

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

ОТЧЕТ
по учебной практике

Тема: Кратчайшие пути в графе. Алгоритм Дейкстры.

Студентка гр. 3381

Деревягина А.П.

Студентка гр. 3381

Кунакова М.Р.

Студент гр. 3388

Потоцкий С.С.

Руководитель

Шестопалов Р.П.

Санкт-Петербург

2025

ЗАДАНИЕ
НА УЧЕБНУЮ ПРАКТИКУ

Студентка Кунакова М.Р. группы 3381

Студентка Деревягина А.П. группы 3381

Студент Потоцкий С.С. группы 3388

Тема практики: Визуализация алгоритма нахождения кратчайших путей в графе. Алгоритм Дейкстры.

Задание на практику:

Командная итеративная разработка визуализатора алгоритма на Kotlin с графическим интерфейсом.

Алгоритм: алгоритм Дейкстры.

Сроки прохождения практики: 25.06.2024 – 08.07.2024

Дата сдачи отчета: 04.07.2024

Дата защиты отчета: 04.07.2024

Студентка _____ Кунакова М.Р.

Студентка _____ Деревягина А.П.

Студент _____ Потоцкий С.С.

Руководитель _____ Шестопалов Р.П.

АННОТАЦИЯ

Разработать и реализовать графическое приложение на Kotlin для визуализации работы алгоритма Дейкстры. Программа должна наглядно демонстрировать процесс нахождения кратчайших путей в графе, предоставляя пользователю возможность наблюдать за каждым этапом выполнения алгоритма.

План практики:

1. Определение требований и проектирование архитектуры приложения;
2. Разработка графического интерфейса пользователя, включающего холст для редактирования графа и кнопки для взаимодействия с алгоритмом и графическими элементами;
3. Защита проекта.

SUMMARY

Develop and implement a graphical application in Kotlin to visualize the operation of Dijkstra's algorithm. The program should display the process of finding the shortest execution paths in the graph, providing the operator with the ability to observe each stage of the algorithm.

Practice plan:

1. Defining requirements and designing the application structure;
2. Developing a graphical user interface that includes a canvas for editing the graph and buttons for interacting with the algorithm and graphical elements;
3. Project defense.

СОДЕРЖАНИЕ

Введение	5
1. Требования к программе	6
1.1. Исходные требования к программе*	6
1.2. Уточнение требований после сдачи прототипа	10
1.3. Уточнение требований после сдачи 1-ой версии	10
1.4. Уточнение требований после сдачи 2-ой версии	10
2. План разработки и распределение ролей в бригаде	11
2.1. План разработки	11
2.2. Распределение ролей в бригаде	11
3. Особенности реализации	13
3.1. Структуры данных	13
3.2. Основные методы	14
4. Тестирование	17
4.1 Тестирование графического интерфейса	17
4.2 Тестирование кода алгоритма	27
Заключение	30
Список использованных источников	32
Приложение А. Исходный код	33

ВВЕДЕНИЕ

Цель данной практики заключается в разработке графического приложения на языке Kotlin, предназначенного для визуализации работы алгоритма Дейкстры, который используется для нахождения кратчайших путей в графах. Визуализация алгоритма способствует более глубокому пониманию его принципов и особенностей, что особенно актуально для студентов, изучающих основы алгоритмизации.

В рамках практики предусмотрены следующие задачи:

1. Проектирование и разработка пользовательского интерфейса приложения;
2. Реализация алгоритма Дейкстры для определения кратчайших путей в графе;
3. Обеспечение возможности задания графа как через редактируемый холст, так и из файла;
4. Создание механизмов визуализации, позволяющих наглядно демонстрировать процесс нахождения кратчайших путей;
5. Интеграция всех компонентов в единую систему и проведение тестирования для обеспечения корректной работы приложения.

1. ТРЕБОВАНИЯ К ПРОГРАММЕ

1.1. Исходные Требования к программе

1.1.1. Требования к визуализации

1.1.1.1. Технология: Compose for Desktop (Kotlin).

1.1.1.2. Основные элементы

1.1.1.2.1. Холст для отображения графа

1.1.1.2.2. Панель управления

1.1.1.2.3. Таблица текущего состояния алгоритма

1.1.1.2.4. Текстовая область для пояснений шагов

1.1.2. Элементы управления графиком

1.1.2.1. Кнопки:

1.1.2.1.1. «Вставить вершину»:

1.1.2.1.1.1. Добавляет вершину в место клика на холсте.

1.1.2.1.1.2. Автоматическое именование (A, B, C, ...)

1.1.2.1.2. «Задать начальную вершину»:

1.1.2.1.2.1. Выбор стартовой вершины (выделяется желтым)

1.1.2.1.3. «Показать таблицу»:

1.1.2.1.3.1. Открывает/скрывает полную таблицу алгоритма

1.1.2.1.4. «Скачать график»:

1.1.2.1.4.1. Сохранение графа в файл (формат JSON)

1.1.2.1.5. «Загрузить график»:

1.1.2.1.5.1. Чтение графа из файла (формат JSON)

1.1.3. Элементы управления алгоритмом

1.1.3.1. Кнопки:

1.1.3.1.1. «Старт/Пауза»:

1.1.3.1.1.1. Запуск/приостановка пошаговой визуализации

1.1.3.1.1.2. При нажатии кнопки после создания графа, запускается визуализация алгоритма

1.1.3.1.1.3. При нажатии во время демонстрации алгоритма, он останавливается на текущем шаге

1.1.3.1.1.4. При нажатии во время паузы демонстрации алгоритма, он возобновляется с того же шага

1.1.3.1.2. Ползунок скорости:

1.1.3.1.2.1. Изменение скорости анимации (мс/шаг)

1.1.4. Визуализация графа

1.1.4.1. Цветовая схема:

1.1.4.1.1. Начальная вершина: желтый

1.1.4.1.2. Посещенные вершины: зеленый

1.1.4.1.3. Непосещенные: серый

1.1.4.2. Ребра:

1.1.4.2.1. Отображение весов рядом с ребрами

1.1.4.2.2. Выделение текущих ребер при обработке: синий

1.1.5. Таблица состояния

1.1.5.1. Строки:

1.1.5.1.1. Вершины (имена)

1.1.5.1.2. Текущий вес (до шага)

1.1.5.1.3. Новый вес (после шага)

1.1.6. Пояснения алгоритма

1.1.6.1. Выделение действия на конкретном шаге

1.1.6.1.1. «Выбрана следующая непосещенная вершина В»

1.1.6.1.2. «Обновлено расстояние до В»

1.1.7. Требования к вводу исходных данных

1.1.7.1. Задание графа:

1.1.7.1.1. Ручной ввод:

1.1.7.1.1.1. Клик по холсту для создания вершины

1.1.7.1.1.2. Клик по двум вершинам для создания ребра

1.1.7.1.1.3. Кнопка для создания вершины

1.1.7.1.2. Загрузка файла:

1.1.7.1.2.1. Формат: JSON

1.1.7.1.2.2. Отображение загруженного графа на холсте

1.1.8. Ограничения

1.1.8.1. Ограничения на граф:

1.1.8.1.1. Запрет отрицательных весов ребер

1.1.8.1.2. Запрет кратных ребер

1.1.8.1.3. Запрет петель

1.1.8.1.4. Ограничение на вес ребер:

1.1.8.1.4. Вес ребра – целое число от 1 до 999

1.1.8.1.5. Ограничение на количество вершин:

1.1.8.1.5. Число вершин – целое число от 2 до 15

1.1.8.1.5. При возникновении запретной ситуации выводить ошибку

с

пояснением

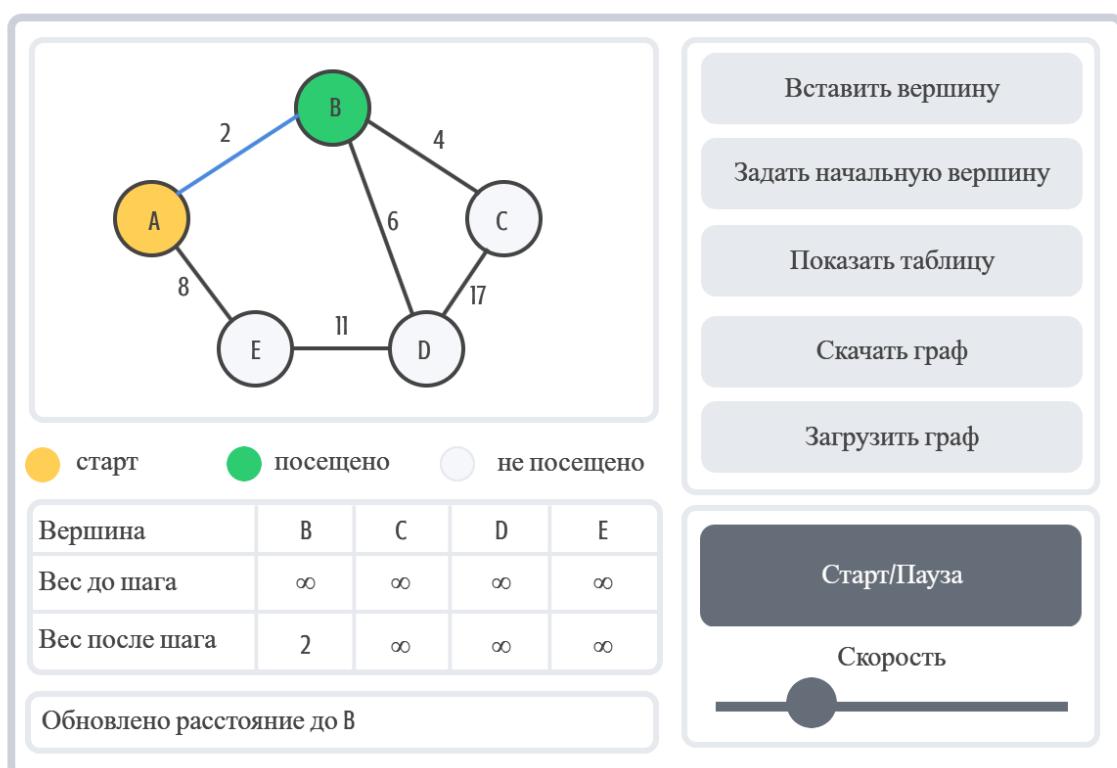


Рисунок 1 - Приблизительный дизайн приложения

1.1.9. План тестирования

1.1.9.1. Проверка корректности обработки ошибок:

1.1.9.1.1. В структуре графа:

1.1.9.1.1.1. Передается ребро с отрицательным весом

1.1.9.1.1.2. Передается ребро с нецелым весом

1.1.9.1.1.3. Передается ребро с большим весом (>999)

1.1.9.1.1.4. Передается граф с большим количеством вершин (>15)

1.1.9.1.1.5. Передаются кратные ребра

1.1.9.1.1.6. Передается петля

1.1.9.1.1.2. В передаче файла:

1.1.9.1.1.2.1. Передается файл, не совпадающий с форматом JSON

1.1.9.2. Проверка корректности работы интерфейса

1.1.9.2.1. Проверка работоспособности кнопок

1.1.9.2.2. Проверка работоспособности холста для работы с графиками

1.1.9.3. Проверка работоспособности алгоритма

1.1.9.3.1. Проверка работоспособности на произвольном графике

1.1.9.3.2. Проверка работоспособности на графике с более чем одной компонентой связности

1.1.9.3.3. Проверка корректности выводимых комментариев алгоритма

1.2. Уточнение требований после сдачи прототипа

1.2.1. Добавить кнопку «В конец»:

1.2.1.1. Кнопка демонстрирует сразу результат работы алгоритма.

1.2.2. Изменить принцип работы кнопки «Вставить вершины»:

1.2.2.1. При нажатии на кнопку включается режим вставки, при каждом нажатии на холст в месте клика появляется вершина

1.2.2.2. При повторном нажатии режим вставки выключается

1.2.3. Добавить редактирование веса ребер:

1.2.3.1. При двойном клике на ребро помимо удаления появляется возможность изменения веса

1.3. Уточнение требований после сдачи 1-ой версии

1.3.1. При сохранении файла с графом сохраняется также результат работы алгоритма

2. ПЛАН РАЗРАБОТКИ И РАСПРЕДЕЛЕНИЕ РОЛЕЙ В БРИГАДЕ

2.1. План разработки

Дата	Этап проекта	Реализованные возможности	Выполнено
25.06.25	Согласование спецификации	Написание требований, плана тестирования.	+
30.06.25	Сдача прототипа	Разработка интерфейса (кнопки для взаимодействия с графом и файлами, подключение холста, кнопки для работы с алгоритмом без его работы)	+
02.07.25	Сдача версии 1	Добавление поддержки алгоритма и его визуализация. Добавление реализации кнопок взаимодействия с алгоритмом.	+
04.07.25	Сдача версии 2	Проведение тестирования. Правки по требованию руководителя.	+
04.07.25	Сдача отчёта		
	Защита отчёта		

2.2. Распределение ролей в бригаде

2.2.1. Реализация графического интерфейса:

2.2.1.1. Общий дизайн приложения и структура кода – Кунакова М.Р.

2.2.1.2. Написание кода для интерфейса приложения, работоспособности кнопок и ползунков – Деревягина А.П.

2.2.2. Реализация алгоритма Дейкстры, его связь с интерфейсом и тестирование – Потоцкий С.С.

3. ОСОБЕННОСТИ РЕАЛИЗАЦИИ

3.1. Структуры данных

Граф:

Класс Graph реализует структуру графа. Для хранения вершин используется список mutableListOf<Vertex>(), а для рёбер — список mutableListOf<Edge>().

- vertices: List<Vertex> — список всех вершин графа.
- edges: List<Edge> — список всех рёбер графа.

Вершина:

Класс Vertex (data class) описывает вершину графа:

- id: Int — уникальный идентификатор вершины.
- xCoordinate: Int, yCoordinate: Int — координаты для визуализации.
- color: VertexColor — цвет вершины для визуализации и состояния.
- name: String — имя вершины.

Ребро:

Класс Edge (data class) описывает ребро графа:

- weight: Int — вес ребра.
- source: Int, target: Int — идентификаторы начальной и конечной вершин.

Цвет вершины:

Перечисление VertexColor определяет возможные состояния/цвета вершины (например, для визуализации работы алгоритма).

Состояние алгоритма Дейкстры:

DijkstraStep (data class): хранит информацию о каждом шаге алгоритма:

- step: Int — номер шага.
- action: String — описание действия.
- currentVertexId: Int? — текущая вершина.
- distances: Map<Int, Int> — расстояния до всех вершин.

- visited: Set<Int> — множество посещённых вершин.
- priorityQueue: List<Pair<Int, Int>> — состояние очереди с приоритетом.

DijkstraResult (data class): итоговый результат работы алгоритма:

- distances: Map<Int, Int> — финальные расстояния.
- previous: Map<Int, Int?> — предыдущие вершины для восстановления пути.
- steps: List<DijkstraStep> — история всех шагов.

Структуры для визуализации:

В визуальных компонентах используются:

- Списки вершин и рёбер для отрисовки (List<Vertex>, List<Edge>).
- Таблицы и строки для отображения промежуточных и итоговых данных алгоритма (List<DijkstraStep>, Map<Int, Int>).
- Для визуализации используется Jetpack Compose (например, компоненты Canvas, Column, Row, Text и др.), что позволяет гибко отображать состояние графа и алгоритма.

3.2. Основные методы

Основные методы программы можно разделить на методы для управления пользовательским интерфейсом, методы для работы с графом и методы для выполнения и визуализации алгоритма Дейкстры.

Методы для управления пользовательским интерфейсом

MainWindowContent (функция-компоновщик):

- Инициализирует главное окно приложения, размещает визуальные компоненты (канвас для графа, таблицы, панель управления).
- Обрабатывает события пользователя: добавление/удаление вершин и рёбер, запуск/пауза/сброс анимации, изменение скорости, загрузка/сохранение графа.

- Управляет состоянием анимации и отображением вспомогательных окон (например, инструкции).

ControlPanel (функция-компоновщик):

- Отвечает за панель управления: кнопки для добавления вершин, задания начальной вершины, запуска/паузы, сброса, загрузки/сохранения графа, управления скоростью анимации.
- Передаёт события в контроллер приложения.

Методы для работы с графом (AppController)

`addVertex(x: Int, y: Int):`

- Добавляет новую вершину с заданными координатами на график.

`deleteVertex(vertexId: Int):`

- Удаляет вершину по идентификатору и связанные с ней ребра.

`addEdge(sourceId: Int, targetId: Int, weight: Int):`

- Добавляет ребро между двумя вершинами с указанным весом.

`deleteEdge(sourceId: Int, targetId: Int):`

- Удаляет ребро между двумя вершинами.

`changeEdgeWeight(sourceId: Int, targetId: Int, newWeight: Int):`

- Изменяет вес существующего ребра.

`changeVertexColor(vertexId: Int, color: VertexColor):`

- Изменяет цвет вершины (для визуализации состояния).

`setStartVertex(vertexId: Int):`

- Устанавливает начальную вершину для запуска алгоритма Дейкстры.

`clearStartVertex():`

- Сбрасывает начальную вершину.

`saveGraph(), loadGraph(filePath: String):`

- Сохраняет график в файл и загружает график из файла.

Методы для выполнения и визуализации алгоритма Дейкстры

`startDijkstraAnimation():`

- Запускает анимацию выполнения алгоритма Дейкстры с текущими параметрами графа и начальной вершиной.

`pauseDijkstraAnimation()`, `resumeDijkstraAnimation()`:

- Управляют паузой и возобновлением анимации.

`nextDijkstraStep()`, `goToEnd()`:

- Переходит к следующему шагу анимации или сразу к финальному состоянию.

`resetDijkstra()`:

- Сбрасывает состояние алгоритма и анимации.

`dijkstra(graph: Graph, startVertexId: Int)`:

- Основная функция алгоритма Дейкстры.
- Инициализирует расстояния, очередь с приоритетом, посещённые вершины.
- На каждом шаге обновляет расстояния до соседей, сохраняет состояние шага для визуализации.
- Возвращает результат с историей шагов и кратчайшими расстояниями.

Методы для визуализации

`GraphCanvas`:

- Отрисовывает вершины и рёбра графа, выделяет текущие, посещённые и начальные вершины.
- Обрабатывает клики пользователя для добавления/удаления/редактирования элементов графа.

`DijkstraTable`, `FullDijkstraTable`:

- Отображают таблицы с промежуточными и итоговыми расстояниями на каждом шаге алгоритма.

4. ТЕСТИРОВАНИЕ

4.1. Тестирование графического интерфейса

Тестирование корректности работы графического интерфейса.

4.1.1. Ввод неподходящего числа:

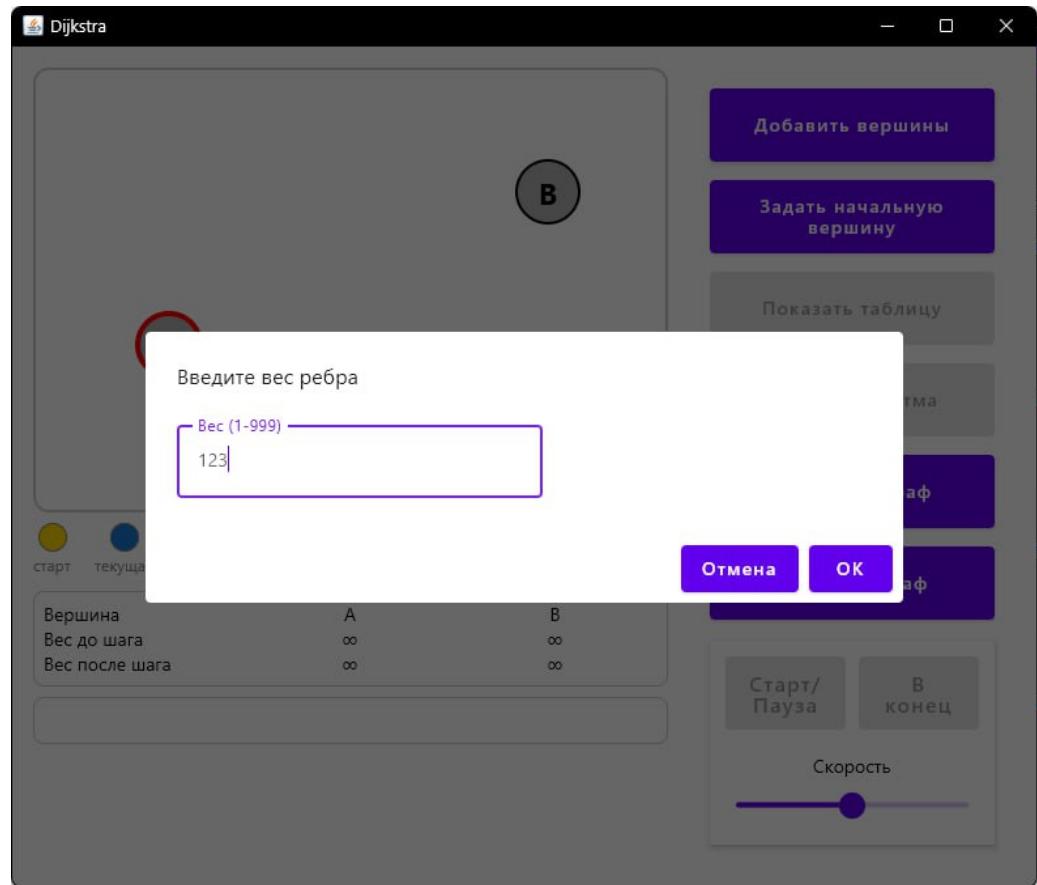


Рисунок 4.1.1. – демонстрация запрета на ввод неверных весов

Результат: невозможно ввести нецелое число, отрицательное число, число, меньшее 1 или большее 999.

4.1.2. Ввод кратных ребер:

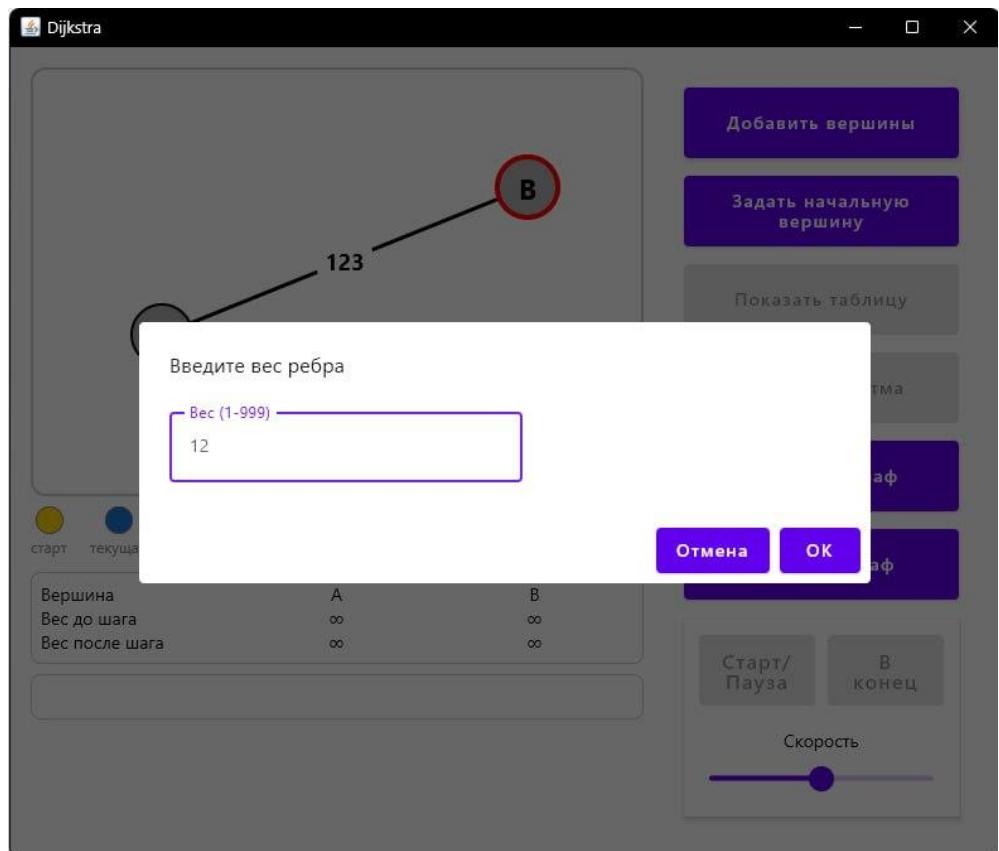


Рисунок 4.1.2.1. Процесс ввода кратного ребра

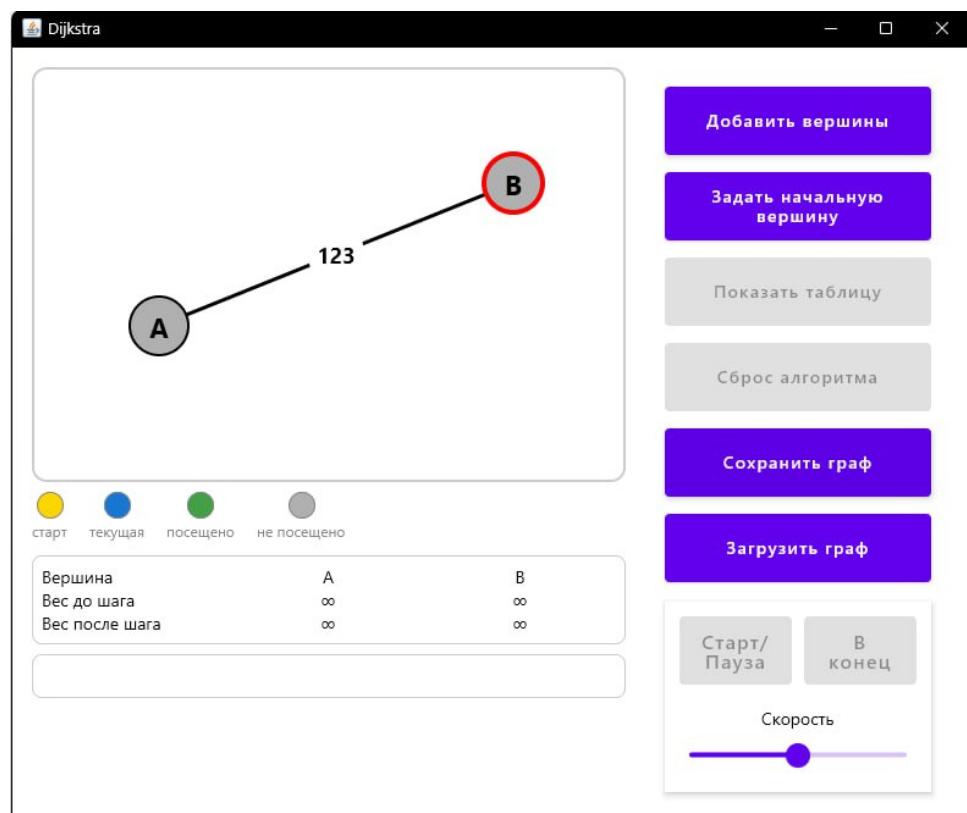


Рисунок 4.1.2.2. Результат ввода кратного ребра

Результат: отображается первое введенное ребро, кратность не учитывается.

4.1.3. Попытка ввода петли:

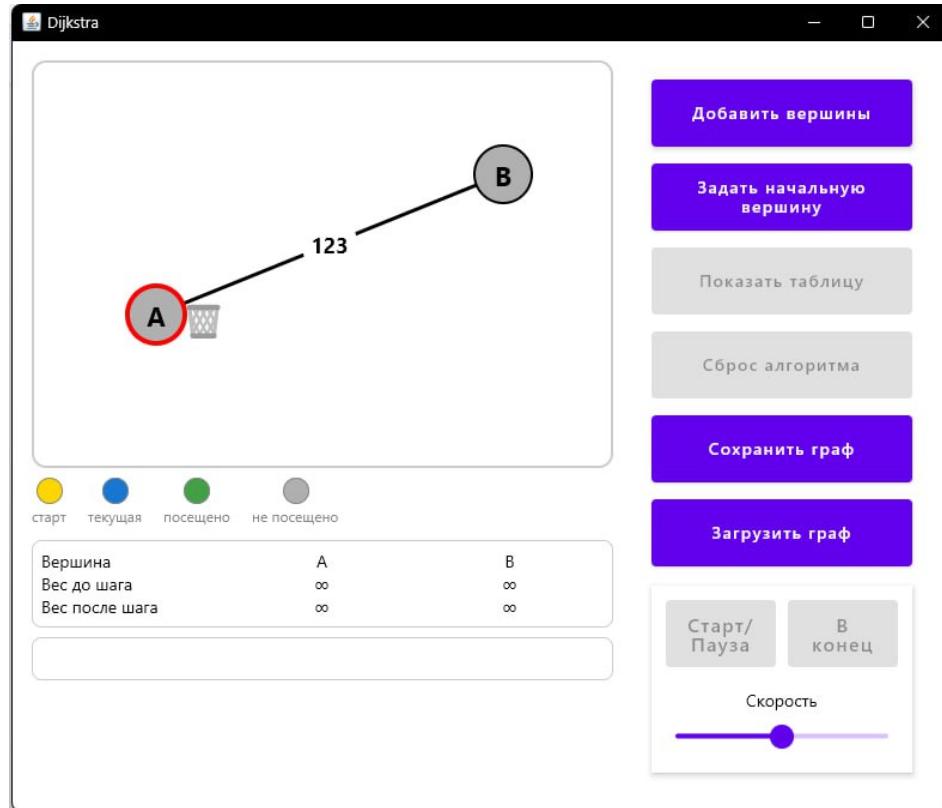


Рисунок 4.1.3. Попытка дважды нажать на вершину с целью создать ребро

Результат: лишь нажатие на 2 различные вершины подразумевает создание ребра. Петлю создать невозможно.

4.1.4. Передача не .json файла:

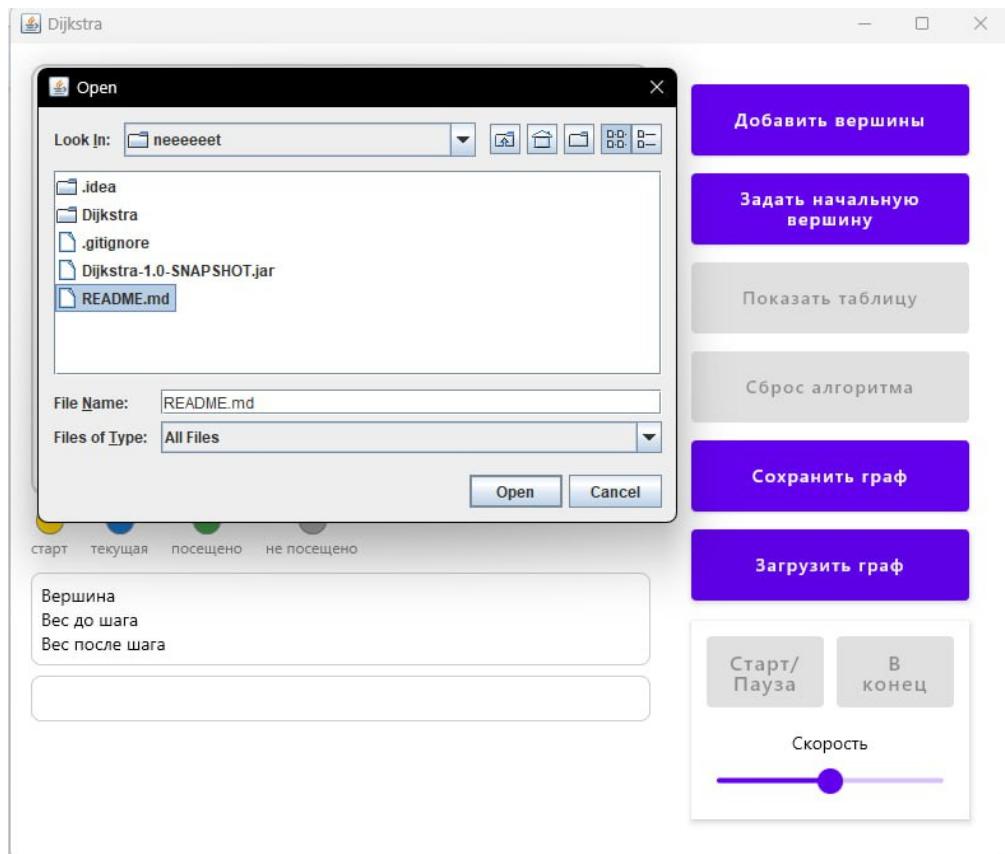


Рисунок 4.1.4. Передача файла без расширения .json

Результат: пустой холст, так как файлы без расширения json приложение не открывает.

4.1.5. Работоспособность кнопки «добавить вершины»:

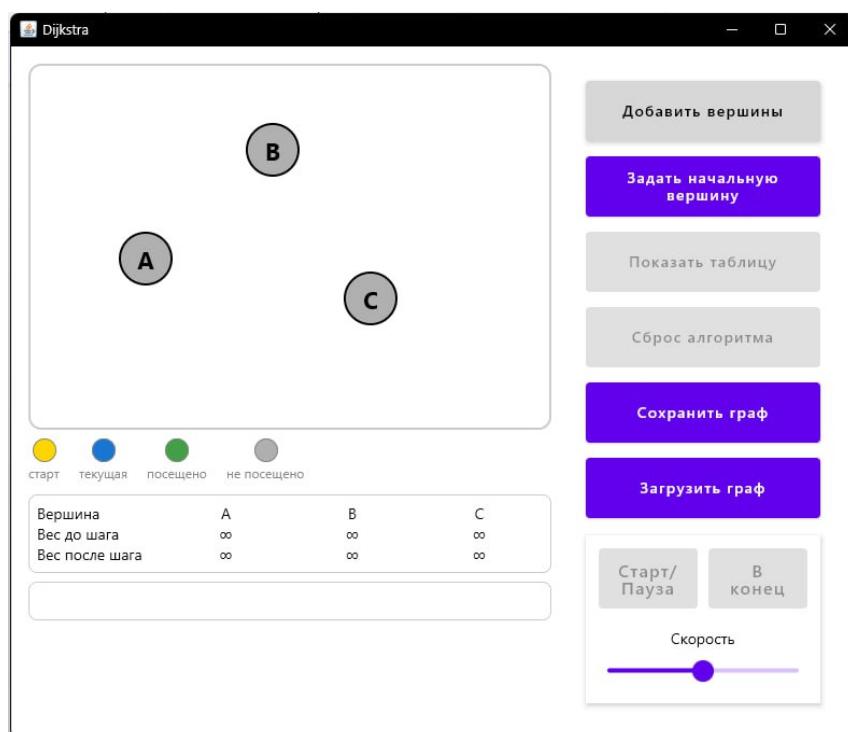


Рисунок 4.1.5. Активная кнопка «добавить вершины»

4.1.6. Двойное нажатие на ребро графа:

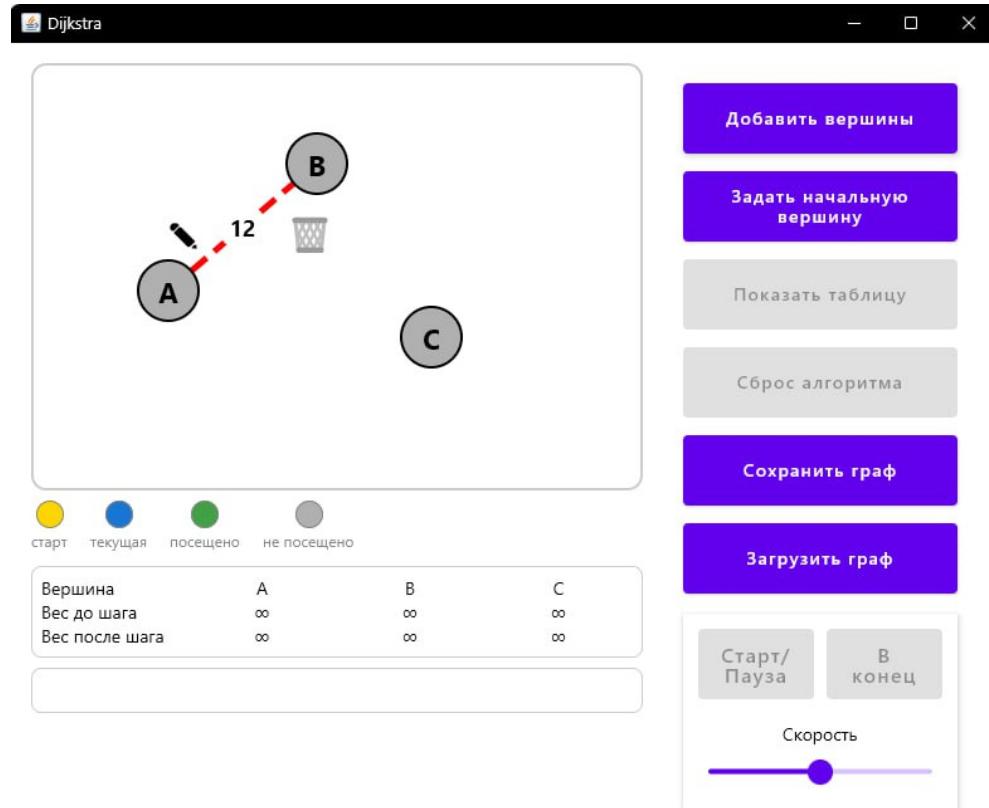


Рисунок 4.1.6. Результат нажатия дважды по ребру АВ

4.1.7. Редактирование веса ребра:

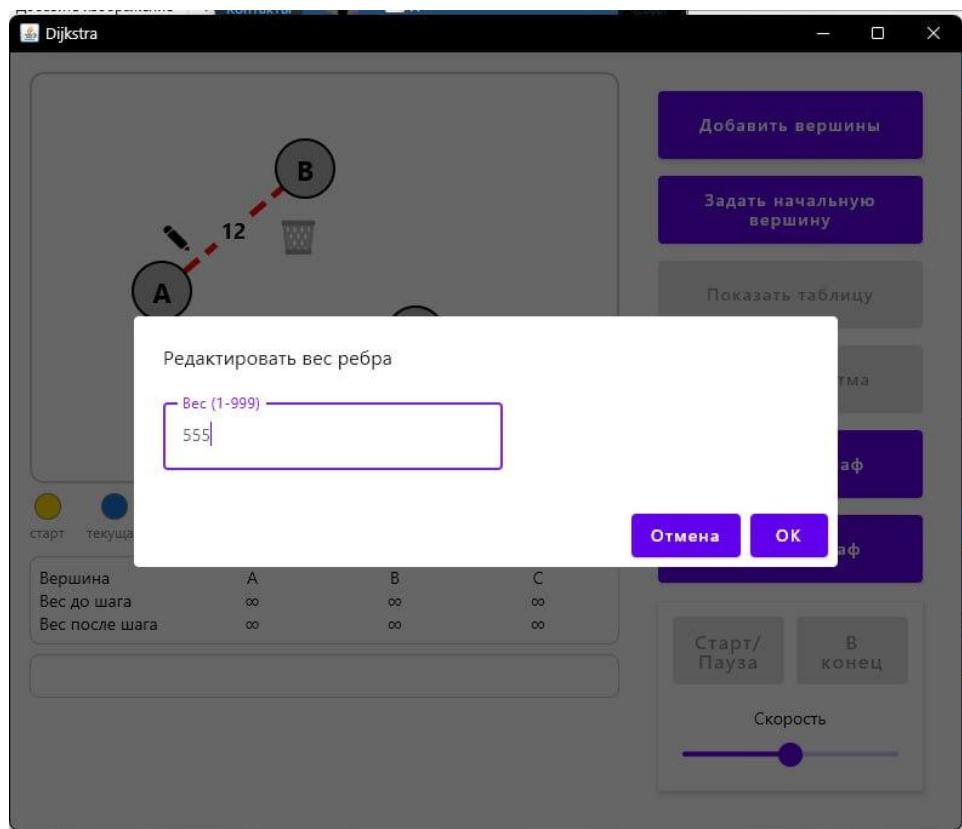


Рисунок 4.1.7.1. Ввод нового веса ребра

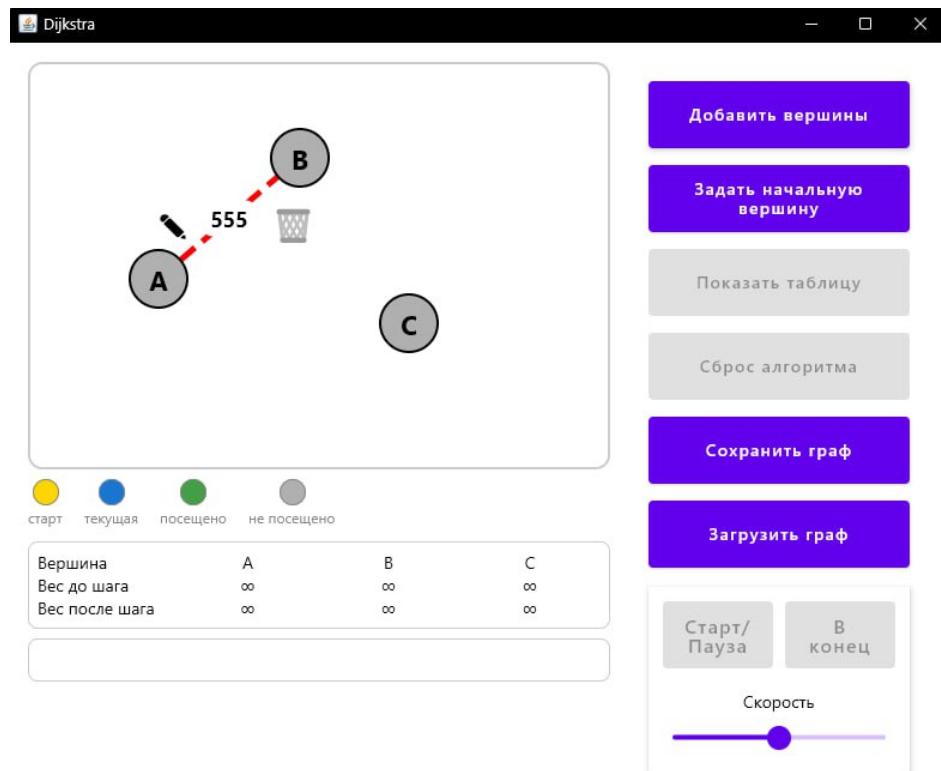


Рисунок 4.1.7.2. Результат редактирования веса ребра АВ

4.1.8. Двойное нажатие на вершину:

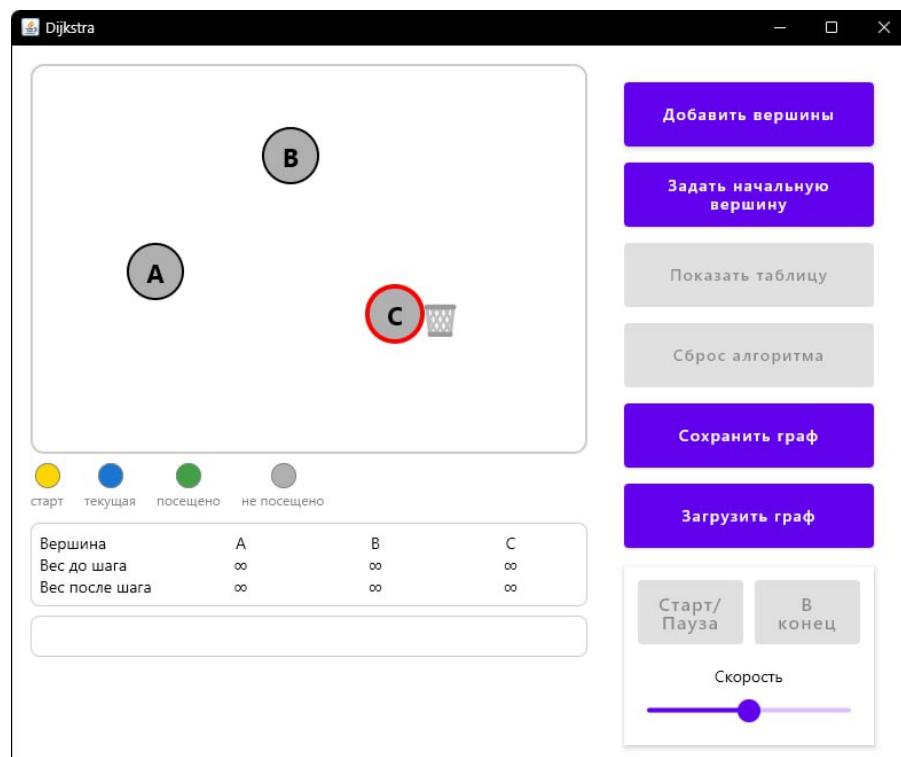


Рисунок 4.1.8. Результат двойного нажатия на вершину С

4.1.9. Задание начальной вершины:

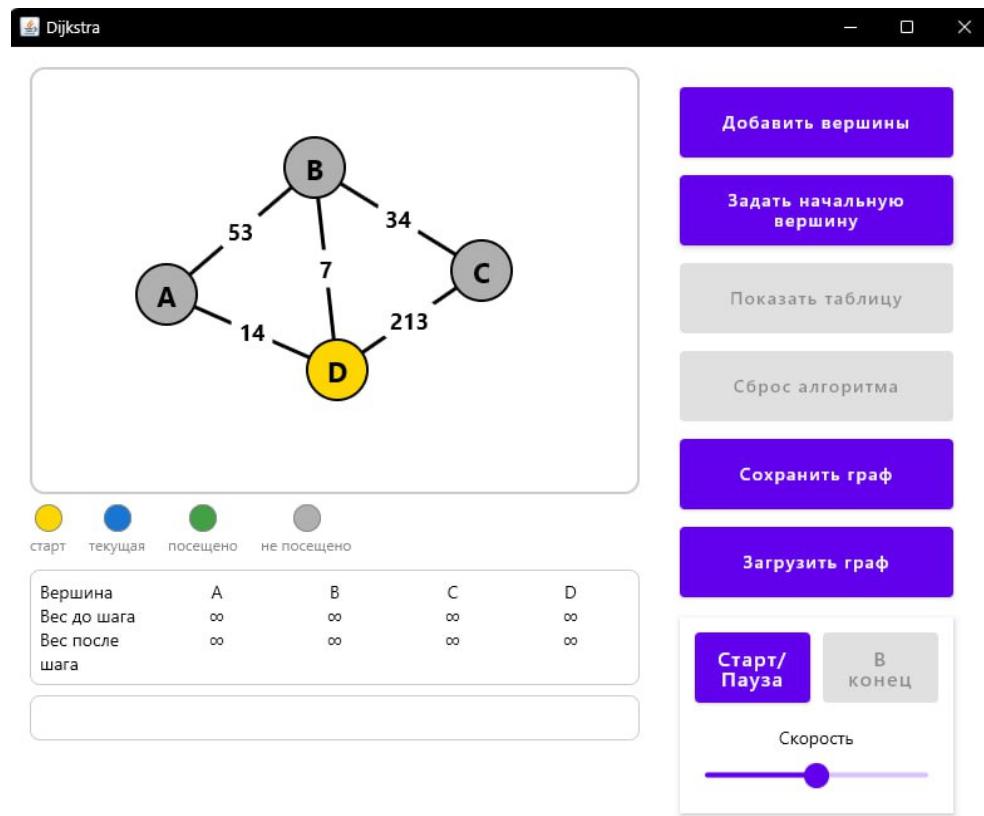


Рисунок 4.1.9. Результат работы кнопки «задать начальную вершину» и нажатия по вершине D

4.1.10. Попытка при уже заданной начальной вершине D задать еще одну:

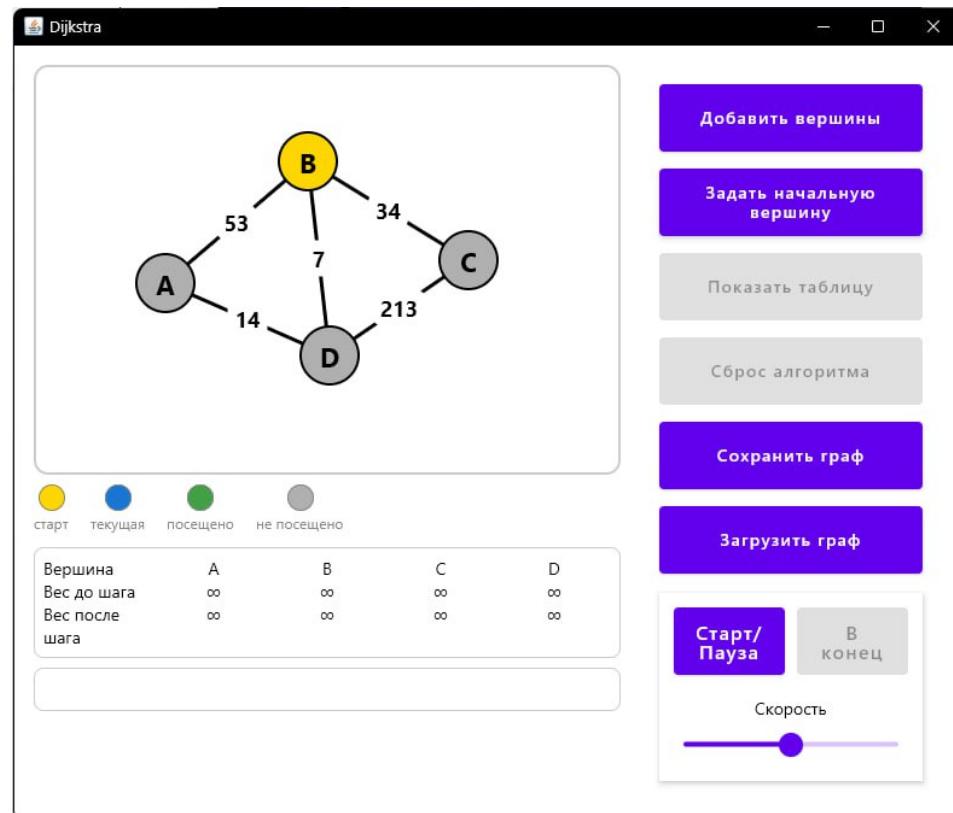


Рисунок 4.1.10 Результат задания начальной вершины В при уже заданной D

Результат: начальной вершиной становится последняя заданная.

4.1.11. Отображение процесса алгоритма:

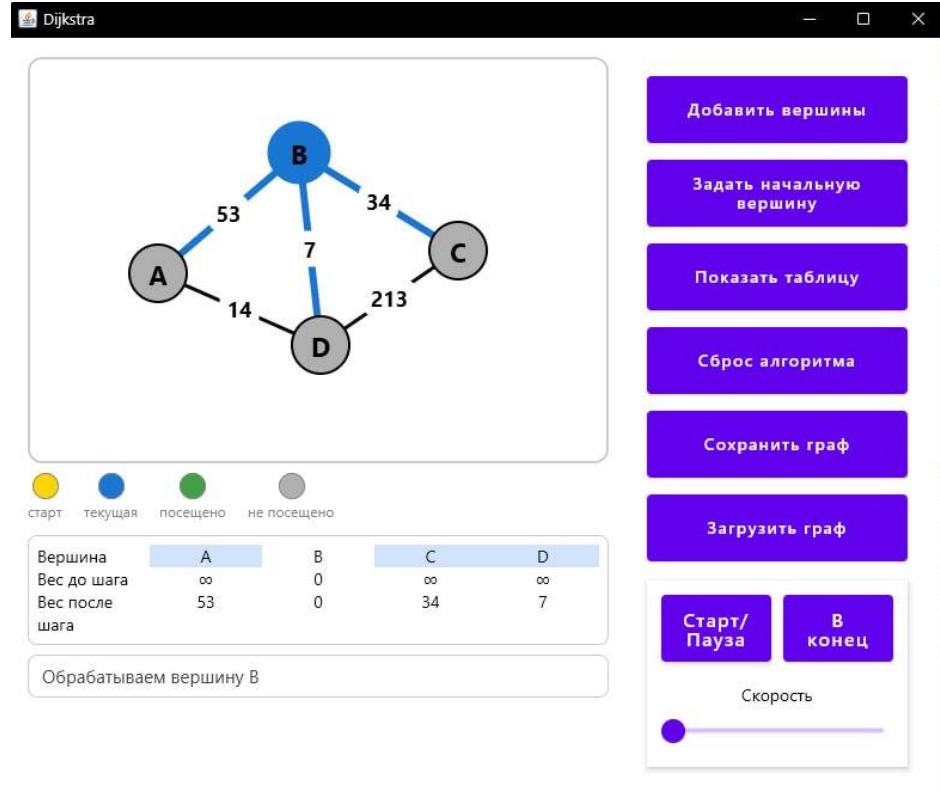


Рисунок 4.1.11.1. Отображение графа во время выполнения алгоритма

Таблица алгоритма Дейкстры					
Шаг	Действие	A	B	C	D
0	Инициализация	∞	0	∞	∞
1	Обрабатываем вершину B	53	0	34	7
2	Обрабатываем вершину D	21	0	34	7
3	Обрабатываем вершину A	21	0	34	7
4	Обрабатываем вершину C	21	0	34	7
Итог		21	0	34	7

Рисунок 4.1.11.2. Отображение полной таблицы во время выполнения алгоритма

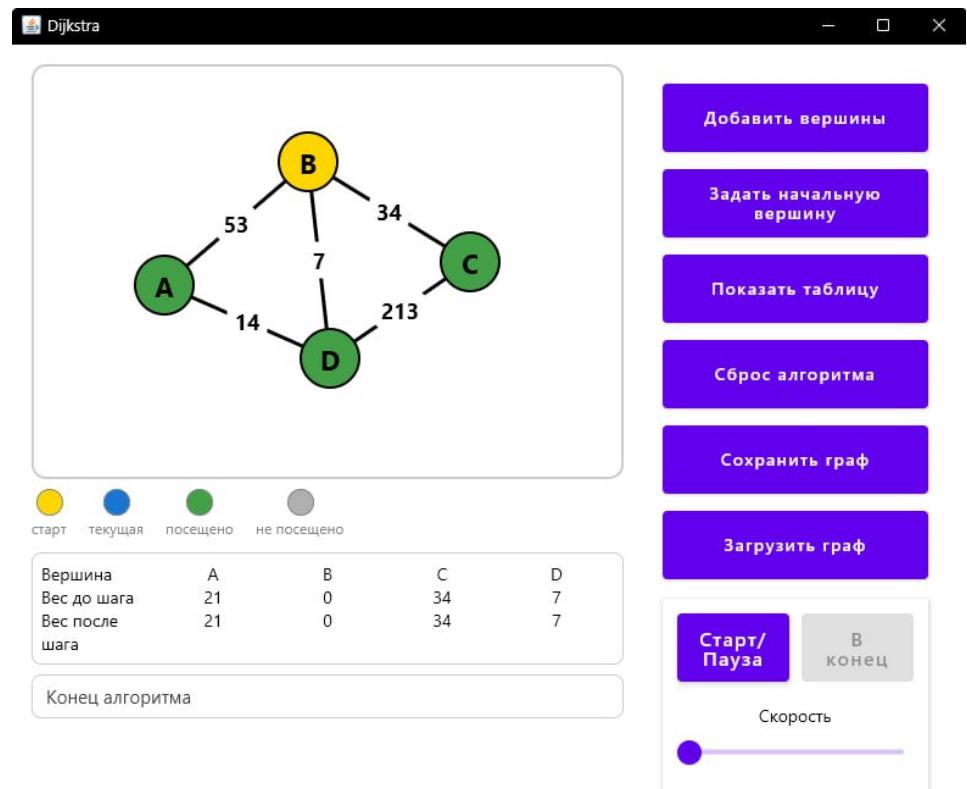


Рисунок 4.1.11.3. Отображение графа в конце алгоритма

Таблица алгоритма Дейкстры					
Шаг	Действие	A	B	C	D
0	Инициализация	∞	0	∞	∞
1	Обрабатываем вершину B	53	0	34	7
2	Обрабатываем вершину D	21	0	34	7
3	Обрабатываем вершину A	21	0	34	7
4	Обрабатываем вершину C	21	0	34	7
Итог		21	0	34	7

Рисунок 4.1.11.4. Отображение таблицы в конце алгоритма

4.1.12. Кнопка «сброс алгоритма»:

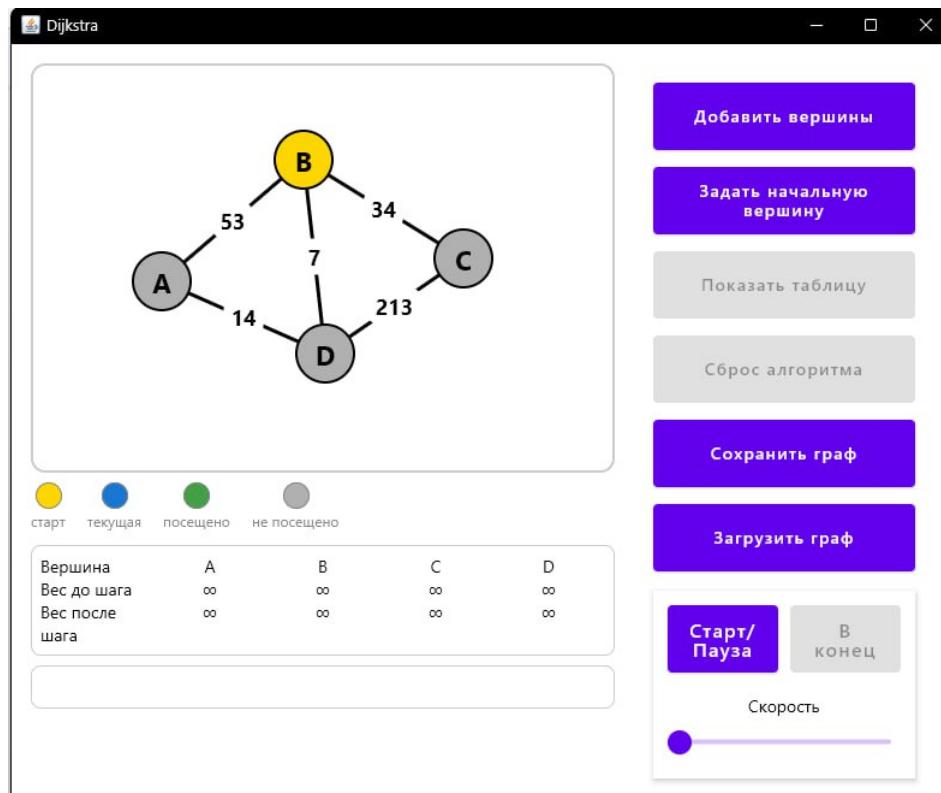


Рисунок 4.1.12. Результат работы кнопки «сброс алгоритма»

4.1.13. Кнопка «В конец»:

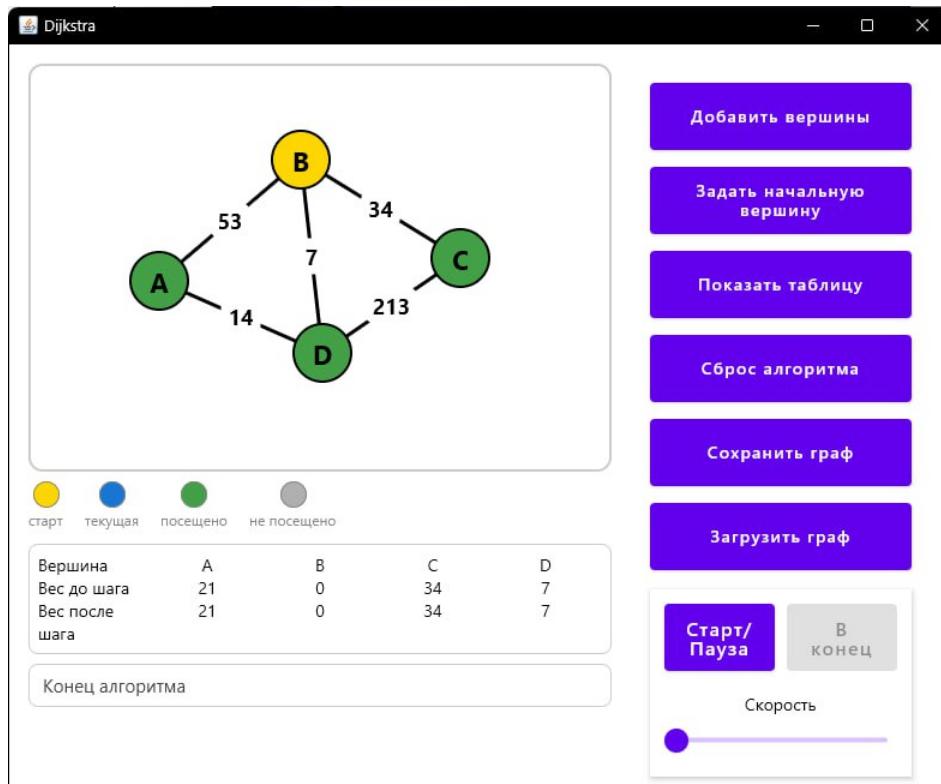


Рисунок 4.1.13 Результат работы кнопки «в конец»

4.2. Тестирование кода алгоритма

4.2.1. Алгоритм на простом графе:

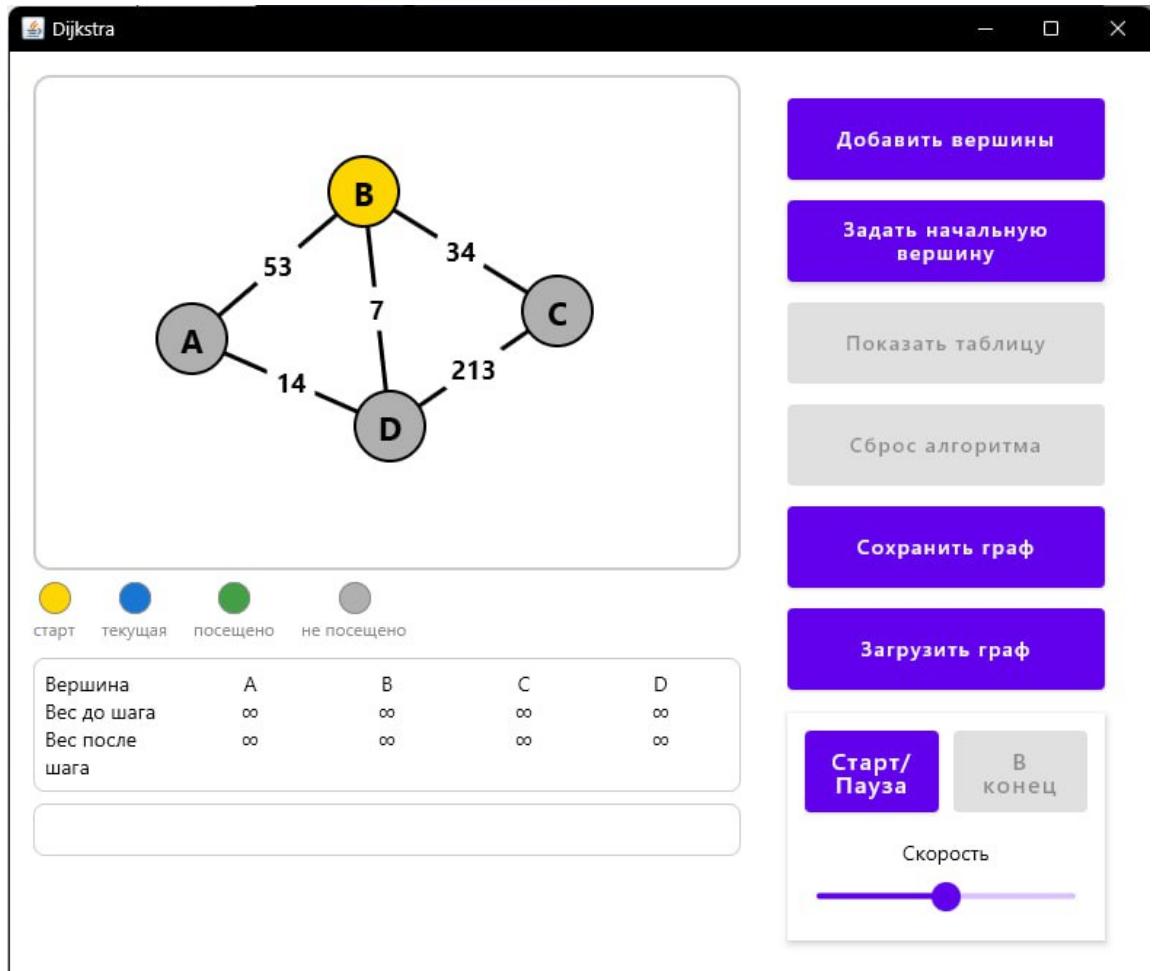


Рисунок 4.2.1.1. Граф до выполнения алгоритма

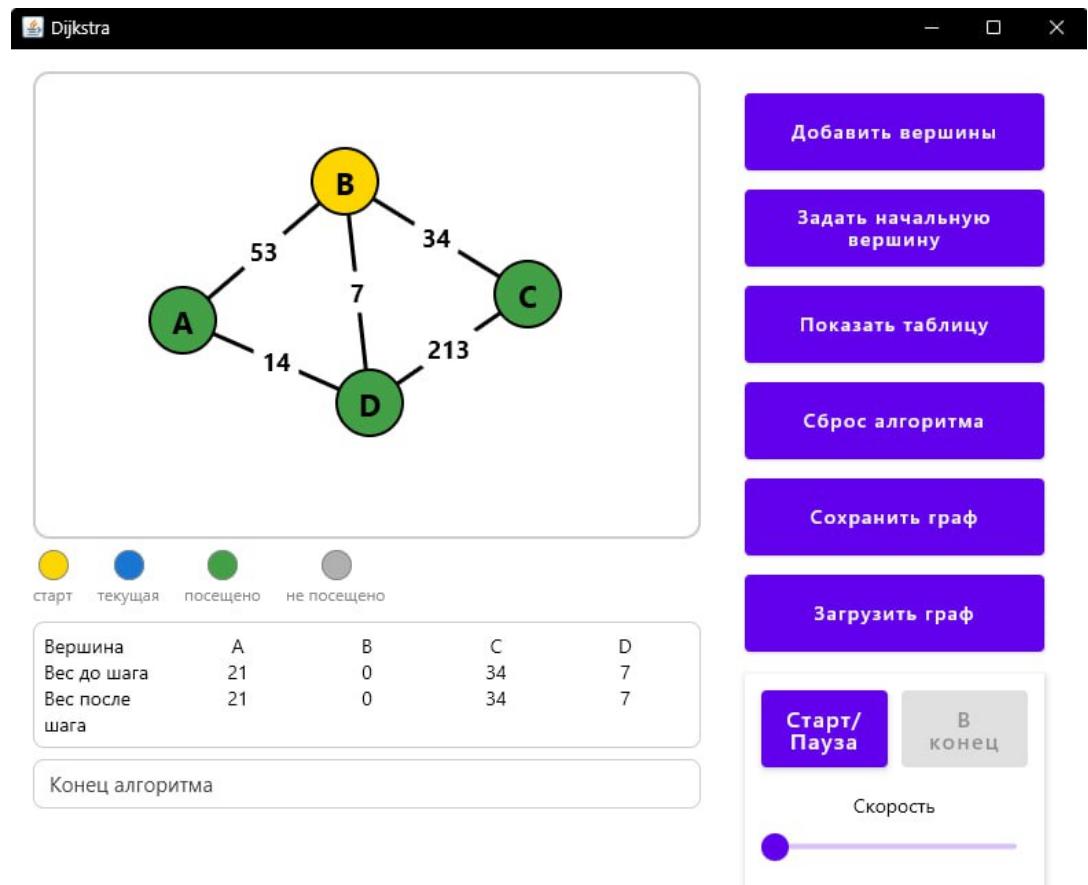


Таблица алгоритма Дейкстры					
Шаг	Действие	A	B	C	D
0	Инициализация	∞	0	∞	∞
1	Обрабатываем вершину B	53	0	34	7
2	Обрабатываем вершину D	21	0	34	7
3	Обрабатываем вершину A	21	0	34	7
4	Обрабатываем вершину C	21	0	34	7
Итог		21	0	34	7

Рисунок 4.2.1.3. Таблица после выполнения алгоритма
Результат: верно

4.2.2. Тестирование на графе с двумя компонентами связности:

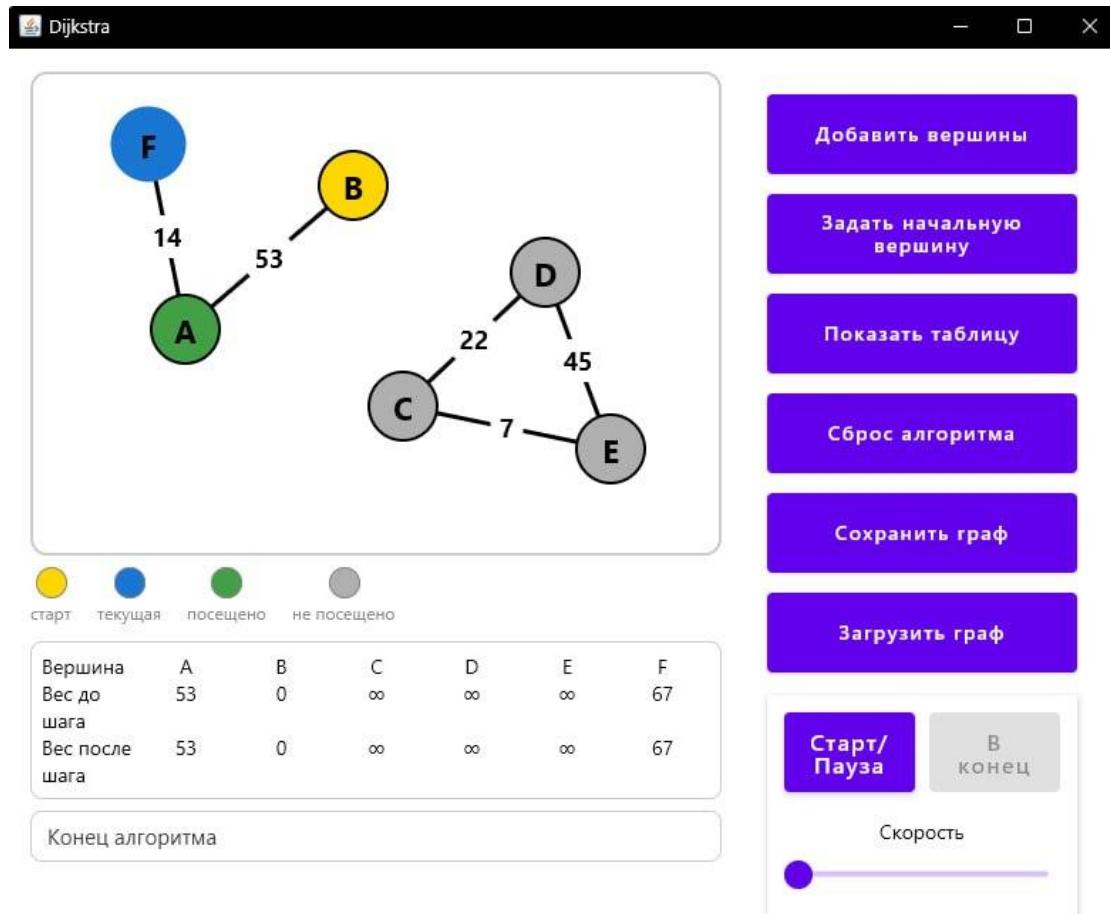


Рисунок 4.2.2. Результат работы алгоритма на графике с двумя КС

Результат: верно, т.к. из начальной вершины В нет пути в те вершины, которые находятся в отдельной компоненте связности.

ЗАКЛЮЧЕНИЕ

В ходе выполнения данной практики было успешно реализовано графическое приложение на языке Kotlin для визуализации работы алгоритма Дейкстры на графах.

Приложение предоставляет пользователю интуитивно понятный интерфейс, позволяющий создавать, редактировать и сохранять графы, а также запускать пошаговую анимацию поиска кратчайших путей. В процессе разработки были достигнуты следующие результаты.

Создан удобный пользовательский интерфейс с помощью библиотеки Compose for Desktop, что позволило реализовать современный и отзывчивый дизайн.

Пользователь может добавлять и удалять вершины и рёбра графа, задавать начальную вершину, изменять веса рёбер, а также сохранять и загружать графы из файлов.

Алгоритм Дейкстры реализован с возможностью пошаговой анимации. Каждый шаг алгоритма отображается в виде таблицы, где пользователь может наблюдать за изменением расстояний до вершин и посещёнными вершинами.

Это позволяет не только увидеть конечный результат, но и подробно проследить логику работы алгоритма на каждом этапе. Для удобства предусмотрены функции паузы, продолжения и мгновенного завершения анимации. Визуализация графа реализована с помощью холста, на котором отображаются вершины, рёбра и их состояния. Цветовое выделение помогает различать начальную, текущую и посещённые вершины, а также отслеживать изменения в процессе работы алгоритма.

Дополнительно реализованы вспомогательные таблицы и пояснения, что способствует лучшему пониманию принципов работы алгоритма Дейкстры. Все компоненты приложения были интегрированы в единую систему, проведено тестирование основных сценариев работы, что обеспечило корректное функционирование интерфейса, алгоритма и визуализации.

Поставленная цель — разработка приложения для визуализации алгоритма Дейкстры — была достигнута в полном объёме. Приложение предоставляет пользователю все необходимые инструменты для интерактивного изучения и анализа работы алгоритма поиска кратчайших путей. Реализованные функции и интерфейс соответствуют задачам, поставленным в начале работы.

Таким образом, данное приложение способствует более глубокому пониманию алгоритмов на графах и может быть использовано как в учебных целях, так и для демонстрации принципов работы алгоритма Дейкстры.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Документация по языку Kotlin. URL:
<https://kotlinlang.org/docs/home.html> (дата обращения: 26.06.2025).
2. Кормен, Т., Лейзерсон, Ч., Ривест, Р., Штайн, К. Алгоритмы: построение и анализ. М.: Вильямс, 2022. 681 с.
3. Скиена, С. С., Ревью, М. А. Алгоритмы. Руководство по разработке. СПб.: БХВ-Петербург, 2020. 228 с.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД

Main.kt

```
import androidx.compose.desktop.ui.tooling.preview.Preview
import androidx.compose.foundation.layout.Row
import androidx.compose.runtime.Composable
import androidx.compose.runtime.remember
import androidx.compose.ui.window.Window
import androidx.compose.ui.window.application
import controller.AppController
import visualisation.ControlPanel
import visualisation.MainWindowContent

fun main() = application {
    val appController = remember { AppController() }

    Window(onCloseRequest = ::exitApplication, title = "Dijkstra") {
        MainWindowContent(appController)
    }
}
```

Edge.kt

```
package graph

// Класс для представления ребра графа
data class Edge(var weight: Int, var source: Int, var target: Int) {

    fun updateWeight(newWeight: Int) {
        if (newWeight >= 0) {
            weight = newWeight
        } else {
            throw IllegalArgumentException("Weight cannot be negative")
        }
    }

    fun checkWeight(): Boolean {
        return weight >= 0
    }

    fun checkLoop(): Boolean {
        return source != target
    }

    fun isValid(): Boolean {
        return checkWeight() && checkLoop()
    }

    override fun toString(): String {
        return "Edge(from=$source to $target, weight=$weight)"
    }
}
```

Graph.kt

```
package graph

// Класс для представления графа
class Graph(var isDirected: Boolean = false) {

    private val _verticies = mutableListOf<Vertex>()
    private val _edges = mutableListOf<Edge>()

    val verticies: List<Vertex>
```

```

get() = _vertecies.toList()

val edges: List<Edge>
    get() = _edges.toList()

fun addVertex(vertex: Vertex) {
    _vertecies.add(vertex)
}

fun addEdge(edge: Edge) {
    _edges.add(edge)
}

// Получаем вершину по ID
fun getVertexById(id: Int): Vertex {
    return _vertecies.find { it.id == id } ?: throw IllegalArgumentException("Вершина с ID $id не найдена")
}

fun deleteVertexByid(id: Int) {
    // Удаляем все рёбра, связанные с этой вершиной
    _edges.removeAll { it.source == id || it.target == id }
    val vertex = getVertexById(id)
    _vertecies.remove(vertex)
}

// Удаляем ребро по ID начальной и конечной вершины
fun deleteEdgeBySourceAndTarget(source: Int, target: Int) {
    _edges.removeAll { it.source == source && it.target == target }
}

override fun toString(): String {
    val sb = StringBuilder()
    sb.append("Граф(isDirected=$isDirected)\n")
    sb.append("Вершины:\n")
    _vertecies.forEach { sb.append(" $it\n") }
    sb.append("Ребра:\n")
    _edges.forEach { sb.append(" $it\n") }
    return sb.toString()
}
}

```

Vertex.kt

package graph

```

// Класс для представления вершины графа
data class Vertex(var id: Int, var xCoordinate: Int, var yCoordinate: Int, var color: VertexColor = VertexColor.GRAY,
var name: String) {

    fun moveTo(newX: Int, newY: Int) {
        xCoordinate = newX
        yCoordinate = newY
    }

    fun paint(newColor: VertexColor) {
        color = newColor
    }

    fun checkCoordinates(): Boolean {
        return xCoordinate >= 0 && yCoordinate >= 0
    }

    fun checkId(): Boolean {

```

```

        return id >= 0
    }

    fun checkColor(): Boolean {
        return try {
            color in VertexColor.values()
        } catch (e: Exception) {
            println("Некорректный цвет вершины: $color")
            false
        }
    }

    fun isValid(): Boolean {
        return checkId() && checkCoordinates() && checkColor()
    }

}

```

VertexColor.kt

package graph

```

// Класс для представления цветов вершин графа
enum class VertexColor {
    YELLOW,
    GRAY,
    GREEN
}

```

GraphChecker.kt

package graph.GraphSerializer

```

import graph.Graph
import graph.Edge
import graph.Vertex
import java.lang.IllegalArgumentException

class GraphChecker {

    // Метод для проверки корректности графа
    fun isValidGraph(graph: Graph){

        // Проверка на наличие вершин
        if(graph.vertices.isEmpty()) {
            throw IllegalArgumentException("Граф не содержит вершин.")
        }

        val vertexIds = graph.vertices.map { it.id }.toSet()

        // Проверка связи вершин и ребер
        for (edge in graph.edges) {
            if (edge.source !in vertexIds || edge.target !in vertexIds) {
                throw IllegalArgumentException("Некорректное ребро: от вершины ${edge.source} к вершине ${edge.target} не существует.")
            }
        }

        // Проверка уникальности ID вершин
        if (vertexIds.size != graph.vertices.size) {
            throw IllegalArgumentException("Найдены дублирующиеся ID вершин.")
        }
    }
}

```

```

    }

fun isValidData(graph: Graph) {

    // Проверка корректности вершин
    for (vertex in graph.vertices) {
        if (!vertex.isValid()) {
            throw IllegalArgumentException("Некорректная вершина: ${vertex.id} с координатами
                (${vertex.xCoordinate}, ${vertex.yCoordinate}) и цветом ${vertex.color}.")

        }
    }

    // Проверка корректности ребер
    for (edge in graph.edges) {
        if (!edge.isValid()) {
            throw IllegalArgumentException("Некорректное ребро: от вершины ${edge.source} к вершине
                ${edge.target} с весом ${edge.weight}.")
        }
    }
}

```

GraphLoader.kt

package graph.GraphSerializer

```

import graph.Graph
import com.google.gson.Gson

// Класс для загрузки графа из файла
class GraphLoader {
    fun loadGraphFromFile(filePath: String): Graph {
        if (filePath.isEmpty()) {
            throw IllegalArgumentException("Файл не может быть пустым")
        }

        if (!filePath.endsWith(suffix = ".JSON", ignoreCase = true)) {
            throw IllegalArgumentException("Файл должен иметь расширение .JSON")
        }

        val gson = Gson()
        val fileContent = java.io.File(filePath).readText()
        val graph = gson.fromJson(fileContent, Graph::class.java)
        val graphChecker = GraphChecker()
        try {
            graphChecker.isValidGraph(graph)
            graphChecker.isValidData(graph)
        } catch (e: IllegalArgumentException) {
            throw IllegalArgumentException("Ошибка валидации графа: ${e.message}")
        }
        return graph
    }
}

```

GraphSaver.kt

package graph.GraphSerializer

```
import java.util.UUID
```

```

import com.google.gson.Gson
import graph.Graph

// Класс для сохранения графа в файл
class GraphSaver {

    fun saveGraphToFile(graph: Graph, filePath: String): String {
        if (filePath.isEmpty()) {
            throw IllegalArgumentException("File path cannot be empty")
        }

        val gson = Gson()
        val jsonContent = gson.toJson(graph)
        val fileName = "graph-data-" + UUID.randomUUID().toString() + ".json" // генератор уникального имени
        файла

        // Формируем полный путь к файлу
        val fullPath = if (filePath.endsWith("/")) {
            filePath + fileName
        } else {
            "$filePath/$fileName"
        }

        java.io.File(fullPath).writeText(jsonContent)
        return fullPath
    }
}

```

Dijkstra.kt

package algorithm

```

import java.util.*
import graph.Graph

// Класс для хранения состояния на каждом шаге алгоритма
data class DijkstraStep(
    val step: Int,
    val action: String,
    val currentVertexId: Int?,
    val distances: Map<Int, Int>,
    val visited: Set<Int>,
    val priorityQueue: List<Pair<Int, Int>>
)

// Класс для хранения итогового результата
data class DijkstraResult(
    val distances: Map<Int, Int>,
    val previous: Map<Int, Int?>,
    val steps: List<DijkstraStep>
)

fun dijkstra(graph: Graph, startVertexId: Int): DijkstraResult {
    // Инициализация
    val distances = mutableMapOf<Int, Int>()
    val previous = mutableMapOf<Int, Int?>()
    val visited = mutableSetOf<Int>()
    val pq = PriorityQueue<Pair<Int, Int>>(compareBy { it.first })

    graph.vertices.forEach {
        distances[it.id] = Int.MAX_VALUE
        previous[it.id] = null
    }
}

```

```

    }

    distances[startVertexId] = 0
    pq.add(0 to startVertexId)

    val steps = mutableListOf<DijkstraStep>()
    var stepCounter = 0

    // Начальное состояние
    steps.add(
        DijkstraStep(
            step = stepCounter,
            action = "Инициализация",
            currentVertexId = null,
            distances = distances.toMap(),
            visited = visited.toSet(),
            priorityQueue = pq.toList().map { it.second to it.first }
        )
    )

    while (pq.isNotEmpty()) {
        stepCounter++
        val (currentDistance, currentVertexId) = pq.poll()

        if (currentVertexId in visited) {
            continue
        }

        visited.add(currentVertexId)

        val action = "Обрабатываем вершину ${graph.getVertexById(currentVertexId).name}"

        // Обновляем расстояния до соседей
        graph.edges
            .filter { it.source == currentVertexId || it.target == currentVertexId }
            .forEach { edge ->
                val neighborId =
                    if (edge.source == currentVertexId) edge.target else edge.source
                if (neighborId !in visited) {
                    val newDistance = currentDistance + edge.weight
                    if (newDistance < (distances[neighborId] ?: Int.MAX_VALUE)) {
                        distances[neighborId] = newDistance
                        previous[neighborId] = currentVertexId
                        pq.add(newDistance to neighborId)
                    }
                }
            }
    }

    steps.add(
        DijkstraStep(
            step = stepCounter,
            action = action,
            currentVertexId = currentVertexId,
            distances = distances.toMap(),
            visited = visited.toSet(),
            priorityQueue = pq.toList().map { it.second to it.first }
        )
    )
}

return DijkstraResult(distances.toMap(), previous.toMap(), steps)
}

```

AppController.kt

package controller

```
import androidx.compose.runtime.getValue
import androidx.compose.runtime.mutableStateOf
import androidx.compose.runtime.setValue
import algorithm.DijkstraResult
import algorithm.dijkstra
import graph.*
import graph.GraphSerializer.GraphLoader
import graph.GraphSerializer.GraphSaver
import graph.VertexColor

class AppController {
    var graph by mutableStateOf(Graph())
        private set

    // Добавляем отслеживание начальной вершины
    var startVertexId by mutableStateOf<Int?>(null)
        private set

    var dijkstraResult by mutableStateOf<DijkstraResult?>(null)
        private set

    var notification by mutableStateOf<String?>(null)
        private set

    var errorMessage by mutableStateOf<String?>(null)
        private set

    // --- Состояние анимации алгоритма Дейкстры ---
    var isRunning by mutableStateOf(false)
        private set
    var isPaused by mutableStateOf(false)
        private set
    var currentStep by mutableStateOf(0)
        private set

    private val saver = GraphSaver()
    private val loader = GraphLoader()

    // Добавление вершины с заданными координатами
    fun addVertex(x: Int, y: Int) {
        val currentGraph = graph
        val name = getNextVertexName()
        val vertexId = if (currentGraph.vertices.isEmpty()) 0 else (currentGraph.vertices.maxOf { it.id } + 1)
        val newVertex = Vertex(vertexId, x, y, name = name)
        currentGraph.addVertex(newVertex)
        graph = createGraphCopy(currentGraph)
        notification = "Вершина добавлена: $name"
    }

    // Удаление вершины по ID
    fun deleteVertex(vertexId: Int) {
        try {
            val currentGraph = graph

            // Если удаляем начальную вершину, сбрасываем её
            if (startVertexId == vertexId) {
                startVertexId = null
            }
        }
    }
}
```

```

    val vertexName = currentGraph.vertices.find { it.id == vertexId }?.name

    currentGraph.deleteVertexByid(vertexId)
    graph = createGraphCopy(currentGraph)
    notification = "Вершина $vertexName удалена"
}
catch (e: Exception) {
    errorMessage = "Ошибка при удалении вершины: ${e.message}"
}
}

// Добавление ребра между вершинами
fun addEdge(sourceId: Int, targetId: Int, weight: Int = 1) {
    val currentGraph = graph
    // Проверка на кратные рёбра (без учёта направления)
    val exists = currentGraph.edges.any {
        (it.source == sourceId && it.target == targetId) ||
        (it.source == targetId && it.target == sourceId)
    }
    if (exists) {
        errorMessage = "Ребро между $sourceId и $targetId уже существует!"
        return
    }
    val newEdge = Edge(weight, sourceId, targetId)
    if (newEdge.isValid()) {
        currentGraph.addEdge(newEdge)
        graph = createGraphCopy(currentGraph)
        notification = "Ребро с весом $weight добавлено"
    } else {
        errorMessage = "Ошибка при добавлении ребра: некорректные данные"
    }
}

// Удаление ребра между вершинами
fun deleteEdge(sourceId: Int, targetId: Int) {
    val currentGraph = graph

    try {
        currentGraph.deleteEdgeBySourceAndTarget(sourceId, targetId)
        graph = createGraphCopy(currentGraph)
        notification = "Ребро удалено"
    }
    catch (e: Exception) {
        errorMessage = "Ошибка при удалении ребра: ${e.message}"
    }
}

// Изменение веса ребра между вершинами
fun changeEdgeWeight(sourceId: Int, targetId: Int, newWeight: Int) {
    val currentGraph = graph
    val edge = currentGraph.edges.find {
        (it.source == sourceId && it.target == targetId) ||
        (it.source == targetId && it.target == sourceId)
    }
    if (edge != null) {
        edge.updateWeight(newWeight)
        graph = createGraphCopy(currentGraph)
        notification = "Вес ребра изменён на $newWeight"
    } else {
        errorMessage = "Ребро между $sourceId и $targetId не найдено"
    }
}

```

```

}

// Изменение цвета вершины
fun changeVertexColor(vertexId: Int, color: VertexColor) {
    val currentGraph = graph
    val vertex = currentGraph.vertices.find { it.id == vertexId }
    if (vertex != null) {
        vertex.color = color
        graph = createGraphCopy(currentGraph)
        notification = "Цвет вершины $vertexId изменен на $color"
    } else {
        errorMessage = "Вершина с ID $vertexId не найдена"
    }
}

// Установка начальной вершины
fun setStartVertex(vertexId: Int) {
    val currentGraph = graph

    // Проверяем, существует ли вершина с таким ID
    val vertex = currentGraph.vertices.find { it.id == vertexId }
    if (vertex == null) {
        errorMessage = "Вершина с ID $vertexId не найдена"
        return
    }

    // Если уже есть начальная вершина, сбрасываем её цвет
    if (startVertexId != null) {
        val previousStartVertex = currentGraph.vertices.find { it.id == startVertexId }
        previousStartVertex?.color = VertexColor.GRAY
    }

    // Устанавливаем новую начальную вершину
    startVertexId = vertexId
    vertex.color = VertexColor.YELLOW

    graph = createGraphCopy(currentGraph)
    notification = "Начальная вершина установлена"
}

// Сброс начальной вершины
fun clearStartVertex() {
    if (startVertexId != null) {
        val currentGraph = graph
        val vertex = currentGraph.vertices.find { it.id == startVertexId }
        vertex?.color = VertexColor.GRAY
        startVertexId = null
        graph = createGraphCopy(currentGraph)
        notification = "Начальная вершина сброшена"
    }
}

// Получение информации о начальной вершине
fun getStartVertex(): Vertex? {
    return if (startVertexId != null) {
        graph.vertices.find { it.id == startVertexId }
    } else null
}

// Сохранение графа в файл
fun saveGraph() {
    try {

```

```

        val savedPath = saver.saveGraphToFile(graph, ".")
        notification = "Граф успешно сохранен в: $savedPath"
    } catch (e: Exception) {
        errorMessage = "Ошибка при сохранении графа: ${e.message}"
    }
}

// Загрузка графа из файла
fun loadGraph(filePath: String) {
    try {
        graph = loader.loadGraphFromFile(filePath)
        val startVertex = graph.vertices.find { it.color == VertexColor.YELLOW }
        startVertexId = startVertex?.id
        notification = "Граф успешно загружен из файла: $filePath"
    } catch (e: Exception) {
        errorMessage = e.message ?: "Ошибка при загрузке графа"
    }
}

// Вспомогательная функция для создания копии графа, чтобы обновить состояние
private fun createGraphCopy(original: Graph): Graph {
    val newGraph = Graph(original.isDirected)
    original.vertices.forEach { newGraph.addVertex(it.copy()) } // Копируем вершины
    original.edges.forEach { newGraph.addEdge(it.copy()) } // Копируем ребра
    return newGraph
}

// Генерация уникального имени для новой вершины
// Имена формируются как в Excel
private fun getNextVertexName(): String {
    val existingNames = graph.vertices.map { it.name }.toSet()
    var i = 0
    while (true) {
        val name = generateName(i)
        if (name !in existingNames) {
            return name
        }
        i++
    }
}

// Генерация имени для вершины по индексу
private fun generateName(index: Int): String {
    var num = index
    val nameBuilder = StringBuilder()
    while (num >= 0) {
        nameBuilder.insert(0, ('A' + num % 26))
        num = num / 26 - 1
    }
    return nameBuilder.toString()
}

// Запуск анимации алгоритма
fun startDijkstraAnimation() {
    if (graph.vertices.isEmpty() || startVertexId == null) return
    dijkstraResult = algorithm.dijkstra(graph, startVertexId!!)
    isRunning = true
    isPaused = false
    currentStep = 0
}

// Пауза анимации

```

```

fun pauseDijkstraAnimation() {
    if (isRunning) {
        isPaused = true
    }
}

// Возобновление анимации
fun resumeDijkstraAnimation() {
    if (isRunning && isPaused) {
        isPaused = false
    }
}

// Остановка анимации
fun stopDijkstraAnimation() {
    isRunning = false
    isPaused = false
    currentStep = 0
}

fun dismissError() {
    errorMessage = null
}

// Следующий шаг анимации
fun nextDijkstraStep() {
    val steps = dijkstraResult?.steps ?: return
    if (currentStep < steps.size - 1) {
        currentStep++
    } else {
        isRunning = false
        isPaused = false
    }
}

fun goToEnd() {
    val steps = dijkstraResult?.steps
    if (steps != null && steps.isNotEmpty()) {
        currentStep = steps.size - 1
        isRunning = false
        isPaused = false
    }
}

fun resetDijkstra() {
    dijkstraResult = null
    isRunning = false
    isPaused = false
    currentStep = 0
}
}

```

ControlPanel.kt

package visualisation

```

import androidx.compose.foundation.layout.*
import androidx.compose.material.Button
import androidx.compose.material.Slider
import androidx.compose.material.Text
import androidx.compose.material.Surface

```

```

import androidx.compose.runtime.Composable
import androidx.compose.ui.Alignment
import androidx.compose.ui.Modifier
import androidx.compose.ui.unit.dp
import androidx.compose.ui.unit.sp
import androidx.compose.ui.text.style.TextAlign
import androidx.compose.ui.graphics.Color
import androidx.compose.material.ButtonDefaults

import controller.AppController

@Composable
fun ControlPanel(
    controller: AppController,
    onAddVertexClick: () -> Unit = {},
    isAddVertexModeActive: Boolean = false,
    onLoadGraphClick: () -> Unit = {},
    onSetStartVertexClick: () -> Unit = {},
    onStartPauseClick: () -> Unit = {},
    isStartPauseEnabled: Boolean = true,
    speedValue: Float = 0.5f,
    onSpeedChange: (Float) -> Unit = {},
    isSpeedEnabled: Boolean = true,
    onToEndClick: () -> Unit = {},
    isToEndEnabled: Boolean = true,
    onShowTableClick: () -> Unit = {},
    isShowTableEnabled: Boolean = true,
    isTableShown: Boolean = false,
    onResetAlgorithmClick: () -> Unit = {},
    isResetEnabled: Boolean = true
) {
    Column(
        modifier = Modifier
            .fillMaxHeight()
            .width(250.dp)
            .padding(16.dp),
        verticalArrangement = Arrangement.SpaceBetween
    ) {
        Column(verticalArrangement = Arrangement.spacedBy(14.dp)) {
            ControlButton(
                text = "Добавить вершины",
                height = 56.dp,
                onClick = {
                    onAddVertexClick()
                },
                enabled = true,
                buttonColors = if (isAddVertexModeActive) ButtonDefaults.buttonColors(buttonColor =
Color(0xFFE0E0E0)) else ButtonDefaults.buttonColors()
            )
            ControlButton(
                text = "Задать начальную вершину",
                height = 56.dp,
                onClick = onSetStartVertexClick,
                enabled = true
            )
            ControlButton(
                text = if (isTableShown) "Скрыть таблицу" else "Показать таблицу",
                height = 56.dp,
                onClick = onShowTableClick,
                enabled = isShowTableEnabled
            )
            ControlButton(

```

```

        text = "Сброс алгоритма",
        height = 56.dp,
        onClick = onResetAlgorithmClick,
        enabled = isResetEnabled
    )
    ControlButton("Сохранить граф",
    height = 56.dp,
    onClick = {
        controller.saveGraph()
    },
    enabled = true
)
ControlButton(
    text = "Загрузить граф",
    height = 56.dp,
    onClick = onLoadGraphClick,
    enabled = true
)
}
Surface(
    modifier = Modifier
        .fillMaxWidth()
        .padding(top = 16.dp),
    elevation = 4.dp
) {
    Column(
        modifier = Modifier.padding(12.dp),
        horizontalAlignment = Alignment.CenterHorizontally
) {
    Row(modifier = Modifier.fillMaxWidth(), horizontalArrangement = Arrangement.spacedBy(10.dp)) {
        ControlButton(
            text = "Старт/Пауза",
            fontSize = 16.sp,
            height = 56.dp,
            enabled = isStartPauseEnabled,
            modifier = Modifier.weight(1f),
            onClick = onStartPauseClick
        )
        ControlButton(
            text = "В конец",
            fontSize = 16.sp,
            height = 56.dp,
            enabled = isToEndEnabled,
            modifier = Modifier.weight(1f),
            onClick = onToEndClick
        )
    }
    Spacer(modifier = Modifier.height(18.dp))
    Text("Скорость")
    Slider(
        value = speedValue,
        onValueChange = onSpeedChange,
        enabled = isSpeedEnabled,
        modifier = Modifier.fillMaxWidth()
    )
}
}
}
}

@Composable
fun ControlButton(

```

```

text: String,
fontSize: androidx.compose.ui.unit.TextUnit = androidx.compose.ui.unit.TextUnit.Unspecified,
height: androidx.compose.ui.unit.Dp = 40.dp,
onClick: () -> Unit = {},
enabled: Boolean = true,
modifier: Modifier = Modifier,
buttonColors: androidx.compose.material.ButtonColors = ButtonDefaults.buttonColors()
) {
    Button(
        onClick = onClick,
        enabled = enabled,
        modifier = modifier
            .fillMaxWidth()
            .height(height),
        colors = buttonColors
    ) {
        Text(
            text,
            modifier = Modifier.fillMaxWidth(),
            textAlign = TextAlign.Center,
            fontSize = fontSize
        )
    }
}

```

DijkstraTable.kt

package visualisation

```

import androidx.compose.foundation.background
import androidx.compose.foundation.border
import androidx.compose.foundation.layout.*
import androidx.compose.material.Text
import androidx.compose.runtime.Composable
import androidx.compose.ui.Alignment
import androidx.compose.ui.Modifier
import androidx.compose.ui.graphics.Color
import androidx.compose.ui.unit.dp
import androidx.compose.ui.unit.sp
import controller.AppController

@Composable
fun DijkstraTable(controller: AppController) {
    val result = controller.dijkstraResult
    val steps = result?.steps ?: emptyList()
    val vertecies = controller.graph.vertecies
    val currentStep = controller.currentStep

    val beforeDistances = if (currentStep > 0 && steps.isNotEmpty()) steps[currentStep - 1].distances else
        vertecies.associate { it.id to "∞" }
    val afterDistances = if (steps.isNotEmpty() && currentStep < steps.size) steps[currentStep].distances else
        vertecies.associate { it.id to "∞" }

    val currentStepData = steps.getOrNull(currentStep)
    val highlightVertexIds = if (currentStepData?.currentVertexId != null)
        controller.graph.edges
            .filter { it.source == currentStepData.currentVertexId || it.target == currentStepData.currentVertexId }
            .map { if (it.source == currentStepData.currentVertexId) it.target else it.source }
            .filter { it !in currentStepData.visited }
    else emptyList()

    Column(
        modifier = Modifier

```

```

.fillMaxWidth()
.background(Color.White, shape = androidx.compose.foundation.shape.RoundedCornerShape(8.dp))
.border(1.dp, Color(0xFFCCCCCC), shape = androidx.compose.foundation.shape.RoundedCornerShape(8.dp))
.padding(8.dp)
) {
    Row(Modifier.fillMaxWidth()) {
        Text("Вершина", modifier = Modifier.weight(1f), fontSize = 14.sp)
        vertecies.forEach { v ->
            val highlight = v.id in highlightVertexIds
            Text(
                v.name,
                modifier = Modifier.weight(1f).background(if (highlight) Color(0xFFD2E3FC) else Color.Transparent),
                fontSize = 14.sp,
                textAlign = androidx.compose.ui.text.style.TextAlign.Center
            )
        }
    }
    Row(Modifier.fillMaxWidth()) {
        Text("Вес до шага", modifier = Modifier.weight(1f), fontSize = 14.sp)
        vertecies.forEach { v ->
            val value = beforeDistances[v.id]?.let { if (it == Int.MAX_VALUE) "∞" else it.toString() } ?: "∞"
            Text(value, modifier = Modifier.weight(1f), fontSize = 14.sp, textAlign =
                androidx.compose.ui.text.style.TextAlign.Center)
        }
    }
    Row(Modifier.fillMaxWidth()) {
        Text("Вес после шага", modifier = Modifier.weight(1f), fontSize = 14.sp)
        vertecies.forEach { v ->
            val value = afterDistances[v.id]?.let { if (it == Int.MAX_VALUE) "∞" else it.toString() } ?: "∞"
            Text(value, modifier = Modifier.weight(1f), fontSize = 14.sp, textAlign =
                androidx.compose.ui.text.style.TextAlign.Center)
        }
    }
}

```

FullDijkstraTable.kt

package visualisation

```

import androidx.compose.foundation.background
import androidx.compose.foundation.border
import androidx.compose.foundation.layout.*
import androidx.compose.material.Divider
import androidx.compose.material.Text
import androidx.compose.runtime.Composable
import androidx.compose.ui.Alignment
import androidx.compose.ui.Modifier
import androidx.compose.ui.graphics.Color
import androidx.compose.ui.text.font.FontWeight
import androidx.compose.ui.text.style.TextAlign
import androidx.compose.ui.unit.dp
import androidx.compose.ui.unit.sp
import controller.AppController

```

```

@Composable
fun FullDijkstraTable(controller: AppController) {
    val result = controller.dijkstraResult ?: return
    val steps = result.steps
    val vertecies = controller.graph.vertecies
    val currentStep = controller.currentStep

```

```

Column(
    modifier = Modifier
        .fillMaxWidth()
        .background(Color.White)
        .border(1.dp, Color(0xFFCCCCCC))
        .padding(8.dp)
) {
    // Header
    Row(Modifier.fillMaxWidth()) {
        Text("Шаг", modifier = Modifier.width(40.dp), fontWeight = FontWeight.Bold, fontSize = 13.sp, textAlign = TextAlign.Center)
        Text("Действие", modifier = Modifier.width(120.dp), fontWeight = FontWeight.Bold, fontSize = 13.sp, textAlign = TextAlign.Center)
        vertecies.forEach { v ->
            Text(v.name, modifier = Modifier.weight(1f), fontWeight = FontWeight.Bold, fontSize = 13.sp, textAlign = TextAlign.Center)
        }
    }
    // Steps
    steps.forEachIndexed { idx, step ->
        val isFuture = idx > currentStep
        val rowColor = if (isFuture) Color(0xFFFF0F0F0) else Color.White
        Row(
            Modifier
                .fillMaxWidth()
                .background(rowColor)
        ) {
            Text(idx.toString(), modifier = Modifier.width(40.dp), fontSize = 13.sp, textAlign = TextAlign.Center)
            Text(step.action, modifier = Modifier.width(120.dp), fontSize = 13.sp, textAlign = TextAlign.Center)
            vertecies.forEach { v ->
                val value = step.distances[v.id]?.let { if (it == Int.MAX_VALUE) "∞" else it.toString() } ?: "∞"
                Text(value, modifier = Modifier.weight(1f), fontSize = 13.sp, textAlign = TextAlign.Center, color = if (isFuture) Color.Gray else Color.Black)
            }
        }
        Divider(color = Color(0xFFCCCCCC), thickness = 1.dp)
    }
    // Итоговая строка с длинами всех путей
    Row(Modifier.fillMaxWidth().background(Color(0xFFE3F2FD))) {
        Text("Итог", modifier = Modifier.width(160.dp), fontWeight = FontWeight.Bold, fontSize = 13.sp, textAlign = TextAlign.Center)
        vertecies.forEach { v ->
            val value = result.distances[v.id]?.let { if (it == Int.MAX_VALUE) "∞" else it.toString() } ?: "∞"
            Text(value, modifier = Modifier.weight(1f), fontWeight = FontWeight.Bold, fontSize = 13.sp, textAlign = TextAlign.Center, color = Color(0xFF1976D2))
        }
    }
}

```

GraphCanvas.kt

package visualisation

```

import androidx.compose.foundation.background
import androidx.compose.foundation.border
import androidx.compose.foundation.layout.Box
import androidx.compose.foundation.layout.fillMaxSize
import androidx.compose.foundation.shape.RoundedCornerShape
import androidx.compose.runtime.Composable
import androidx.compose.ui.Modifier
import androidx.compose.ui.graphics.Color
import androidx.compose.ui.unit.dp
import graph.Vertex

```

```

import graph.VertexColor
import androidx.compose.foundation.Canvas
import androidx.compose.ui.geometry.Offset
import androidx.compose.ui.graphics.drawscope.Stroke
import androidx.compose.ui.text.TextStyle
import androidx.compose.ui.unit.sp
import androidx.compose.ui.graphics.nativeCanvas
import androidx.compose.ui.text.font.FontWeight
import androidx.compose.ui.Alignment
import androidx.compose.ui.input.pointer.pointerInput
import androidx.compose.foundation.gestures.detectTapGestures
import androidx.compose.ui.platform.LocalDensity
import androidx.compose.ui.text.font.FontFamily
import androidx.compose.ui.text.style.TextAlign
import androidx.compose.ui.text.rememberTextMeasurer
import androidx.compose.ui.graphics.drawscope.drawIntoCanvas
import androidx.compose.ui.text.drawText
import graph.Edge
import androidx.compose.runtime.getValue
import androidx.compose.runtime.setValue
import androidx.compose.runtime.mutableStateOf
import androidx.compose.material.AlertDialog
import androidx.compose.material.OutlinedTextField
import androidx.compose.material.Button
import androidx.compose.material.Text
import androidx.compose.runtime.*
import androidx.compose.ui.graphics.PathEffect
import androidx.compose.ui.res.painterResource
import androidx.compose.ui.unit.DpOffset
import algorithm.DijkstraStep

@Composable
fun GraphCanvas(
    vertices: List<Vertex>,
    edges: List<Edge> = emptyList(),
    currentStepData: DijkstraStep? = null,
    onCanvasClick: ((x: Int, y: Int) -> Unit)? = null,
    onAddEdge: ((sourceId: Int, targetId: Int, weight: Int) -> Unit)? = null,
    onDeleteEdge: ((sourceId: Int, targetId: Int) -> Unit)? = null,
    onDeleteVertex: ((vertexId: Int) -> Unit)? = null
) {
    val density = LocalDensity.current
    val textMeasurer = rememberTextMeasurer()
    var selectedVertices by remember { mutableStateOf(listOf<Int>()) }
    var showWeightDialog by remember { mutableStateOf(false) }
    var edgeWeightInput by remember { mutableStateOf("") }
    var selectedEdge by remember { mutableStateOf<Pair<Int, Int>?>(null) }
    var showDeleteIcon by remember { mutableStateOf(false) }
    var selectedVertex by remember { mutableStateOf<Int?>(null) }
    var showDeleteVertexIcon by remember { mutableStateOf(false) }
    var showEditWeightDialog by remember { mutableStateOf(false) }
    var editWeightInput by remember { mutableStateOf("") }

    // Размер иконки
    val iconSize = with(density) { 28.dp.toPx() }

    Box(
        modifier = Modifier
            .fillMaxSize()
            .background(Color.White, shape = RoundedCornerShape(12.dp))
            .border(2.dp, Color(0xFFCCCCCC), shape = RoundedCornerShape(12.dp))
    )
}

```

```

.pointerInput(onCanvasClick, edges, selectedEdge, showDeleteIcon, selectedVertex, showDeleteVertexIcon,
showEditWeightDialog) {
    detectTapGestures(
        onDoubleTap = { offset ->
            // Проверка двойного клика по ребру
            var foundEdge: Pair<Int, Int>? = null
            for (edge in edges) {
                val from = vertices.find { it.id == edge.source }
                val to = vertices.find { it.id == edge.target }
                if (from != null && to != null) {
                    val start = Offset(from.xCoordinate.toFloat(), from.yCoordinate.toFloat())
                    val end = Offset(to.xCoordinate.toFloat(), to.yCoordinate.toFloat())
                    val dist = distancePointToSegment(offset, start, end)
                    if (dist < 16f) {
                        foundEdge = Pair(edge.source, edge.target)
                        break
                    }
                }
            }
            if (foundEdge != null) {
                selectedEdge = foundEdge
                showDeleteIcon = true
                selectedVertex = null
                showDeleteVertexIcon = false
            } else {
                // Проверка двойного клика по вершине
                val clickedVertex = vertices.indexOfLast {
                    val dx = it.xCoordinate - offset.x
                    val dy = it.yCoordinate - offset.y
                    dx * dx + dy * dy <= 24f * 24f
                }
                val clickedVertexId = if (clickedVertex != -1) vertices[clickedVertex].id else null
                if (clickedVertexId != null) {
                    selectedVertex = clickedVertexId
                    showDeleteVertexIcon = true
                    selectedEdge = null
                    showDeleteIcon = false
                } else {
                    selectedVertex = null
                    showDeleteVertexIcon = false
                    selectedEdge = null
                    showDeleteIcon = false
                }
            }
        },
        onTap = { offset ->
            // Удаление вершины
            if (showDeleteVertexIcon && selectedVertex != null) {
                val v = vertices.find { it.id == selectedVertex }
                if (v != null) {
                    val center = Offset(v.xCoordinate.toFloat(), v.yCoordinate.toFloat())
                    val iconRect = androidx.compose.ui.geometry.Rect(
                        center.x + iconSize * 0.7f,
                        center.y - iconSize / 2,
                        center.x + iconSize * 0.7f + iconSize,
                        center.y + iconSize / 2
                    )
                    if (iconRect.contains(offset)) {
                        onDeleteVertex?.invoke(selectedVertex!!)
                        selectedVertex = null
                        showDeleteVertexIcon = false
                    }
                }
            }
        }
    )
}

```

```

        }
    }
}
// Удаление ребра
if(showDeleteIcon && selectedEdge != null) {
    val (sourceId, targetId) = selectedEdge!!
    val from = vertices.find { it.id == sourceId }
    val to = vertices.find { it.id == targetId }
    if (from != null && to != null) {
        val mid = Offset((from.xCoordinate + to.xCoordinate) / 2f, (from.yCoordinate + to.yCoordinate) /
2f)
        val trashX = mid.x + iconSize * 1.2f
        val trashY = mid.y - iconSize / 2
        val trashRect = androidx.compose.ui.geometry.Rect(
            trashX - iconSize * 0.3f,
            trashY - iconSize * 0.3f,
            trashX + iconSize * 1.3f,
            trashY + iconSize * 1.3f
        )
        if (trashRect.contains(offset)) {
            onDeleteEdge?.invoke(sourceId, targetId)
            selectedEdge = null
            showDeleteIcon = false
            return@detectTapGestures
        }
        // Обработка клика по иконке карандаша
        val pencilX = mid.x - iconSize * 2.2f
        val pencilY = mid.y - iconSize / 2
        val pencilRect = androidx.compose.ui.geometry.Rect(
            pencilX - iconSize * 0.3f,
            pencilY - iconSize * 0.3f,
            pencilX + iconSize * 1.3f,
            pencilY + iconSize * 1.3f
        )
        if (pencilRect.contains(offset)) {
            editWeightInput = edges.find { (it.source == sourceId && it.target == targetId) || (it.source ==
targetId && it.target == sourceId) }?.weight?.toString() ?: ""
            showEditWeightDialog = true
            return@detectTapGestures
        }
    }
}
// Добавление вершины
if(onCanvasClick != null) {
    onCanvasClick(offset.x.toInt(), offset.y.toInt())
    selectedEdge = null
    showDeleteIcon = false
    selectedVertex = null
    showDeleteVertexIcon = false
    return@detectTapGestures
}
// Проверка клика по вершине
val clickedVertex = vertices.indexOfLast {
    val dx = it.xCoordinate - offset.x
    val dy = it.yCoordinate - offset.y
    dx * dx + dy * dy <= 24f * 24f
}
val clickedVertexId = if (clickedVertex != -1) vertices[clickedVertex].id else null
if(clickedVertexId != null) {
    if(selectedVertex == clickedVertexId && !showDeleteVertexIcon) {
        showDeleteVertexIcon = true
    } else {

```

```

        selectedVertex = clickedVertexId
        showDeleteVertexIcon = false
    }
    selectedEdge = null
    showDeleteIcon = false
    // Добавление ребра
    selectedVertices =
        if (selectedVertices.size == 2) listOf(clickedVertexId)
        else selectedVertices + clickedVertexId
    if (selectedVertices.size == 2) {
        if (selectedVertices[0] != selectedVertices[1]) {
            showWeightDialog = true
        } else {
            selectedVertices = emptyList()
        }
    } else {
        selectedVertex = null
        showDeleteVertexIcon = false
        // Проверка клика по ребру
        var foundEdge: Pair<Int, Int>? = null
        for (edge in edges) {
            val from = vertices.find { it.id == edge.source }
            val to = vertices.find { it.id == edge.target }
            if (from != null && to != null) {
                val start = Offset(from.xCoordinate.toFloat(), from.yCoordinate.toFloat())
                val end = Offset(to.xCoordinate.toFloat(), to.yCoordinate.toFloat())
                val dist = distancePointToSegment(offset, start, end)
                if (dist < 16f) {
                    foundEdge = Pair(edge.source, edge.target)
                    break
                }
            }
        }
        if (foundEdge != null) {
            if (selectedEdge == foundEdge && !showDeleteIcon) {
                showDeleteIcon = true
            } else {
                selectedEdge = foundEdge
                showDeleteIcon = false
            }
            selectedVertex = null
            showDeleteVertexIcon = false
        } else {
            selectedEdge = null
            showDeleteIcon = false
        }
    }
}
Canvas(modifier = Modifier.fillMaxSize()) {
    // Рисуем рёбра
    edges.forEach { edge ->
        val from = vertices.find { it.id == edge.source }
        val to = vertices.find { it.id == edge.target }
        if (from != null && to != null) {
            val start = Offset(from.xCoordinate.toFloat(), from.yCoordinate.toFloat())
            val end = Offset(to.xCoordinate.toFloat(), to.yCoordinate.toFloat())
            val isSelected = selectedEdge == Pair(edge.source, edge.target) || selectedEdge == Pair(edge.target, edge.source)

```

```

// Выделение рёбер текущего шага
val isCurrentStepEdge = currentStepData?.let { step ->
    val cur = step.currentVertexId
    if (cur == null) false else
        (edge.source == cur && edge.target !in step.visited) ||
        (edge.target == cur && edge.source !in step.visited)
} ?: false
drawLine(
    color = when {
        isCurrentStepEdge -> Color(0xFF1976D2) // синий
        isSelected -> Color.Red
        else -> Color.Black
    },
    start = start,
    end = end,
    strokeWidth = when {
        isCurrentStepEdge -> 6f
        isSelected -> 5f
        else -> 3f
    },
    pathEffect = if (isSelected) PathEffect.dashPathEffect(floatArrayOf(16f, 8f)) else null
)
// Рисуем вес ребра с белым фоном
val mid = Offset((start.x + end.x) / 2, (start.y + end.y) / 2)
val weightText = edge.weight.toString()
val textLayout = textMeasurer.measure(
    weightText,
    style = TextStyle(
        fontSize = 18.sp,
        fontWeight = FontWeight.Bold,
        color = Color.Black,
        textAlign = TextAlign.Center,
        fontFamily = FontFamily.Default
    )
)
val padding = 6.dp.toPx()
val bgWidth = textLayout.size.width + padding * 2
val bgHeight = textLayout.size.height + padding
drawRoundRect(
    color = Color.White,
    topLeft = Offset(
        mid.x - bgWidth / 2f,
        mid.y - bgHeight / 2f
    ),
    size = androidx.compose.ui.geometry.Size(bgWidth, bgHeight),
    cornerRadius = androidx.compose.ui.geometry.CornerRadius(8.dp.toPx(), 8.dp.toPx())
)
drawText(
    textLayout,
    topLeft = Offset(
        mid.x - textLayout.size.width / 2f,
        mid.y - textLayout.size.height / 2f
    )
)
// Если ребро выделено и showDeleteIcon — рисуем иконки
if (isSelected && showDeleteIcon) {
    // Иконка мусорки (справа)
    val iconText = "trash"
    val iconLayout = textMeasurer.measure(
        iconText,
        style = TextStyle(
            fontSize = 28.sp,

```

```

        fontWeight = FontWeight.Bold,
        color = Color.Black,
        textAlign = TextAlign.Center,
        fontFamily = FontFamily.Default
    )
)
val trashX = mid.x + iconSize * 1.2f
val trashY = mid.y - iconSize / 2
drawText(
    iconLayout,
    topLeft = Offset(
        trashX,
        trashY
    )
)
// Иконка карандаша (слева)
val pencilText = "↖"
val pencilLayout = textMeasurer.measure(
    pencilText,
    style = TextStyle(
        fontSize = 28.sp,
        fontWeight = FontWeight.Bold,
        color = Color.Black,
        textAlign = TextAlign.Center,
        fontFamily = FontFamily.Default
    )
)
val pencilX = mid.x - iconSize * 2.2f
val pencilY = mid.y - iconSize / 2
drawText(
    pencilLayout,
    topLeft = Offset(
        pencilX,
        pencilY
    )
)
}
}
}
vertices.forEach { vertex ->
    val isVisited = currentStepData?.visited?.contains(vertex.id) == true
    // Не выделяем синим, если шаг последний (все посещены или currentVertexId уже в visited)
    val isCurrent = currentStepData?.currentVertexId == vertex.id &&
        (currentStepData.visited.contains(vertex.id).not() || currentStepData.visited.size != vertices.size)
    val color = when {
        isCurrent -> Color(0xFF1976D2) // синий
        vertex.color == VertexColor.YELLOW -> Color(0xFFFFD600)
        isVisited -> Color(0xFF43A047)
        else -> Color(0xFFB0B0B0)
    }
    val center = Offset(vertex.xCoordinate.toFloat(), vertex.yCoordinate.toFloat())
    val isSelected = selectedVertex == vertex.id
    drawCircle(
        color = color,
        radius = 24f,
        center = center
    )
    drawCircle(
        color = if (isCurrent) Color(0xFF1976D2) else if (isSelected) Color.Red else Color.Black,
        radius = 24f,
        center = center,
        style = Stroke(width = if (isCurrent) 5f else if (isSelected) 4f else 2f)
    )
}

```

```

        )
        // Имя вершины (A, B, C, ...)
        val name = vertex.name
        val textLayout = textMeasurer.measure(
            name,
            style = TextStyle(
                fontSize = 22.sp,
                fontWeight = FontWeight.Bold,
                color = Color.Black,
                textAlign = TextAlign.Center,
                fontFamily = FontFamily.Default
            )
        )
        drawText(
            textLayout,
            topLeft = Offset(
                center.x - textLayout.size.width / 2f,
                center.y - textLayout.size.height / 2f
            )
        )
    )
    // Если вершина выделена и showDeleteVertexIcon — рисуем иконку мусорки
    if (isSelected && showDeleteVertexIcon) {
        val iconText = "🗑"
        val iconLayout = textMeasurer.measure(
            iconText,
            style = TextStyle(
                fontSize = 28.sp,
                fontWeight = FontWeight.Bold,
                color = Color.Black,
                textAlign = TextAlign.Center,
                fontFamily = FontFamily.Default
            )
        )
        val iconX = center.x + iconSize * 0.7f
        val iconY = center.y - iconSize / 2
        drawText(
            iconLayout,
            topLeft = Offset(
                iconX,
                iconY
            )
        )
    }
}
// Диалог для ввода веса ребра
if (showWeightDialog && selectedVertices.size == 2) {
    AlertDialog(
        onDismissRequest = { showWeightDialog = false; selectedVertices = emptyList() },
        title = { Text("Введите вес ребра") },
        text = {
            OutlinedTextField(
                value = edgeWeightInput,
                onValueChange = { edgeWeightInput = it.filter { c -> c.isDigit() }.take(3) },
                label = { Text("Bec (1-999)") }
            )
        },
        confirmButton = {
            Button(onClick = {
                val weight = edgeWeightInput.toIntOrNull()
                if (weight != null && weight in 1..999 && onAddEdge != null) {
                    onAddEdge(selectedVertices[0], selectedVertices[1], weight)
                }
            })
        }
    )
}

```

```

        showWeightDialog = false
        selectedVertices = emptyList()
        edgeWeightInput = ""
    }
})
} { Text("OK") }
},
dismissButton = {
    Button(onClick = {
        showWeightDialog = false
        selectedVertices = emptyList()
        edgeWeightInput = ""
    }) { Text("Отмена") }
}
)
}
// Диалог для редактирования веса ребра
if(showEditWeightDialog && selectedEdge != null) {
    AlertDialog(
        onDismissRequest = { showEditWeightDialog = false },
        title = { Text("Редактировать вес ребра") },
        text = {
            OutlinedTextField(
                value = editWeightInput,
                onValueChange = { editWeightInput = it.filter { c -> c.isDigit() }.take(3) },
                label = { Text("Bec (1-999)") }
            )
        },
        confirmButton = {
            Button(onClick = {
                val weight = editWeightInput.toIntOrNull()
                val (sourceId, targetId) = selectedEdge!!
                if (weight != null && weight in 1..999 && onAddEdge != null) {
                    // Удаляем старое ребро и добавляем новое с новым весом
                    onDeleteEdge?.invoke(sourceId, targetId)
                    onAddEdge(sourceId, targetId, weight)
                    showEditWeightDialog = false
                }
            }) { Text("OK") }
        },
        dismissButton = {
            Button(onClick = {
                showEditWeightDialog = false
            }) { Text("Отмена") }
        }
    )
}
}

// Вспомогательная функция для расстояния от точки до отрезка
fun distancePointToSegment(p: Offset, a: Offset, b: Offset): Float {
    val ab = b - a
    val ap = p - a
    val abLen = ab.getDistance()
    if (abLen == 0f) return ap.getDistance()
    val t = ((ap.x * ab.x + ap.y * ab.y) / (abLen * abLen)).coerceIn(0f, 1f)
    val proj = Offset(a.x + ab.x * t, a.y + ab.y * t)
    return (p - proj).getDistance()
}

```

MainWindow.kt

package visualisation

```

import androidx.compose.ui.window.Window
import androidx.compose.ui.window.application
import androidx.compose.material.Text
import androidx.compose.runtime.Composable
import androidx.compose.ui.unit.dp
import androidx.compose.foundation.layout.*
import androidx.compose.ui.Modifier
import androidx.compose.ui.graphics.Color
import androidx.compose.ui.unit.sp
import androidx.compose.foundation.background
import androidx.compose.foundation.border
import androidx.compose.foundation.layout.Box
import androidx.compose.foundation.layout.Spacer
import androidx.compose.foundation.layout.fillMaxWidth
import androidx.compose.foundation.layout.height
import androidx.compose.foundation.layout.padding
import androidx.compose.foundation.layout.width
import androidx.compose.foundation.shape.RoundedCornerShape
import controller.AppController
import androidx.compose.runtime.remember
import androidx.compose.runtime.mutableStateOf
import androidx.compose.ui.platform.LocalDensity
import androidx.compose.runtime.getValue
import androidx.compose.runtime.setValue
import androidx.compose.ui.geometry.Size
import androidx.compose.ui.layout.onGloballyPositioned
import javax.swing.JFileChooser
import kotlinx.coroutines.Dispatchers
import kotlinx.coroutines.withContext
import androidx.compose.runtime.LaunchedEffect
import kotlinx.coroutines.delay
import androidx.compose.ui.Alignment
import androidx.compose.material.icons(Icons)
import androidx.compose.material.icons.filled.Close
import androidx.compose.material.icons.filled.Menu
import androidx.compose.foundation.verticalScroll
import androidx.compose.foundation.rememberScrollState

@Composable
fun MainWindowContent(controller: AppController) {
    val addVertexMode = remember { mutableStateOf(false) }
    var canvasSize by remember { mutableStateOf(Size.Zero) }
    val density = LocalDensity.current
    var showFileDialog by remember { mutableStateOf(false) }
    var setStartVertexMode by remember { mutableStateOf(false) }
    var animationTick by remember { mutableStateOf(0L) }
    var speedValue by remember { mutableStateOf(0.5f) } // 0.0..1.0
    val minDelay = 1000L
    val maxDelay = 5000L
    val animationDelay = ((1.0f - speedValue) * (maxDelay - minDelay) + minDelay).toLong()
    val isStartPauseEnabled = controller.graph.vertices.size > 1 && controller.startVertexId != null
    val isSpeedEnabled = controller.isRunning || !controller.isRunning
    var showFullTable by remember { mutableStateOf(false) }
    val isShowTableEnabled = controller.dijkstraResult != null && (controller.isPaused || !controller.isRunning)
    val isResetEnabled = controller.dijkstraResult != null

    fun handleStartPause() {
        when {
            !controller.isRunning -> controller.startDijkstraAnimation()
            controller.isRunning && !controller.isPaused -> controller.pauseDijkstraAnimation()
            controller.isRunning && controller.isPaused -> controller.resumeDijkstraAnimation()
        }
    }
}

```

```

        }

    fun handleToEnd() {
        controller.goToEnd()
    }

    val isToEndEnabled = controller.dijkstraResult?.steps?.isEmpty() == true && controller.currentStep <
(controller.dijkstraResult?.steps?.size ?: 1) - 1

    // --- Анимация шагов алгоритма Дейкстры ---
    androidx.compose.runtime.LaunchedEffect(controller.isRunning, controller.isPaused, controller.currentStep,
animationDelay) {
        while (controller.isRunning && !controller.isPaused) {
            kotlinx.coroutines.delay(animationDelay)
            controller.nextDijkstraStep()
        }
    }

Row(
    modifier = Modifier
        .fillMaxSize()
        .padding(16.dp)
) {
    Column(
        modifier = Modifier
            .weight(1f)
            .fillMaxHeight()
    ) {
        Box(
            modifier = Modifier
                .fillMaxWidth()
                .height(340.dp)
                .onGloballyPositioned { coordinates ->
                    canvasSize = Size(
                        coordinates.size.width.toFloat(),
                        coordinates.size.height.toFloat()
                    )
                }
        )
    }
}

    val paddingPx = with(density) { (24.dp + 2.dp + 6.dp).toPx() }
    GraphCanvas(
        vertices = controller.graph.vertices.map {
            if (it.id == controller.startVertexId) it.copy(color = graph.VertexColor.YELLOW)
            else it.copy(color = graph.VertexColor.GRAY)
        },
        edges = controller.graph.edges,
        currentStepData = controller.dijkstraResult?.steps?.getOrNull(controller.currentStep),
        onCanvasClick = if (!controller.isRunning && addVertexMode.value) { x, y ->
            if (
                x > paddingPx && x < canvasSize.width - paddingPx &&
                y > paddingPx && y < canvasSize.height - paddingPx
            ) {
                controller.addVertex(x, y)
            }
        } else if (!controller.isRunning && setStartVertexMode) { x, y ->
            val clickedVertex = controller.graph.vertices.find {
                val dx = it.xCoordinate - x
                val dy = it.yCoordinate - y
                dx * dx + dy * dy <= 24f * 24f
            }
            if (clickedVertex != null) {

```

```

        controller.setStartVertex(clickedVertex.id)
        setStartVertexMode = false
    }
} else null,
onAddEdge = if (!controller.isRunning) { { sourceId, targetId, weight ->
    controller.addEdge(sourceId, targetId, weight)
} } else null,
onDeleteEdge = if (!controller.isRunning) { { sourceId, targetId ->
    controller.deleteEdge(sourceId, targetId)
} } else null,
onDeleteVertex = if (!controller.isRunning) { { vertexId ->
    controller.deleteVertex(vertexId)
} } else null
)
}
// Легенда цветов
Spacer(modifier = Modifier.height(8.dp))
Row(modifier = Modifier.fillMaxWidth(), horizontalArrangement = Arrangement.spacedBy(18.dp)) {
    LegendCircle(Color(0xFFFFD600), "старт")
    LegendCircle(Color(0xFF1976D2), "текущая")
    LegendCircle(Color(0xFF43A047), "посещено")
    LegendCircle(Color(0xFFB0B0B0), "не посещено")
    Row(verticalAlignment = Alignment.CenterVertically) {

        Spacer(modifier = Modifier.width(8.dp))
        var showInstruction by remember { mutableStateOf(false) }
        androidx.compose.material.Button(onClick = { showInstruction = true }) {
            Text("Инструкция")
        }
        if (showInstruction) {
            InstructionDialog(onClose = { showInstruction = false })
        }
    }
}
Spacer(modifier = Modifier.height(12.dp))
DijkstraTable(controller)
Spacer(modifier = Modifier.height(8.dp))
StepExplanation(controller)
}
Spacer(modifier = Modifier.width(16.dp))
ControlPanel(
    controller,
    onAddVertexClick = { if (!controller.isRunning) addVertexMode.value = !addVertexMode.value },
    isAddVertexModeActive = addVertexMode.value && !controller.isRunning,
    onLoadGraphClick = {
        if (!controller.isRunning && controller.graph.vertices.isEmpty() && controller.graph.edges.isEmpty()) {
            showDialog = true
        }
    },
    onSetStartVertexClick = {
        if (!controller.isRunning) setStartVertexMode = true
    },
    onStartPauseClick = { handleStartPause() },
    isStartPauseEnabled = isStartPauseEnabled,
    speedValue = speedValue,
    onSpeedChange = { speedValue = it },
    isSpeedEnabled = isSpeedEnabled,
    onToEndClick = { handleToEnd() },
    isToEndEnabled = isToEndEnabled,
    onShowTableClick = { showFullTable = !showFullTable },
    isShowTableEnabled = isShowTableEnabled,
    isTableShown = showFullTable,
)

```

```

        onResetAlgorithmClick = { controller.resetDijkstra() },
        isResetEnabled = isResetEnabled
    )
    if(showFileDialog) {
        LaunchedEffect(Unit) {
            val filePath = withContext(Dispatchers.IO) { showFileChooser() }
            showFileDialog = false
            if(filePath != null) {
                controller.loadGraph(filePath)
            }
        }
    }
    if(showFullTable) {
        androidx.compose.ui.window.Window(
            onCloseRequest = { showFullTable = false },
            title = "Таблица алгоритма Дейкстры",
            resizable = true,
            alwaysOnTop = true
        ) {
            Box(Modifier.fillMaxSize().background(Color.White)) {
                FullDijkstraTable(controller)
            }
        }
    }
}

@Composable
fun StateTable() {
    Box(
        modifier = Modifier
            .fillMaxWidth()
            .height(110.dp)
            .background(Color.White, shape = RoundedCornerShape(8.dp))
            .border(1.dp, Color(0xFFCCCCCC), shape = RoundedCornerShape(8.dp))
    )
}

@Composable
fun StepExplanation(controller: AppController) {
    val steps = controller.dijkstraResult?.steps
    val currentStep = controller.currentStep
    val isLast = steps != null && currentStep == steps.lastIndex && steps.isNotEmpty()

    val dijkstraExplanation = when {
        isLast -> "Конец алгоритма"
        else -> steps?.getOrNull(currentStep)?.action
    }

    val explanation = dijkstraExplanation ?: controller.notification ?: ""
    Box(
        modifier = Modifier
            .fillMaxWidth()
            .height(36.dp)
            .background(Color.White, shape = RoundedCornerShape(8.dp))
            .border(1.dp, Color(0xFFCCCCCC), shape = RoundedCornerShape(8.dp)),
            contentAlignment = Alignment.CenterStart
    ) {
        Text(explanation, fontSize = 15.sp, color = Color(0xFF444444), modifier = Modifier.padding(start = 12.dp))
    }
}

```

```

fun showFileChooser(): String? {
    val chooser = JFileChooser()
    val result = chooser.showOpenDialog(null)
    return if (result == JFileChooser.APPROVE_OPTION) chooser.selectedFile.getAbsolutePath else null
}

@Composable
fun LegendCircle(color: Color, label: String) {
    Column(horizontalAlignment = Alignment.CenterHorizontally) {
        Box(
            modifier = Modifier
                .size(22.dp)
                .background(color, shape = androidx.compose.foundation.shape.CircleShape)
                .border(1.dp, Color(0xFF888888), shape = androidx.compose.foundation.shape.CircleShape)
        )
        Spacer(modifier = Modifier.height(2.dp))
        Text(label, fontSize = 12.sp, color = Color.Gray)
    }
}

fun showMainWindow() = application {
    val controller = remember { AppController() }
    Window(
        onCloseRequest = ::exitApplication,
        title = "Dijkstra visualisation",
        resizable = true,
        state = androidx.compose.ui.window.rememberWindowState(width = 900.dp, height = 600.dp)
    ) {
        MainWindowContent(controller)
    }
}

@Composable
fun InstructionDialog(onClose: () -> Unit) {
    val instructionText = remember { mutableStateOf("") }
    LaunchedEffect(Unit) {
        try {
            instructionText.value = java.io.File("README_instruction.txt").readText()
        } catch (e: Exception) {
            instructionText.value = "Не удалось загрузить инструкцию."
        }
    }
    androidx.compose.ui.window.DialogWindow(
        onCloseRequest = onClose,
        title = "Инструкция",
        state = androidx.compose.ui.window.rememberDialogState(width = 700.dp, height = 500.dp)
    ) {
        val scrollState = rememberScrollState()
        Box(Modifier.fillMaxSize().background(Color.White).padding(24.dp)) {
            Column(Modifier.fillMaxSize()) {
                Box(Modifier.weight(1f).verticalScroll(scrollState)) {
                    Text(
                        instructionText.value,
                        fontSize = 15.sp,
                        color = Color(0xFF222222),
                        modifier = Modifier.fillMaxWidth().padding(end = 12.dp)
                    )
                }
                Spacer(Modifier.height(16.dp))
                androidx.compose.material.Button(onClick = onClose, modifier = Modifier.align(Alignment.End)) {
                    Text("Закрыть")
                }
            }
        }
    }
}

```

} } }