

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

ОТЧЕТ
по учебной практике
Тема: Визуализация алгоритма Дейкстры поиска кратчайших путей в
графе.

Студент гр. 2381	_____	Долотов Н.А.
Студент гр. 2381	_____	Богатов И.А.
Студент гр. 2381	_____	Бочаров Г.С.
Руководитель	_____	Фирсов М.А.

Санкт-Петербург
2024

ЗАДАНИЕ НА УЧЕБНУЮ ПРАКТИКУ

Студент Долотов Н.А. группы 2381

Студент Богатов И.А. группы 2381

Студент Бочаров Г.С. группы 2381

Тема практики: Командная итеративная разработка пошагового визуализатора алгоритма Дейкстры поиска кратчайших путей в графе с графическим интерфейсом на *Java*.

Задание на практику:

Командная итеративная разработка визуализатора алгоритма на *Java* с графическим интерфейсом.

Алгоритм: алгоритм Дейкстры поиска кратчайших путей в графе.

Сроки прохождения практики: 26.06.2024 – 09.07.2024

Дата сдачи отчета: 08.07.2024

Дата защиты отчета: 08.07.2024

Студент	_____	Долотов Н.А.
Студент	_____	Богатов И.А.
Студент	_____	Бочаров Г.С.
Руководитель	_____	Фирсов М.А.

АННОТАЦИЯ

Целью мини-проекта является овладение новым объектно-ориентированным языком программирования *Java*, а также применение полученных знаний на практике. Задача состоит в реализации визуализатора алгоритма Дейкстры поиска кратчайших путей в графе с графическим интерфейсом.

SUMMARY

The purpose of the mini-project is to master new object-oriented Java programming languages, as well as to apply the knowledge gained in practice. The task is to implement the visualizer of the Dijkstra algorithm for finding shortest paths in a graph with a graphical interface.

СОДЕРЖАНИЕ

	Введение	5
1.	Требования к программе	7
1.1.	Исходные требования к программе	7
1.1.1	Требования к функциональности	7
1.1.2	Требования к визуализации	8
1.2.	Уточнение требований после сдачи прототипа	9
1.3.	Уточнение требований после сдачи 1-ой версии	9
1.4	Уточнение требований после сдачи 2-ой версии	9
2.	План разработки и распределение ролей в бригаде	10
2.1.	План разработки	10
2.2.	Распределение ролей в бригаде	10
3.	Особенности реализации	11
3.1.	Основные классы модели	11
3.2.	Основные классы графического интерфейса	15
4.	Тестирование	20
4.1	Тестирование графического интерфейса	20
4.2	Тестирование кода графа	25
4.3	Тестирование кода алгоритма	26
4.4	Тестирование кода сохранения графа	27
	Заключение	29
	Список использованных источников	30
	Приложение А. Исходный код программы	31

ВВЕДЕНИЕ

Основная цель – реализация визуализатора алгоритма Дейкстры поиска кратчайших путей в графе в виде приложения с графическим интерфейсом. Для её достижения требуется изучить основные понятия объектно-ориентированного языка программирования *Java*, составить спецификацию и план разработки, распределить обязанности в бригаде для выполнения итеративной разработки проекта.

Алгоритм Дейкстры используется для поиска кратчайших путей в графе. Он работает путем постепенного расширения множества вершин, для которых известен кратчайший путь от начальной вершины, выбора вершины с минимальным известным расстоянием и обновления расстояний до ее соседей. Этот процесс продолжается, пока не будут найдены кратчайшие пути до всех вершин в графе.

Псевдокод:

```
// V - множество вершин графа
// E - множество рёбер графа
// start - начальная вершина
// dist[] - массив минимальных расстояний от начальной вершины
// used[] - массив обработанных вершин
// w(i) - вес i-го ребра
// e.to - вершина, в которую ведёт ребро e от текущей вершины v

function dijkstra(start):
    for v ∈ V:
        dist[v] = ∞
        used[v] = false
    dist[start] = 0
    for i ∈ V:
        v = null
        for j ∈ V:
            if !used[j] and (v == null or dist[j] < dist[v]):
                v = j
        if dist[v] = ∞:
            break
        used[v] = true
        for e ∈ E (e - edge coming from vertex v):
            if dist[v] + w(e) < dist[e.to]:
                dist[e.to] = dist[v] + w(e)
```

Алгоритм Дейкстры используется в следующих системах:

- Сетевые маршрутизаторы: определение оптимальных маршрутов для передачи данных в компьютерных сетях;
- Географические информационные системы: нахождение кратчайших путей на картах (навигация);
- Игровая разработка: определение путей для персонажей и объектов в игровых мирах;
- Логистика и транспорт: оптимизация маршрутов доставки и передвижения транспорта.

1. ТРЕБОВАНИЯ К ПРОГРАММЕ

1.1 Исходные требования к программе

1.1.1 Требования к функциональности

1. Построение графа в среде приложения на холсте:
 - 1.1. Режим *“Edit Mode”*:
 - 1.1.1. Добавление вершины – клик ЛКМ;
 - 1.1.2. Перемещение вершины – зажатие ЛКМ на вершине и перемещение мыши;
 - 1.1.3. Добавление ребра – два последовательных клика ЛКМ по вершинам;
 - 1.1.4. Задание веса ребру – двойной клик ЛКМ по ребру и ввод веса.
 - 1.2. Режим *“Delete Mode”*: удаление вершины/ребра – клик ЛКМ по вершине/ребру.
 - 1.3. Действие *“Clear the Field”*: очистка холста.
 - 1.4. Действие *“Switch Type Graph”*: изменение ориентации графа (ориентированный/неориентированный).
2. Загрузка/сохранение графа из файла *.json*:
 - 2.1. Действие *“Load the Graph”*: загрузка графа.
 - 2.2. Действие *“Save the Graph”*: сохранение графа.
3. Выполнение алгоритма:
 - 3.1. Действие *“Run Completely”*: запуск алгоритма с начальной вершины без пошаговой визуализации.
 - 3.2. Действие *“Step Back”*: возврат к предыдущему шагу работы алгоритма.
 - 3.3. Действие *“Step Forward”*: переход к следующему шагу работы алгоритма.

1.1.2 Требования к визуализации

Визуализация работы алгоритма непосредственно на графе:

1. Выбранная начальная вершина графа обозначена цветом;
2. Текущая обрабатываемая вершина выделена цветом;
3. Текущее обрабатываемое ребро выделено цветом;
4. Текущая вершина-сосед выделена цветом;
5. Текущее новое расстояние отображается рядом с вершиной в виде неравенства (сравнение нового расстояния с текущим минимальным);
6. Минимальные расстояния отображены рядом с каждой вершиной.

Текстовые пояснения визуализации отображены в области вывода шагов алгоритма.

Запустить	Шаг назад	Шаг вперёд	Строить	Удалить	Импорт	Экспорт
Шаги работа алгоритма			Поле для построения и визуализации графа			

Рисунок 1 – схема графического интерфейса приложения

1.2 Уточнение требований после сдачи прототипа

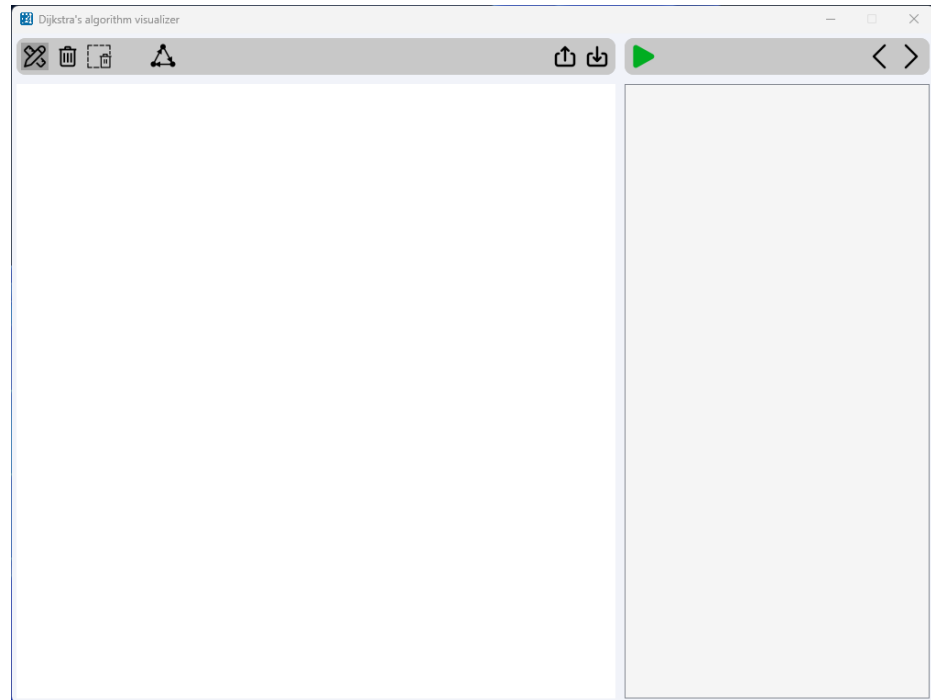


Рисунок 2 – графический интерфейс приложения после сдачи прототипа

1.3 Уточнение требований после сдачи 1-ой версии

- Режим “*Edit Mode*”: задание веса ребру – клик ПКМ по ребру и ввод веса;
- Окно ввода веса ребра должно поддерживать нажатие на клавиатуре “*Enter*” для подтверждения;
- Номера вершин должны выводиться поверх рёбер.

1.4 Уточнение требований после сдачи 2-ой версии

- Стартовая вершина должна сохраняться/загружаться из файла вместе с графом;
- При попытке запустить алгоритм без стартовой вершины требуется сообщить о том, что нужно её выбрать;
- Увеличить размер шрифта в области вывода текстовых пояснений.

2. ПЛАН РАЗРАБОТКИ И РАСПРЕДЕЛЕНИЕ РОЛЕЙ В БРИГАДЕ

2.1. План разработки

Дата	Этап проекта	Реализованные возможности	Выполнено
28.06.2024	Согласование спецификации	-	-
01.07.2024	Сдача прототипа	Графический интерфейс без функциональных возможностей	Запланированный функционал
03.07.2024	Сдача версии 1	Построение графа на холсте; запуск алгоритма без пошаговой визуализации; вывод текстовых пояснений работы алгоритма	Запланированный функционал
06.07.2024	Сдача версии 2	Пошаговое выполнение алгоритма; пошаговая визуализация; загрузка/сохранение графа в файл	Запланированный функционал
08.07.2024	Сдача версии 3	Внесение правок	Запланированный функционал
08.07.2024	Сдача отчёта	-	-
08.07.2024	Защита отчёта	-	-

Таблица 1 – План разработки

2.2. Распределение ролей в бригаде

- Долотов Никита – API графа, реализация алгоритма, интерфейс.
- Богатов Илья – Визуализация алгоритма, тестирование, оформление отчёта.
- Бочаров Глеб – Реализация двунаправленного поиска пути в алгоритме Дейкстры.

3. ОСОБЕННОСТИ РЕАЛИЗАЦИИ

3.1. Основные классы модели

Класс *Vertex*.

Содержит приватные поля: идентификатор вершины *id*, строковая метка *label*, координаты вершины *x* и *y*, цвет *color*.

Реализованы сеттеры и геттеры полей.

Реализован публичный метод *String toJSON()*, преобразующий объект вершины в *JSON*-строку и возвращает её.

Реализован публичный метод *Vertex fromJSON(String json)*, принимающий строку *json* и создающий из неё объект вершины.

Класс *Edge*.

Содержит приватные поля: начальная вершина *fromV*, конечная вершина *toV*, вес *weight*, цвет *color*.

Реализованы сеттеры и геттеры полей.

Реализован публичный метод *String toJSON()*; преобразует объект ребра в *JSON*-строку и возвращает её.

Реализован публичный метод *Edge fromJSON(String json, DirectedGraph graph)*, принимающий граф *graph*, строку *json*; создает из *json* объект вершины.

Класс *DirectedGraph*.

Содержит приватные поля: словарь вершин по идентификаторам *vertices*, словарь исходящих ребер по вершинам (список смежности) *adjacencyList*, флаг ориентированности графа *isDirected*, метка для следующей вершины *nextLabel*.

Реализован публичный метод *boolean isDirected()*; возвращает флаг, указывающий, является ли граф ориентированным.

Реализован публичный метод *void addVertex(Vertex vertex)*, принимающий вершину *vertex*; добавляет *vertex* в граф, присваивая метку.

Реализован публичный метод *void addEdge(Edge edge)*, принимающий ребро *edge*; добавляет *edge* в граф, если такого ребра ещё нет.

Реализован публичный метод *void removeVertex(Vertex vertex)*, принимающий вершину *vertex*; удаляет *vertex* и все рёбра, инцидентные ей, из графа.

Реализован публичный метод *void removeEdge(Edge edge)*, принимающий ребро *edge*; удаляет *edge* из графа.

Реализован публичный метод *void clear()*; очищает граф, удаляя все вершины и рёбра.

Реализован публичный метод *void setEdgeWeight(Edge edge, Integer weight)*, принимающий ребро *edge*, вес *weight*; устанавливает вес *weight* ребру *edge*.

Реализован публичный метод *List<Vertex> getVertices()*; возвращает список всех вершин в графе.

Реализован публичный метод *List<Edge> getEdgesFrom(Vertex vertex)*, принимающий вершину *vertex*; возвращает список всех рёбер, исходящих из *vertex*.

Реализован публичный метод *Vertex getVertexById(UUID id)*, принимающий идентификатор вершины *id*; возвращает вершину по её *id*.

Реализован публичный метод *Vertex getVertexByColor(Color color)*, принимающий цвет *color*; возвращает вершину по её *color*, если такая существует.

Реализован публичный метод *String toJSON()*; преобразует объект ориентированного графа в *JSON*-строку, включая информацию о вершинах и рёбрах, и возвращает её.

Реализован публичный метод *DirectedGraph fromJSON(String json)*, принимающий строку *json*; создает из *json* объект ориентированного графа, восстанавливая вершины и рёбра из *JSON*-данных.

Реализован приватный метод *void reassignLabels()*; переназначает метки вершинам после удаления вершины.

Класс *UndirectedGraph*.

Наследуется от *DirectedGraph*.

Переопределен публичный метод *void addEdge(Edge edge)*, принимающий ребро *edge*; добавляет *edge* в граф и автоматически добавляет обратное ребро, чтобы сохранить неориентированность.

Переопределен публичный метод *void removeEdge(Edge edge)*, принимающий ребро *edge*; удаляет *edge* и соответствующее ему обратное ребро.

Переопределен публичный метод *void setEdgeWeight(Edge edge, Integer weight)*, принимающий ребро *edge*, вес *weight*; устанавливает вес *weight* ребра *edge* и соответствующего ему обратного ребра.

Переопределен публичный метод *UndirectedGraph fromJSON(String json)*, принимающий строку *json*; создает из *json* объект неориентированного графа, восстанавливая вершины и рёбра из *JSON*-данных.

Класс *DijkstraState*.

Содержит приватные поля: словарь расстояний по вершинам *distances*, множество посещённых вершин *visited*, список шагов *steps*, текущую вершину *currentVertex*, текущее ребро *currentEdge*, соседнюю вершину *neighborVertex* и строку неравенства *inequality*.

Реализован конструктор и геттеры полей.

Класс *DijkstraAlgorithm*.

Содержит приватные поля: граф *graph*, словарь расстояний по вершинам *distances*, множество посещённых вершин *visited*, приоритетная очередь вершин *queue*, список шагов *steps*, список состояний *states*.

Реализован публичный метод *void process(Vertex start)*, принимающий вершину *start*; запускает алгоритм Дейкстры, начиная с указанной стартовой вершины; обрабатывает вершины, обновляет расстояния и сохраняет состояния на каждом шаге.

Реализован публичный метод *DijkstraState* *getState(Integer index)*, принимающий индекс *index*; возвращает состояние алгоритма по *index*.

Реализован публичный метод *Integer* *getNumberStates()*, возвращает количество сохранённых состояний алгоритма.

Реализован приватный метод *void* *saveState(Vertex vertex, Edge edge, Vertex neighbor, String inequality)*, принимающий текущую вершину *vertex*, текущее ребро *edge*, вершину-соседа *neighbor*, строку неравенства *inequality*; сохраняет текущее состояние алгоритма, включая словарь расстояний по вершинам, множество посещённых вершин, список шагов, текущую вершину, текущее ребро, соседнюю вершину и неравенство.

Класс *AlgorithmManager*.

Содержит приватные поля: объект приложения *app*, объект алгоритма *DijkstraAlgorithm dijkstra*, флаг выполнения алгоритма *isRun* и индекс текущего состояния *stateIndex*.

Реализован публичный метод *DijkstraState* *getState()*; возвращает текущее состояние алгоритма Дейкстры по индексу *stateIndex*.

Реализован публичный метод *boolean* *isRun()*; возвращает флаг выполнения алгоритма *isRun*.

Реализован публичный метод *void* *reset()*; сбрасывает алгоритм и граф в исходное состояние, обнуляет флаг выполнения и индекс состояния, устанавливает исходные цвета для вершин и рёбер.

Реализован публичный метод *void* *stepBack()*; выполняет шаг назад по состояниям алгоритма.

Реализован публичный метод *void* *stepForward()*; выполняет шаг вперёд по состояниям алгоритма

Реализован публичный метод *void* *runFull()*; полностью выполняет алгоритм, переходя к последнему шагу.

Реализован приватный метод *void* *run()*; запускает выполнение алгоритма Дейкстры, начиная с выбранной стартовой вершины, сохраняя состояния.

Реализован приватный метод *void update()*; обновляет текстовое пояснение шагов алгоритма и перерисовывает холст с графом.

3.2. Основные классы графического интерфейса

Класс *ControlPanelsManager*.

Содержит приватные поля: объект приложения *app*, объект *constraints* для размещения компонентов *gbc*, объект *ButtonsManager buttonsManager*, кнопки *editButton*, *deleteButton* и *switchGraphTypeButton*.

Реализован конструктор *ControlPanelsManager(App app)*, принимающий объект приложения *app*; инициализирует менеджер панелей управления с указанным приложением; создает кнопки и панели управления.

Реализован публичный метод *JToggleButton getEditButton()*; возвращает кнопку переключения режима редактирования.

Реализован публичный метод *JToggleButton getDeleteButton()*; возвращает кнопку переключения режима удаления.

Реализован приватный метод *void initControlPanelLeft()*; инициализирует левую панель управления, добавляя кнопки редактирования, удаления, переключения типа графа, сохранения и загрузки графа, а также добавляет обработчики событий для этих кнопок.

Реализован приватный метод *void initControlPanelRight()*; инициализирует правую панель управления, добавляя кнопки запуска алгоритма, шага назад и шага вперёд, а также добавляет обработчики событий для этих кнопок.

Реализован приватный метод *void deleteAll()*; очищает холст графа, удаляет все вершины и рёбра, сбрасывает алгоритм и перерисовывает холст.

Реализован приватный метод *void switchGraphType()*; переключает тип графа (ориентированный/неориентированный), очищает граф, сбрасывает алгоритм и обновляет иконку кнопки переключения типа графа.

Реализован приватный метод *void save()*; сохраняет текущий граф в *JSON*-файл через диалоговое окно сохранения файла.

Реализован приватный метод *void load()*; загружает граф из *JSON*-файла через диалоговое окно открытия файла, устанавливает загруженный граф в приложении и обновляет отображение.

Реализован приватный метод *void updateToggleButtonStyles(ButtonGroup toggleGroup)*, принимающий группу кнопок-переключателей *toggleGroup*; обновляет стили кнопок-переключателей в зависимости от их состояния (выбрана/не выбрана).

Реализован приватный метод *void changeSwitchGraphTypeButtonIcon(JButton button, boolean isDirected)*, принимающий кнопку *button*, флаг ориентированности графа *isDirected*; обновляет иконку и подсказку кнопки переключения типа графа в зависимости от текущего типа графа.

Реализован вложенный приватный класс *RoundedPanel*, наследуемый от *JPanel*; реализует закругленные панели с переопределенным методом *paintComponent()* для отрисовки панели с закругленными углами.

Класс *GraphFieldManager*.

Содержит приватные поля: объект приложения *app*, панель для отображения графа *graphField*, выбранную вершину *selectedVertex*, первую вершину для добавления ребра *firstVertex* и точку начального клика мыши *initClick*.

Реализован конструктор *GraphFieldManager(App app)*, принимающий объект приложения *app*; инициализирует менеджер поля графа с указанным приложением; создает панель для графа, добавляет слушатели мыши и настраивает отображение панели.

Реализован публичный метод *JPanel getGraphField()*; возвращает панель для отображения графа.

Реализован публичный метод *void setFirstVertex(Vertex vertex)*, принимающий первую вершину *vertex*; устанавливает первую вершину для добавления ребра.

Реализован публичный метод *Vertex getFirstVertex()*; возвращает первую вершину для добавления ребра.

Реализован публичный метод *void reset()*; сбрасывает выбранную вершину, первую вершину для добавления ребра и точку начального клика мыши.

Реализован приватный метод *void addVertex(Point point)*, принимающий точку *point*; добавляет вершину в граф по *point*, обновляет алгоритм и перерисовывает графическое поле.

Реализован приватный метод *void addEdge(Vertex from, Vertex to)*, принимающий начальную *from* и конечную *to* вершину; добавляет ребро в граф между *from* и *to*, обновляет алгоритм и перерисовывает графическое поле.

Реализован приватный метод *void removeVertex(Vertex vertex)*, принимающий вершину *vertex*; удаляет *vertex* из графа, обновляет алгоритм и перерисовывает графическое поле.

Реализован приватный метод *void removeEdge(Edge edge)*, принимающий ребро *edge*; удаляет *edge* из графа, обновляет алгоритм и перерисовывает графическое поле.

Реализован приватный метод *Vertex getVertexAt(Point point)*, принимающий точку *point*; возвращает вершину, находящуюся в *point*, или *null*, если вершина не найдена.

Реализован приватный метод *Edge getEdgeAt(Point point)*, принимающий точку *point*; возвращает ребро, находящееся в *point*, или *null*, если ребро не найдено.

Реализован приватный метод *boolean isPointOnLine(Point point, Point from, Point to)*, принимающий точки *point*, *from*, *to*; проверяет, находится ли *point* на линии между *from* и *to*.

Реализован вложенный приватный класс *GraphPainter*, наследуемый от *JPanel* (отрисовка графа); переопределяет метод *paintComponent(Graphics g)* для отрисовки вершин, рёбер и текущего состояния алгоритма Дейкстры.

Реализован вложенный приватный класс *MouseListener*, наследуемый от *MouseAdapter*; переопределяет слушатель событий мыши для обработки кликов и нажатий.

Реализован вложенный приватный класс *MouseMotionListener*, наследуемый от *MouseMotionAdapter*; переопределяет слушатель событий движения мыши для обработки перетаскивания вершин.

Класс *StepsFieldManager*.

Содержит приватные поля: объект приложения *app* и текстовую область *stepsField* для отображения шагов алгоритма.

Реализован конструктор *StepsFieldManager(App app)*; инициализирует менеджер поля шагов с указанным приложением; создаёт текстовую область *stepsField*, настраивает её внешний вид и прокручиваемую панель для неё, а также добавляет эту панель в интерфейс приложения.

Реализован публичный метод *void display()*; отображает текущие шаги алгоритма Дейкстры в текстовой области.

Реализован публичный метод *void clear()*; очищает текстовую область шагов.

Класс *App*.

Содержит приватные поля: объект *constraints* для размещения компонентов *gbc*, объект графа *graph*, и объекты менеджеров интерфейса *controlPanelsManager*, *graphFieldManager*, *stepsFieldManager* и *algorithmManager*.

Реализован сеттер графа *void setGraph(DirectedGraph graph)* и геттеры объекта *constraints* для размещения компонентов *GridBagConstraints* *getGBC()*, графа *DirectedGraph* *getGraph()*, менеджера панелей управления *ControlPanelsManager* *getControlPanelsManager()*, менеджера холста графа *GraphFieldManager* *getGraphFieldManager()*, менеджера поля шагов

StepsFieldManager *getStepsFieldManager()*, менеджера алгоритмов
AlgorithmManager *getAlgorithmManager()*.

Реализован конструктор *App()*; инициализирует приложение; настраивает основные параметры окна, инициализирует компоненты интерфейса и добавляет их в окно.

4. ТЕСТИРОВАНИЕ

4.1. Тестирование графического интерфейса

Было произведено ручное тестирование на двух операционных системах (*Linux Ubuntu* и *Windows*) всех элементов интерфейса, а именно:

- Режим изменения графа;
- Режим удаления графа;
- Воспроизведение алгоритма Дейкстры;
- Панели инструментов (*Clear the Field*, *Switch Graph Type*);
- Инструментов сохранения и загрузки графа;
- Окна логирования.

Во время тестирования режима изменения графа была проверена корректная работоспособность каждого инструмента: создание вершины, создание ребра, установка веса ребра, выбор начальной вершины для алгоритма Дейкстры (см. рис. 3), перетаскивание вершин графа (см. рис. 4), очистка графа, удаление ребра/вершины (см. рис. 5). Инструменты работают исправно, конфликта не возникает. При выборе инструмента, который отвечает за редактирование графа, графически подсвечивается какая кнопка была выбрана пользователем, при этом невозможно выбрать одновременно два режима, только один.

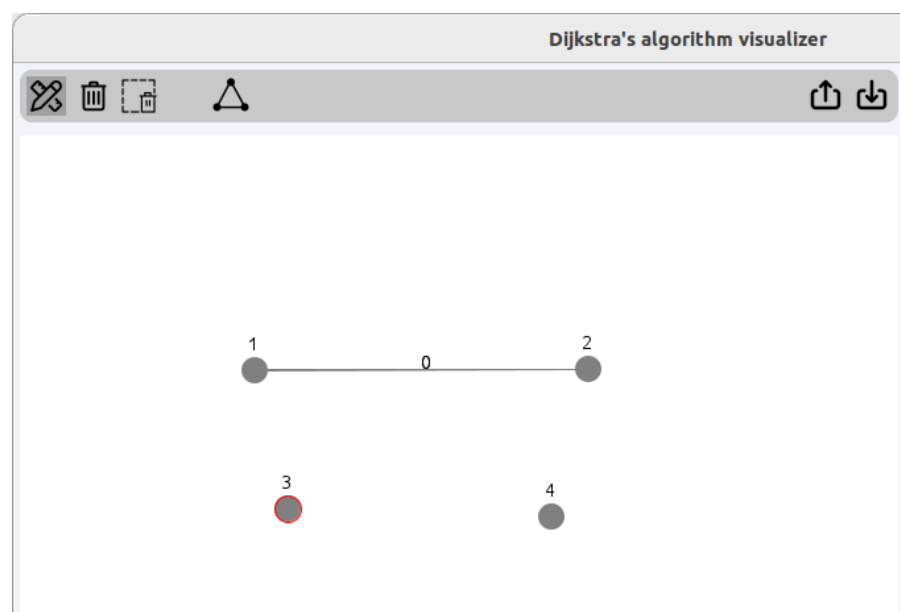


Рисунок 3 – Тест создания вершин, ребра, выбор начальной вершины

Во время тестирования воспроизведения алгоритма Дейкстры также была проверена работоспособность каждого инструмента: полный запуск (рис. 4), следующий шаг (см. рис. 6), предыдущий шаг (см. рис. 7). Если пользователь не выбрал начальную вершину – появляется окно с предупреждением о том, что требуется выбрать начальную вершину. Были протестированы ситуации, когда пользователь, во время проигрывания алгоритма Дейкстры, начинает менять настройки или как-либо взаимодействовать с инструментами, которые ему доступны, никаких конфликтов не возникает, в случае изменения графа или начальной вершины алгоритм прекращает свою работу, пользователю необходимо заново его начать, так как он изменил какие-либо настройки. В случае, когда настройки не были изменены – алгоритм продолжает воспроизводиться.

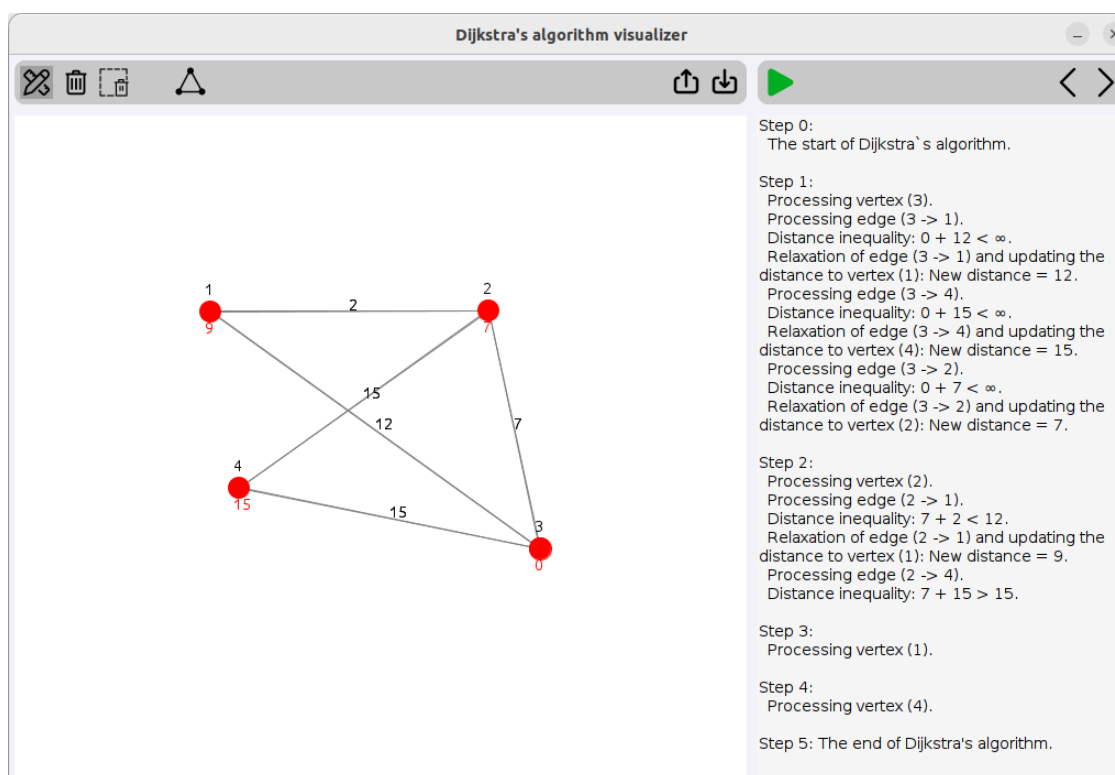


Рисунок 4 – Тест перетаскивания вершин и работы алгоритма Дейкстры

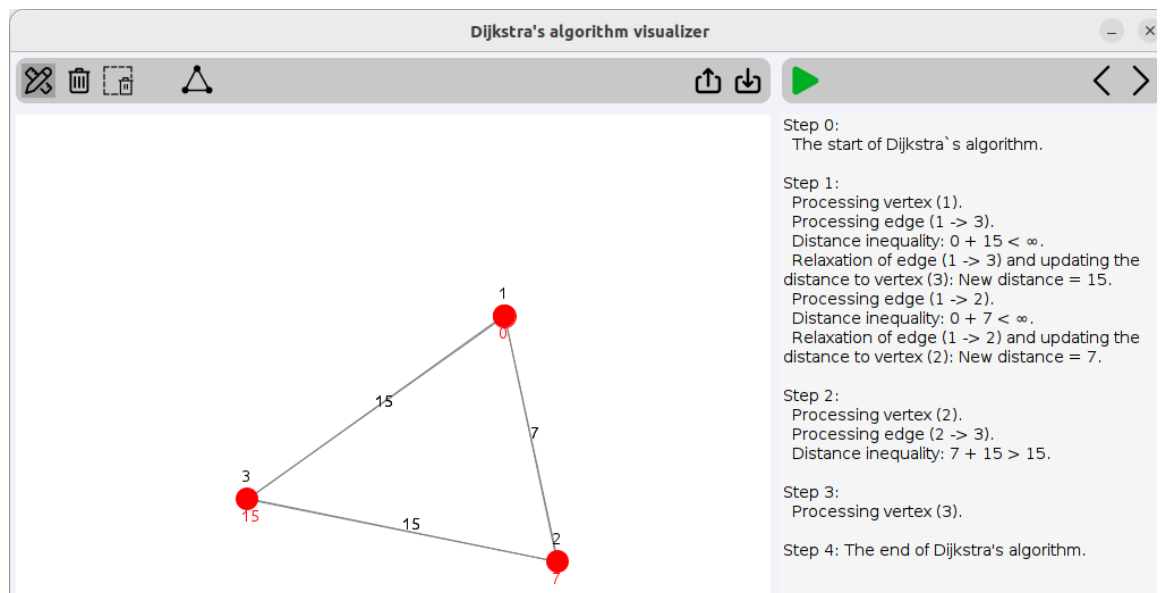


Рисунок 5 – Тест удаления вершины и повторного запуска алгоритма

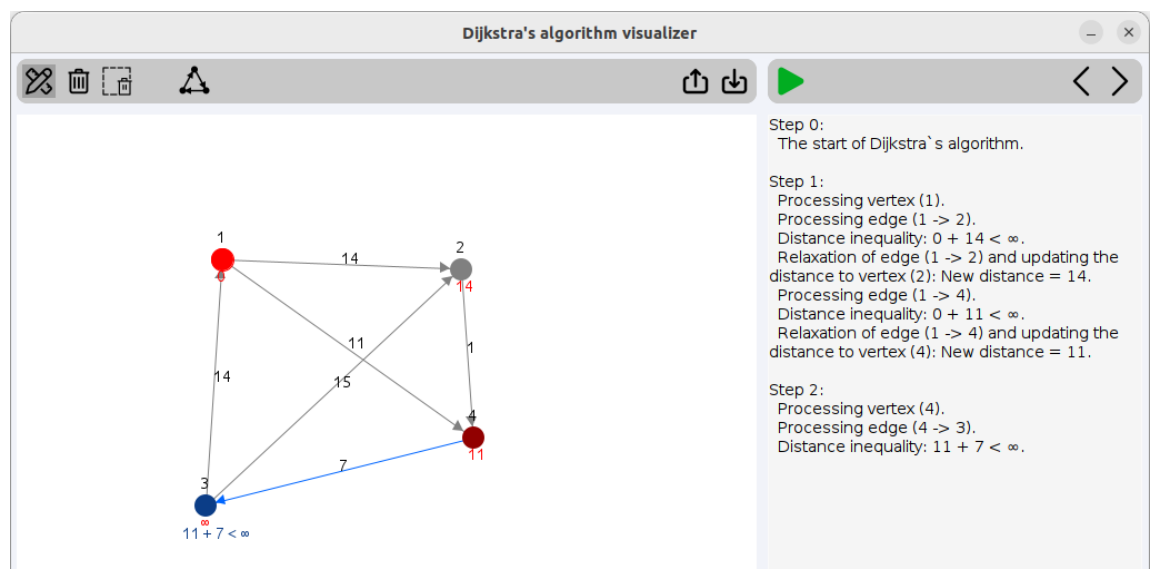


Рисунок 6 – Тест воспроизведения работы алгоритма по шагам

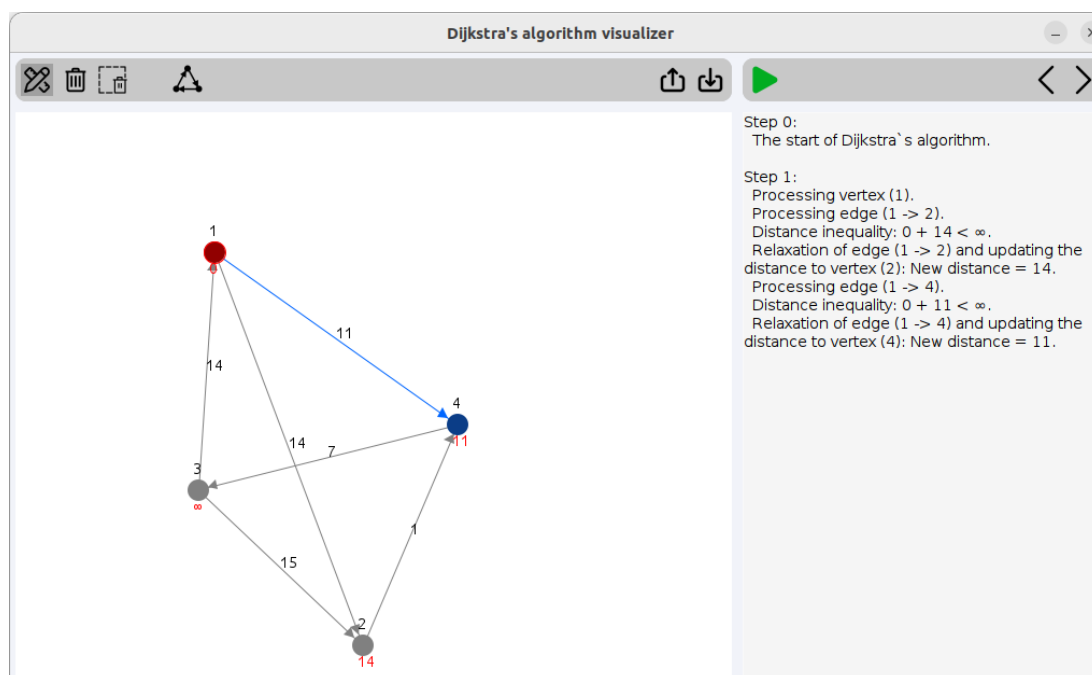


Рисунок 7 – Тест перемещения вершины во время работы алгоритма, выбор шага назад

Инструменты сохранения и загрузки работают корректно. При сохранении графа с помощью инструмента “*Save the Graph*” (см. рис. 8) – будет открыт проводник, где нужно выбрать путь сохранения. Загрузка происходит при помощи инструмента “*Load the Graph*” (см. рис. 9, 10). Расширение у сохраняемых файлов будет *.json*. Инструмент загрузки файлов будет загружать только целые файлы формата *.json*, то есть, если файл был поврежден каким-либо образом – приложение не сможет его загрузить, конфликта не возникнет.

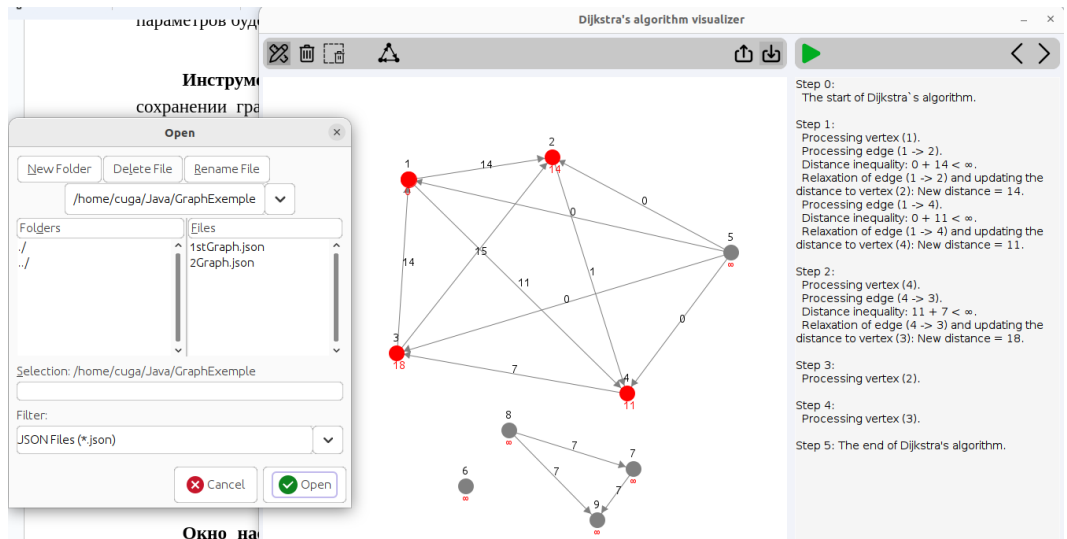


Рисунок 8 – Тестирование сохранения графа в файл

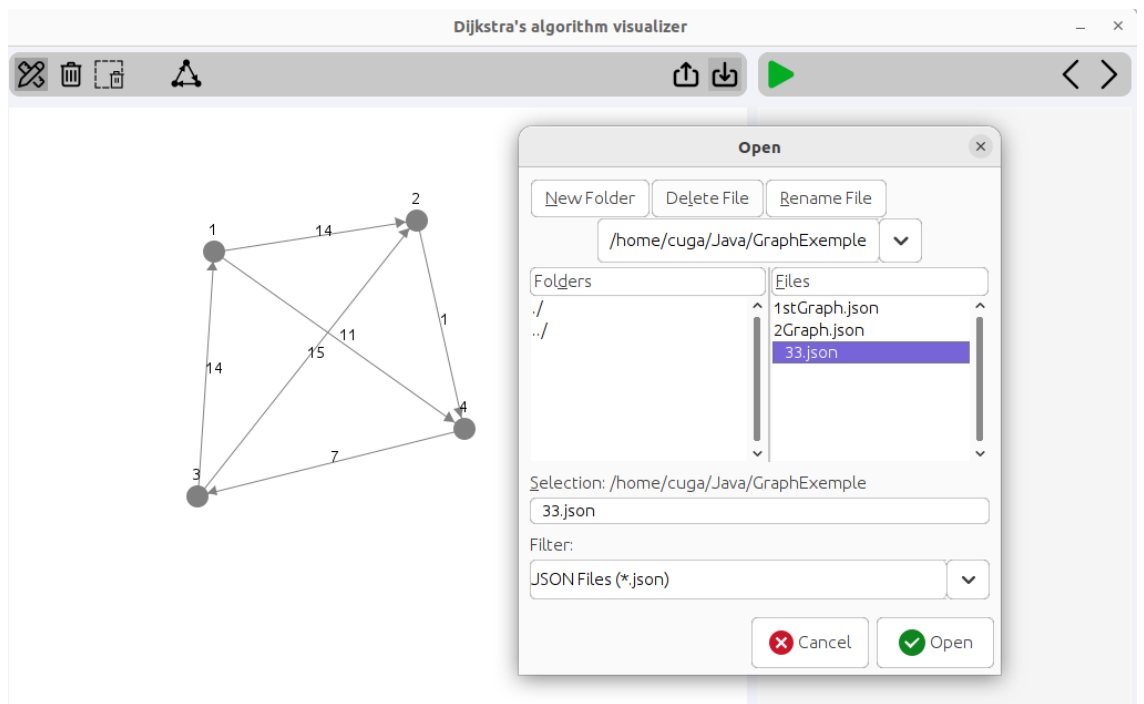


Рисунок 9 – Тестирование загрузки графа из файла

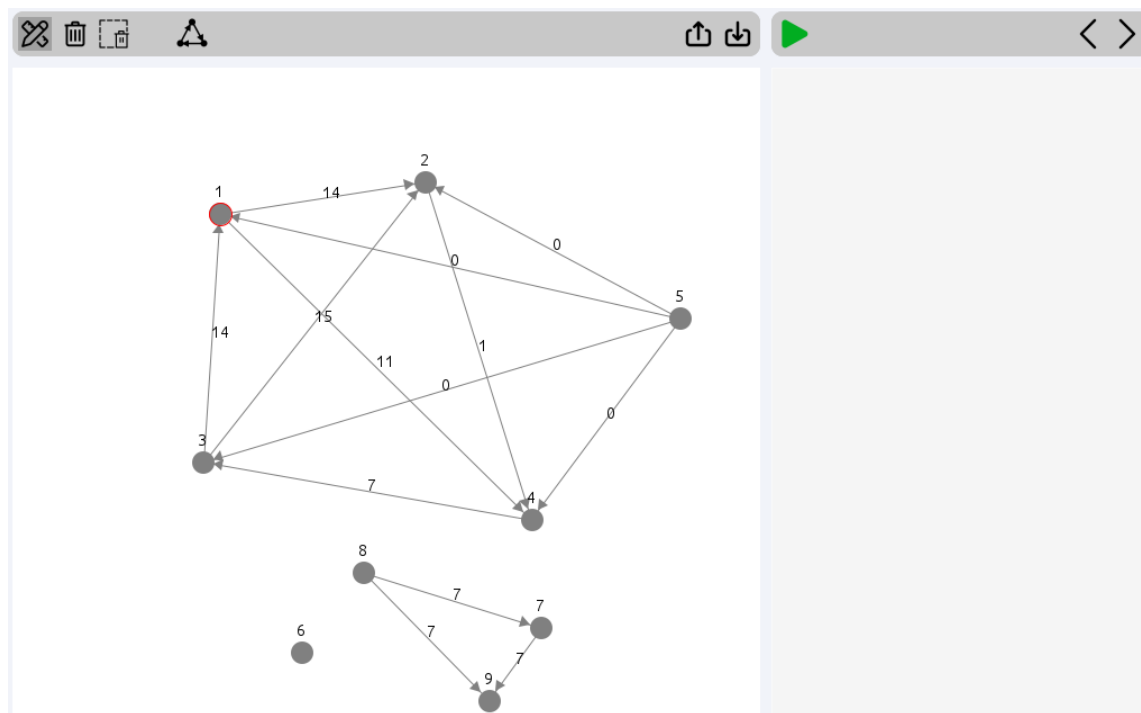


Рисунок 10 – Тестирование загрузки графа из файла

Окно логирования не доступно для редактирования пользователем, что было протестировано. Любое действие пользователя записывается в окно логирования.

4.2. Тестирования кода графа

Тестирования реализации графа подразумевает под собой создание двух групп автоматического тестирования для каждого вида графа, т.к. реализация некоторых функций у разных типов разная.

Для тестирования использовалась библиотека *JUnit*, которая сильно упростила этот процесс. Удобной возможностью оказалась функция, которая вызывается перед каждым тестом, в которой производился сброс графа.

Каждая функция в обоих видах тестировалась минимум тремя тестами: тестом на нормальную работу, на получение ошибки при передаче *null* значения или на получения ошибки при других обстоятельствах.

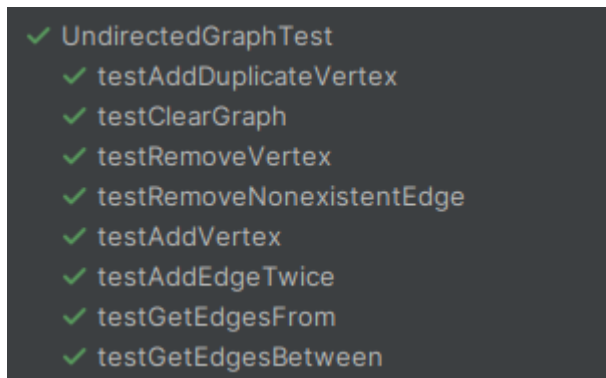


Рисунок 11 – Тесты для неориентированного графа

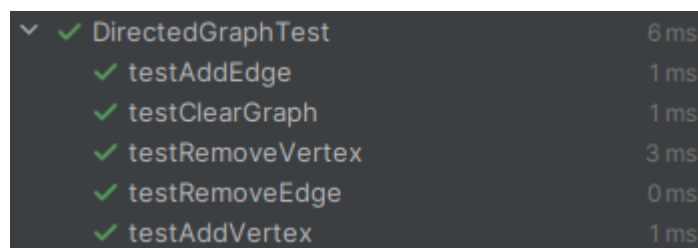


Рисунок 12 – Тесты для ориентированного графа

4.3. Тестирование кода алгоритма

Для тестирования кода алгоритма были созданы несколько автоматических тестов с помощью библиотеки *JUnit*. С её помощью тестировался сам алгоритм на ориентированном и неориентированном графе, а также ситуации, когда путь не существует, либо же когда существуют несколько кратчайших путей. Сами тесты подбирались, чтобы охватывать все описанные случаи.

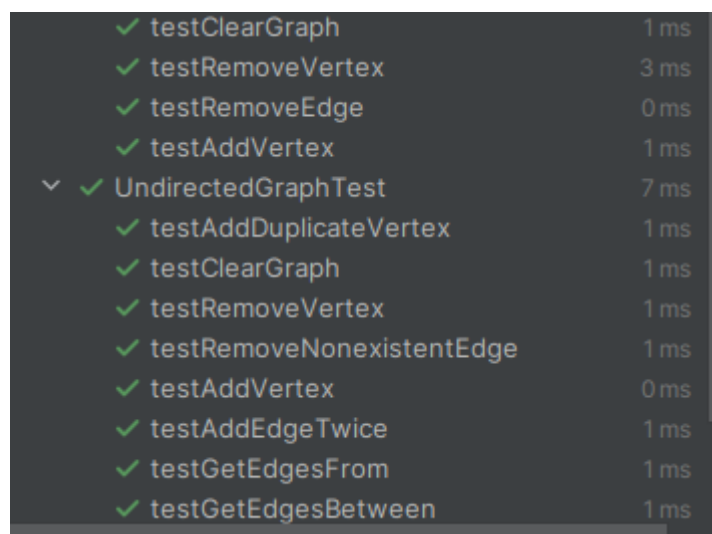


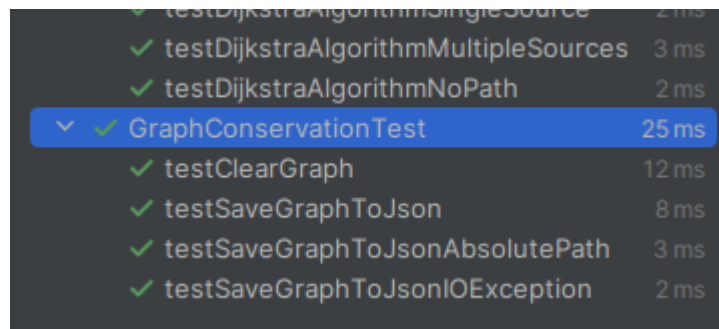
Рисунок 13 – Тестирование алгоритма на неориентированном графе

✓ EdgeTest	61 ms
✓ testEdgeSetters	60 ms
✓ testEdgeInitialization	0 ms
✓ testNotEqualsMethod	1 ms
✓ VertexTest	0 ms
✓ testVertex	0 ms
✓ DijkstraStateTest	3 ms
✓ testDijkstraStateCopy	1 ms
✓ testDijkstraStateInitialization	1 ms
✓ testDijkstraStateWithEmptyParameters	1 ms
✓ DirectedGraphTest	6 ms
✓ testAddEdge	1 ms
✓ testClearGraph	1 ms
✓ testRemoveVertex	3 ms
✓ testRemoveEdge	0 ms
✓ testAddVertex	1 ms

Рисунок 14 – Тестирование алгоритма на ориентированном графе

4.4. Тестирование кода сохранения графа

Для тестирования кода сохранения графа были созданы несколько автоматических тестов с помощью библиотеки *JUnit*. С её помощью тестировалась как загрузка, так и записи графа в текстовый файл формата *JSON*, причем сохранение тестировалось в разных директориях, как относительно запущенной программы, так и абсолютно начиная с дисков, а также ситуации, если невозможно открыть файл. Сами тесты были подобраны таким образом, чтобы охватить все описанные случаи, причем над ними была проведена дополнительная настройка, чтобы порядок выполнения тестов соответствовал их порядку в коде, так как некоторые из них взаимосвязаны. К тому же, после выполнения тестов все созданные в тестах файлы с графами автоматически удаляются с компьютера.



✓	testDijkstraAlgorithmSingleSource	2 ms
✓	testDijkstraAlgorithmMultipleSources	3 ms
✓	testDijkstraAlgorithmNoPath	2 ms
✓	GraphConservationTest	25 ms
✓	testClearGraph	12 ms
✓	testSaveGraphToJson	8 ms
✓	testSaveGraphToJsonAbsolutePath	3 ms
✓	testSaveGraphToJsonIOException	2 ms

Рисунок 15 – Тестирование сохранения графа

ЗАКЛЮЧЕНИЕ

В ходе выполнения мини-проекта было реализовано приложение с графическим интерфейсом, содержащее графический редактор графов, демонстрирующее пошаговое выполнение алгоритма Дейкстры поиска кратчайших путей в графе. Получены навыки программирования на объектно-ориентированном языке программирования *Java*.

Разработанное приложение соответствует требованиям, предъявленным в начале работы, а также уточнениям, которые были получены в процессе итеративной разработки.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Репозиторий бригады // *GitHub*. URL: https://github.com/Do1otov/Dijkstra_Algo_Visualization.
2. Система вопросов и ответов о программировании // *Stack Overflow*. URL: <https://stackoverflow.com/> (дата обращения: 30.06.2024, 01.07.2024, 05.07.2024).
3. Стоковая графика для дизайна // *Icons8*. URL: <https://icons8.ru/> (дата обращения: 30.06.2024).
4. Википедия конспектов // *neerc.ifmo*. URL: https://neerc.ifmo.ru/wiki/index.php?title=%D0%90%D0%BB%D0%B3%D0%BE%D1%80%D0%B8%D1%82%D0%BC_%D0%94%D0%B5%D0%B9%D0%BA%D1%81%D1%82%D1%80%D1%8B (дата обращения: 01.07.2024).
5. Платформа для изучения программирования на *Java* // *JavaRush*. URL: <https://javarush.com/> (дата обращения: 02.07.2024, 05.07.2024).
6. Платформа для изучения программирования на *Java* // *Progoschool*. URL: <https://progoschool.ru/java/java-swing/> (дата обращения: 30.06.2024).

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: *Vertex.java*

```
package model;

import java.awt.*;
import java.util.UUID;

public class Vertex {
    private UUID id;
    private String label;
    private Integer x;
    private Integer y;
    private Color color;

    public Vertex(String label, Integer x, Integer y, Color color) {
        this.id = UUID.randomUUID();
        this.label = label;
        this.x = x;
        this.y = y;
        this.color = color;
    }

    public void setLabel(String label) {
        this.label = label;
    }

    public void setX(Integer x) {
        this.x = x;
    }

    public void setY(Integer y) {
        this.y = y;
    }

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

    public UUID getId() {
        return id;
    }

    public String getLabel() {
        return label;
    }

    public Integer getX() {
        return x;
    }

    public Integer getY() {
        return y;
    }

    public Color getColor() {
        return color;
    }

    public String toJSON() {
```

```

        return String.format(
            "{\\\"id\\\":\\\"%s\\\",\\\"label\\\":\\\"%s\\\",\\\"x\\\":%d,\\\"y\\\":%d,\\\"color\\\":\\\"%d\\\"}",
            id.toString(), label, x, y, color.getRGB()
        );
    }

    public static Vertex fromJSON(String json) {
        String[] fields = json.replaceAll("[{}\\\"]", "").split(",");
        UUID id = UUID.fromString(fields[0].split(":")[1]);
        String label = fields[1].split(":")[1];
        Integer x = Integer.parseInt(fields[2].split(":")[1]);
        Integer y = Integer.parseInt(fields[3].split(":")[1]);
        Color color = new Color(Integer.parseInt(fields[4].split(":")[1]));

        Vertex vertex = new Vertex(label, x, y, color);
        vertex.id = id;
        return vertex;
    }
}

```

Название файла: *Edge.java*

```

package model;

import java.awt.*;
import java.util.UUID;

public class Edge {
    private final Vertex fromV;
    private final Vertex toV;
    private Integer weight;
    private Color color;

    public Edge(Vertex fromV, Vertex toV, Color color) {
        this.fromV = fromV;
        this.toV = toV;
        this.weight = 0;
        this.color = color;
    }

    public void setWeight(Integer weight) {
        this.weight = weight;
    }

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

    public Vertex getFromV() {
        return fromV;
    }

    public Vertex getToV() {
        return toV;
    }

    public Integer getWeight() {
        return weight;
    }

    public Color getColor() {
        return color;
    }
}

```



```

    }

    public String toJSON() {
        return String.format(
            "{\\"from\\":\\"%s\\",\\"to\\":\\"%s\\",\\"weight\\":%d,\\"color\\":\\"%d\\"}",
            fromV.getId().toString(), toV.getId().toString(), weight,
            color.getRGB()
        );
    }

    public static Edge fromJSON(String json, DirectedGraph graph) {
        String[] fields = json.replaceAll("[{}\\"]", "").split(",");
        UUID fromId = UUID.fromString(fields[0].split(":")[1]);
        UUID toId = UUID.fromString(fields[1].split(":")[1]);
        Integer weight = Integer.parseInt(fields[2].split(":")[1]);
        Color color = new Color(Integer.parseInt(fields[3].split(":")[1]));

        Vertex fromV = graph.getVertexById(fromId);
        Vertex toV = graph.getVertexById(toId);

        Edge edge = new Edge(fromV, toV, color);
        edge.setWeight(weight);
        return edge;
    }
}

```

Название файла: *DirectedGraph.java*

```

package model;

import java.awt.*;
import java.util.*;
import java.util.List;

public class DirectedGraph {
    private final Map<UUID, Vertex> vertices;
    private final Map<Vertex, List<Edge>> adjacencyList;
    protected boolean isDirected;
    private Integer nextLabel;

    public DirectedGraph() {
        this.vertices = new HashMap<>();
        this.adjacencyList = new HashMap<>();
        this.isDirected = true;
        this.nextLabel = 1;
    }

    public boolean isDirected() {
        return isDirected;
    }

    public void addVertex(Vertex vertex) {
        if (vertex.getLabel() == null || vertex.getLabel().isEmpty()) {
            vertex.setLabel(String.valueOf(nextLabel++));
        } else {
            int vertexLabel = Integer.parseInt(vertex.getLabel());
            if (vertexLabel >= nextLabel) {
                nextLabel = vertexLabel + 1;
            }
        }
        vertices.put(vertex.getId(), vertex);
        adjacencyList.put(vertex, new ArrayList<>());
    }
}

```

```

    }

    public void addEdge(Edge edge) {
        List<Edge> edgesFrom = adjacencyList.get(edge.getFromV());
        for (Edge e : edgesFrom) {
            if (e.getToV().equals(edge.getToV())) {
                return;
            }
        }
        edgesFrom.add(edge);
    }

    public void removeVertex(Vertex vertex) {
        Vertex removed = vertices.remove(vertex.getId());
        if (removed != null) {
            adjacencyList.remove(removed);
            for (List<Edge> edges : adjacencyList.values()) {
                edges.removeIf(e -> e.getToV().equals(removed));
            }
            reassignLabels();
        }
    }

    public void removeEdge(Edge edge) {
        List<Edge> edgesFrom = adjacencyList.get(edge.getFromV());
        if (edgesFrom != null) {
            edgesFrom.remove(edge);
        }
    }

    public void clear() {
        vertices.clear();
        adjacencyList.clear();
        nextLabel = 1;
    }

    public void setEdgeWeight(Edge edge, Integer weight) {
        edge.setWeight(weight);
    }

    public List<Vertex> getVertices() {
        return new ArrayList<>(vertices.values());
    }

    public List<Edge> getEdgesFrom(Vertex vertex) {
        return new ArrayList<>(adjacencyList.get(vertex));
    }

    public Vertex getVertexById(UUID id) {
        return vertices.get(id);
    }

    public Vertex getVertexByColor(Color color) {
        for (Vertex vertex : vertices.values()) {
            if (vertex.getColor().equals(color)) {
                return vertex;
            }
        }
        return null;
    }

    public String toJSON() {
        StringBuilder verticesJSON = new StringBuilder();

```

```

        StringBuilder edgesJSON = new StringBuilder();

        for (Vertex vertex : vertices.values()) {
            verticesJSON.append(vertex.toJSON()).append(",");
        }

        for (List<Edge> edges : adjacencyList.values()) {
            for (Edge edge : edges) {
                edgesJSON.append(edge.toJSON()).append(",");
            }
        }

        if (!verticesJSON.isEmpty())
verticesJSON.setLength(verticesJSON.length() - 1);
        if (!edgesJSON.isEmpty()) edgesJSON.setLength(edgesJSON.length() - 1);

        return
String.format("{\\"isDirected\":%b,\\"nextLabel\":%d,\\"vertices\":[%s],\\"edges\":[
%s]}",
                isDirected, nextLabel, verticesJSON, edgesJSON);
    }

    public static DirectedGraph fromJSON(String json) {
        DirectedGraph graph = new DirectedGraph();
        json = json.trim();

        String isDirectedStr = json.substring(json.indexOf("\\"isDirected\":")) +
13, json.indexOf(",", json.indexOf("\\"isDirected\":")));
        graph.isDirected = Boolean.parseBoolean(isDirectedStr.trim());

        String nextLabelStr = json.substring(json.indexOf("\\"nextLabel\":")) +
12, json.indexOf(",", json.indexOf("\\"nextLabel\":")));
        graph.nextLabel = Integer.parseInt(nextLabelStr.trim());

        int verticesStart = json.indexOf("\\"vertices\":[") + 12;
        int verticesEnd = json.indexOf("]",",", verticesStart);
        String verticesData = json.substring(verticesStart, verticesEnd);

        if (!verticesData.isEmpty()) {
            String[] vertexArray = verticesData.split("\\},");
            for (String vertexData : vertexArray) {
                vertexData = vertexData.endsWith("}") ? vertexData : vertexData
+ "}";

                Vertex vertex = Vertex.fromJSON(vertexData);
                graph.addVertex(vertex);
            }
        }

        int edgesStart = json.indexOf("\\"edges\":[") + 9;
        int edgesEnd = json.lastIndexOf("]");
        String edgesData = json.substring(edgesStart, edgesEnd);

        if (!edgesData.isEmpty()) {
            String[] edgeArray = edgesData.split("\\},");
            for (String edgeData : edgeArray) {
                edgeData = edgeData.endsWith("}") ? edgeData : edgeData + "}";
                Edge edge = Edge.fromJSON(edgeData, graph);
                graph.addEdge(edge);
            }
        }

        return graph;
    }
}

```

```

private void reassignLabels() {
    List<Vertex> vertexList = new ArrayList<>(vertices.values());
    vertexList.sort(Comparator.comparing(Vertex::getLabel));
    int label = 1;
    for (Vertex vertex : vertexList) {
        vertex.setLabel(String.valueOf(label++));
    }
    nextLabel = label;
}
}

```

Название файла: *UndirectedGraph.java*

```

package model;

import java.util.*;

public class UndirectedGraph extends DirectedGraph {
    public UndirectedGraph() {
        super();
        this.isDirected = false;
    }

    @Override
    public void addEdge(Edge edge) {
        super.addEdge(edge);
        Edge reversed = new Edge(edge.getToV(), edge.getFromV(),
edge.getColor());
        reversed.setWeight(edge.getWeight());
        super.addEdge(reversed);
    }

    @Override
    public void removeEdge(Edge edge) {
        super.removeEdge(edge);
        List<Edge> edgesFrom = super.getEdgesFrom(edge.getToV());
        Edge reversed = null;
        for (Edge e : edgesFrom) {
            if (e.getToV().equals(edge.getFromV())) {
                reversed = e;
                break;
            }
        }
        if (reversed != null) {
            super.removeEdge(reversed);
        }
    }

    @Override
    public void setEdgeWeight(Edge edge, Integer weight) {
        super.setEdgeWeight(edge, weight);
        List<Edge> edgesFrom = super.getEdgesFrom(edge.getToV());
        for (Edge e : edgesFrom) {
            if (e.getToV().equals(edge.getFromV())) {
                super.setEdgeWeight(e, weight);
                break;
            }
        }
    }

    public static UndirectedGraph fromJSON(String json) {
        DirectedGraph tempGraph = DirectedGraph.fromJSON(json);
        UndirectedGraph undirectedGraph = new UndirectedGraph();
    }
}

```

```

        for (Vertex vertex : tempGraph.getVertices()) {
            undirectedGraph.addVertex(vertex);
        }
        for (Vertex vertex : tempGraph.getVertices()) {
            for (Edge edge : tempGraph.getEdgesFrom(vertex)) {
                undirectedGraph.addEdge(edge);
            }
        }
        return undirectedGraph;
    }
}

```

Название файла: *DijkstraState.java*

```

package model;

import java.util.*;

public class DijkstraState {
    private final Map<Vertex, Integer> distances;
    private final Set<Vertex> visited;
    private final List<String> steps;

    private final Vertex currentVertex;
    private final Edge currentEdge;
    private final Vertex neighborVertex;
    private final String inequality;

    public DijkstraState(Map<Vertex, Integer> distances, Set<Vertex> visited,
        ArrayList<String> steps, Vertex currentVertex, Edge currentEdge, Vertex
        neighborVertex, String inequality) {
        this.distances = distances;
        this.visited = visited;
        this.steps = steps;
        this.currentVertex = currentVertex;
        this.currentEdge = currentEdge;
        this.neighborVertex = neighborVertex;
        this.inequality = inequality;
    }

    public Map<Vertex, Integer> getDistances() {
        return distances;
    }

    public Set<Vertex> getVisited() {
        return visited;
    }

    public List<String> getSteps() {
        return steps;
    }

    public Vertex getCurrentVertex() {
        return currentVertex;
    }

    public Edge getCurrentEdge() {
        return currentEdge;
    }

    public Vertex getNeighborVertex() {
        return neighborVertex;
    }
}

```

```

    }

    public String getInequality() {
        return inequality;
    }
}

```

Название файла: *DijkstraAlgorithm.java*

```

package model;

import java.util.*;

public class DijkstraAlgorithm {
    private final DirectedGraph graph;
    private final Map<Vertex, Integer> distances;
    private final Set<Vertex> visited;
    private final PriorityQueue<Vertex> queue;

    private final List<String> steps;
    private final List<DijkstraState> states;

    public DijkstraAlgorithm(DirectedGraph graph) {
        this.graph = graph;
        this.distances = new HashMap<>();
        this.visited = new HashSet<>();
        this.queue = new
PriorityQueue<>(Comparator.comparingInt(distances::get));

        this.steps = new ArrayList<>();
        this.states = new ArrayList<>();

        for (Vertex vertex : graph.getVertices()) {
            distances.put(vertex, Integer.MAX_VALUE);
        }
    }

    public void process(Vertex start) {
        steps.add("Step 0:\n The start of Dijkstra`s algorithm.\n");
        saveState(null, null, null, null);

        distances.put(start, 0);
        queue.add(start);
        int count = 1;

        while (!queue.isEmpty()) {
            Vertex curr = queue.poll();
            if (visited.contains(curr)) {
                continue;
            }
            visited.add(curr);

            steps.add(String.format("\nStep %d:\n Processing vertex (%s).\n",
count, curr.getLabel()));
            saveState(curr, null, null, null);

            for (Edge edge : graph.getEdgesFrom(curr)) {
                Vertex neighbor = edge.getToV();
                if (!visited.contains(neighbor)) {
                    steps.add(String.format(" Processing edge (%s -> %s).\n",
curr.getLabel(), neighbor.getLabel()));
                    saveState(curr, edge, neighbor, null);
                }
            }
        }
    }
}

```

```

        int newDist = distances.get(curr) + edge.getWeight();
        String stringSign = newDist < distances.get(neighbor) ? " <
" : " > ";
        String stringDist = distances.get(neighbor) <
Integer.MAX_VALUE ? distances.get(neighbor).toString() : "∞";
        String inequality = distances.get(curr) + " + " +
edge.getWeight() + stringSign + stringDist;

        steps.add(String.format(" Distance inequality: %s.\n",
inequality));
        saveState(curr, edge, neighbor, inequality);

        if (newDist < distances.get(neighbor)) {
            distances.put(neighbor, newDist);
            queue.add(neighbor);

            steps.add(String.format(" Relaxation of edge (%s -> %s)
and updating the distance to vertex (%s): New distance = %d.\n",
curr.getLabel(), neighbor.getLabel(), neighbor.getLabel(), newDist));
            saveState(curr, edge, neighbor, null);
        }
    }
    count++;
}
steps.add(String.format("\nStep %d: The end of Dijkstra's algorithm.\n",
count));
saveState(null, null, null, null);
}

private void saveState(Vertex vertex, Edge edge, Vertex neighbor, String
inequality) {
    states.add(new DijkstraState(new HashMap<>(distances), new
HashSet<>(visited), new ArrayList<>(steps), vertex, edge, neighbor,
inequality));
}

public DijkstraState getState(Integer index) {
    return states.get(index);
}

public Integer getNumberStates() {
    return states.size();
}
}

```

Название файла: *AlgorithmManager.java*

```

package model;

import gui.*;

import static gui.Settings.*;

public class AlgorithmManager {
    private final App app;

    private DijkstraAlgorithm dijkstra;
    private boolean isRun;
    private Integer stateIndex;

    public AlgorithmManager(App app) {
        this.app = app;
    }
}

```

```

        this.isRun = false;
        this.stateIndex = 0;
    }

    public DijkstraState getState() {
        return dijkstra.getState(stateIndex);
    }

    public boolean isRun() {
        return isRun;
    }

    public void reset() {
        dijkstra = null;
        isRun = false;
        stateIndex = 0;

        for (Vertex vertex : app.getGraph().getVertices()) {
            vertex.setColor(VERTEX_COLOR);
            for (Edge edge : app.getGraph().getEdgesFrom(vertex)) {
                edge.setColor(EDGE_COLOR);
            }
        }
        app.getStepsFieldManager().clear();
    }

    public void stepBack() {
        if (isRun && stateIndex > 0) {
            stateIndex--;
            update();
        }
    }

    public void stepForward() {
        if (stateIndex.equals(0)) {
            run();
            if (isRun) {
                stateIndex++;
                update();
            }
        } else if (isRun && stateIndex < dijkstra.getNumberStates() - 1) {
            stateIndex++;
            update();
        }
    }

    public void runFull() {
        run();
        if (isRun) {
            stateIndex = dijkstra.getNumberStates() - 1;
            update();
        }
    }

    private void run() {
        Vertex startVertex = app.getGraphFieldManager().getFirstVertex();
        if (startVertex == null) {
            CustomMessageDialog.showMessageDialog(app.getGraphFieldManager().getGraphField(),
            "Error", "Start vertex is not selected.", 250, 100);
            return;
        }
    }

```



```

        reset();
        isRun = true;
        dijkstra = new DijkstraAlgorithm(app.getGraph());
        dijkstra.process(startVertex);
    }

    private void update() {
        app.getStepsFieldManager().display();
        app.getGraphFieldManager().getGraphField().repaint();
    }
}

```

Название файла: *App.java*

```

package gui;

import model.*;
import static gui.Settings.*;

import javax.swing.*;
import java.awt.*;

public class App extends JFrame {
    private final GridBagConstraints gbc;

    private DirectedGraph graph;

    private final ControlPanelsManager controlPanelsManager;
    private final GraphFieldManager graphFieldManager;
    private final StepsFieldManager stepsFieldManager;
    private final AlgorithmManager algorithmManager;

    public App() {
        setTitle("Dijkstra's algorithm visualizer");
        setSize(1024, 768);
        setResizable(false);
        setLocationRelativeTo(null);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setLayout(new GridBagLayout());
        ImageIcon icon = new ImageIcon(getClass().getResource("/app.png"));
        setIconImage(icon.getImage());
        getContentPane().setBackground(APP_COLOR);

        gbc = new GridBagConstraints();
        gbc.insets = new Insets(5, 5, 5, 5);

        UIManager.put("ToolTip.background", APP_COLOR);
        UIManager.put("ToolTip.foreground", TITLE_COLOR);
        UIManager.put("ToolTip.border",
            BorderFactory.createLineBorder(CONTROL_PANEL_COLOR));

        this.graph = new DirectedGraph();

        controlPanelsManager = new ControlPanelsManager(this);
        graphFieldManager = new GraphFieldManager(this);
        stepsFieldManager = new StepsFieldManager(this);
        algorithmManager = new AlgorithmManager(this);
    }

    public GridBagConstraints getGBC() {
        return gbc;
    }
}

```

```

    public void setGraph(DirectedGraph graph) {
        this.graph = graph;
    }

    public DirectedGraph getGraph() {
        return graph;
    }

    public ControlPanelsManager getControlPanelsManager() {
        return controlPanelsManager;
    }

    public GraphFieldManager getGraphFieldManager() {
        return graphFieldManager;
    }

    public StepsFieldManager getStepsFieldManager() {
        return stepsFieldManager;
    }

    public AlgorithmManager getAlgorithmManager() {
        return algorithmManager;
    }

    public static void main(String[] args) {
        try {
            UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
        } catch (ClassNotFoundException | InstantiationException |
IllegalAccessRuntimeException | UnsupportedOperationException e) {
            e.printStackTrace();
        }

        SwingUtilities.invokeLater(() -> {
            App app = new App();
            app.setVisible(true);
        });
    }
}

```

Название файла: *ControlPanelsManager.java*

```

package gui;

import model.*;
import static gui.Settings.*;

import javax.swing.*.*;
import javax.swing.filechooser.FileFilter;
import java.awt.*.*;
import java.io.*;
import java.io.FileReader;
import java.io.FileWriter;
import java.io.IOException;
import java.util.Collections;

public class ControlPanelsManager {
    private final App app;

    private final GridBagConstraints gbc;
    private final ButtonsManager buttonsManager;

    private JToggleButton editButton;
    private JToggleButton deleteButton;

```

```

private JButton switchGraphTypeButton;

public ControlPanelsManager(App app) {
    this.app = app;
    this.gbc = app.getGBC();
    buttonsManager = new ButtonsManager();

    initControlPanelLeft();
    initControlPanelRight();
}

public JToggleButton getEditButton() {
    return editButton;
}

public JToggleButton getDeleteButton() {
    return deleteButton;
}

private void initControlPanelLeft() {
    JPanel controlPanelLeft = new RoundedPanel();
    controlPanelLeft.setLayout(new BorderLayout());
    controlPanelLeft.setBackground(CONTROL_PANEL_COLOR);

    RoundedPanel leftPanel = new RoundedPanel();
    leftPanel.setLayout(new FlowLayout(FlowLayout.LEFT));
    leftPanel.setBackground(CONTROL_PANEL_COLOR);

    RoundedPanel rightPanel = new RoundedPanel();
    rightPanel.setLayout(new FlowLayout(FlowLayout.RIGHT));
    rightPanel.setBackground(CONTROL_PANEL_COLOR);

    controlPanelLeft.add(leftPanel, BorderLayout.WEST);
    controlPanelLeft.add(rightPanel, BorderLayout.EAST);

    gbc.gridx = 0;
    gbc.gridy = 0;
    gbc.gridwidth = 1;
    gbc.gridheight = 1;
    gbc.weightx = 0.75;
    gbc.weighty = 0.0;
    gbc.fill = GridBagConstraints.HORIZONTAL;

    ButtonGroup toggleGroup = new ButtonGroup();

    editButton =
buttonsManager.createToggleButton("/gui_icons/construct.png", "Edit Mode",
toggleGroup);
    deleteButton =
buttonsManager.createToggleButton("/gui_icons/delete.png", "Delete Mode",
toggleGroup);
    editButton.setSelected(true);
    updateToggleButtonsStyles(toggleGroup);

    JButton deleteAllButton =
buttonsManager.createButton("/gui_icons/delete_all.png", "Clear the Field");
    switchGraphTypeButton =
buttonsManager.createButton("/gui_icons/switch_directed.png", "Switch Graph Type
(Current: Directed)");
    JButton saveButton = buttonsManager.createButton("/gui_icons/save.png",
"Save the Graph");
    JButton loadButton = buttonsManager.createButton("/gui_icons/load.png",
"Load the Graph");

```

```

        leftPanel.add(editButton);
        leftPanel.add(deleteButton);
        leftPanel.add(deleteAllButton);
        leftPanel.add(Box.createHorizontalStrut(BUTTON_SIZE));
        leftPanel.add(switchGraphTypeButton);
        rightPanel.add(saveButton);
        rightPanel.add(loadButton);

        deleteAllButton.addActionListener(e -> deleteAll());
        switchGraphTypeButton.addActionListener(e -> switchGraphType());
        saveButton.addActionListener(e -> save());
        loadButton.addActionListener(e -> load());

        app.add(controlPanelLeft, gbc);
    }

    private void initControlPanelRight() {
        JPanel controlPanelRight = new RoundedPanel();
        controlPanelRight.setLayout(new BorderLayout());
        controlPanelRight.setBackground(CONTROL_PANEL_COLOR);

        RoundedPanel leftPanel = new RoundedPanel();
        leftPanel.setLayout(new FlowLayout(FlowLayout.LEFT));
        leftPanel.setBackground(CONTROL_PANEL_COLOR);

        RoundedPanel rightPanel = new RoundedPanel();
        rightPanel.setLayout(new FlowLayout(FlowLayout.RIGHT));
        rightPanel.setBackground(CONTROL_PANEL_COLOR);

        controlPanelRight.add(leftPanel, BorderLayout.WEST);
        controlPanelRight.add(rightPanel, BorderLayout.EAST);

        gbc.gridx = 1;
        gbc.gridy = 0;
        gbc.gridwidth = 1;
        gbc.gridheight = 1;
        gbc.weightx = 0.25;
        gbc.weighty = 0.0;
        gbc.fill = GridBagConstraints.HORIZONTAL;

        JButton runButton = buttonsManager.createButton("/gui_icons/run.png",
"Run Completely");
        JButton stepBackButton =
buttonsManager.createButton("/gui_icons/step_back.png", "Step Back");
        JButton stepForwardButton =
buttonsManager.createButton("/gui_icons/step_forward.png", "Step Forward / Run
by Steps");

        leftPanel.add(runButton);
        rightPanel.add(stepBackButton);
        rightPanel.add(stepForwardButton);

        runButton.addActionListener(e -> app.getAlgorithmManager().runFull());
        stepBackButton.addActionListener(e ->
app.getAlgorithmManager().stepBack());
        stepForwardButton.addActionListener(e ->
app.getAlgorithmManager().stepForward());

        app.add(controlPanelRight, gbc);
    }

    private void deleteAll() {

```

```

        app.getGraphFieldManager().reset();
        app.getGraph().clear();
        app.getAlgorithmManager().reset();
        app.getGraphFieldManager().getGraphField().repaint();
    }

    private void switchGraphType() {
        app.getGraph().clear();
        app.getAlgorithmManager().reset();
        app.setGraph(app.getGraph().isDirected() ? new UndirectedGraph() : new
DirectedGraph());
        app.getGraphFieldManager().getGraphField().repaint();
        changeSwitchGraphTypeButtonIcon(switchGraphTypeButton,
app.getGraph().isDirected());
    }

    private void save() {
        JFileChooser fileChooser = new JFileChooser();
        fileChooser.setFileFilter(new FileFilter() {
            @Override
            public boolean accept(File f) {
                return f.isDirectory() ||
f.getName().toLowerCase().endsWith(".json");
            }

            @Override
            public String getDescription() {
                return "JSON Files (*.json)";
            }
        });

        int returnValue = fileChooser.showSaveDialog(null);
        if (returnValue == JFileChooser.APPROVE_OPTION) {
            File selectedFile = fileChooser.getSelectedFile();
            String path = selectedFile.getAbsolutePath();
            if (!path.toLowerCase().endsWith(".json")) {
                selectedFile = new File(path + ".json");
            }
            try (FileWriter file = new FileWriter(selectedFile)) {
                app.getAlgorithmManager().reset();
                app.getGraphFieldManager().getGraphField().repaint();
                if (app.getGraphFieldManager().getFirstVertex() != null) {

app.getGraphFieldManager().getFirstVertex().setColor(OUTLINE_SELECTED_VERTEX_COL
OR);

                    }
                    file.write(app.getGraph().toJSON());
                    if (app.getGraphFieldManager().getFirstVertex() != null) {

app.getGraphFieldManager().getFirstVertex().setColor(VERTEX_COLOR);

                    }
                } catch (IOException ex) {
                    ex.printStackTrace();
                }
            }
        }

        private void load() {
            JFileChooser fileChooser = new JFileChooser();
            fileChooser.setFileFilter(new FileFilter() {
                @Override
                public boolean accept(File f) {

```

```

        return f.isDirectory() ||
f.getName().toLowerCase().endsWith(".json");
    }

    @Override
    public String getDescription() {
        return "JSON Files (*.json)";
    }
});

int returnValue = fileChooser.showOpenDialog(null);
if (returnValue == JFileChooser.APPROVE_OPTION) {
    File selectedFile = fileChooser.getSelectedFile();
    try (FileReader reader = new FileReader(selectedFile)) {
        StringBuilder jsonBuilder = new StringBuilder();
        int c;
        while ((c = reader.read()) != -1) {
            jsonBuilder.append((char) c);
        }
        String json = jsonBuilder.toString();
        DirectedGraph loadedGraph = DirectedGraph.fromJSON(json);
        if (loadedGraph.isDirected()) {
            app.setGraph(loadedGraph);
        } else {
            UndirectedGraph undirectedGraph =
UndirectedGraph.fromJSON(json);
            app.setGraph(undirectedGraph);
        }
        changeSwitchGraphTypeButtonIcon(switchGraphTypeButton,
loadedGraph.isDirected());
        app.getGraphFieldManager().reset();
        Vertex firstVertex =
app.getGraph().getVertexByColor(OUTLINE_SELECTED_VERTEX_COLOR);
        if (firstVertex != null) {
            app.getGraphFieldManager().setFirstVertex(firstVertex);
        }
        app.getAlgorithmManager().reset();
        app.getGraphFieldManager().getGraphField().repaint();
    } catch (IOException ex) {
        ex.printStackTrace();
    }
}

private void updateToggleButtonStyles(ButtonGroup toggleGroup) {
    for (AbstractButton button :
Collections.list(toggleGroup.getElements())) {
        if (button.isSelected()) {
            button.setContentAreaFilled(true);
            button.setOpaque(true);
            button.setBackground(PRESSED_BUTTON_COLOR);

button.setBorder(BorderFactory.createLineBorder(MOUSE_ENTERED_BORDER_COLOR));
        } else {
            button.setContentAreaFilled(false);
            button.setOpaque(false);
            button.setBackground(UIManager.getColor("control"));
            button.setBorder(BorderFactory.createEmptyBorder());
        }
    }
}
}

```

```

        private void changeSwitchGraphTypeButtonIcon(JButton button, boolean
isDirected) {
            String iconPath = isDirected ? "/gui_icons/switch_directed.png" :
"/gui_icons/switch_undirected.png";
            String tooltipText = isDirected ? "Switch Graph Type (Current:
Directed)" : "Switch Graph Type (Current: Undirected)";
            ImageIcon icon = new ImageIcon(getClass().getResource(iconPath));
            Image img = icon.getImage();
            Image resizedImg = img.getScaledInstance(BUTTON_SIZE, BUTTON_SIZE,
Image.SCALE_SMOOTH);
            button.setIcon(new ImageIcon(resizedImg));
            button.setToolTipText(tooltipText);
        }

        private static class RoundedPanel extends JPanel {
            public RoundedPanel() {
                super();
                setOpaque(false);
            }

            @Override
            protected void paintComponent(Graphics g) {
                super.paintComponent(g);
                Graphics2D g2 = (Graphics2D) g.create();
                g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
RenderingHints.VALUE_ANTIALIAS_ON);
                g2.setColor(getBackground());
                g2.fillRoundRect(0, 0, getWidth(), getHeight(), 20, 20);
                g2.dispose();
            }
        }
    }
}

```

Название файла: *GraphFieldManager.java*

```

package gui;

import model.*;
import static gui.Settings.*;

import javax.swing.*;
import java.awt.*;
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;
import java.awt.event.MouseMotionAdapter;

public class GraphFieldManager {
    private final App app;
    private final JPanel graphField;

    private Vertex selectedVertex;
    private Vertex firstVertex;
    private Point initClick;

    public GraphFieldManager(App app) {
        this.app = app;
        this.graphField = new GraphPainter();
        reset();

        graphField.setBackground(GRAPH_FIELD_COLOR);
        graphField.addMouseListener(new MouseListener());
        graphField.addMouseMotionListener(new MouseMotionListener());
    }
}

```

```

        GridBagConstraints gbc = app.getGBC();
        gbc.gridx = 0;
        gbc.gridy = 1;
        gbc.gridwidth = 1;
        gbc.gridheight = 1;
        gbc.weightx = 0.75;
        gbc.weighty = 1.0;
        gbc.fill = GridBagConstraints.BOTH;

        app.add(graphField, gbc);
    }

    public JPanel getGraphField() {
        return graphField;
    }

    public void setFirstVertex(Vertex vertex) {
        this.firstVertex = vertex;
    }

    public Vertex getFirstVertex() {
        return firstVertex;
    }

    public void reset() {
        this.selectedVertex = null;
        this.firstVertex = null;
        this.initClick = null;
    }

    private void addVertex(Point point) {
        Vertex vertex = new Vertex("", point.x, point.y, VERTEX_COLOR);
        app.getGraph().addVertex(vertex);
        app.getAlgorithmManager().reset();
        graphField.repaint();
    }

    private void addEdge(Vertex from, Vertex to) {
        Edge edge = new Edge(from, to, EDGE_COLOR);
        app.getGraph().addEdge(edge);
        app.getAlgorithmManager().reset();
        graphField.repaint();
    }

    private void removeVertex(Vertex vertex) {
        app.getGraph().removeVertex(vertex);
        app.getAlgorithmManager().reset();
        graphField.repaint();
    }

    private void removeEdge(Edge edge) {
        app.getGraph().removeEdge(edge);
        app.getAlgorithmManager().reset();
        graphField.repaint();
    }

    private Vertex getVertexAt(Point point) {
        for (Vertex vertex : app.getGraph().getVertices()) {
            if (Math.pow(point.x - vertex.getX(), 2) + Math.pow(point.y -
vertex.getY(), 2) <= Math.pow(VERTEX_RADIUS, 2)) {
                return vertex;
            }
        }
    }
}

```



```

        app.getAlgorithmManager().reset();
        return null;
    }

    private Edge getEdgeAt(Point point) {
        for (Vertex vertex : app.getGraph().getVertices()) {
            for (Edge edge : app.getGraph().getEdgesFrom(vertex)) {
                Point from = new Point(edge.getFromV().getX(),
edge.getFromV().getY());
                Point to = new Point(edge.getToV().getX(),
edge.getToV().getY());
                if (isPointOnLine(point, from, to)) {
                    return edge;
                }
            }
        }
        return null;
    }

    private boolean isPointOnLine(Point point, Point from, Point to) {
        double distance = point.distance(from) + point.distance(to);
        double lineLength = from.distance(to);
        return Math.abs(distance - lineLength) < 3;
    }

    private class GraphPainter extends JPanel {
        @Override
        protected void paintComponent(Graphics g) {
            super.paintComponent(g);
            Graphics2D g2 = (Graphics2D) g.create();
            g2.setRenderingHint(RenderingHints.KEY_ANTIALIASING,
RenderingHints.VALUE_ANTIALIAS_ON);

            if (app.getAlgorithmManager().isRun()) {
                DijkstraState state = app.getAlgorithmManager().getState();
                for (Vertex vertex : app.getGraph().getVertices()) {
                    vertex.setColor(VERTEX_COLOR);
                    for (Edge edge : app.getGraph().getEdgesFrom(vertex)) {
                        edge.setColor(EDGE_COLOR);
                    }
                }

                for (Vertex vertex : state.getDistances().keySet()) {
                    if (state.getVisited().contains(vertex)) {
                        vertex.setColor(VISITED_VERTEX_COLOR);
                    }
                }

                if (state.getCurrentVertex() != null) {
                    state.getCurrentVertex().setColor(CURRENT_VERTEX_COLOR);
                }

                if (state.getNeighborVertex() != null) {
                    state.getNeighborVertex().setColor(NEIGHBOR_VERTEX_COLOR);
                }

                if (state.getCurrentEdge() != null) {
                    state.getCurrentEdge().setColor(PROCESSED_EDGE_COLOR);
                }
            }

            for (Vertex vertex : app.getGraph().getVertices()) {
                for (Edge edge : app.getGraph().getEdgesFrom(vertex)) {
                    Point from = new Point(edge.getFromV().getX(),
edge.getFromV().getY());

```

```

        Point to = new Point(edge.getToV().getX(),
edge.getToV().getY());
        Point intersection = getIntersection(from, to);
        g2.setColor(edge.getColor());
        g2.drawLine(from.x, from.y, intersection.x, intersection.y);
        if (app.getGraph().isDirected()) {
            drawArrow(g2, from.x, from.y, intersection.x,
intersection.y);
        }

        int midX = (from.x + to.x) / 2;
        int midY = (from.y + to.y) / 2;
        g2.setColor(TITLE_COLOR);
        g2.drawString(edge.getWeight().toString(), midX, midY);
    }
}

for (Vertex vertex : app.getGraph().getVertices()) {
    int x = vertex.getX() - VERTEX_RADIUS;
    int y = vertex.getY() - VERTEX_RADIUS;
    g2.setColor(vertex.getColor());
    g2.fillOval(x, y, VERTEX_RADIUS * 2, VERTEX_RADIUS * 2);
    g2.setColor(TITLE_COLOR);
    g2.drawString(vertex.getLabel(), vertex.getX() - VERTEX_RADIUS /
2, vertex.getY() - (VERTEX_RADIUS + VERTEX_RADIUS / 2));

    if (vertex.equals(firstVertex)) {
        g2.setColor(OUTLINE_SELECTED_VERTEX_COLOR);
        g2.drawOval(x, y, VERTEX_RADIUS * 2, VERTEX_RADIUS * 2);
    }
}

if (app.getAlgorithmManager().isRun()) {
    DijkstraState state = app.getAlgorithmManager().getState();
    for (Vertex vertex : state.getDistances().keySet()) {
        g2.setColor(DISTANCE_COLOR);
        g2.drawString(state.getDistances().get(vertex) <
Integer.MAX_VALUE ? String.valueOf(state.getDistances().get(vertex)) : "∞",
vertex.getX() - VERTEX_RADIUS / 2, vertex.getY() + 2 * VERTEX_RADIUS);
    }

    if (state.getInequality() != null) {
        g2.setColor(INEQUALITY_COLOR);
        g2.drawString(state.getInequality(),
state.getNeighborVertex().getX() - VERTEX_RADIUS / 2 - 16,
state.getNeighborVertex().getY() + 3 * VERTEX_RADIUS);
    }
}
g2.dispose();
}

private Point getIntersection(Point from, Point to) {
    int radius = VERTEX_RADIUS;
    double dx = to.x - from.x;
    double dy = to.y - from.y;
    double dist = Math.sqrt(dx * dx + dy * dy);
    double newX = to.x - dx * radius / dist;
    double newY = to.y - dy * radius / dist;
    return new Point((int) newX, (int) newY);
}

private void drawArrow(Graphics2D g2, int x1, int y1, int x2, int y2) {

```

```

        int arrowSize = ARROW_SIZE;
        double angle = Math.atan2(y2 - y1, x2 - x1);
        int x = (int) (x2 - arrowSize * Math.cos(angle - Math.PI / 6));
        int y = (int) (y2 - arrowSize * Math.sin(angle - Math.PI / 6));
        int x3 = (int) (x2 - arrowSize * Math.cos(angle + Math.PI / 6));
        int y3 = (int) (y2 - arrowSize * Math.sin(angle + Math.PI / 6));
        int[] xPoints = {x2, x, x3};
        int[] yPoints = {y2, y, y3};
        g2.fillPolygon(xPoints, yPoints, 3);
    }
}

private class MouseListener extends MouseAdapter {
    @Override
    public void mouseClicked(MouseEvent e) {
        if (app.getControlPanelsManager().getEditButton().isSelected()) {
            if (SwingUtilities.isLeftMouseButton(e) && e.getClickCount() ==
1) {
                if (firstVertex == null) {
                    firstVertex = getVertexAt(e.getPoint());
                    if (firstVertex == null) {
                        addVertex(e.getPoint());
                    } else {
                        graphField.repaint();
                    }
                } else {
                    Vertex secondVertex = getVertexAt(e.getPoint());
                    if (secondVertex != null &&
!secondVertex.equals(firstVertex)) {
                        addEdge(firstVertex, secondVertex);
                        firstVertex = null;
                        graphField.repaint();
                    } else if (secondVertex == null) {
                        firstVertex = null;
                    }
                }
            } else if (SwingUtilities.isRightMouseButton(e) &&
e.getClickCount() == 1) {
                Edge edge = getEdgeAt(e.getPoint());
                if (edge != null) {
                    String weightStr =
CustomDialog.showInputDialog(graphField, "Edge Weight", "Enter Non-Negative Edge
Weight:", 250, 125);
                    if (weightStr != null && !weightStr.trim().isEmpty()) {
                        try {
                            int weight = Integer.parseInt(weightStr);
                            if (weight < 0) {
                                CustomMessageDialog.showMessageDialog(graphField, "Error", "Edge weight must be
non-negative.", 250, 100);
                            } else {
                                app.getGraph().setEdgeWeight(edge, weight);
                                app.getAlgorithmManager().reset();
                                graphField.repaint();
                            }
                        } catch (NumberFormatException ex) {
                            CustomMessageDialog.showMessageDialog(graphField, "Error", "Invalid input.
Please enter a number.", 250, 100);
                        }
                    }
                }
            }
        }
    }
}

```

```

        } else if
(app.getControlPanelsManager().getDeleteButton().isSelected()) {
        if (SwingUtilities.isLeftMouseButton(e) && e.getClickCount() ==
1) {
            Vertex vertex = getVertexAt(e.getPoint());
            if (vertex != null) {
                removeVertex(vertex);
            } else {
                Edge edge = getEdgeAt(e.getPoint());
                if (edge != null) {
                    removeEdge(edge);
                }
            }
        }
        graphField.repaint();
    }

    @Override
    public void mousePressed(MouseEvent e) {
        if (SwingUtilities.isLeftMouseButton(e)) {
            selectedVertex = getVertexAt(e.getPoint());
            initClick = e.getPoint();
        }
    }

    @Override
    public void mouseReleased(MouseEvent e) {
        selectedVertex = null;
        initClick = null;
    }
}

private class MouseMotionListener extends MouseMotionAdapter {
    @Override
    public void mouseDragged(MouseEvent e) {
        if (selectedVertex != null &&
(app.getControlPanelsManager().getEditButton().isSelected() ||
app.getControlPanelsManager().getDeleteButton().isSelected())) {
            int deltaX = e.getX() - initClick.x;
            int deltaY = e.getY() - initClick.y;
            selectedVertex.setX(selectedVertex.getX() + deltaX);
            selectedVertex.setY(selectedVertex.getY() + deltaY);
            initClick = e.getPoint();
            graphField.repaint();
        }
    }
}
}

```

Название файла: *StepsFieldManager.java*

```

package gui;

import model.DijkstraState;
import static gui.Settings.*;

import javax.swing.*.*;
import java.awt.*.*;

public class StepsFieldManager {
    private final App app;
    private final JTextArea stepsField;

```

```

public StepsFieldManager(App app) {
    this.app = app;

    stepsField = new JTextArea();
    stepsField.setBackground(STEPS_FIELD_COLOR);
    stepsField.setEditable(false);
    stepsField.setLineWrap(true);
    stepsField.setWrapStyleWord(true);

    stepsField.setFont(new Font("Arial", Font.PLAIN, STEPS_FIELD_FONT));

    JScrollPane scrollPane = new JScrollPane(stepsField);
    int width = 200, height = 500;
    scrollPane.setPreferredSize(new Dimension(width, height));
    scrollPane.setMinimumSize(new Dimension(width, height));
    scrollPane.setMaximumSize(new Dimension(width, height));

    JPanel stepsPanel = new JPanel();
    stepsPanel.setLayout(new BorderLayout());
    stepsPanel.add(scrollPane, BorderLayout.CENTER);

    GridBagConstraints gbc = app.getGBC();
    gbc.gridx = 1;
    gbc.gridy = 1;
    gbc.gridwidth = 1;
    gbc.gridheight = 1;
    gbc.weightx = 0.25;
    gbc.weighty = 1.0;
    gbc.fill = GridBagConstraints.BOTH;

    app.add(stepsPanel, gbc);
}

public void display() {
    clear();
    DijkstraState state = app.getAlgorithmManager().getState();
    for (String step : state.getSteps()) {
        stepsField.append(step);
    }
}

public void clear() {
    stepsField.setText("");
}
}

```

Название файла: *ButtonsManager.java*

```

package gui;

import static gui.Settings.*;

import javax.swing.*;
import javax.swing.plaf.basic.BasicButtonUI;
import java.awt.*;
import java.awt.event.MouseAdapter;
import java.awt.event.MouseEvent;
import java.util.Collections;

public class ButtonsManager {
    public JButton createButton(String iconPath, String toolTipText) {
        ImageIcon icon = new ImageIcon(getClass().getResource(iconPath));
    }
}

```

```

        Image img = icon.getImage();
        Image resizedImg = img.getScaledInstance(BUTTON_SIZE, BUTTON_SIZE,
Image.SCALE_SMOOTH);
        icon = new ImageIcon(resizedImg);
        JButton button = new JButton(icon);
        button.setPreferredSize(new Dimension(BUTTON_SIZE, BUTTON_SIZE));
        button.setContentAreaFilled(false);
        button.setBorderPainted(false);
        button.setToolTipText(toolTipText);

        button.setUI(new BasicButtonUI() {
            @Override
            protected void paintButtonPressed(Graphics g, AbstractButton b) {
                g.setColor(PRESSED_BUTTON_COLOR);
                g.fillRect(0, 0, b.getWidth(), b.getHeight());
            }
        });

        button.addMouseListener(new MouseAdapter() {
            @Override
            public void mouseEntered(MouseEvent e) {
                button.setContentAreaFilled(true);
                button.setOpaque(true);
                button.setBackground(MOUSE_ENTERED_BACKGROUND_COLOR);
            }

            @Override
            public void mouseExited(MouseEvent e) {
                button.setContentAreaFilled(false);
                button.setOpaque(false);
                button.setBackground(UIManager.getColor("control"));
                button.setBorder(BorderFactory.createEmptyBorder());
            }
        });
        return button;
    }

    public JToggleButton createToggleButton(String iconPath, String toolTipText,
ButtonGroup toggleGroup) {
        ImageIcon icon = new ImageIcon(getClass().getResource(iconPath));
        Image img = icon.getImage();
        Image resizedImg = img.getScaledInstance(BUTTON_SIZE, BUTTON_SIZE,
Image.SCALE_SMOOTH);
        icon = new ImageIcon(resizedImg);
        JToggleButton button = new JToggleButton(icon);
        button.setPreferredSize(new Dimension(BUTTON_SIZE, BUTTON_SIZE));
        button.setContentAreaFilled(false);
        button.setBorderPainted(false);
        button.setToolTipText(toolTipText);
        toggleGroup.add(button);

        button.setUI(new BasicButtonUI() {
            @Override
            protected void paintButtonPressed(Graphics g, AbstractButton b) {
                if (b.isContentAreaFilled()) {
                    g.setColor(PRESSED_BUTTON_COLOR);
                    g.fillRect(0, 0, b.getWidth(), b.getHeight());
                }
            }
        });
    }

```

```

        button.addMouseListener(new MouseAdapter() {
            @Override
            public void mouseEntered(MouseEvent e) {
                if (!button.isSelected()) {
                    button.setContentAreaFilled(true);
                    button.setOpaque(true);
                    button.setBackground(MOUSE_ENTERED_BACKGROUND_COLOR);
                }
            }

            @Override
            public void mouseExited(MouseEvent e) {
                if (!button.isSelected()) {
                    button.setContentAreaFilled(false);
                    button.setOpaque(false);
                    button.setBackground(UIManager.getColor("control"));
                    button.setBorder(BorderFactory.createEmptyBorder());
                }
            }
        });

        button.addActionListener(e -> {
            if (button.isSelected()) {
                button.setContentAreaFilled(true);
                button.setOpaque(true);
                button.setBackground(PRESSED_BUTTON_COLOR);
            } else {
                button.setContentAreaFilled(false);
                button.setOpaque(false);
                button.setBackground(UIManager.getColor("control"));
                button.setBorder(BorderFactory.createEmptyBorder());
            }
        });

        for (AbstractButton btn :
Collections.list(toggleGroup.getElements())) {
            if (btn != button) {
                btn.setContentAreaFilled(false);
                btn.setOpaque(false);
                btn.setBackground(UIManager.getColor("control"));
                btn.setBorder(BorderFactory.createEmptyBorder());
            }
        }
    });
    return button;
}
}

```

Название файла: *CustomDialog.java*

```

package gui;

import static gui.Settings.*;

import javax.swing.*;
import javax.swing.plaf.basic.BasicButtonUI;
import java.awt.*;
import java.awt.event.KeyEvent;

public class CustomDialog {

```

```

    public static String showInputDialog(Component parent, String title, String
message, Integer width, Integer height) {
        JDialog dialog = new JDialog(SwingUtilities.getWindowAncestor(parent),
title, Dialog.ModalityType.APPLICATION_MODAL);
        dialog.setLayout(new BorderLayout());
        dialog.getContentPane().setBackground(DIALOG_BACKGROUND_COLOR);

        JPanel messagePanel = new JPanel();
        messagePanel.setBackground(Settings.DIALOG_BACKGROUND_COLOR);
        messagePanel.add(new JLabel(message));
        dialog.add(messagePanel, BorderLayout.NORTH);

        JTextField textField = new JTextField(10);
        JPanel inputPanel = new JPanel();
        inputPanel.setBackground(DIALOG_BACKGROUND_COLOR);
        inputPanel.add(textField);
        dialog.add(inputPanel, BorderLayout.CENTER);

        JPanel buttonPanel = new JPanel();
        buttonPanel.setBackground(DIALOG_BACKGROUND_COLOR);

        JButton okButton = createStyledButton("OK");
        okButton.addActionListener(e -> dialog.dispose());

        JButton cancelButton = createStyledButton("Cancel");
        cancelButton.addActionListener(e -> {
            textField.setText(null);
            dialog.dispose();
        });

        buttonPanel.add(okButton);
        buttonPanel.add(cancelButton);
        dialog.add(buttonPanel, BorderLayout.SOUTH);

        dialog.getRootPane().setDefaultButton(okButton);
        dialog.getRootPane().registerKeyboardAction(e ->
textField.setText(null),
            KeyStroke.getKeyStroke(KeyEvent.VK_ESCAPE, 0),
            JComponent.WHEN_IN_FOCUSED_WINDOW);
        dialog.getRootPane().registerKeyboardAction(e -> okButton.doClick(),
            KeyStroke.getKeyStroke(KeyEvent.VK_ENTER, 0),
            JComponent.WHEN_IN_FOCUSED_WINDOW);

        dialog.setSize(width, height);
        dialog.setResizable(false);
        dialog.setLocationRelativeTo(parent);
        dialog.setVisible(true);

        return textField.getText();
    }

    private static JButton createStyledButton(String text) {
        JButton button = new JButton(text);
        button.setPreferredSize(new Dimension(80, 25));
        button.setBackground(CONTROL_PANEL_COLOR);
        button.setUI(new BasicButtonUI() {
            @Override
            protected void paintButtonPressed(Graphics g, AbstractButton b) {
                g.setColor(PRESSED_BUTTON_COLOR);
                g.fillRect(0, 0, b.getWidth(), b.getHeight());
            }
        });
    }
}

```



```

        button.addMouseListener(new java.awt.event.MouseAdapter() {
            @Override
            public void mouseEntered(java.awt.event.MouseEvent evt) {

button.setBorder(BorderFactory.createLineBorder(MOUSE_ENTERED_BORDER_COLOR));
        button.setBackground(MOUSE_ENTERED_BACKGROUND_COLOR);
            }

            @Override
            public void mouseExited(java.awt.event.MouseEvent evt) {
                button.setBorder(BorderFactory.createEmptyBorder());
                button.setBackground(CONTROL_PANEL_COLOR);
            }

            @Override
            public void mousePressed(java.awt.event.MouseEvent evt) {
                button.setBackground(PRESSED_BUTTON_COLOR);
            }

            @Override
            public void mouseReleased(java.awt.event.MouseEvent evt) {
                button.setBackground(CONTROL_PANEL_COLOR);
            }
        });
        return button;
    }
}

```

Название файла: *CustomMessageDialog.java*

```

package gui;

import static gui.Settings.*;

import javax.swing.*;
import javax.swing.plaf.basic.BasicButtonUI;
import java.awt.*;
import java.awt.event.KeyEvent;

public class CustomMessageDialog {
    public static void showMessageDialog(Component parent, String title, String
message, Integer width, Integer height) {
        JDialog dialog = new JDialog(SwingUtilities.getWindowAncestor(parent),
title, Dialog.ModalityType.APPLICATION_MODAL);
        dialog.setLayout(new BorderLayout());
        dialog.getContentPane().setBackground(DIALOG_BACKGROUND_COLOR);

        JPanel messagePanel = new JPanel();
        messagePanel.setBackground(DIALOG_BACKGROUND_COLOR);
        messagePanel.add(new JLabel(message));
        dialog.add(messagePanel, BorderLayout.CENTER);

        JPanel buttonPanel = new JPanel();
        buttonPanel.setBackground(DIALOG_BACKGROUND_COLOR);

        JButton okButton = createStyledButton("OK");
        okButton.addActionListener(e -> dialog.dispose());

        buttonPanel.add(okButton);
        dialog.add(buttonPanel, BorderLayout.SOUTH);

        dialog.getRootPane().setDefaultButton(okButton);
        dialog.getRootPane().registerKeyboardAction(e -> dialog.dispose(),

```

```

        KeyStroke.getKeyStroke(KeyEvent.VK_ESCAPE, 0),
        JComponent.WHEN_IN_FOCUSED_WINDOW);
dialog.getRootPane().registerKeyboardAction(e -> okButton.doClick(),
        KeyStroke.getKeyStroke(KeyEvent.VK_ENTER, 0),
        JComponent.WHEN_IN_FOCUSED_WINDOW);

dialog.setSize(width, height);
dialog.setResizable(false);
dialog.setLocationRelativeTo(parent);
dialog.setVisible(true);
}

private static JButton createStyledButton(String text) {
    JButton button = new JButton(text);
    button.setPreferredSize(new Dimension(80, 25));
    button.setBackground(CONTROL_PANEL_COLOR);
    button.setUI(new BasicButtonUI() {
        @Override
        protected void paintButtonPressed(Graphics g, AbstractButton b) {
            g.setColor(PRESSED_BUTTON_COLOR);
            g.fillRect(0, 0, b.getWidth(), b.getHeight());
        }
    });

    button.addMouseListener(new java.awt.event.MouseAdapter() {
        @Override
        public void mouseEntered(java.awt.event.MouseEvent evt) {
            button.setBorder(BorderFactory.createLineBorder(MOUSE_ENTERED_BORDER_COLOR));
            button.setBackground(MOUSE_ENTERED_BACKGROUND_COLOR);
        }

        @Override
        public void mouseExited(java.awt.event.MouseEvent evt) {
            button.setBorder(BorderFactory.createEmptyBorder());
            button.setBackground(CONTROL_PANEL_COLOR);
        }

        @Override
        public void mousePressed(java.awt.event.MouseEvent evt) {
            button.setBackground(PRESSED_BUTTON_COLOR);
        }

        @Override
        public void mouseReleased(java.awt.event.MouseEvent evt) {
            button.setBackground(CONTROL_PANEL_COLOR);
        }
    });
    return button;
}
}

```

Название файла: *Settings.java*

```

package gui;

import java.awt.*;

public class Settings {
    public static final Color APP_COLOR = new Color(241, 243, 249);

    public static final Color CONTROL_PANEL_COLOR = new Color(200, 200, 200);
    public static final Color STEPS_FIELD_COLOR = new Color(245, 245, 245);
}

```

```

    public static final Integer STEPS_FIELD_FONT = 14;

    public static final Color GRAPH_FIELD_COLOR = new Color(255, 255, 255);

    public static final Color DIALOG_BACKGROUND_COLOR = new Color(255, 255,
255);

    public static final Color TITLE_COLOR = Color.BLACK;

    public static final Integer BUTTON_SIZE = 30;

    public static final Color MOUSE_ENTERED_BACKGROUND_COLOR = new Color(180,
180, 180);
    public static final Color MOUSE_ENTERED_BORDER_COLOR = new Color(150, 150,
150);

    public static final Color PRESSED_BUTTON_COLOR = new Color(160, 160, 160);

    public static final Integer VERTEX_RADIUS = 10;

    public static final Color VERTEX_COLOR = Color.GRAY;
    public static final Color EDGE_COLOR = Color.GRAY;
    public static final Color OUTLINE_SELECTED_VERTEX_COLOR = Color.RED;

    public static final Integer ARROW_SIZE = 10;

    public static final Color VISITED_VERTEX_COLOR = Color.RED;
    public static final Color CURRENT_VERTEX_COLOR = new Color(145, 0, 0);
    public static final Color NEIGHBOR_VERTEX_COLOR = new Color(11, 61, 135);

    public static final Color PROCESSED_EDGE_COLOR = new Color(0, 102, 255);

    public static final Color DISTANCE_COLOR = Color.RED;
    public static final Color INEQUALITY_COLOR = new Color(11, 61, 135);
}

```