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

# ОТЧЕТ

**по учебной практике**

**Тема: Реализация алгоритма Флойда-Уоршелла на языке Kotlin с визуализацией**

Студентка гр. 2381 Малюская Д.И.

Студентка гр. 2381 Биктагирова Д.С.

Руководитель Шестопалов Р.П

Санкт-Петербург
2024

**ЗАДАНИЕ**

**НА УЧЕБНУЮ ПРАКТИКУ**

Студентка Малюская Д.И.

Студентка Биктагирова Д.И.

Группа 2381

Тема практики: Реализация алгоритма Флойда-Уоршелла на языке Kotlin с визуализацией

Задание на практику: Разработка визуализатора алгоритма Флойда-Уоршелла на языке программирования Kotlin с графическим интерфейсом.

Сроки прохождения практики: 26.06.2024 – 09.07.2024

Дата сдачи отчета: 09.07.2024

Дата защиты отчета:  09.07.2024

| | | |
|---|---|---|
| Студентка гр. 2381 | | Малюская Д.И. |
| Студентка гр. 2381 | | Биктагирова Д.С. |
| Руководитель | | Шестопалов Р.П |

## АННОТАЦИЯ

Цель практики заключается в изучении и применении языка программирования Kotlin для разработки визуализатора алгоритма Флойда с графическим интерфейсом. Алгоритм Флойда-Уоршелла используется для нахождения кратчайших путей во всех парах вершин взвешенного ориентированного графа. Он позволяет найти кратчайшие расстояния между всеми парами вершин в графе. Визуализация алгоритма выполнена средствами библиотеки JavaFX.



## SUMMARY

The goal of the practice is to study and apply the Kotlin programming language for developing a visualizer for the Floyd algorithm with a graphical user interface. The Floyd-Warshall algorithm is used to find the shortest paths between all pairs of vertices in a weighted directed graph. It allows finding the shortest distances between all pairs of vertices in the graph. The visualization of the algorithm is implemented using the JavaFX library.

# СОДЕРЖАНИЕ

# ВВЕДЕНИЕ

Цель данной практики заключается в разработке программы, которая представит визуализацию алгоритма поиска кратчайших путей в графе с использованием алгоритма Флойда-Уоршелла. Интерфейс программы будет включать в себя отображение графа, выделение кратчайших путей между вершинами и изменение матрицы весов на графе. Пользователь сможет создавать граф, модифицировать его через матрицу весов. Кроме того, пользователь сможет просматривать работу алгоритма по этапам или сразу видеть окончательный результат его работы. Разработка данной программы поможет понять принципы работы алгоритма Флойда-Уоршелла и его применение на практике через визуализацию шагов алгоритма на графе.

# 1. СПЕЦИФИКАЦИЯ

## 1.1. Формальная постановка задачи

**Входные данные:**

1. Граф G = (V, E), где V - множество вершин, E - множество ребер.
2. Матрица весов W размером |V| × |V|, где W[i][j] представляет вес ребра между вершинами i и j. Если ребра между вершинами нет, то вес равен бесконечности.

**Задача:**

Разработать программу для визуализации алгоритма поиска кратчайших путей в графе с использованием алгоритма Флойда-Уоршелла.

**Выходные данные:**

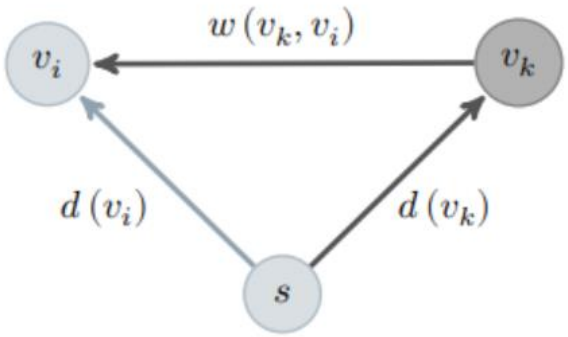
1. Визуализация графа с выделенными кратчайшими путями между вершинами.
2. Интерактивная матрица весов, отображающая изменения на каждом шаге алгоритма Флойда-Уоршелла.
3. Возможность пользователю создавать граф, вводить и модифицировать его через матрицу весов.
4. Возможность просмотра работы алгоритма по шагам или непосредственно окончательного результата его работы.

### 1.2. Описание алгоритма

Алгоритм начинает работу с матрицы весов (матрицы пометок □(□, □)), рассматриваемых как длины кратчайших путей от вершины □_□ к вершине □_□ , не имеющих ни одной промежуточной вершины, т. е. вес ребра (дуги) (v_i , □_□ ). Далее поочередно каждую вершину представляют в роли промежуточной и через нее осуществляют модификацию пометки, т. е. на каждом шаге одну из вершин добавляют к множеству промежуточных вершин. Обозначим пометку вершины A как вещественное число □(□), которое по окончании работы алгоритма означает кратчайшее расстояние, тогда новое значение пометки, согласно алгоритму, будет рассчитываться как

□(□) : = min{□(□), □(□) + □(□, □)}

Данное выражение называется правилом модификации пометок , которое можно проиллюстрировать следующим рисунком:




Условие применимости - в графе нет циклов отрицательной величины. В случае, когда такой цикл есть, алгоритм позволяет найти хотя бы один такой цикл.

Алгоритм работает за O(n^3) времени и использует O(n^2) памяти.

Для того, чтобы алгоритм возвращал не только длину кратчайшего пути но и сам путь (дерево кратчайших путей), введем дополнительную матрицу □, в которой для каждой пары вершин □ и □ будем хранить номер первой промежуточной вершины кратчайшего пути из □ в □.

Опишем работу алгоритма в виде псевдокода:

```
// Инициализация
for □ ← 1, □ do
    for □ ← 1, □ do
        □(□, □) ← □(□, □) // Матрицу пометок заполняем весами ребер
        □(□, □) ← □
        // □_□ — первая промежуточная вершина пути □_□ → □_□
    end for
end for
// Основной алгоритм
// Внешний цикл идет по промежуточным вершинам □, внутри него для каждой
// пары вершин проверяется возможность улучшения их пометки через □
for □ ← 1, □ do
    for □ ← 1, □ do
        for □ ← 1, □ do
            if □(□, □) + □(□, □) < □(□, □) then
                □(□, □) ← □(□, □) + □(□, □)
                □(□, □) ← □(□, □)
            end if
        end for
    end for
end for // Нахождение кратчайших путей для каждой пары вершин □_□ , □_□
// □(□, □) — список вершин кратчайшего пути □_□ → □_□
for □ ← 1, □ do
    for □ ← 1, □ do
        □(□, □) ← ∅
        if □(□, □) ≠ ∞ then
            □ = □ // □□ — первая вершина пути □(□, □)
            do
                □(□, □) ← □(□, □) U {□}
                □ ← □(□, □)
            while □ ≠ □(□, □)
// Если промежуточная вершина подпути □□ → □□ совпала с его
// началом, то других промежуточных вершин на этом подпути нет
                □(□, □) ← □(□, □) U {□}
            // Добавляем в список □(□, □) последнюю вершину пути □□ → □□
        end if
    end for
end for
```

### 1.3. Интерфейс программы

- Отображение графа: Программа будет позволять пользователю визуально создавать и редактировать граф, добавляя вершины и рёбра с редактирования матрицы весов.

Предполагается, что матрица весов и граф будут выглядеть следующим

$$\begin{pmatrix} \times & 1 & 6 & \infty \\ \infty & \times & 4 & 1 \\ \infty & \infty & \times & \infty \\ \infty & \infty & 1 & \times \end{pmatrix}$$

образом:

Рисунок 1 – Матрица весов

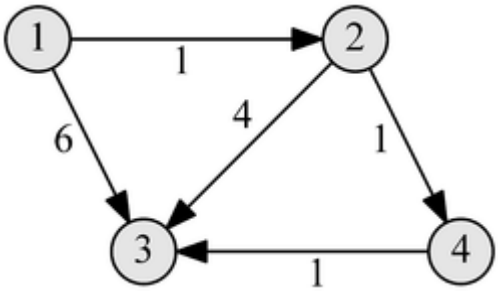



Рисунок 2 – Визуальное представление графа

Примерная схема интерфейса:

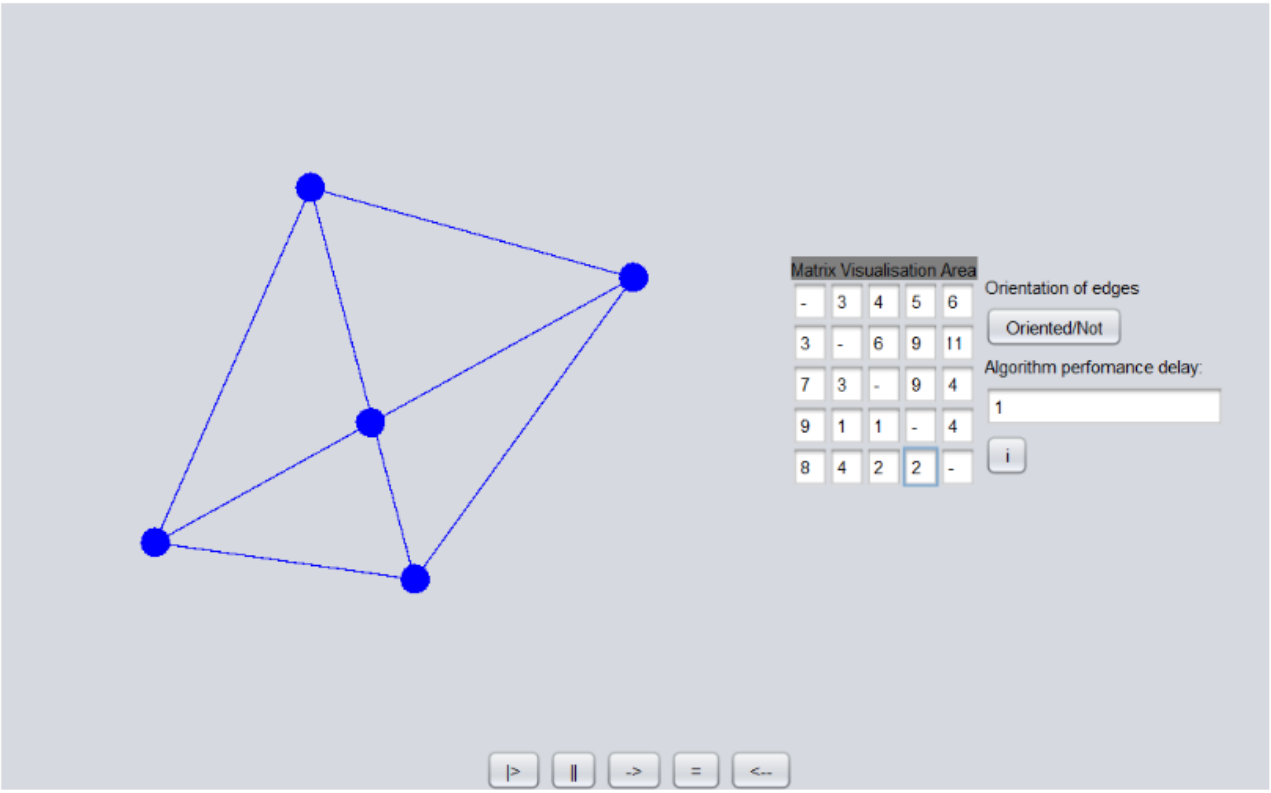


Рисунок 3 – Схема программы

- Анимация алгоритма: Пользователь сможет запустить алгоритм Флойда-Уоршелла на созданном графе и наблюдать шаг за шагом, как алгоритм находит кратчайшие пути между всеми парами вершин. Визуализация будет показывать изменения в матрице расстояний на каждом шаге алгоритма.
- Интерактивность: Пользователь сможет взаимодействовать с визуализацией, приостанавливая, возобновляя или шагая по алгоритму по своему усмотрению. Также будет добавлена возможность выбора режима - показывать работу алгоритма по шагам или готовое представление дерева кратчайших путей.

## 2. ПЛАН РАЗРАБОТКИ И РАСПРЕДЕЛЕНИЕ РОЛЕЙ В КОМАНДЕ

### 2.1. План разработки

1. 1 июля (понедельник): сдача прототипа проекта.
    - Разработка базового функционала программы, включая создание графического интерфейса для добавления вершин и рёбер.
    - Реализация основной логики алгоритма Флойда-Уоршелла для нахождения кратчайших путей между всеми парами вершин.
2. 5 июля (пятница): сдача первой версии.
    - Доработка интерфейса программы.
    - Полная реализация алгоритма в соответствии с описанием в предыдущем пункте.
3. 8 июля (понедельник): сдача второй версии.
    - Оптимизация кода и улучшение производительности программы.
    - Добавление функционала для визуализации шагов алгоритма Флойда-Уоршелла.
    - Подготовка к финальной сдаче, проверка соответствия всем требованиям проекта.
4. 9 июля (вторник): сдача финальной версии программы с отчётом.

- Финальная проверка программы на наличие ошибок и исправление последних недочетов.
- Подготовка отчёта о проделанной работе, включая описание функционала программы, инструкции по использованию и примеры работы.
- Сдача финальной версии программы и отчёта.

### 2.2. Распределение ролей в команде

1. Дарья Биктагирова - отвечает за разработку основной логики программы, включая алгоритмы и функциональность, а также за написание отчета по фрагментам своего кода.
2. Дарья Малюская - отвечает за дизайн и внешний вид программы, а также за написание основной части отчета, описывающей общую концепцию проекта и его результаты.

## 3. ОСОБЕННОСТИ РЕАЛИЗАЦИИ

### 3.1. Архитектура программы

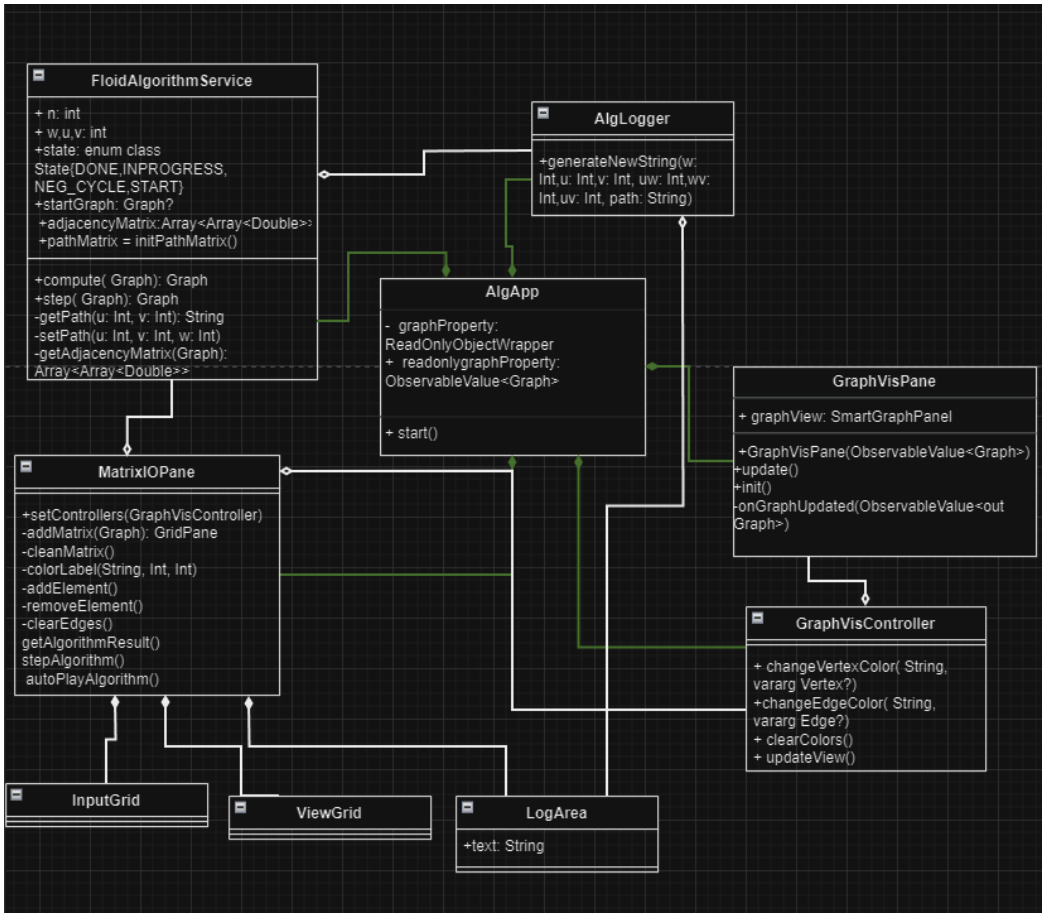



Схема архитектуры приложения представлена в виде UML на рисунке 4:

Рисунок 4 – UML-диаграмма приложения

### 3.2. Использованные структуры данных

Класс *GraphVisPane* используется для визуализации графов и имеет следующие поля:

1. *graphObservable: ObservableValue*. Объект, который хранит информацию о графе и обновляется при изменениях в графе.
2. *diGraph: DigraphEdgeList.* Объект, который представляет собой ориентированный граф с использованием списка ребер. Он хранит вершины и связи между ними.
3. *graphView: SmartGraphPanel*. Объект, который представляет панель для визуализации графов. Он использует стратегии размещения вершин, такие как *SmartCircularSortedPlacementStrategy* и *ForceDirectedSpringGravityLayoutStrategy*, для отображения графа.

4. *vertexCache: mutableMap*. Объект, который хранит кешированные объекты вершин графа для быстрого доступа к ним при обновлении графа.
5. *edgeCache: mutableMap*. Объект, который хранит кешированные объекты ребер графа для быстрого доступа к ним при обновлении графа.

И следующие методы:

1. *onGraphUpdated:* Это метод, который обрабатывает изменения в графе при обновлении *graphObservable.*
2. *init():* Этот метод инициализирует *graphView* и вызывает *onGraphUpdated* для обновления представления графа.

Класс *MatrixIOPane* используется для отображения матрицы смежности графа и управления элементами графа. Его поля:

1. *dim:* Переменная типа *SimpleIntegerProperty*, используется для хранения размерности матрицы.
2. *vbox:* Объект типа *VBox*, представляет вертикальный контейнер для размещения элементов.
3. *grid:* Объект типа *GridPane*, используется для представления сетки вершин и ребер графа.
4. *matrixGrid:* Объект типа *GridPane*, представляет графическое представление матрицы смежности графа.
5. *vertexCache: mutableMap*, хранит вершины графа для быстрого доступа к ним.
6. *graphProperty: ReadOnlyObjectWrapper*, хранит информацию о текущем состоянии графа.
7. *step: Int* - переменная, используется для отслеживания текущего шага алгоритма.

Этот класс имеет следующие методы:

1. *addMatrix(graph: Graph): GridPane* - Данный метод создает и добавляет матрицу смежности графа на экран. Он принимает объект Graph, извлекает из него информацию о вершинах и ребрах, а затем визуализирует матрицу смежности на основе этой информации.
2. *cleanMatrix()* - Метод для очистки матрицы смежности с экрана. Удаляет предыдущее отображение матрицы.
3. *addTextField(i: Int, j: Int)* - Метод, который добавляет текстовые поля для ввода веса ребра между вершинами i и j. При вводе значения и нажатии клавиши "Enter" создается новое ребро между вершинами и обновляется граф.
4. *addElement()* - Метод для добавления новой вершины в граф. Создает новую вершину, обновляет связи с существующими вершинами и визуализирует изменения на экране.
5. *removeElement()* - Метод для удаления последней добавленной вершины из графа. Удаляет вершину и все связанные с ней ребра, затем обновляет граф и матрицу смежности.
6. *startAlgorithm()* - Метод для запуска алгоритма. Очищает матрицу смежности, запускает вычисления по алгоритму, визуализирует результат и обновляет граф.
7. *stepAlgorithm()* - Метод для пошагового выполнения алгоритма. Поочередно проходит шаги алгоритма, обновляя матрицу смежности и граф.
8. *automaticAlgorithmMode()* - Метод для автоматического выполнения алгоритма. Запускает вычисления с определенной периодичностью, с заданной задержкой между шагами.

Класс *FloidAlg* представляет собой класс, содержащий реализацию алгоритма и операций для работы с графом. Он использует матрицу весов ребер для расчетов и обновления кратчайших путей с учетом текущего шага или полного выполнения алгоритма на графе.

1. *step(graph: Graph, step: Int): Graph* - Метод для выполнения шага алгоритма с заданным номером. Принимает объект *Graph* и номер шага. Внутри этого метода осуществляется выполнение алгоритма Флойда, постепенно обновляя матрицу расстояний между вершинами на каждом шаге. Возвращает обновленный граф после выполнения указанного шага.
2. *getAdjacencyMatrix(graph: Graph): Array* - Метод для получения матрицы смежности из графа. Принимает объект *Graph* и возвращает двумерный массив значений типа *Double,* представляющий матрицу смежности графа с учетом весов ребер.
3. *compute(graph: Graph): Graph* - Метод для полного выполнения алгоритма на всем графе. Выполняет алгоритм Флойда для нахождения кратчайших путей между всеми парами вершин в графе. Обновляет матрицу расстояний и возвращает обновленный граф.

*class AlgoApp : Application()* представляет собой графическое приложение, использующее JavaFX для визуализации графа и выполнения алгоритма. Создержит методы:

1. *start(primaryStage: Stage)* - Переопределенный метод старта приложения, где создается интерфейс приложения, настраиваются компоненты и устанавливается основное окно.

Приложение состоит из следующих основных компонентов:

1. *MatrixIOPane* - это пользовательский интерфейс для работы с матрицей смежности графа. Создается экземпляр класса *MatrixIOPane*, который предоставляет функциональность для добавления и удаления вершин, управления ребрами графа и выполнения алгоритма.
2. *GraphVisPane* - это компонент для визуализации графа. Создается экземпляр класса *GraphVisPane* и используется для отображения изменений в графе.

3. *SplitPane* - используется для разделения главного окна на две части: визуализацию графа и пользовательский интерфейс для работы с матрицей смежности.
4. *Scene* - создается новый экземпляр Scene, в который добавляется SplitPane и устанавливаются стили.
5. *PrimaryStage* - основное окно приложения, которое отображает сцену.

Классы *Edge, Graph* и *Vertex* представляют основные структуры данных, используемые для представления графа и его компонентов.

*data class Edge* - класс, представляющий ребро графа. Свойства:

1. *from* - вершина, из которой исходит ребро
2. *to* - вершина, в которую входит ребро
3. *cost* - стоимость ребра (вес)

Методы:

1. *toString()* - переопределенный метод, возвращающий строковое представление веса ребра.

*data class Graph:* - класс, представляющий граф. Свойства:

1. *vertices* - множество вершин графа
2. *edges* - множество ребер графа

*data class Vertex*: - класс, представляющий вершину графа. Свойства:

1. *name* - название вершины

Методы:

1. *toString()* - переопределенный метод, возвращающий строковое представление названия вершины.

# 4. ТЕСТИРОВАНИЕ

Окно приложения при запуске выглядит следующим образом:

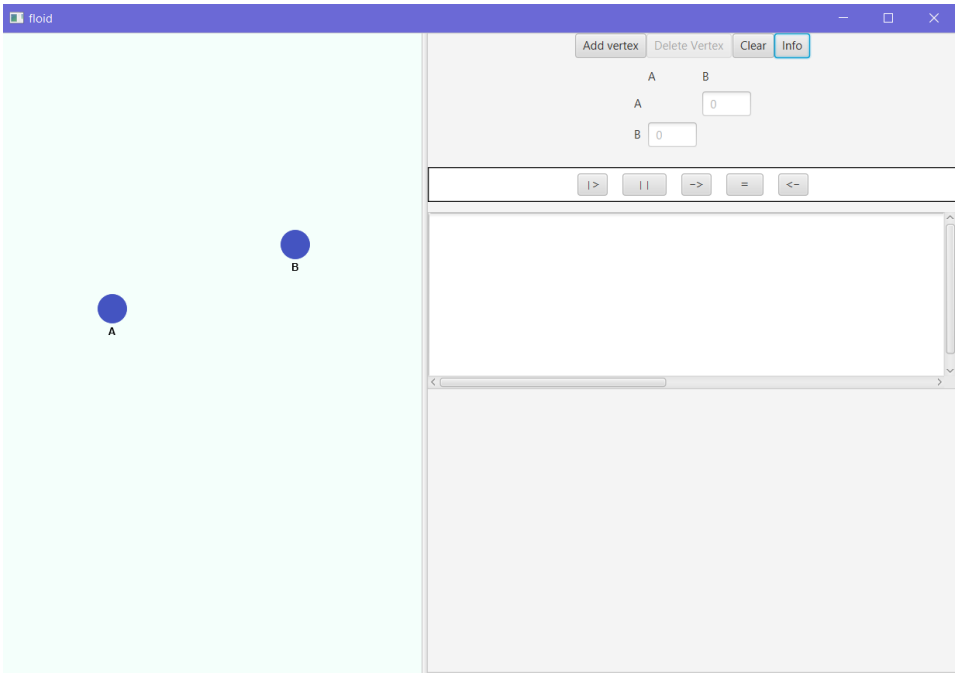



Рисунок 5 – Стартовое окно

Удаление вершин A и B не допускается, чтобы не возникало ошибок при запуске алгоритма.

Протестируем насколько корректно работают заявленные функции.

1) Добавление вершин по кнопке также добавляет ячейки для ввода веса возможных ребер. При большом количестве вершин может понадобиться открыть окно в полный экран.

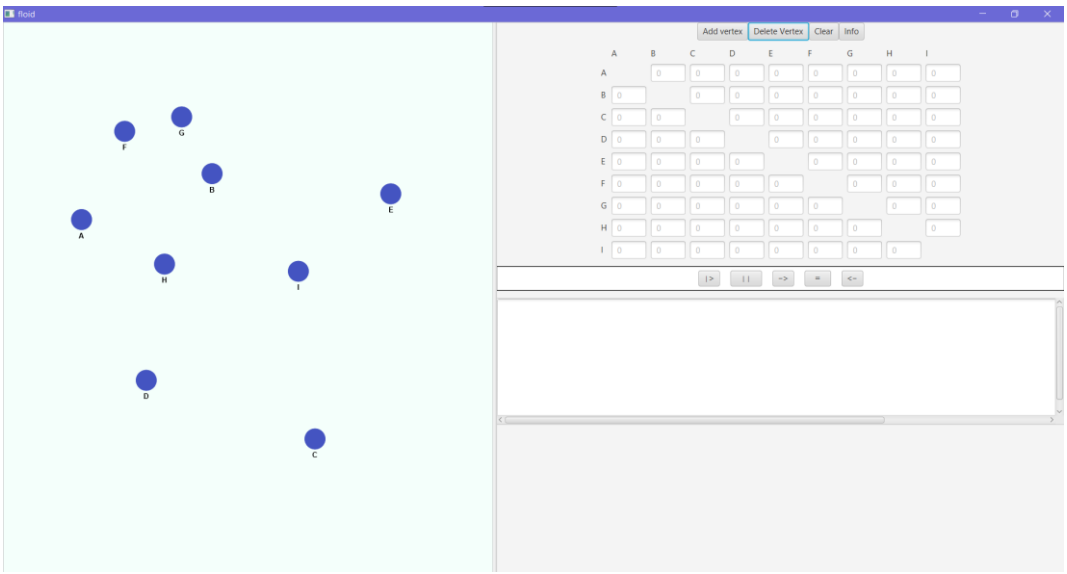



Рисунок 7 – Добавление вершин

Добавление ребер происходит корректно - не допускается дублирующих, при повторном введении веса в одну ячейку старое ребро перезапишется.

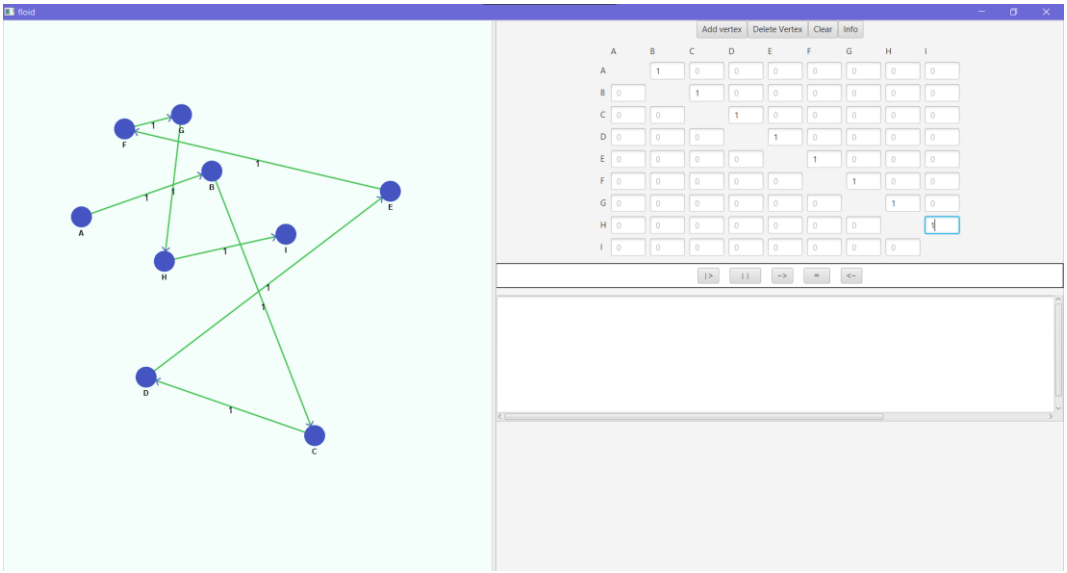



Рисунок 6 – Добавление ребер

2) Кнопка получения моментального результата работает правильно - у итогового графа зеленым цветом показаны ребра, измененные в ходе работы алгоритма.

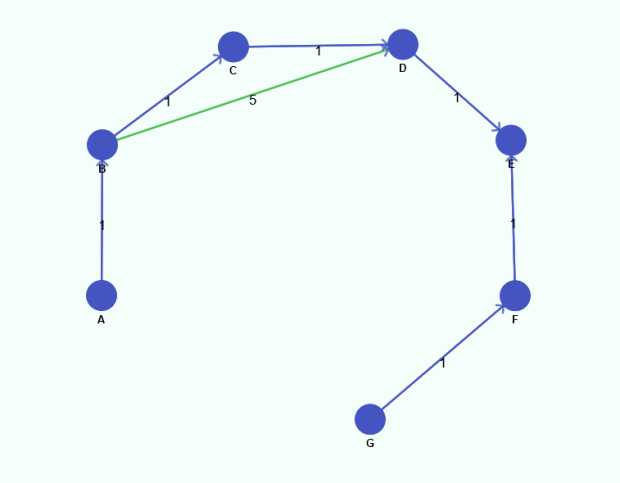



Рисунок 8 – Исходный граф(1)

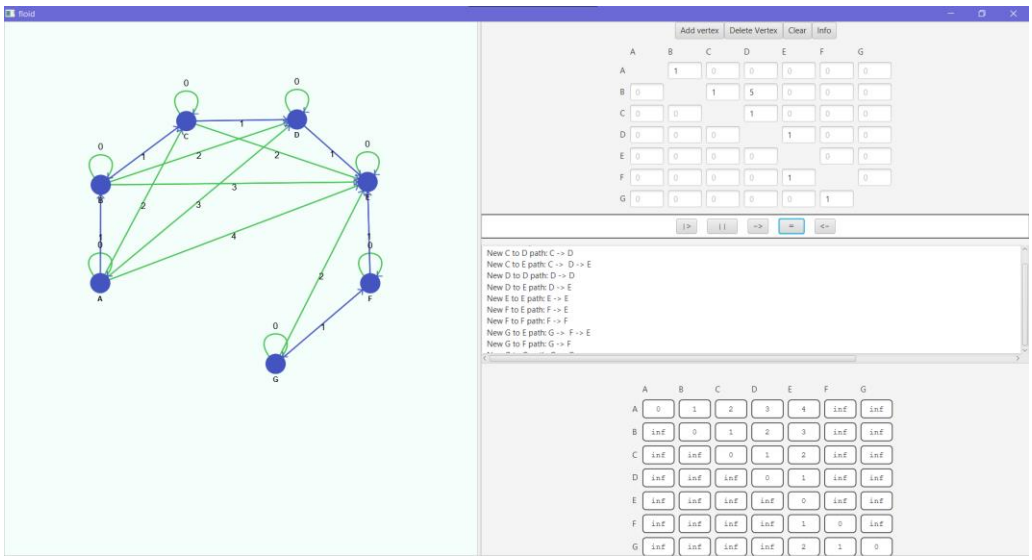


Рисунок 9 – Итоговый граф(1)

В текстовом поле можно пролистать описание выполненных шагов и в конце работы корректно выводятся все минимальные пути между вершинами.

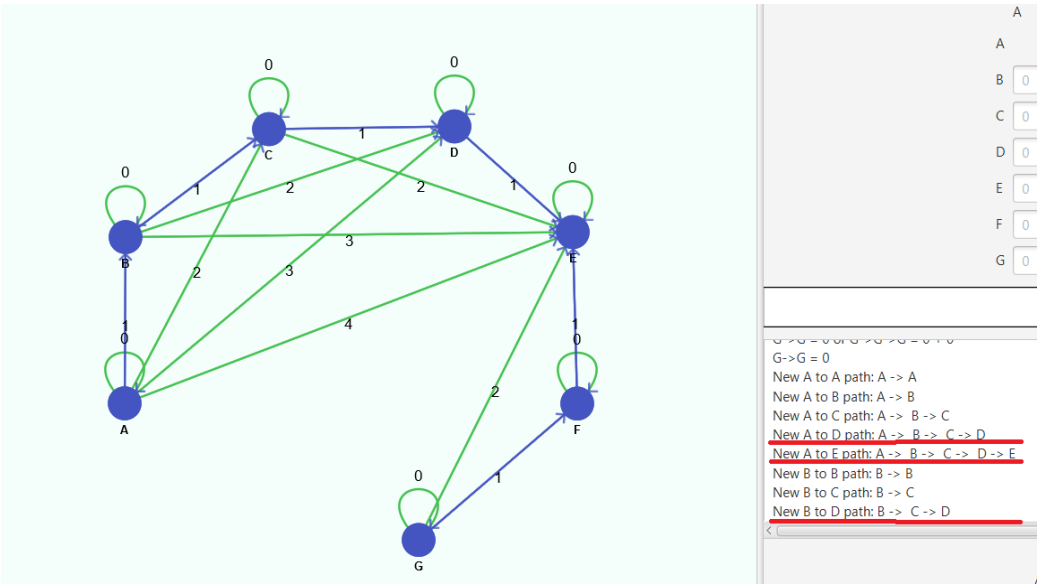



Рисунок 10 – Вывод путей

3)В ходе выполнения алгоритма пошагово изменяются цвет ребер и вершин, а также выделяются ячейки в матрице. Это просиходит по следующим правилам:

- цвет вершины розовый, если она является промежуточной;
- цвет вершины красный, если она является начальной или конечной;
- цвет ребер розовый, если они входят в путь через промежуточную вершину;

- цвет ребра красный, если на этом шаге оно было оставлено как мин. путь между нач. и кон. вершинами;
- цвет ребра зеленый, если на этом шаге мин. путь между нач. и кон. вершинами был перезаписан(т.е. путь через промеж. верш. меньше исходного);

На рисунках ниже конкретные примеры:

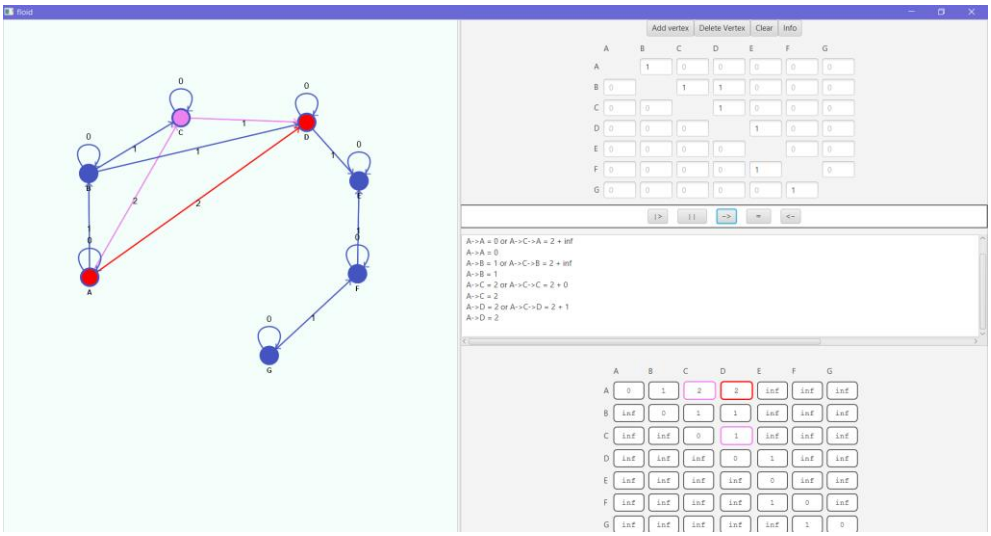


Рисунок 10 – AD < AC+CD

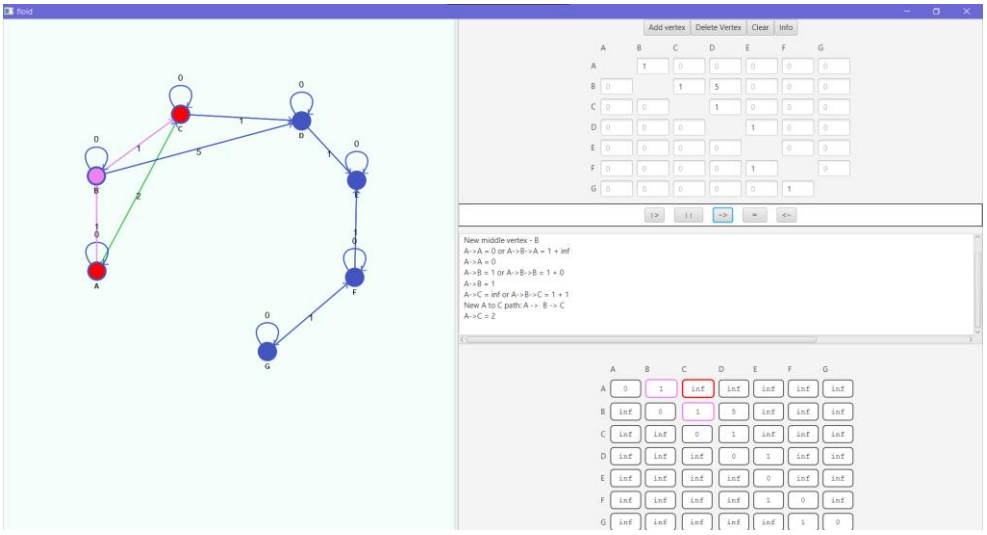


Рисунок 11 – Ребро AC добавлено - путь A->B->C

4)Если в графе есть отрицательный цикл - он будет найдет и выведен в панель с логом. После этого объект алгоритма получит state = State.NEG_CYCLE и дальнейшее выполнение будет прекращено (нажатие кнопок не будет давать никакого эффекта).

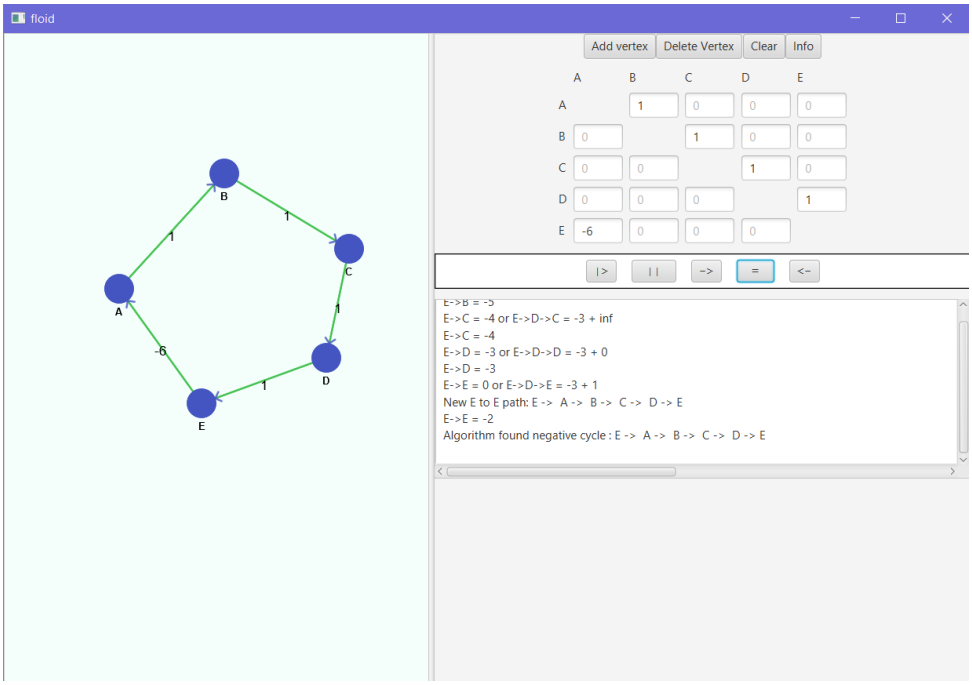



Рисунок 12 – Результат на графе с отр. циклом

На нуль-графе алгоритм просто добавляет петли.

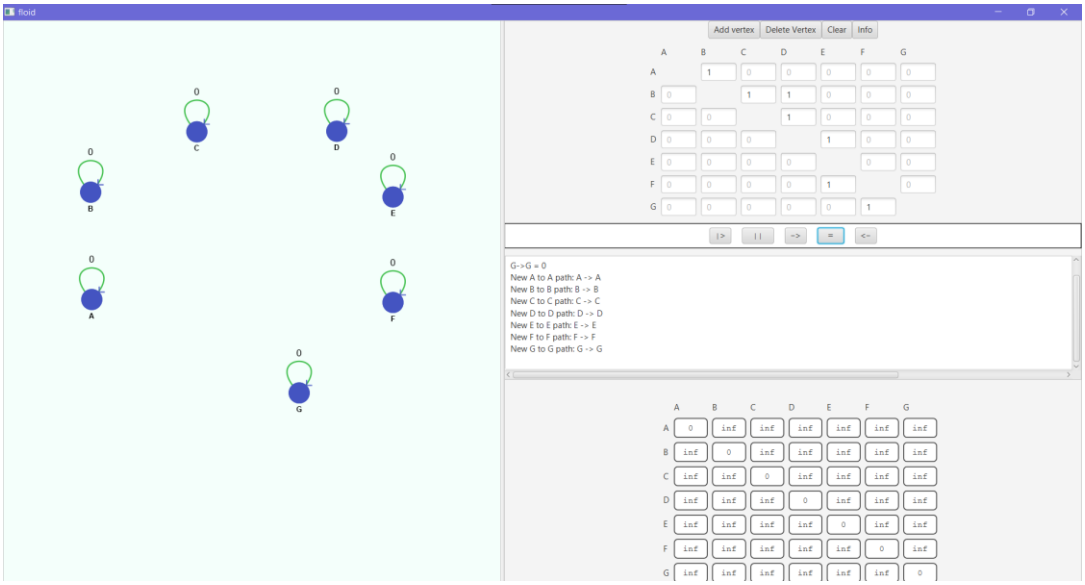



Рисунок 13 – Результат на нуль-графе

5) Протестируем алгоритм на произвольном графе без отр. циклов. Для получения результата используем кнопку автоматического выполнения.

К сожалению, паузу реализовать не получилось. По факту в данном случае функция шага вызывается $n^3$ раз (с задержкой в 1 секунду), без возможности внешнего прерывания. Все изменения цветов и матрицы на каждом шаге видны.

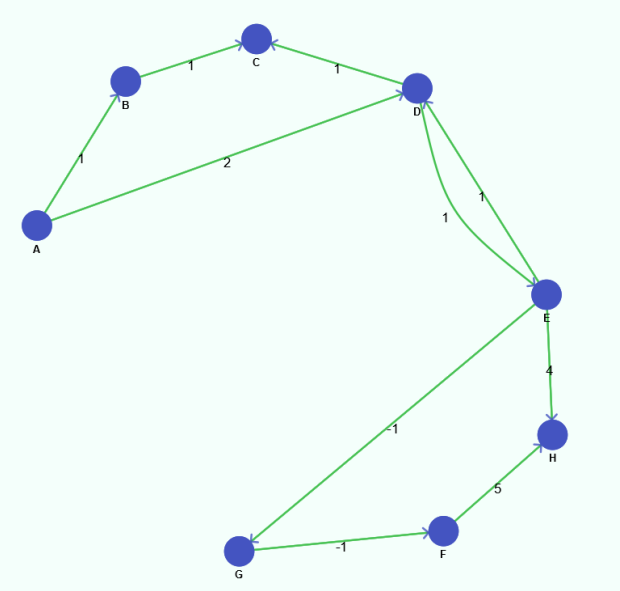



Рисунок 14 – Исходный граф(2)

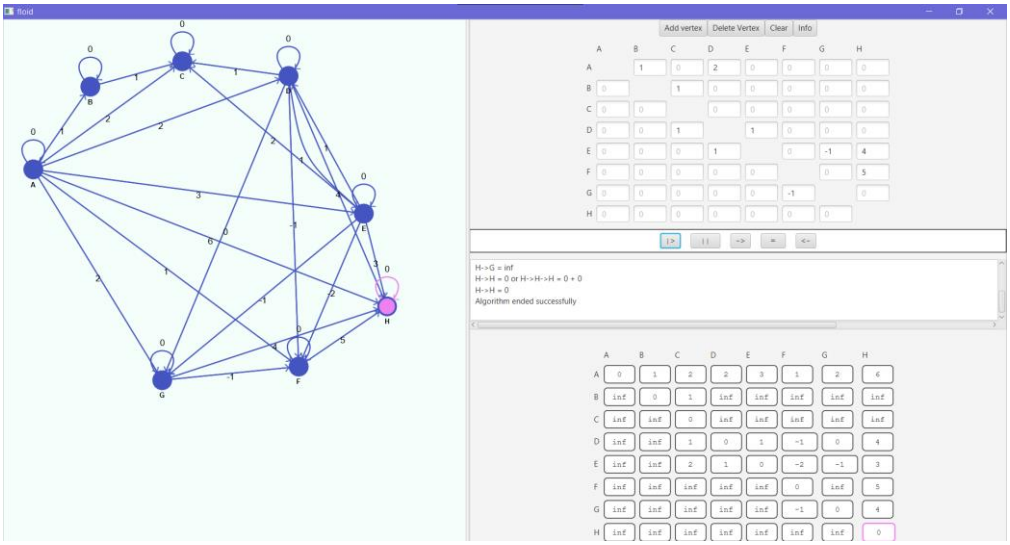



Рисунок 14 – Итоговый граф(2)

## ЗАКЛЮЧЕНИЕ

В ходе данной практики было осуществлено изучение и применение языка программирования Kotlin для создания визуализатора алгоритма Флойда-Уоршелла с графическим интерфейсом. Алгоритм Флойда-Уоршелла используется для нахождения кратчайших путей во всех парах вершин взвешенного ориентированного графа, и его визуализация была реализована с использованием библиотеки JavaFX.

В результате проведенной работы был разработан визуализатор, который позволяет наглядно отображать взаимосвязи между вершинами графа и демонстрировать процесс нахождения кратчайших путей в графе посредством алгоритма Флойда-Уоршелла. Полученный опыт разработки с использованием Kotlin и интеграции с библиотекой JavaFX позволяет дополнительно углубить понимание работы с графическими интерфейсами и алгоритмами нахождения кратчайших путей в графах.

Проделанная работа позволяет эффективно применять полученные знания в области разработки программного обеспечения, а также улучшить навыки работы с Kotlin и визуализацией алгоритмов для решения задач оптимизации и анализа графов.

## ПРИЛОЖЕНИЕ А
## НАЗВАНИЕ ПРИЛОЖЕНИЯ