArrayList<Pair<Integer, Integer>> timeOut;
//время выхода
    private ArrayList<Event> events;
//список событий для пошаговой визуализации
    private ArrayList<Pair<Integer, Integer>> listEdge;
//список ребер

    private int dfsTimer = 1;
//таймер для определения времени
    private int component = 1;

    public AlgorithmKosarayu() {
        listEdge = new ArrayList<Pair<Integer, Integer>>();
        timeOut = new ArrayList<Pair<Integer, Integer>>();
        events = new ArrayList<Event>();
    }

    public ArrayList<Event> start(ArrayList<Pair<Integer,
Integer>> listEdge) {

        //вызов методов, в последовательности задаваемой
алгоритмом
        this.listEdge = listEdge;
    }
}
```

```

        initGraph();
//инициализация графа, заданного списком ребер

        for (int i = 0; i < graph.size(); i++) {
//запуск поиска в глубину с фиксированием времени выхода

            if (graph.get(i).getColor() == -1) {

                events.add(new Event(1));
                events.get(events.size() -
1).setNameVertex(graph.get(i).getNAME());
                events.get(events.size() - 1).setTextHints("Запуск
поиска в глубину на исходном графе из вершины "
                    + graph.get(i).getNAME() + ".\n");
                //событие: начало дфс - с какой вершины

                dfs1(i);
            }
        }

        this.listEdge = inverting();
//инвертирование инвертированного списка рёбер

        events.add(new Event(5));
        events.get(events.size() -
1).setInventedListEdge(this.listEdge);
        events.get(events.size() - 1).setTextHints("Изменение
направлений ребер на противоположные. " +
            "Инвертированный граф изображен на экране.\n");
        //событие: инвертирование графа

        graph.clear();
        initGraph();
//инвертированный граф

        for (int i = timeOut.size() - 1; i >= 0; i--) {
//идем по убыванию времени выхода

```

```

        int index = find(timeOut.get(i).getKey());
//индекс вершины в графе
        events.add(new Event(10));
        events.get(events.size() -
1).setNameVertex(graph.get(index).getName());
        events.get(events.size() - 1).setTextHints("Следующая
вершина по порядку времени выхода " +
            graph.get(index).getName() + ".\n");
        if (graph.get(index).getColor() == -1) {
//если не посещена ранее
            graph.get(index).setComponent(component++);
//фиксируем принадлежность к компоненте

            events.add(new Event(6));
            events.get(events.size() -
1).setTransition(graph.get(index).getName(),
graph.get(index).getComponent());
            events.get(events.size() - 1).setTextHints("Запуск
поиска в глубину из вершины " + graph.get(index).getName() +
                " в обратном порядке, текущая компонента
связности " + graph.get(index).getComponent() + ".\n");
            //событие: начало dfs - вершина, компонента

            dfs2(index);
        }
    }

    this.listEdge = inverting();
//инвертированный список ребер

    events.add(new Event(5));
    events.get(events.size() -
1).setInventedListEdge(this.listEdge);
    events.get(events.size() - 1).setTextHints("Изменение
направлений ребер на противоположные. " +
        "Инвертированный граф изображен на экране.\n");
//событие: инвертирование графа

```

```

        return events;
    }

    private void initGraph() {
        //принимает список ребер и преобразовывает в списки
смежности вершин

        for(int i = 0; i < listEdge.size(); i++) {

            Vertex temp1 = new Vertex(listEdge.get(i).getKey());
//добавляемая вершина
            Vertex temp2 = new Vertex(listEdge.get(i).getValue());
//добавляемая вершина

            int index1 = 0;
//индекс вхождения в граф
            int index2 = 0;
//индекс вхождения в граф

            if (graph.isEmpty()) {
//в графе нет ни одной вершины
                graph.add(temp1);
                if (temp1.getName() != temp2.getName() &&
temp2.getName() != -1) { //если не петля или изолированная
вершина
                    graph.add(temp2);
                }
                if(temp2.getName() != -1)
graph.get(0).setAdjacentVertex(temp2); //если не изолированная
вершина

            } else {
//если граф не пуст
                for (int j = 0; j < graph.size(); j++) {
                    index1 = find(temp1.getName());
                    if (index1 == -1) {
//вершина еще не добавлена в граф

```

```

        graph.add(temp1);

//добавляем

        index1 = graph.size() - 1;

//запоминаем индекс
    }
    index2 = find(temp2.getName());
    if (index2 == -1) {
//вершина еще не добавлена в граф
        if (temp1.getName() != temp2.getName() &&
temp2.getName() != -1) {
            graph.add(temp2);

//добавляем

        }
        index2 = graph.size() - 1;

//запоминаем индекс
    }
    if (temp2.getName() != -1)
graph.get(index1).setAdjacentVertex(graph.get(index2));
//добавляем смежную вершину
        break;
    }
}
}

/*Color:
 * -1: вершина не обработана
 * 0: вершина обрабатывается
 * 1: вершина обработана */

private void dfs1(int index) {

    // graph.get(index).setTimeIn(dfsTimer++);
//время входа
    graph.get(index).setColor(0);
//обрабатывается

```

```

        for(int i = 0; i <
graph.get(index).getAdjacentVertex().size(); i++) {

    if(graph.get(index).getAdjacentVertex().get(i).getColor() == -1) {
        //если есть смежная непосещенная
            if(graph.get(index).getName() !=
graph.get(index).getAdjacentVertex().get(i).getName())

graph.get(index).getAdjacentVertex().get(i).setWhence(graph.get(in
dex).getName());           //запоминаем как в нее пришли

                events.add(new Event(3));
                events.get(events.size() -
1).setTransition(graph.get(index).getName(),
graph.get(index).getAdjacentVertex().get(i).getName());
                events.get(events.size() -
1).setTextHints("Найдено ребро между вершинами " +
graph.get(index).getName() +
                    " и " +
graph.get(index).getAdjacentVertex().get(i).getName() + ". Переход
из " +
                    graph.get(index).getName() + " в " +
graph.get(index).getAdjacentVertex().get(i).getName() + ".\n");
                //событие: переход по ребру - откуда, куда

dfs1(find(graph.get(index).getAdjacentVertex().get(i).getName()));
//запускаем поиск от нее
    }
}
graph.get(index).setColor(1);
//обработана
graph.get(index).setTimeout(dfsTimer++);
//время выхода
    timeout.add(new Pair(graph.get(index).getName(),
graph.get(index).getTimeout()));

```

```

        if(graph.get(index).getWhence() != -1) {
            events.add(new Event(4));
            events.get(events.size() -
1).setTransition(graph.get(index).getName(),
graph.get(index).getWhence());
            events.get(events.size() - 1).setTextHints("У вершины
" + graph.get(index).getName() +
                " больше нет непосещенных соседних вершин.
Возврат на " + graph.get(index).getWhence() + ".\n");
            //событие: возврат - вершина, из которой возвращаемся,
вершина в которую возвращаемся
        }
        else{
            events.add(new Event(2));
            events.get(events.size() -
1).setNameVertex(graph.get(index).getName());
            events.get(events.size() - 1).setTextHints("У вершины
" + graph.get(index).getName() +
                " больше нет непосещенных соседних
вершин.\n");
            //событие: конец дфс - когда по дфс идти больше некуда
и вернуться на другую вершину нельзя
        }
    }

    private ArrayList<Pair<Integer, Integer>> invertng() {

        ArrayList<Pair<Integer, Integer>> inventedListEdge = new
ArrayList<Pair<Integer, Integer>>();

        for(int i = 0; i < listEdge.size(); i++) {
            if(listEdge.get(i).getValue() == -1)
//изолированная вершина
                inventedListEdge.add(new
Pair(listEdge.get(i).getKey(), listEdge.get(i).getValue()));
            else

```

```

        inventedListEdge.add(new
Pair(listEdge.get(i).getValue(), listEdge.get(i).getKey()));
    }
    return inventedListEdge;
//возвращаем инвертированный список ребер
}

private void dfs2(int index) {

    graph.get(index).setColor(0);
//обрабатывается
    for(int i = 0; i <
graph.get(index).getAdjacentVertex().size(); i++) {

if(graph.get(index).getAdjacentVertex().get(i).getColor() == -1) {
//если есть смежная непосещенная
        if(graph.get(index).getNAME() !=
graph.get(index).getAdjacentVertex().get(i).getNAME()) {

graph.get(index).getAdjacentVertex().get(i).setWhence(graph.get(in
dex).getNAME()); //запоминаем как в нее пришли

graph.get(index).getAdjacentVertex().get(i).setComponent(graph.get
(index).getComponent()); //принадлежность к компоненте

            events.add(new Event(8));
            events.get(events.size() -
1).setDataDFS2(graph.get(index).getNAME(),

graph.get(index).getAdjacentVertex().get(i).getNAME(),
graph.get(index).getComponent());

            events.get(events.size() -
1).setTextHints("Найдено ребро между вершинами " +
graph.get(index).getNAME() +

                " и " +

graph.get(index).getAdjacentVertex().get(i).getNAME() +

```



```

        ". Переход из " +
graph.get(index).getNAME() + " в " +
graph.get(index).getAdjacentVertex().get(i).getNAME() +
        ". Текущая компонента связности: " +
graph.get(index).getComponent() + ".\n");
        //событие: переход по ребру - вершина, из
которой вышли, вершина куда пришли

dfs2(find(graph.get(index).getAdjacentVertex().get(i).getNAME()));
//запускаем поиск от нее
    }
    }
    }
    graph.get(index).setColor(1);
//обработана
    if(graph.get(index).getWhence() != -1) {
        events.add(new Event(9));
        events.get(events.size() -
1).setTransition(graph.get(index).getNAME(),
graph.get(index).getWhence());
        events.get(events.size() - 1).setTextHints("У вершины
" + graph.get(index).getNAME() +
        " больше нет непосещенных соседних вершин.
Возврат на " + graph.get(index).getWhence() + ".\n");
        //событие: возврат - вершина, из которой возвращаемся,
вершина в которую возвращаемся
    }
    else{
        events.add(new Event(7));
        events.get(events.size() -
1).setNameVertex(graph.get(index).getNAME());
        events.get(events.size() - 1).setTextHints("У вершины
" + graph.get(index).getNAME() +
        " больше нет непосещенных соседних
вершин.\n");

```

```
        //событие: конец dfs - когда по dfs идти больше некуда  
        и вернуться на другую вершину нельзя
```

```
    }  
}
```

```
    private int find(Integer name) {  
        //возвращает индекс вхождения вершины в граф  
        for(int i = 0; i < graph.size(); i++){  
            //или -1  
            if (graph.get(i).getName() == name) {  
                return i;  
            }  
        }  
        return -1;  
    }  
}
```

```
//Event.java
```

```
package com.Algorithm;  
import java.util.ArrayList;  
import javafx.util.Pair;
```

```
public class Event {
```

```
    private final Integer EVENT_NUMB;  
    //номер события  
    private Integer nameVertex = -1;  
    //для событий: 1, 2, 7  
    private Integer[] dataDFS2;  
    //для события: 8
```

```
    private Pair <Integer, Integer > transition;  
    //для событий: 3, 4, 6, 9  
    private ArrayList<Pair<Integer, Integer>> inventedListEdge;  
    //для событий: 5
```

```

        private String textHints;
//текстовые пояснения

        public Event(Integer EVENT_NUMB) {
            this.EVENT_NUMB = EVENT_NUMB;
            inventedListEdge = new ArrayList<Pair<Integer,
Integer>>();
            dataDFS2 = new Integer[3];
        }
//сеттеры
        public void setNameVertex(Integer nameVertex) {
            this.nameVertex = nameVertex;
        }

        public void setTransition(Integer value1, Integer value2) {
            transition = new Pair <Integer, Integer >(value1, value2);
        }

        public void setInventedListEdge(ArrayList<Pair<Integer,
Integer>> inventedListEdge) {
            this.inventedListEdge = inventedListEdge;
        }

        public void setDataDFS2(Integer value1, Integer value2,
Integer value3) {
            dataDFS2[0] = value1;
            dataDFS2[1] = value2;
            dataDFS2[2] = value3;
        }

        public void setTextHints(String textHints){
            this.textHints = textHints;
        }

//геттеры
        public Integer getName_EVENT() {

```

```

        return EVENT_NUMB;
    }

    public Integer getNameVertex() {
        return nameVertex;
    }

    public Pair <Integer, Integer > getTransition() {
        return transition;
    }

    public ArrayList<Pair<Integer, Integer>> getInventedListEdge()
{
        return inventedListEdge;
    }

    public Integer[] getDataDFS2() {
        return dataDFS2;
    }

    public String getTextHints() {
        return textHints;
    }
}

//Vertex.java
package com.Algorithm;
import javafx.util.Pair;
import java.util.ArrayList;

public class Vertex {

    private final Integer NAME;                //имя
    private Integer whence = -1;                //из какой вершины
попали
    private ArrayList<Vertex> adjacentVertex;  //список смежных
вершин

```

```

        private int color = -1;                                //цвет, в который
вершина окрашена (для dfs)
        private int component = -1;                            //компонента
связности
        private int timeOut = -1;                              //время выхода
(для dfs)

```

```

public Vertex(Integer NAME) {
    this.NAME = NAME;
    adjacentVertex = new ArrayList<Vertex>();
}
//сеттеры
public void setWhence(Integer whence) {
    this.whence = whence;
}

public void setColor(int color) {
    this.color = color;
}

public void setComponent(int component) {
    this.component = component;
}

public void setTimeout(int timeOut) {
    this.timeOut = timeOut;
}

public void setAdjacentVertex(Vertex value) {
    adjacentVertex.add(value);
}
//геттеры
public Integer getNAME() {
    return NAME;
}

```

```

        public Integer getWhence() {
            return whence;
        }

        public int getColor() {
            return color;
        }

        public int getComponent() {
            return component;
        }

        public int getTimeout() {
            return timeout;
        }

        public ArrayList<Vertex> getAdjacentVertex() {
            return adjacentVertex;
        }
    }
}
//Controller.java
package sample;

import javafx.animation.KeyFrame;
import javafx.animation.Timeline;
import javafx.fxml.FXML;
import javafx.scene.canvas.Canvas;
import javafx.scene.control.Button;
import javafx.scene.control.Label;
import javafx.scene.control.TextArea;
import javafx.scene.layout.HBox;
import javafx.scene.layout.Pane;
import javafx.util.Duration;

public class Controller {
    boolean isBtnAddVClicked = false;

```

```

boolean isBtnAddRClicked = false;
boolean isBtnDelVClicked = false;
boolean isBtnDelRClicked = false;
boolean isDraggedProcess = false;
boolean isVisualiseProcess = false;

@FXML
// Function for reacting to a button click
private void isBtnAddVertClicked() {
    if(!isBtnAddVClicked) {
        mainTextArea.setText("Нажмите на поле для добавления
вершины.");
        isBtnAddVClicked = true;
        isBtnAddRClicked = false;
        isBtnDelVClicked = false;
        isBtnDelRClicked = false;
    }
    else isBtnAddVClicked = false;
    setPrevChosenVertex(null);
}

@FXML
private void isBtnAddRibClicked() {
    if(!isBtnAddRClicked) {
        mainTextArea.setText("Выберите две вершины для
добавления ребра.");
        isBtnAddRClicked = true;
        isBtnAddVClicked = false;
        isBtnDelVClicked = false;
        isBtnDelRClicked = false;
    }
    else {
        isBtnAddRClicked = false;
        mainTextArea.setText("");
    }
    setPrevChosenVertex(null);
}

```

```

@FXML
private void isBtnDelVertClicked() {
    if(!isBtnDelVClicked) {
        mainTextArea.setText("Выберите вершины для
удаления.");
        isBtnDelVClicked = true;
        isBtnDelRClicked = false;
        isBtnAddRClicked = false;
        isBtnAddVClicked = false;
    }
    else {
        isBtnDelVClicked = false;
        mainTextArea.setText("");
    }
    setPrevChosenVertex(null);
}

@FXML
private void isBtnDelRibClicked() {
    if(!isBtnDelRClicked) {
        mainTextArea.setText("Выберите две вершины для
удаления ребра.");
        isBtnDelRClicked = true;
        isBtnDelVClicked = false;
        isBtnAddRClicked = false;
        isBtnAddVClicked = false;
    }
    else isBtnDelRClicked = false;
    setPrevChosenVertex(null);
}

// обработка нажатия на область для добавления графа
private void canvasClick(javafx.scene.input.MouseEvent
mouseEvent) {
    Vertex curChosenVertex =
graph.checkCollision(mouseEvent.getX(), mouseEvent.getY(), 0);

```



```

        if (isBtnAddVClicked || isBtnAddRClicked) {
            if (curChosenVertex == null && isBtnAddVClicked) {
                int numb = 1;
                while(graph.findVertex(numb) != null)
                    numb++;
                graph.addVertex(new Vertex(mouseEvent.getX(),
mouseEvent.getY(), numb));
                setPrevChosenVertex(null);
            }
            else if (curChosenVertex != null && prevChosenVertex
!= null) {
                graph.addEdge(prevChosenVertex, curChosenVertex);
                setPrevChosenVertex(null);
            }
            else if (isBtnAddRClicked) {
                setPrevChosenVertex(curChosenVertex);
            }
        }
        else if (isBtnDelVClicked) {
            if (curChosenVertex != null) {
                graph.removeVertex(curChosenVertex);
                setPrevChosenVertex(null);
            }
        }
        else if (isBtnDelRClicked) {
            if (curChosenVertex != null && prevChosenVertex !=
null) {
                graph.removeEdge(prevChosenVertex,
curChosenVertex);
                setPrevChosenVertex(null);
            }
            else {
                setPrevChosenVertex(curChosenVertex);
            }
        }
    }
}

```

```

@FXML
private void canvasMousePressed(javafx.scene.input.MouseEvent
mouseEvent) {
    draggedVertex = graph.checkCollision(mouseEvent.getX(),
mouseEvent.getY(), 0);
}

@FXML
private void canvasMouseReleased(javafx.scene.input.MouseEvent
mouseEvent) {
    if (!isVisualiseProcess && !isDraggedProcess) {
        canvasClick(mouseEvent);
    }
    draggedVertex = null;
    isDraggedProcess = false;
}

@FXML
private void canvasMouseDragged(javafx.scene.input.MouseEvent
mouseEvent) {
    if (draggedVertex != null) {
        draggedVertex.setX(mouseEvent.getX());
        draggedVertex.setY(mouseEvent.getY());
        isDraggedProcess = true;
    }
}

@FXML
private void runAlgorithm() {
    isBtnAddVClicked = false;
    isBtnAddRClicked = false;
    isBtnDelVClicked = false;
    isBtnDelRClicked = false;
    setPrevChosenVertex(null);
    if (graph.isEmpty()) {
        mainTextArea.setText("Граф пуст.\n");
        return;
    }
}

```

```

    }

    switchVisualize();
    mainTextArea.setText("Алгоритм запущен.\n");
    mainLabel.getChildren().add(new Label("Порядок вершин:"));
    graph.runAlgorithm();
}

```

@FXML

```

private void clearButton() {
    isBtnAddVClicked = false;
    isBtnAddRClicked = false;
    isBtnDelVClicked = false;
    isBtnDelRClicked = false;
    setPrevChosenVertex(null);
    mainTextArea.setText("Поле очищено.\n");
    graph = new Graph();
}

```

@FXML

```

private void stopButton() {
    isBtnAddVClicked = false;
    isBtnAddRClicked = false;
    isBtnDelVClicked = false;
    isBtnDelRClicked = false;
    setPrevChosenVertex(null);
    graph.endAlgorithm(mainTextArea, mainLabel);
    mainTextArea.setText("Алгоритм остановлен.\n");
    switchVisualize();
}

```

@FXML

```

private void stepForward() {
    if (graph.isLastEvent()) {
        graph.endAlgorithm(mainTextArea, mainLabel);
        disableButton(stepButton);
        disableButton(showButton);
    }
}

```

```

    }
    else
        graph.visualiseStep(mainTextArea, mainLabel);
    activeButton(stepButtonBack);
}

@FXML
private void stepBack() {
    graph.visualiseStepBack(mainTextArea, mainLabel);
    if (graph.isFirstEvent()) {
        disableButton(stepButtonBack);
    }
    activeButton(stepButton);
    activeButton(showButton);
    if (graph.isLastEvent()) {
        Timeline timeline = new Timeline(
            new KeyFrame(
                Duration.millis(50),
                ae ->
mainTextArea.setScrollTop(Double.MAX_VALUE)
            )
        );
        timeline.setCycleCount(1);
        timeline.play();
    }
}

@FXML
private void showResult() {
    graph.endAlgorithm(mainTextArea, mainLabel);
    disableButton(stepButton);
    disableButton(showButton);
    activeButton(stepButtonBack);
}

@FXML
private void importG() throws Throwable

```

```

    {
        ModalWindow.newWindow("Импорт из файла");
        graph = new Graph();
        graph.inputFileGraph(mainCanvas.getGraphicsContext2D(),
mainTextArea);
    }

    public void resizeCanvas() {

mainCanvas.setHeight(((Pane) (mainCanvas.getParent())) .getHeight())
;

mainCanvas.setWidth(((Pane) (mainCanvas.getParent())) .getWidth());
        graph.drawAll(mainCanvas.getGraphicsContext2D());
    }

    //переключение между режимами добавления графа и показом
алгоритма
    private void switchVisualize() {
        graph.standardView(mainLabel);
        if (isVisualiseProcess) {
            activeButton(importButton);
            activeButton(addVButton);
            activeButton(addEButton);
            activeButton(delVButton);
            activeButton(deleteButton);
            activeButton(runButton);
            disableButton(stepButton);
            disableButton(stepButtonBack);
            disableButton(showButton);
            disableButton(stopButton);
            activeButton(clearButton);
        }
        else {
            disableButton(importButton);
            disableButton(addVButton);
            disableButton(addEButton);

```

```

        disableButton(delVButton);
        disableButton(delEButton);
        disableButton(runButton);
        activeButton(stepButton);
        disableButton(stepButtonBack);
        activeButton(showButton);
        activeButton(stopButton);
        disableButton(clearButton);
    }
    isVisualiseProcess = !isVisualiseProcess;
}

private void disableButton(Button button) {
    button.setOpacity(0.75);
    button.setDisable(true);
}

private void activeButton(Button button) {
    button.setOpacity(1.0);
    button.setDisable(false);
}

// запоминание предыдущей выбраной вершины для добавления и
удаления рёбер
private void setPrevChosenVertex(Vertex vert) {
    if (prevChosenVertex != null)
prevChosenVertex.setRinged(false);
    prevChosenVertex = vert;
    if (prevChosenVertex != null)
prevChosenVertex.setRinged(true);
}

@FXML
private Canvas mainCanvas;

@FXML
private TextArea mainTextArea;

```

```

    @FXML
    private HBox mainLabel;

    @FXML
    private Button importButton, addVButton, addEButton,
    delVButton, delEButton,
        runButton, stepButton, stepButtonBack, showButton,
    stopButton, clearButton;

    private Graph graph = new Graph();

    private Vertex prevChosenVertex;
    private Vertex draggedVertex;

}
//Graph.java
package sample;

import com.Algorithm.AlgorithmKosarayu;
import com.Algorithm.Event;
import javafx.scene.canvas.GraphicsContext;
import javafx.scene.control.Label;
import javafx.scene.control.TextArea;
import javafx.scene.layout.HBox;
import javafx.scene.paint.Color;
import javafx.util.Pair;

import java.io.IOException;
import java.util.ArrayList;
import java.util.HashSet;
import java.util.Random;

public class Graph {
    public Graph() {
        Vertex.vertexCount = 1;
    }

```

```

public void addVertex(Vertex vertex) {
    vertexes.add(vertex);
}

public void removeVertex(Vertex vert) {
    for (Vertex nextVert: vertexes) {
        nextVert.getAdj().remove(vert);
    }
    vertexes.remove(vert);
}

public void addEdge(Vertex vert1, Vertex vert2) {
    if (vert1.equals(vert2)) vert1.setHasLoop(true); //
добавление петли
    else vert1.addAdj(vert2);
}

public void removeEdge(Vertex vert1, Vertex vert2) {
    if (vert1.equals(vert2)) vert1.setHasLoop(false); //
удаление петли
    vert1.getAdj().remove(vert2);
    vert2.getAdj().remove(vert1);
}

// отрисовка изображения графа
public void drawAll(GraphicsContext gc) {
    clearCanvas(gc);
    Pair<Integer, Color> edge;
    for (Vertex vert: vertexes) {
        vert.draw(gc);
        if (vert.getRinged()) drawRing(gc, vert.getX(),
vert.getY()); // отрисовка кольца вокруг вершины
        // отрисовка всех рёбер исходящих из вершины
        for (Vertex nextVert: vert.getAdj()) {
            if (vert != nextVert) {
                if (nextVert.getNumber() ==
vert.getNextSelectedEdge())

```



```

        drawArrow(gc, vert.getX(), vert.getY(),
nextVert.getX(), nextVert.getY(),

Color.RED, false);

        else if(!vert.getSelectedEdges().isEmpty() &&
            (edge =
vert.findSelectedEdge(nextVert.getNumber())) != null)
            drawArrow(gc, vert.getX(),
vert.getY(), nextVert.getX(), nextVert.getY(),

edge.getValue(), false);

        else if
(ver.getColor().equals(nextVert.getColor()))
        {
            if (!nextVert.getSelectedEdges().isEmpty()
&& nextVert.findSelectedEdge(vert.getNumber())
                != null || vert.getNumber() ==
nextVert.getNextSelectedEdge())
                drawArrow(gc, vert.getX(),
vert.getY(), nextVert.getX(), nextVert.getY(),

Color.BLACK, true);

            else
                drawArrow(gc, vert.getX(),
vert.getY(), nextVert.getX(), nextVert.getY(),

vert.getColor(), false);
        }
        else
            drawArrow(gc, vert.getX(),
vert.getY(), nextVert.getX(), nextVert.getY(),

Color.BLACK, false);

    }
}

```

```

    }
}

private void drawArrow(GraphicsContext gc, double node1X,
double node1Y,
double node2X, double node2Y, Color
color, boolean isOnlyArrow) {
    double arrowAngle = Math.toRadians(30.0);
    double arrowLength = Vertex.RADIUS;
    double dx = node1X - node2X;
    double dy = node1Y - node2Y;
    double angle = Math.atan2(dy, dx);

    node1X = node1X - Math.cos(angle) * Vertex.RADIUS;
    node1Y = node1Y - Math.sin(angle) * Vertex.RADIUS;
    node2X = node2X + Math.cos(angle) * Vertex.RADIUS;
    node2Y = node2Y + Math.sin(angle) * Vertex.RADIUS;
    gc.setStroke(color);
    gc.setLineWidth(2);
    if(!isOnlyArrow)
        gc.strokeLine(node1X, node1Y, node2X, node2Y);

    double x1 = Math.cos(angle + arrowAngle) * arrowLength +
node2X;
    double y1 = Math.sin(angle + arrowAngle) * arrowLength +
node2Y;

    double x2 = Math.cos(angle - arrowAngle) * arrowLength +
node2X;
    double y2 = Math.sin(angle - arrowAngle) * arrowLength +
node2Y;

    // отрисовка наконечника стрелочки
    gc.strokeLine(node2X, node2Y, x1, y1);
    gc.strokeLine(node2X, node2Y, x2, y2);
}

```

```

        private static void drawRing(GraphicsContext gc, double x,
double y) {
            gc.setStroke(Color.RED);
            gc.setLineWidth(3);
            gc.strokeOval(x - Vertex.RADIUS, y-Vertex.RADIUS,
Vertex.RADIUS*2, Vertex.RADIUS*2);
        }

        public Vertex checkCollision(double dotX, double dotY, double
diff) {
            for (Vertex vert: vertexes) {
                if (vert.isDotInRadius(dotX, dotY, diff)) {
                    return vert;
                }
            }
            return null;
        }

        // получение списка рёбер и передача алгоритму
        public void runAlgorithm() {
            ArrayList<Pair<Integer, Integer>> adjList = new
ArrayList<>();
            for (Vertex vert: vertexes) {
                for (Vertex curVert: vert.getAdj()) {
                    adjList.add(new Pair(vert.getNumber(),
curVert.getNumber()));
                }
            }
            // выделение изолированных вершин
            HashSet<Integer> markedVertexes = new HashSet<>();
            for (Pair<Integer, Integer> edge: adjList) {
                markedVertexes.add(edge.getKey());
                markedVertexes.add(edge.getValue());
            }
            for (Vertex vert: vertexes) {
                if (!markedVertexes.contains(vert.getNumber())) {
                    adjList.add(new Pair(vert.getNumber(), -1));
                }
            }
        }
    }
}

```

```

        }
    }

    AlgorithmKosarayu algorithm = new AlgorithmKosarayu();
    events = algorithm.start(adjList);
    eventIndex = 0;
}

// визуализация одного шага алгоритма, согласно списку
событий, полученному от алгоритма
public void visualiseStep(TextArea textArea, HBox labelBox) {
    Event event = events.get(eventIndex++);
    textArea.appendText(event.getTextHints());
    switch (event.getNAME_EVENT()) {
        case 1:
            VisualSteps.event1(vertexes,
event.getNameVertex());
            break;
        case 2:
            VisualSteps.event2(vertexes,
event.getNameVertex(), labelBox);
            break;
        case 3:
            VisualSteps.event3(vertexes,
event.getTransition().getKey(), event.getTransition().getValue());
            break;
        case 4:
            VisualSteps.event4(vertexes,
event.getTransition().getKey(),
event.getTransition().getValue(),
labelBox);
            break;
        case 5:
            VisualSteps.event5(vertexes);
            for (Pair<Integer, Integer> pair :
event.getInventedListEdge()) {
                if (pair.getValue() != -1)

```

```

        addEdge(findVertex(pair.getKey()),
findVertex(pair.getValue()));
    }
    break;
    case 6:
        VisualSteps.event6(vertexes,
event.getTransition().getKey(), event.getTransition().getValue());
        break;
    case 7:
        VisualSteps.event7(vertexes,
event.getNameVertex());
        break;
    case 8:
        VisualSteps.event8(vertexes,
event.getDataDFS2()[0], event.getDataDFS2()[1],
event.getDataDFS2()[2]);
        break;
    case 9:
        VisualSteps.event9(vertexes,
event.getTransition().getKey(), event.getTransition().getValue());
        break;
    case 10:
        if (orderIndex != 0)
            ((Label) (labelBox.getChildren().
                get(labelBox.getChildren().size() -
orderIndex))).setTextFill(Color.BLACK);
            ((Label) (labelBox.getChildren().
                get(labelBox.getChildren().size() -
orderIndex++ - 1))).setTextFill(Color.RED);
            VisualSteps.event10(vertexes,
event.getNameVertex());
            break;
    }
}

// визуализация шага назад

```

```

    public void visualiseStepBack(TextArea textArea, HBox
labelBox) {
        int saveEventIndex = eventIndex - 1;
        endAlgorithm(textArea, labelBox);
        standardView(labelBox);
        eventIndex = 0;
        textArea.setText("Алгоритм запущен.\n");
        labelBox.getChildren().add(new Label("Порядок вершин:"));
        for (int i = 0; i < saveEventIndex; i++) {
            visualiseStep(textArea, labelBox);
        }
    }

    // сброс изображения графа к стандартному виду
    public void standardView(HBox labelBox) {
        for (Vertex vert: vertexes) {
            vert.setColor(Color.BLACK);
            vert.setRinged(false);
            vert.getSelectedEdges().clear();
            vert.setNextSelectedEdge(0);
        }
        orderIndex = 0;
        labelBox.getChildren().clear();
    }

    // ввод графа из файла
    public void inputFileGraph(GraphicsContext gc, TextArea
mainTextArea) throws IOException {
        ImportManager graph = new
ImportManager(ModalWindow.fileName, mainTextArea);
        ArrayList<Pair<Integer, Integer>> list = graph.getGraph();
        if (list == null) { return; }
        mainTextArea.setText("Граф импортирован.");

        double fromX = 40;
        double toX = gc.getCanvas().getWidth() - 30;

```

```

double fromY = 60;
double toY = gc.getCanvas().getHeight() - 30;
double X;
double Y;
double diffVertexes = 60;    // расстояние между вершинами
Random randomX = new Random();
Random randomY = new Random();
X = fromX + randomX.nextInt((int) (toX - fromX));
Y = fromY + randomY.nextInt((int) (toY - fromY));
double repeat = 1;

for (Pair<Integer, Integer> rib : list) {
    Vertex vertexFrom = findVertex(rib.getKey());

    if (vertexFrom == null) {    // если данной вершины не
существует

        // выбор места, непересекающегося с другими
вершинами

        while (checkCollision(X, Y, diffVertexes) != null
&& X < toX && Y < toY) {
            X = fromX + randomX.nextInt((int) (toX -
fromX));

            Y = fromY + randomY.nextInt((int) (toY -
fromY));

            if(repeat++ > 150) {
                mainTextArea.setText("Скорее всего,
введенный граф слишком большой и не помещается на" +
                " поле текущего размера.
Разверните программу на полный экран и попробуйте снова.");
                vertexes.clear();
                clearCanvas(gc);
                return;
            }
        }
        vertexFrom = new Vertex(X, Y, rib.getKey());
        addVertex(vertexFrom);
    }
}

```

```

        if (rib.getValue() == -1) continue;

        Vertex vertexTo = findVertex(rib.getValue());
        repeat = 1;

        if (vertexTo == null) { // если данной вершины не
существует
            // выбор места, непересекающегося с другими
вершинами
            while (checkCollision(X, Y, diffVertexes) != null
&& X < toX && Y < toY) {
                X = fromX + randomX.nextInt((int) (toX -
fromX));

                Y = fromY + randomY.nextInt((int) (toY -
fromY));

                if(repeat++ > 150) {
                    mainTextArea.setText("Скорее всего,
введенный граф слишком большой и не помещается на" +
" поле текущего размера.
Разверните программу на полный экран и попробуйте снова.");
                    vertexes.clear();
                    clearCanvas(gc);
                    return;
                }
            }
            vertexTo = new Vertex(X, Y, rib.getValue());
            addVertex(vertexTo);
        }
        addEdge(vertexFrom, vertexTo); // добавление ребра
    }
}

public Vertex findVertex(int number) {
    for(Vertex vertex : vertexes) {
        if(vertex.getNumber() == number)
            return vertex;
    }
}

```



```

        }
        return null;
    }

    private void clearCanvas(GraphicsContext gc) {
        gc.clearRect(0,0, gc.getCanvas().getWidth(),
gc.getCanvas().getHeight());
    }

    public boolean isLastEvent() {
        return events.size()-1 == eventIndex;
    }
    public boolean isFirstEvent() {return eventIndex == 0;}
    public boolean isEmpty() {return vertexes.isEmpty();}

    public void endAlgorithm(TextArea textArea, HBox label) {
        for (int i = eventIndex; i < events.size(); i++) {
            visualiseStep(textArea, label);
        }
        textArea.setText("Результат работы алгоритма изображен на
экране.\n");
    }

    private final ArrayList<Vertex> vertexes = new ArrayList<>();

    private ArrayList<Event> events = new ArrayList<>();

    private int orderIndex = 0;

    private int eventIndex = 0;
}
//ImportManager.java
package sample;

import javafx.scene.control.TextArea;
import javafx.util.Pair;

```

```

import java.io.BufferedReader;
import java.io.IOException;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.util.ArrayList;

/*
 * Класс для импорта графа из файла. Файл следует хранить в
 * корневой папке
 * "folder_project/name.txt"
 */

public class ImportManager {

    private boolean isOpenAndReadOK;
    private final ArrayList<Pair<Integer, Integer>> graph;

    public ImportManager(String name, TextArea mainTextArea)
    throws IOException {

        Path pathImport = Paths.get(name);
        graph = new ArrayList<Pair<Integer, Integer>>();

        try(BufferedReader in =
Files.newBufferedReader(pathImport)){
            String line = in.readLine();
            while (line != null){

                String[] rib = (line).split(" ", 2);

                if (rib.length < 2 || rib[0].equals("") ||
rib[1].equals(""))
                {
                    throw new NumberFormatException(line);
                }
                Integer start = Integer.parseInt(rib[0]);

```

```

        Integer end = Integer.parseInt(rib[1]);

        if (start < 0 || end == 0 || end < -1)
        {
            throw new NumberFormatException(line);
        }
        graph.add(new Pair(start, end));

        line = in.readLine();
    }
    isOpenAndReadOK = true;
}
catch(IOException e) {
    mainTextArea.setText("Файл для импорта не найден.");
}
catch(ClassCastException|NumberFormatException e){
    mainTextArea.setText("Формат текста в файле
некорректный: \"" + e.getMessage() + "\"");
}
}

public ArrayList<Pair<Integer, Integer>> getGraph(){
    return isOpenAndReadOK ? graph : null;
}

```

```

}

```

```

//Main.java

```

```

package sample;

```

```

import javafx.application.Application;
import javafx.application.Platform;
import javafx.fxml.FXMLLoader;
import javafx.scene.Parent;
import javafx.scene.Scene;
import javafx.stage.Stage;

```

```

import java.util.Timer;

```

```

import java.util.TimerTask;

public class Main extends Application {

    @Override
    public void start(Stage primaryStage) throws Exception {
        FXMLLoader loader = new
FXMLLoader(Main.class.getResource("sample.fxml"));
        Parent root = loader.load();
        Controller contr = loader.getController();

        Timer timer = new java.util.Timer();
        timer.schedule(new TimerTask() {
            public void run() {
                Platform.runLater(contr::resizeCanvas);
            }
        }, 0, 50);
        primaryStage.setOnCloseRequest(windowEvent ->
{timer.cancel();});
        primaryStage.setTitle("Strong connectivity");
        primaryStage.setScene(new Scene(root, 700, 500));
        primaryStage.show();
    }

    public static void main(String[] args) {
        launch(args);
    }
}
//ModalWindow.java
package sample;

import javafx.geometry.Insets;
import javafx.geometry.Pos;
import javafx.scene.Scene;
import javafx.scene.control.Button;
import javafx.scene.control.Label;

```

```

import javafx.scene.control.TextField;
import javafx.scene.layout.StackPane;
import javafx.scene.layout.VBox;
import javafx.stage.Modality;
import javafx.stage.Stage;

// окно, которое создаётся при нажатии кнопки считывания из файла
public class ModalWindow {
    public static void closeFile(String str, Stage window) {
        fileName = str;
        window.close();
    }

    public static void newWindow(String title) {
        fileName = "";
        Stage window = new Stage();
        window.initModality(Modality.APPLICATION_MODAL);
        StackPane pane = new StackPane();
        double width = 500;
        double height = 300;

        Label label = new Label("Введите в файл рёбра в виде пар
чисел через перенос строки. Далее введите " +
        "имя файла(name.txt) в поле ниже для импорта графа
из файла.\n" +
        "В папке Tests находятся тестовые файлы, которые
можно использовать для тестирования. Для этого введите
\"Tests/test{номер теста}.txt.\" +
        "\n\n" +
        "Формат ввода: \n" +
        "
1 1 -
ориентированное ребро,\n" +
        "
3 -1 -
изолированная вершина,\n" +
        "
7 7 - петля\n");

        label.setMaxWidth(width*9/10);
    }
}

```

```

        label.setWrapText(true);
        TextField textField = new TextField();
        textField.setMaxWidth(width*9/10);
        Button btn = new Button("OK");
        btn.setMinWidth(width*3/20);
        btn.setDefaultButton(true);

        VBox vbox = new VBox(label, textField);
        VBox.setMargin(textField, new Insets(5, 10, 5, 10));
        VBox.setMargin(label, new Insets(5, 10, width*1/40, 10));

        vbox.setAlignment(Pos.CENTER);

        btn.setOnAction(event->closeFile(textField.getText(),
window));

        pane.getChildren().addAll(vbox, btn);
        pane.setMargin(btn, new Insets(Math.max(width*1/40, 5),
width*1/20, width*1/40, 10));
        pane.setAlignment(btn, Pos.BOTTOM_RIGHT);

        Scene scene = new Scene(pane, width, height);
        window.setScene(scene);
        window.setTitle(title);
        window.showAndWait();
    }

    public static String fileName = "";
}
//Vertex.java
package sample;

import javafx.scene.canvas.GraphicsContext;
import javafx.scene.paint.Color;
import javafx.scene.text.TextAlignment;
import javafx.util.Pair;

```

```

import java.util.ArrayList;

public class Vertex {
    public Vertex(double x, double y, int number) {
        this.x = x;
        this.y = y;
        this.number = number;
    }

    // отрисовка вершины
    public void draw(GraphicsContext gc) {
        gc.setFill(currentColor);
        if (isHasLoop) drawLoop(gc);
        gc.fillOval(x-RADIUS, y-RADIUS, RADIUS*2, RADIUS*2);
        gc.setFill(Color.WHITE);
        gc.setTextAlign(TextAlignment.CENTER);
        gc.fillText(String.valueOf(number), x, y + 5);
    }

    // отрисовка кольца вокруг вершины, обозначающего выделение
    private void drawLoop(GraphicsContext gc) {
        gc.setStroke(currentColor);
        gc.setLineWidth(2);
        gc.strokeOval(x, y - 2 * (RADIUS - 1), (RADIUS - 1) * 2,
(RADIUS - 1) * 2);
        gc.strokeLine(x + RADIUS, y, x + 2 * RADIUS, y);

        double angle = Math.toRadians(70.0);
        double X1 = x + RADIUS;
        double Y1 = y;
        double X2 = X1 + Math.cos(angle) * RADIUS;
        double Y2 = Y1 - Math.sin(angle) * RADIUS;

        gc.strokeLine(X1, Y1, X2, Y2);
    }
}

```

```

    public boolean isDotInRadius(double dotX, double dotY, double
diff) {
        return (dotX - x)*(dotX - x) + (dotY - y)*(dotY - y) <=
(RADIUS * 3.0/2 + diff)*(RADIUS * 3.0/2 + diff);
    }

    public void addAdj(Vertex vert) {
        adj.add(vert);
    }

    public int getNumber() {
        return number;
    }

    public ArrayList<Vertex> getAdj() {
        return adj;
    }

    public void setX(double newX) {x = newX;}
    public void setY(double newY) {y = newY;}
    public double getX() {return x;}
    public double getY() {return y;}

    public void setRinged(boolean bool) {isRinged = bool;}
    public Boolean getRinged() {return isRinged;}

    public void setColor(Color color) {currentColor = color;}
    public Color getColor() {return currentColor;}

    public void setNextSelectedEdge(int nextSelectedEdge)
{this.nextSelectedEdge = nextSelectedEdge;}
    public int getNextSelectedEdge() {
        return nextSelectedEdge;
    }

    public void addSelectedEdge(int numb, Color color)
{selectedEdges.add(new Pair<Integer, Color>(numb, color));}

```



```

public ArrayList<Pair<Integer, Color>> getSelectedEdges() {
    return selectedEdges;
}

public void setHasLoop(boolean bool) {isHasLoop = bool;}

public Pair<Integer, Color> findSelectedEdge(int number) {
    for(Pair<Integer, Color> edge : selectedEdges) {
        if(edge.getKey().equals(number))
            return edge;
    }
    return null;
}

private final ArrayList<Pair<Integer, Color>> selectedEdges =
new ArrayList<>();

// вершины, в которые идут рёбра из текущей
private final ArrayList<Vertex> adj = new ArrayList<>();

private double x;
private double y;

private final int number;
private int nextSelectedEdge = 0;
private boolean isHasLoop = false;
private boolean isRinged = false;
private Color currentColor = Color.BLACK;

public static int vertexCount = 1;
public static int RADIUS = 16;

public static Color computeColor(int numb) {
    return colors[(numb-1) % colors.length];
}

// список цветов для обозначений компонент связности

```

```

        public static Color[] colors = {Color.GREEN, Color.BLUE,
Color.BISQUE, Color.PURPLE,
            Color.PINK, Color.GREENYELLOW, Color.CYAN, Color.LIME,
Color.BROWN,
            Color.GRAY, Color.CORAL, Color.DARKGREEN,
Color.ORANGE};
    }
//VisualSteps.java
package sample;

import javafx.scene.control.Label;
import javafx.scene.layout.HBox;
import javafx.scene.paint.Color;

import java.util.ArrayList;

// класс для пошаговой визуализации
// события получаютс в виде списка при завершении работы
алгоритма
public class VisualSteps {
    // событие начала очередного dfs1
    public static void event1(ArrayList<Vertex> vertexes, int
number) {
        for (Vertex vert: vertexes) {
            vert.setNextSelectedEdge(0);
            if (vert.getNumber() == number) {
                vert.setRinged(true);
            }
        }
    }

    // событие окончания очередного dfs1
    public static void event2(ArrayList<Vertex> vertexes, int
number, HBox labelBox) {
        labelBox.getChildren().add(new Label(" " + number));
        for (Vertex vert: vertexes) {
            vert.setNextSelectedEdge(0);

```

```

        if (vert.getNumber() == number) {
            vert.setRinged(false);
        }
    }
}

// событие перехода по ребру в dfs1
public static void event3(ArrayList<Vertex> vertexes, int
fromNumber, int toNumber) {
    for (Vertex vert: vertexes) {
        vert.setNextSelectedEdge(0);
        if (vert.getNumber() == fromNumber) {
            vert.setRinged(false);
            vert.setNextSelectedEdge(toNumber);
            vert.addSelectedEdge(toNumber, Color.DARKRED);
        }
        if (vert.getNumber() == toNumber)
vert.setRinged(true);
    }
}

// событие возврата в dfs1
public static void event4(ArrayList<Vertex> vertexes, int
fromNumber, int toNumber, HBox labelBox) {
    labelBox.getChildren().add(new Label(" " + fromNumber));
    for (Vertex vert: vertexes) {
        vert.setNextSelectedEdge(0);
        if (vert.getNumber() == fromNumber)
vert.setRinged(false);
        if (vert.getNumber() == toNumber)
vert.setRinged(true);
    }
}

// событие инвертирования всех рёбер графа
public static void event5(ArrayList<Vertex> vertexes) {
    for (Vertex vert: vertexes) {

```

```

        vert.setRinged(false);
        vert.getSelectedEdges().clear();
        vert.setNextSelectedEdge(0);
        vert.getAdj().clear();
    }
}

// событие начала очередного dfs2
public static void event6(ArrayList<Vertex> vertexes, int
number, int colorNumber) {
    for (Vertex vert: vertexes) {
        vert.setNextSelectedEdge(0);
        if (vert.getNumber() == number) {
            vert.setRinged(true);
            vert.setColor(Vertex.computeColor(colorNumber));
        }
    }
}

// событие окончания очередного dfs2
public static void event7(ArrayList<Vertex> vertexes, int
number) {
    for (Vertex vert: vertexes) {
        vert.setNextSelectedEdge(0);
        if (vert.getNumber() == number) {
            vert.setRinged(false);
        }
    }
}

// событие перехода по ребру в dfs2
public static void event8(ArrayList<Vertex> vertexes, int
fromNumber, int toNumber, int colorNumber) {
    for (Vertex vert: vertexes) {
        vert.setNextSelectedEdge(0);
        if (vert.getNumber() == fromNumber) {
            vert.setRinged(false);

```

```

        vert.setNextSelectedEdge(toNumber);
    }
    if (vert.getNumber() == toNumber) {
        vert.setRinged(true);
        vert.setColor(Vertex.computeColor(colorNumber));
    }
}

// событие возврата в dfs2
public static void event9(ArrayList<Vertex> vertexes, int
fromNumber, int toNumber) {
    for (Vertex vert: vertexes) {
        vert.setNextSelectedEdge(0);
        if (vert.getNumber() == fromNumber)
vert.setRinged(false);
        if (vert.getNumber() == toNumber)
vert.setRinged(true);
    }
}

// событие просмотра нового кандидата для начала dfs2
public static void event10(ArrayList<Vertex> vertexes, int
number) {
    for (Vertex vert: vertexes) {
        vert.setRinged(false);
        if (vert.getNumber() == number) {
            vert.setRinged(true);
        }
    }
}
}

```

