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.

Студент гр. 7304 _____ Абдульманов Э.М

Студентка гр. 7304 _____ Каляева А.В

Студент гр. 7304 _____ Комаров А.О

Руководитель _____ Жангиров Т.Р.

Санкт-Петербург

2019

ЗАДАНИЕ

НА УЧЕБНУЮ ПРАКТИКУ

Студент Абдульманов Э.М. группы 7304
Студентка Каляева А.В. группы 7304
Студент Комаров А.О. группы 7304

Тема практики: Кратчайшие пути в графе. Алгоритм Дейкстры.

Задание на практику:

Командная итеративная разработка визуализатора алгоритма на Java с графическим интерфейсом.

Алгоритм: Дейкстры.

Сроки прохождения практики: 01.07.2019 – 14.07.2019

Дата сдачи отчета: 12.07.2019

Дата защиты отчета: 12.07.2019

Студент	_____	Абдульманов Э.М
Студентка	_____	Каляева А.В
Студент	_____	Комаров А.О
Руководитель	_____	Жангиров Т.Р.

АННОТАЦИЯ

В ходе выполнения задания учебной практики была реализована программа, предназначенная для нахождения кратчайшего пути в графе. Поиск кратчайшего пути осуществляется с использованием алгоритма Дейкстры. Разработанная программа детально показывает этапы работы алгоритма при построении кратчайшего пути. Программа была написана на языке программирования Java, в среде разработки IntelliJ Idea.

SUMMARY

During the educational practice task, a program was implemented to find the shortest path in the graph. The search for the shortest path is carried out using the Dijkstra algorithm. The developed program shows in detail the stages of the algorithm when building the shortest path. The program was written in the Java programming language in the IntelliJ Idea development environment.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

В ходе выполнения задания учебной практики требуется разработать программу для поиска кратчайшего пути в графе с помощью алгоритма Дейкстры.

Алгоритм Дейкстры - алгоритм на графах, изобретённый нидерландским учёным Эдсгером Дейкстрой в 1959 году.[1] Данный алгоритм находит кратчайшие пути от одной из вершин графа до всех остальных вершин. Алгоритм работает только для графов без рёбер отрицательного веса. Алгоритм широко применяется в программировании и технологиях, например, при планировании автомобильных и авиамаршрутов, при разводке электронных плат, в протоколах маршрутизации.

1. ТРЕБОВАНИЯ К ПРОГРАММЕ

1.1. Исходные Требования к программе

1.1.1. Требования к вводу исходных данных

На вход алгоритму должен подаваться взвешенный ориентированный граф. Именем вершины могут быть буквы (как строчные, так и прописные), а так же строки и цифры.

1.1.2. Требования к визуализации

Пользовательский интерфейс должен представлять собой диалоговое окно, содержащее набор кнопок, предназначенных для управления состоянием программы.

Диалоговое окно должно состоять из:

- Рабочей области для построения графа.
- Кнопки «Сохранить», которая должна позволять пользователю сохранить созданный в рабочей области граф в виде .txt файла.
- Кнопки «Загрузить», которая должна позволять пользователю загрузить граф, для которого необходимо применить алгоритм Дейкстры, в виде .txt файла.
- Кнопки «Перемещение», которая должна позволять перемещать вершины созданного графа внутри рабочей области.
- Кнопки «Добавить вершину», которая должна создать вершину графа в рабочей области после клика мышью. Вновь созданной вершине пользователь должен дать имя, допустимое для данной программы.
- Кнопки «Соединить вершины», которая должна создать направленное ребро между двумя вершинами и задать его вес.
- Кнопки «Удалить», которая должна удалить выбранный пользователем элемент графа.

- Кнопки «Вперед», которая должна отобразить следующую итерацию алгоритма.
- Кнопки «Назад», которая должна отобразить предыдущую итерацию алгоритма.

1.2. Шаблоны архитектуры

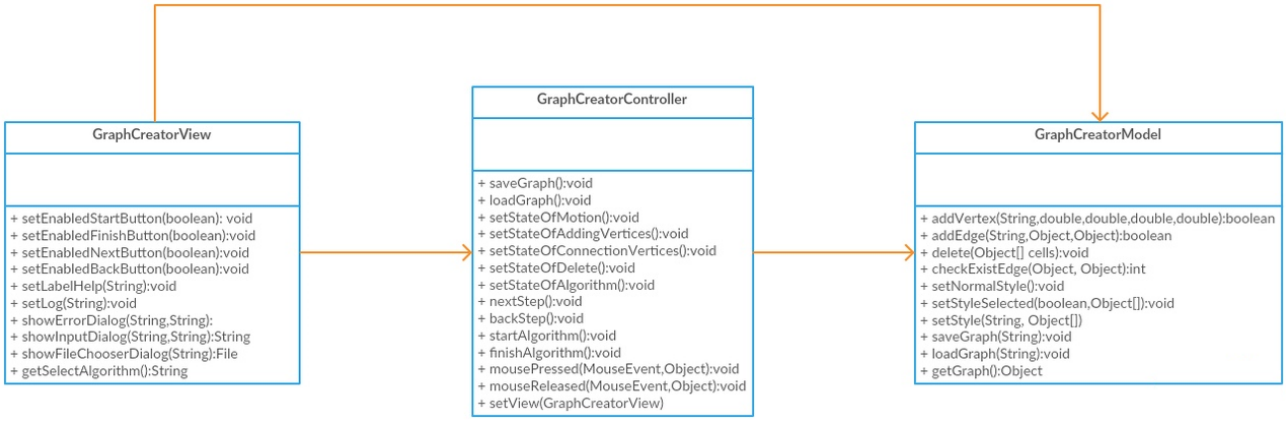

Рисунок 1- UML диаграмма MVC данного приложения

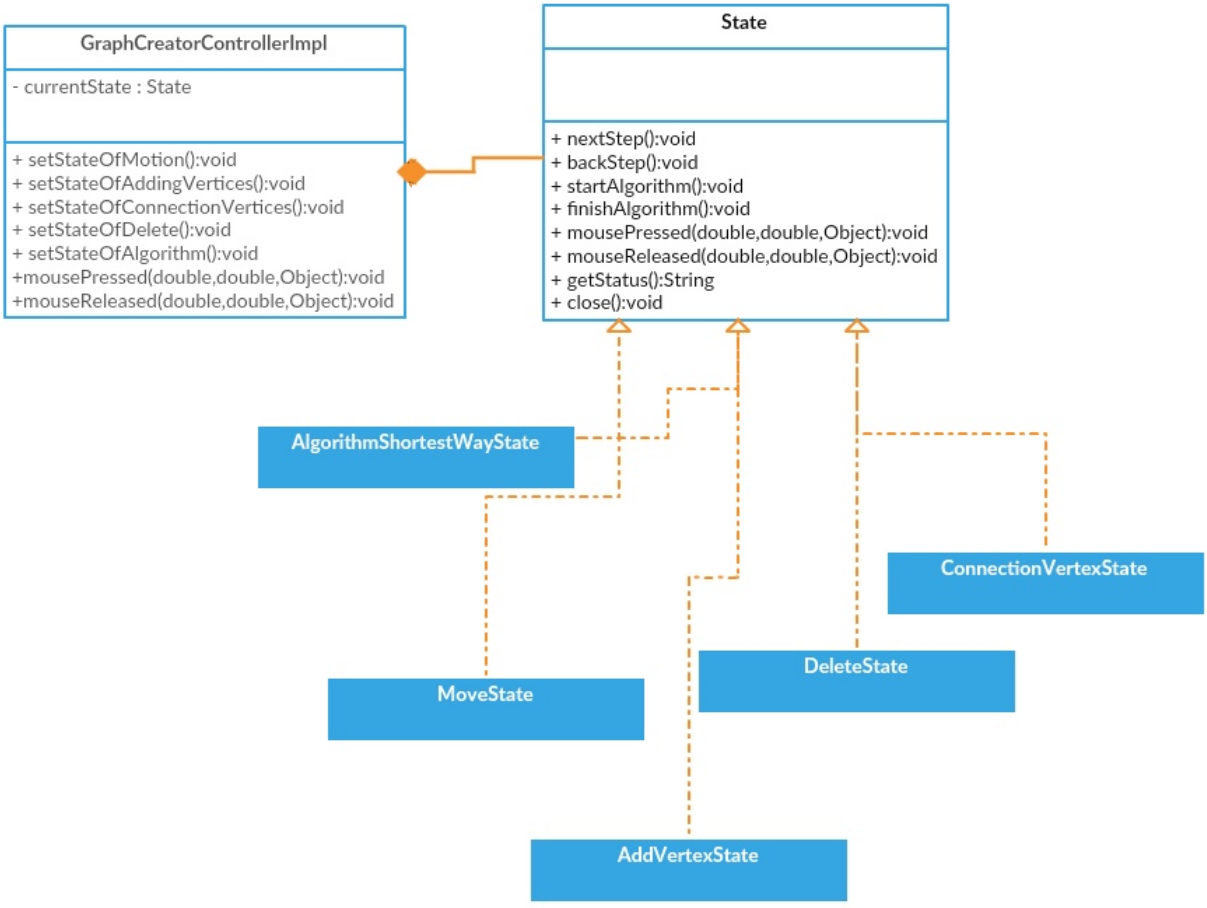

Рисунок 2 - UML диаграмма работы с графом с использованием шаблона State

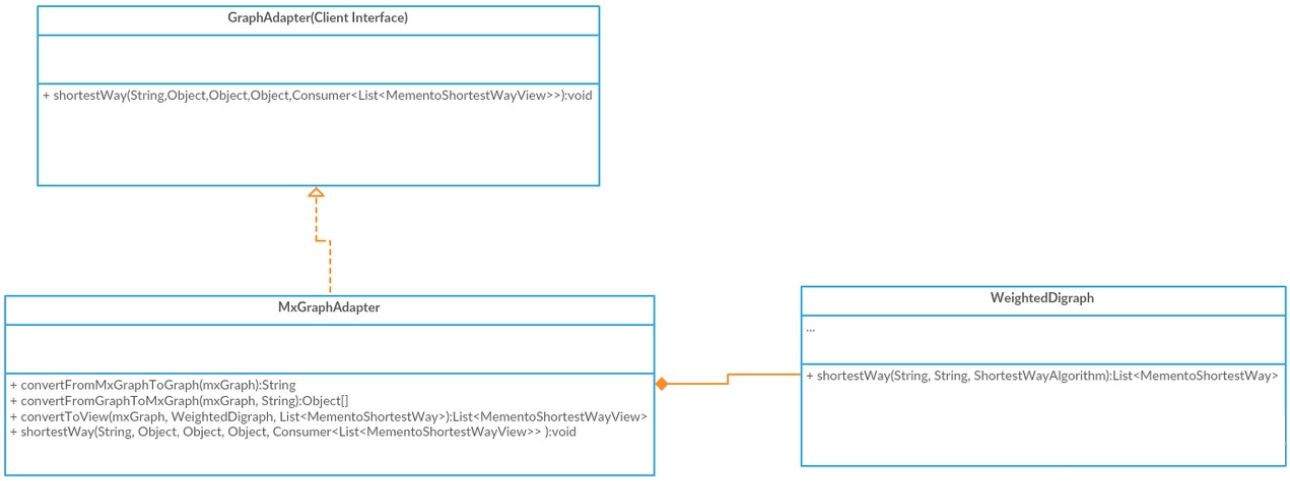

Рисунок 3- UML диаграмма реализации алгоритма

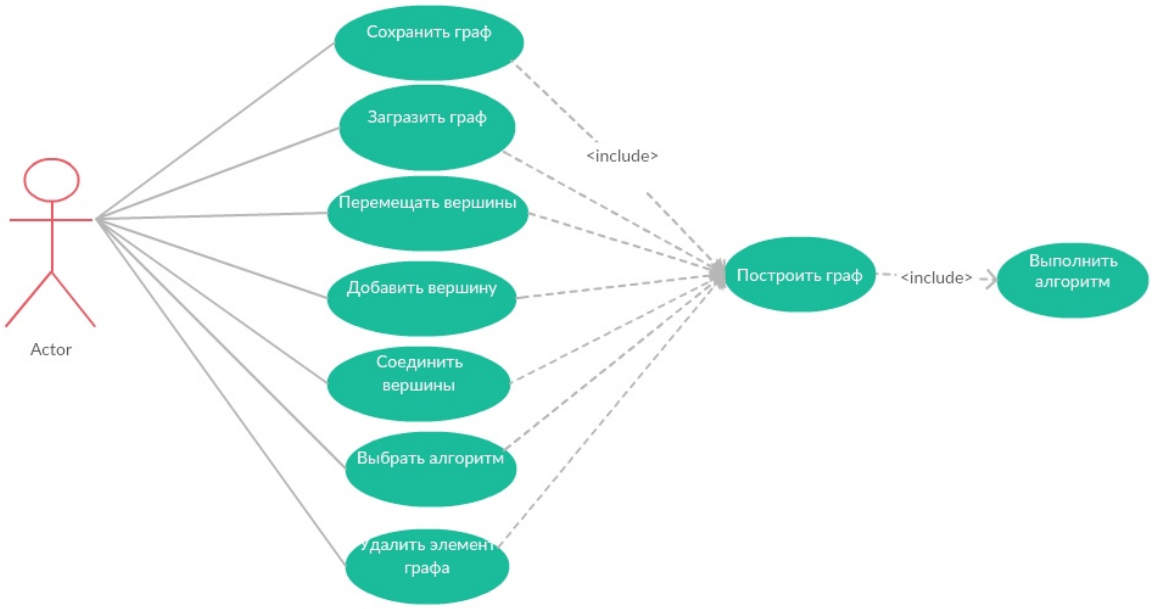

Рисунок 4- Use case диаграмма

1.3. Уточнение требований после сдачи прототипа

- Необходимо добавить кнопку, отвечающую за старт работы алгоритма, на построенном графе.
- Необходимо добавить кнопку «Завершить», отвечающую за отображение результата работы алгоритма Дейкстры.
- Необходимо добавить окно логирования.

1.4. Уточнение требований после дачи 1-ой версии

- Необходимо изменить цветовую гамму графа.
- Необходимо добавить кнопку «Сброс».
- Необходимо добавить возможность отображения всех логов после нажатия кнопки «Завершить».
- Необходимо добавить тестирование программы.

2. ПЛАН РАЗРАБОТКИ И РАСПРЕДЕЛЕНИЕ РОЛЕЙ В БРИГАДЕ

2.1. План разработки

№ п/п	Наименование работ	Срок выполнения
1	Проектирование прототипа и построение UML диаграмм	03.07-04.07
2	Написание кода алгоритма, разработка обработчиков кнопок интерфейса.	03.07-06.07
3	Написание кода, для демонстрации пошаговой работы алгоритма	06.07-07.07
4	Исправление недочетов проекта, добавление необходимых кнопок	08.07-11.07
5	Написание тестов к проекту	08.07-11.07

2.2. Распределение ролей в бригаде

Абдульманов Э. - архитектура программы.

Каляева А. – реализация алгоритма.

Комаров А. – визуализация работы алгоритма.

3. ОСОБЕННОСТИ РЕАЛИЗАЦИИ

3.1. Архитектура программы

- Архитектура данного приложения построена на основе шаблона проектирования MVC (Model View Controller) (см. рис. 1). Суть данного шаблона заключается в том, что любое действие пользователя обрабатывается в контроллере, который имеет доступ к модели. Контроллер, в свою очередь, изменяет состояние модели. View отображает текущее состояние модели. Таким образом, выделив 3 интерфейса (GraphCreatorController, GraphCreator View, GraphCreatorModel) разрабатывать данное приложение можно независимо в трех «плоскостях».[2]
- Для того, чтобы пользователь мог добавлять, перемещать, соединять вершины, а так же удалять элементы графа был применен шаблон проектирования State (Состояние) (см. рис. 2). Пользователь имеет 2 рычага управления – нажать и отпустить кнопку мыши. В разные моменты времени эти два рычага выполняют разные действия: добавить вершину, удалить элемент графа и так далее. Чтобы не писать большие условные конструкции был применен шаблон проектирования State. Были выделены следующие состояния: DeleteState, MoveState, AddVertexState, ConnectionVertexState, AlgorithmShortestWayState. Контекстом для данных состояний стал контроллер, который имеет поле currentState и обрабатывает действия кнопки мыши. Смена состояний происходит путем обработки нажатий на соответствующие кнопки.[3]
- Следующая задача заключалась в том, чтобы разработать алгоритм таким образом, чтобы он не зависел от его отображения. Первая идея была в том, чтобы вынести реализацию алгоритма в другой модуль и написать его так, будто выполняется консольное приложение. Вторая идея была в том, чтобы применить шаблон проектирование Adapter. Прежде чем запустить алгоритм, данные преобразовываются в необходимый

формат, алгоритм выполняется, а затем данные обратно преобразовываются для отображения (см. рис. 3).

- Следующая проблема заключалась в реализации пошаговой визуализации работы алгоритма. Решить эту задачу помогла идея шаблона проектирования Снимок. Был создан класс MementoShortestWay, который хранит в себе текущую вершину, обработанные вершины, вершины в очереди, текущие пути до вершин. В определенные моменты алгоритма в контейнер закидываются снимки текущего состояния алгоритма. Затем, когда необходимо сделать шаг вперед просто отображается i+1 снимок.
- Так же был продуман и реализован пользовательский интерфейс (см. рис. 4).

3.2. Используемые структуры данных

Для реализации алгоритма Дейкстры были разработаны следующие структуры данных:

- Класс DirectedEdge, который хранит информацию о ребре графа.

Поля:

private final int source - хранит номер исходной вершины ребра.

private final int target – хранит номер конечной вершины ребра.

private final double weight – хранит вес ребра.

Методы:

public int getFrom() - позволяет получить номер начальной вершины ребра.

public int getTo() – позволяет получить номер конечной вершины ребра.

public double getWeight() – позволяет получить информацию о весе ребра.

public String toString() – позволяет получить строковое представление.

- Класс Digraph, который хранит информацию о количестве вершин графа, а так же список всех ребер графа.

Поля:

private int vertexCount – количество вершин в графе.

private List<Set<DirectedEdge>> edges = new ArrayList<>() – список всех ребер графа.

Методы:

public int getVertexCount() – позволяет получить информацию о количестве вершин в заданном графе.

public void addEdge(DirectedEdge edge) – позволяет добавить новое ребро.

public Iterable<DirectedEdge> getEdgesForVertex(int vertex) – позволяет получить информацию о всех ребрах смежных заданной вершине.

- Класс WeightedDigraph, которых хранит информацию о графе. Данный класс предназначен для того, чтобы не перегружать класс Digraph. Используется для реализации принципа разделения ответственности, который заключается в разделения компьютерной программы на функциональные блоки, как можно меньше перекрывающие функции друг друга.

Поля:

private Digraph graph – содержит информацию о компонентах графа.

private Map<String, Integer> vertexNameOfNumber – содержит пару «ключ, значение», где ключ – это имя вершины, а значение - номер данной вершины в графе.

Методы:

private List<List<String>> prepareGraphStringRepresentation(String stream, String separator) – производит разбор графа в строковом виде для его преобразования в более удобный вид для дальнейшей обработки.

private void buildGraphFromStringRepresentation(List<List<String>> graphStringRepresentation, int graphSize) – строит граф по заданному строковому представлению.

public int index(String s) – позволяет получить порядковый номер вершины в графе по ее имени.

public String name(int vertex) – позволяет получить имя вершины по ее порядковому номеру в графе.

public List<MementoShortestWay> shortestWay(String source, String target, ShortestWayAlgorithm algorithm) –преобразовывает найденный путь в массив снимков для дальнейшей визуализации.

- Класс Entry, который хранит в себе пару «ключ, значение», необходимую для работы с очередью приоритетов.

Поля:

private double key – хранит метку до текущей вершины.

private int value – хранит номер вершины в графе.

Методы:

public Double getKey() – позволяет получить информацию о текущей метке вершины.

public Integer getValue() – позволяет получить информацию о номере вершины.

public int compareTo(Entry other) – позволяет производить сравнение.

public boolean equals(Object o) – выполняет проверку на равенство.

3.3 Основные методы

Для реализации алгоритма Дейкстры были написаны следующие классы:

- Интерфейс public interface ShortestWayAlgorithm, который хранит методы, которые реализуют работу любого алгоритма на поиск кратчайших путей в графе.

- Класс MementoShortestWay, который хранит информацию о снимках, т.е о текущем состоянии алгоритма.

Поля:

private final int currentVertex – текущая обрабатываемая вершина.

private final boolean[] processedVertices – массив обработанных вершин.

private final DirectedEdge[] currentWays – массив просматриваемых ребер.

private final PriorityQueue<Entry> inQueueVertices – очередь с приоритетом для выбора следующей вершины, предназначенной для обработки.

private final String[] log – массив строк, содержащий информацию о работе алгоритма.

Методы:

public String getCurrentVertex (WeightedDigraph digraph, String separator) – позволяет получить информацию о вершине, находящейся в обработке, в строковом виде.

public String getProcessedVertices(WeightedDigraph digraph,String separator) – позволяет получить информацию об уже обработанных вершинах в строковом виде.

public String getCurrentWays(WeightedDigraph digraph,String separator) – позволяет получить текущий путь в строковом виде.

public String getInQueueVertices(WeightedDigraph digraph,String separator) - позволяет получить информацию о вершинах, находящихся в очереди с приоритетом.

public String[] getLog() – позволяет получить логи.

- Класс Dijkstra, который реализует интерфейс и осуществляет работу алгоритма Дейксры.

Поля:

private DirectedEdge[] edgeTo – массив ребер.

private double[] distTo – массив меток до вершин.

private boolean[] visitedVertex – массив просмотренных вершин.

private PriorityQueue<Entry> priorityQueue – очередь с приоритетом, которая хранит элементы типа Entry для выбора последующей вершины.

private List<MementoShortestWay> steps –список состояний графа во время работы алгоритма.

Методы:

public List<MementoShortestWay> buildWay(Digraph graph, int source, int target) – строит путь из заданной вершины в конечную. Данный метод по сути является реализацией алгоритма Дейкстры. На вход принимает анализируемый граф, а так же номера ствртовой и конечной вершин в задонном графе. Для изменения очереди с приоритетом внутри данного метода вызывается метод relax(…).

private void relax(Digraph graph, int vertex) - производит релаксацию метки. Если метка указанной вершины может быть уменьшена, то производится релаксация. После чего меняется положение данной вершины в очереди с приоритетом.

public boolean hasPathTo(int v)- позволяет проверить существует ли путь до указанной вершины.

public Iterable<DirectedEdge> pathTo(int v) – возвращает путь до заданной вершины.

- Интерфейс GraphCreatorController предназначен для обработки действий пользователя.

Методы:

void saveGraph() – сохраняет граф в файле. В цикле обходятся все элементы графа, сереализуются и затем записываются в файл. В случае возникновения ошибки открывается диалоговое окно, которое сообщает о данной проблеме.

void loadGraph() – загружает граф из файла. При вызове данного метода происходит десериализация всех элементов графа. Если произошла

ошибка, то открывается диалоговое окно, которое сообщает о данной проблеме.

void setStateOfMotion() – устанавливает состояние перемещения вершин в графе при нажатии пользователя на кнопку «Перемещение». Теперь currentState = MoveState

void setStateOfAddingVertices – устанавливает состояние добавления вершин в редактор при нажатии пользователем на кнопку «Добавить вершину». CurrentState = AddVertexState

void setStateOfConnectionVertices() – устанавливает состояние соединения вершин в графе при нажатии пользователем на кнопку «Соединить вершины». CurrentState = ConnectionVertexState

void setStateOfDelete() – устанавливает состояние удаления элементов графа при нажатии пользователем на кнопку «Удалить». CurrentState = DeleteState

void setStateOfAlgorithm() – устанавливает состояние обработки алгоритма при выборе пользователем алгоритма.

void nextStep() – следующий шаг алгоритма, делегирует работу конкретному состоянию. Когда пользователь находится в каком-то состоянии алгоритма, то при нажатии на кнопку «Вперед» выполняется этот метод. Он отображает следующий снимок алгоритма.

void backStep() – предыдущий шаг алгоритма, делегирует работу конкретному состоянию. Когда пользователь находится в каком-то состоянии алгоритма, то при нажатии на кнопку «Назад» выполняется этот метод. Он отображается предыдущий снимок алгоритма.

void startAlgorithm() –начать алгоритм (показать первый шаг), делегирует работу конкретному состоянию. Когда пользователь находится в каком-то состоянии алгоритма, то при нажатии на кнопку «Старт» выполняется этот метод. Он отображается первый снимок алгоритма.

void finishAlgorithm() – закончить алгоритм (показать конечный результат), делегирует работу конкретному состоянию. Когда пользователь находится в каком-то состоянии алгоритма, то при нажатии на кнопку «Завершить» выполняется этот метод. Он отображается результат алгоритма на экране и дописывает в логи все шаги алгоритма.

void mousePressed(MouseEvent e,Object cell) – обрабатывает нажатие кнопки мыши на поле редактора, делегирует работу конкретному состоянию. Состояние добавления вершин – открывается диалоговое окно, при корректном вводе в редакторе появляется вершина. Состояние перемещения – при нажатии выделяется вершина и перемещается. Состояние соединения вершина – при попадании по вершине она выделяется, если это конечная вершина, то в редакторе появляется ребро. Состоянии удаления – при попадании на элемент графа он удаляется, если это вершина, то еще удаляются смежные ребра. Состояние алгоритма – обрабатывает действия для конкретного алгоритма.

void mouseReleased(MouseEvent e, Object cell) – обрабатывает отпускание кнопки мыши на поле редактора, делегирует работу конкретному состоянию. Это реакция на действие пользователя обрабатывается только в MoveState. Когда пользователь отпускает кнопку мыши, то вершина прекращает быть выделенной и перестает перемещаться.

- Класс GraphCreatorControllerImpl является конкретным контроллером, который реализует интерфейс GraphCreatorController.
- Интерфейс GraphCreatorModel является моделью графа.

Методы:

Boolean addVertex(String name,double posX, double posY,double width, double height) – добавляет вершину в граф в позицию (posX,posY) размера (width,height). Если вершина с таким именем уже существует, то функция вернет false, иначе true

Boolean addEdge(String weight, Object vertex1,Object vertex2) – добавляет ребро между двумя вершинами. Если вес вершины не может быть преобразован в double, то функция возвращает false, иначе true.

Void delete(Object[] cells) – удаляет элемент графа.

Int checkExistEdge(Object s,Object t) – проверяет есть ли ребро между двумя вершинами, если ребро существует, то возвращается индекс этого ребра, иначе -1

Void setNormalStyle() – устанавливает нормальный стиль для всего графа. Выполняется проход по всем элементам графа и установка стиля. Если это ребро MY_CUSTOM_EDGE_NORMAL_STYLE, если вершина – MY_CUSTOM_VERTEX_NORMAL_STYLE.

Void setStyleSelected(Boolean flag,Object[] cells) – устанавливает для набора элементов графа стиль выделенного. Если это ребро MY_CUSTOM_EDGE_SELECT_STYLE, если вершина – MY_CUSTOM_VERTEX_SELECT _STYLE.

Void setStyle(String style,Object[] cells) – добавляет стиль для набора элементов графа.

Void saveGraph(String fileName) – сохраняет граф в файле. Выполняется сереализация всех элементов графа. Если произошла ошибка, то вылетает exception.

Void loadGraph(String fileName) – загружает граф из файла. Выполняется десеарилизация всех элементов графа. Если произошла ошибка, то вылетает exception.

Object getGraph() – выдает объект графа.

- StyleManager - класс, который для определенного имени устанавливает определенный стиль. Используются функции библиотеки mxGraph. Реализуются функции для установки стиля для нормальной вершины, для выделенной вершины, для выделенного ребра, для

нормального ребра, для вершины, которая находится в очереди, для вершины, которая является текущей.

- Интерфейс State описывает состояние редактора, которому делегируются вызовы методов контроллера.

Методы:

Void nextStep() – следующий шаг алгоритма. Реализуется только в состояниях, которые выполняют алгоритм. Отображают на экране следующий снимок и увеличивает индекс, который указывает на текущий снимок.

Void backStep() – предыдущий шаг алгоритма. Реализуется только в состояниях, которые выполняют алгоритм. Отображают на экране предыдущий снимок и уменьшает индекс, который указывает на текущий снимок.

Void startAlgorithm – начать алгоритм, предает управление адаптеру, который конвертирует граф в вид, необходимый данному алгоритму. После того, как алгоритм выполнился, он возвращает список снимков, которые конвертируются в необходимый для отображения вид. В конце вызывается callback. Все это выполняется в другом потоке, чтобы пользовательский поток не зависал.

Void finishAlgorithm – закончить алгоритм. Показывает конечный результат алгоритма, то есть путь от начальной до конечной вершины и выводит в логи все шаги алгоритма.

Void mousePressed(double posX,double posY,Object cell) – обработать нажатие кнопки мыши.

Void mouseReleased(double posX,double posY,Object cell) – обработать отпускание кнопки мыши.

String getStatus() - выдает помощь пользователю для конкретного состояния.

Void close() – устанавливает нормальное состояния после своей работы. Вызывается при смене состояний. Так как состоянии может изменять модель, то при смене состояний, все должно «вернуться на свои места».

AddVertexState, AlgorithmShortestWayState, ConnectionVertexState, DeleteState, MoveState - конкретные состояния, реализующие интерфейс State.

- Класс MementoShorrtestWayView – класс, который является снимком для отображения шага алгоритма.

Поля:

Object currentVertext – текущая вершина.

Object[] processedVertices – обработанные вершины.

Object[] currentWays – текущие пути до вершин.

Object[] inQueueVertices – вершины в очереди.

Object[] answer – элементы графа, которые являются ответом.

- GraphAdapter - интерфейс адаптера, который соединят сам алгоритм, с его отображением.

Методы:

shortestWay(String alg,Object gr,Object s,Object t,Consumer<List<MementoShortestWayView>> callbackEnd) – сначала визуализированный граф конвертируется в вид, необходимый для данного алгоритма. После этого выполняется сам алгоритм, а затем происходит обратная конвертация графа в вид необходимый для отображения. В самом конце вызывается callback.

- MxGraphAdapter - класс конкретного адаптера, который реализует интерфейс GraphAdapter.

Методы:

String converFromMxGraphToGraph(mxGraph graph) – позволяет переводить визуализированный граф в вид, необходимый для работы алгоритма. Выполняется проход по всем элементам графа. Элементы

добавляются в stringBuilder. Если это вершина, то «имя_вершины$\n», если ребро – то «имя_вершины1$имя_вершины2$вес\n».

Object[] convertFromGraphToMxGraph(mxGraph graph,String substring) – переводит данные, которыми оперирует алгоритм, в данные для отображения. Строка разбивается на строки по символу $ и если это ребро, то ищется соответствующие ребро в графе, если вершина – то вершина.

List<MementoShortestWayView> convertToView(mxGraph graph, WeightedDigraph digraph, List<MementoShortestWay> mementos) –

метод, который переводит снимки алгоритма в снимки для его отображения. Выполняется проход по всем снимкам алгоритма и для каждого поля снимка вызывается метод convertFromGraphToMxGraph(mxGraph graph,String substring), возвращенный результат добавляется в снимок для отображения, а сам снимок в список снимков, который затем возвращает метод.

- GraphCreatorView - интерфейс, который отвечает за отображения всего приложения.

Void setEnabledStartButton(Boolean show) – устанавливает возможность взаимодействовать с кнопкой старт. Метод вызывается из состоянии алгоритма.

Void setEnabledFinishButton(Boolean show) – устанавливает возможность взаимодействовать с кнопкой финиш. Метод вызывается из состоянии алгоритма.

Void setEnabledNextButton(Boolean show) – устанавливает возможность взаимодействовать с кнопкой следующий. Метод вызывается из состоянии алгоритма.

Void setEnabledBackButton(Boolean show) – устанавливает возможность взаимодействовать с кнопкой назад. Метод вызывается из состоянии алгоритма.

Void setLabelHelp(String strHelp) – показывает помощь для пользователя для характерного состояния, чтобы пользователь понимал, как пользоваться приложением.

Void setLog(String message) – показывает логи.

Void showErrorDialog(String title, String message) – показывает диалог с ошибками. Вызывается, когда пользователь неправильный формат данных для веса ребра, и когда вершина с таким именем уже существует.

String showInputDialog(String title, String message) – показывает диалог для ввода текста. Вызывается, когда пользователь находится в состояниях добавления вершин или соединении вершин

File showFileChooserDialog(String title) - показывает диалог для выбора файла. Вызывает, когда пользователь хочет сохранить граф или загрузить граф из файла.

String getSelectAlgorithm() – выдает выбранный алгоритм на данный момент. Выдает строку, которая была выбрана в JComboBox.

- GraphCreatorViewImpl – конкретный класс отображения, реализующий интерфейс GraphCreatorView.

4. ТЕСТИРОВАНИЕ

4.1 Тестирование работы алгоритма

Для проверки работы алгоритма Дейкстры, реализованного на языке программирования Java был создан класс DijkstraTest, содержащий следующие тесты:

testThatPathHasToIsTrueIfPathReallyExistOnDirectWay() - проверяет корректность работы метода hasPathTo() на простом графе. Для этого создается граф из двух вершин, соединенных ребром. Ожидаемый результат: true.

testThatPathIsCorrectOnDirectWay() – проверяет корректность работы метода pathTo() на простом графе. Для этого создается граф из двух вершин, соединенных ребром. Ожидаемый результат: 1.

testThatPathToIsEmptyIfPathDoesNotExist() – проверяет корректность работы метода pathTo() в случае, когда пути не существует.

testThatPathHasToIsFalseIfPathDoesNotExistOnDirectWay() - проверяет на корректность метод hasPathTo() в случае, когда пути не существует.

testThatPathHasToIsTrueIfPathReallyExistOnComplexWay() - проверяет корректность работы метода hasPathTo() для графа, содержащего большее количество вершин и ребер.

testThatPathHasToIsThrowExceptionIfVertexDoesNotExist() – проверяет на наличие исключения при попытке найти путь до несуществующей вершины.

testThatWeTryToFindPathOnEmptyGraph() – проверяет на наличие исключения при попытке построить путь на пустом графе.

testThatWeTryToFindPathOnGraphWithAloneVertex() – проверяет корректность построения пути для единственной вершины в графе.

testThatWeTryToFindPathOnSimpleGraphWithDirectWay() – проверяет корректность построения пути на простом графе.

testThatWeTryToFindPathOnSimpleGraphWithDirectWayButWhenExistMoreComplexWay() – проверяет на корректность построения кратчайшего пути при наличии нескольких вариантов путей из стартовой вершины в конечную.

testThatWeTryToFindWayOnMoreComplexGraph() - проверяет корректность работы на более полном графе.

Затем работа реализованного алгоритма Дейкстры была проверена на данных тестах (см. рис. 5).

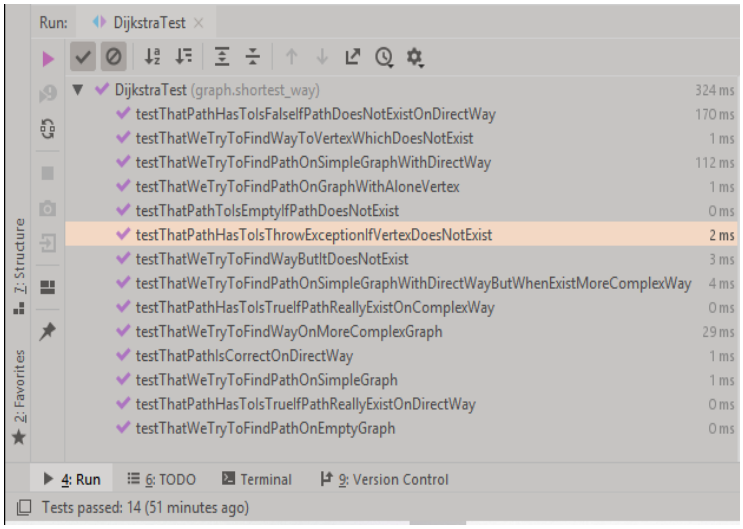

Рисунок 5 – Результат тестирования алгоритма Дейкстры

4.2 Тестирование визуализации

- Класс GraphCreatorContollerImplTest – тестирует конкретные состояния (перемещение, удаление, добавление вершин, соединение вершин) путем создания экземпляра конкретного State и вызыва метода getStatus() (см. рис. 6). Затем выполняется сравнение с соответствующей строкой.

Методы:

Public void setStateOfMotion() – сравнивает возвращаемое значение метода getStatus() со строкой «Выделите и перемещайте объекты».

Public void setStateOfAddingVertices() – сравнивает возвращаемое значение метода getStatus() со строкой «Кликните на рабочую область, чтобы добавить вершину»

Public void setStateOfConnectionVertices() – сравнивает возвращаемое значение метода getStatus() со строкой «Выделите первую вершину для создания дуги»

Public void setStateOfDelete – сравнивает возвращаемое значение метода getStatus() со строкой «Кликните по объекту, который хотите удалить»

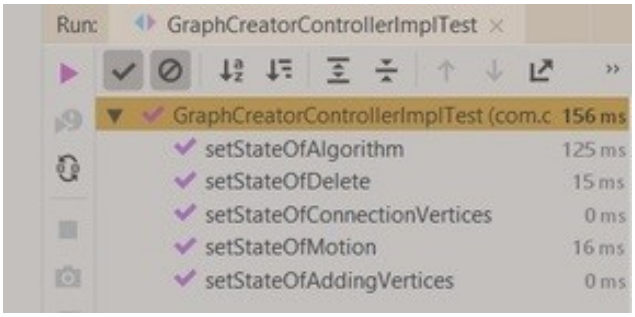

Рисунок 6 – Результат тестирования конкретных состояний

- · Класс GraphCreatorViewTest – тестирует нажатие кнопок смены состояний (см. рис. 7). Для кнопки вызывается метод doClick(), который инициирует нажатие кнопки, затем view.getToolBar().getLabelHelp().getText(), после чего выполняется сравнение с соответствующей строкой.

Методы:

Public void TestClickDeleteButton() – сравнивает возвращаемое значение метода view.getToolBar().getLabelHelp().getText() со строкой «Кликните по объекту, который хотите удалить»

Public void TestClickMoveButton() – сравнивает возвращаемое значение метода view.getToolBar().getLabelHelp().getText() со строкой «Выделите и перемещайте объекты»

Public void TestClickAddVertexButton() – сравнивает возвращаемое значение метода view.getToolBar().getLabelHelp().getText() со строкой «Кликните на рабочую область, чтобы добавить вершину»

Public void TestClickConnectionVertexButton() – сравнивает возвращаемое значение метода view.getToolBar().getLabelHelp().getText() со строкой «Выделите первую вершину для создания дуги».

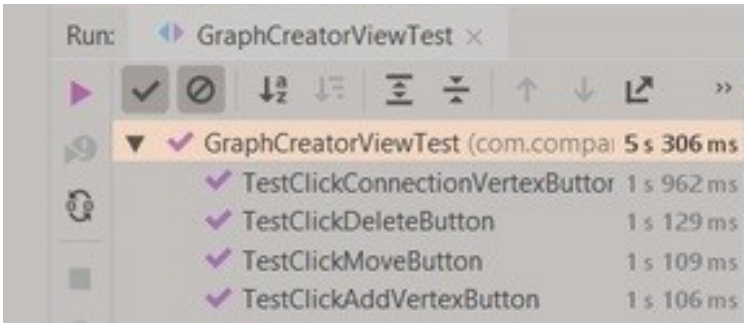

Рисунок 7 – Результат тестирования кнопок смены состояний

- · Класс GraphCreatorModelImplTest – тестирует работу с графом (см. рис. 8). Данный метод проверяет работу следующих методов: добавить вершину, добавить ребро, удалить элемент графа.

Методы:

Public void addVertex() –данный метод вызывает model.addVertex(…), а затем проверяет, существует ли в графе созданная вершина или нет.

Public void addEdge() – метод вызывает model.addEdge(…), а затем сравнивает возвращаемое значение метода checkExistEdge(…) (метод проверяет существование ребра с true).

Public void delete() – метод создает вершину и вызывает model.delete(), а затем проверяет graph.getChildCells(graph.getDefaultParent()).lenght на равенство нулю.

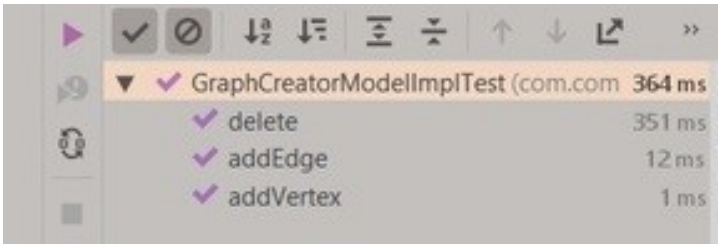

Рисунок 8 – Результат тестирования работы методов графа

ЗАКЛЮЧЕНИЕ

В ходе выполнения задания учебной практики были изучены основы программирования на языке Java. Для изучения данного языка был пройден интерактивный курс «Java.Базовый курс» на платформе Stepik. После чего была разработана программа, которая находит кратчайшие пути в графе с помощью алгоритма Дейкстры. Разработанная программа соответствует всем заявленным требованиям спецификации.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1) Р.Седжвик, К. Уэйн книга Алгоритмы на Java. Изд-во ”Вильямс”, 2013.– 846 с.

2) Java Базовый курс.[Электронный ресурс]

URL: stepik.org/course/Java-Базовый-курс-187/syllabus(дата обращения 7.07.2019)

3) The Java Tutorials.[Электронный ресурс]

URL:https://docs.oracle.com/javase/tutorial/index.html(дата обращения 7.07.2019)

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

```
package graph.data_structures;

import java.util.*;

public class Digraph {
    private int vertexCount;
    private List<Set<DirectedEdge>> edges = new ArrayList<>();

    public Digraph(int vertexCount) {
        this.vertexCount = vertexCount;
        for (int vertexNumber = 0; vertexNumber < vertexCount; vertexNumber++) {
            this.edges.add(new LinkedHashSet<>());
        }
    }

    public int getVertexCount() {
        return vertexCount;
    }

    public void addEdge(DirectedEdge edge) {
        edges.get(edge.getFrom()).add(edge);
    }

    public Iterable<DirectedEdge> getEdgesForVertex(int vertex) {
        return edges.get(vertex);
    }
}

package graph.data_structures;

public class DirectedEdge {
    private final int source;
    private final int target;
    private final double weight;

    public DirectedEdge(int source, int target, double weight) {
        this.source = source;
        this.target = target;
        this.weight = weight;
    }

    public int getTo() {
        return target;
    }

    public int getFrom() {
        return source;
    }
```

```
    public double getWeight() {
        return weight;
    }
}

package graph.data_structures;

import java.util.Objects;

public class Entry implements Comparable<Entry> {
    private double  key;
    private int value;

    public Entry(Double key, Integer value) {
        this.key = key;
        this.value = value;
    }

    public Entry(Integer value) {
        this.value = value;
    }

    public Double getKey() {
        return key;
    }

    public Integer getValue() {
        return value;
    }

    @Override
    public int compareTo(Entry other) {
        return this.getKey().compareTo(other.getKey());
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Entry entry = (Entry) o;
        return Objects.equals(value, entry.value);
    }

    @Override
    public int hashCode() {
        return Objects.hash(value);
    }
}

package graph.data_structures;
```

```
import graph.shortest_way.MementoShortestWay;
import graph.shortest_way.ShortestWayAlgorithm;

import java.util.*;
import java.util.stream.Stream;

public class WeightedDigraph {
   private Digraph graph;
   private Map<String, Integer> vertexNameOfNumber;

   public WeightedDigraph(String stream, String separator) {
      final List<List<String>> graphStringRepresentation =
prepareGraphStringRepresentation(stream, separator);
      final int graphSize =
graphStringRepresentation.stream().map(List::size).reduce(Integer::sum).orElse(0);
      buildGraphFromStringRepresentation(graphStringRepresentation, graphSize);
   }

   private List<List<String>> prepareGraphStringRepresentation(String stream, String separator) {
      final List<List<String>> graphStringRepresentation = new ArrayList<>();
      final Scanner scanner = new Scanner(stream);
      while (scanner.hasNext()) {
         List<String> stringGraph = Arrays.asList(scanner.nextLine().split(String.format("[\\%s]",
separator)));
         graphStringRepresentation.add(stringGraph);
      }
      return graphStringRepresentation;
   }

   private void buildGraphFromStringRepresentation(List<List<String>>
graphStringRepresentation, int graphSize) {
      graph = new Digraph(graphSize);
      vertexNameOfNumber = new HashMap<>();
      for (List<String> graphLine : graphStringRepresentation) {
         final String sourceVertexName = graphLine.get(0);
         if (graphLine.size() > 1) {
            final String targetVertexName = graphLine.get(1);

            vertexNameOfNumber.putIfAbsent(sourceVertexName, vertexNameOfNumber.size());
            vertexNameOfNumber.putIfAbsent(targetVertexName, vertexNameOfNumber.size());

            final int sourceVertexNumber = vertexNameOfNumber.get(sourceVertexName);
            final int targetVertexNumber = vertexNameOfNumber.get(targetVertexName);
            final double weight = Double.parseDouble(graphLine.get(2));
            graph.addEdge(new DirectedEdge(sourceVertexNumber, targetVertexNumber, weight));
         } else {
            vertexNameOfNumber.putIfAbsent(sourceVertexName, vertexNameOfNumber.size());
         }
      }
   }
```

```
    public int index(String s) {
        return vertexNameOfNumber.get(s);
    }

    public String name(int vertex) {
        return new ArrayList<>(vertexNameOfNumber.keySet()).stream()
                                            .filter(el -> vertexNameOfNumber.get(el)
                                                                    .equals(vertex))
                                            .findFirst()
                                            .get();
    }

    public List<MementoShortestWay> shortestWay(String source, String target,
ShortestWayAlgorithm algorithm) {
        return algorithm.buildWay(graph, index(source), index(target));
    }
}
```

```
package graph.shortest_way;

import graph.data_structures.Digraph;
import graph.data_structures.DirectedEdge;
import graph.data_structures.Entry;

import java.util.*;

public class Dijkstra implements ShortestWayAlgorithm {

    private DirectedEdge[] edgeTo;
    private double[] distTo;
    private boolean[] visitedVertex;
    private PriorityQueue<Entry> priorityQueue;
    private List<MementoShortestWay> steps;

    private void relax(Digraph graph, int vertex) {
        steps.add(new MementoShortestWay(vertex, visitedVertex, edgeTo, priorityQueue,
                new String[]{"Выбрана текущая вершина: %s", Integer.toString(vertex)}));
        for (DirectedEdge edge : graph.getEdgesForVertex(vertex)) {
            int vertexTo = edge.getTo();
            if (distTo[vertexTo] > (distTo[vertex] + edge.getWeight())) {
                String log = "Была произведена релаксация, метка вершины %s была изменена с "
                        + distTo[vertexTo] + " на " + (distTo[vertex] + edge.getWeight());
                distTo[vertexTo] = distTo[vertex] + edge.getWeight();
                this.edgeTo[vertexTo] = edge;
                if (priorityQueue.contains(new Entry(vertexTo))) {
                    final Iterator<Entry> iterator = priorityQueue.iterator();
                    //boolean flag = false;
                    while (iterator.hasNext()) {
                        final Entry currentElement = iterator.next();
                        if (currentElement.getValue().equals(vertexTo)) {
                            iterator.remove();
```

```
                    //flag = true;
                    break;
                }
            }
            //if (flag) {
                priorityQueue.offer(new Entry(distTo[vertexTo], vertexTo));
            //}
        } else {
            priorityQueue.offer(new Entry(distTo[vertexTo], vertexTo));
        }
        steps.add(new MementoShortestWay(vertex, visitedVertex, this.edgeTo, priorityQueue,
                new String[]{log, Integer.toString(vertexTo)}));
      }
    }
    visitedVertex[vertex] = true;
    steps.add(new MementoShortestWay(-1, visitedVertex, edgeTo, priorityQueue,
            new String[]{"Обаботка вершины %s была закончена", Integer.toString(vertex)}));
  }

  @Override
  public List<MementoShortestWay> buildWay(Digraph graph, int source, int target) {
    steps = new ArrayList<>();
    edgeTo = new DirectedEdge[graph.getVertexCount()];
    distTo = new double[graph.getVertexCount()];
    visitedVertex = new boolean[graph.getVertexCount()];
    priorityQueue = new PriorityQueue<>();
    for (int vertexNumber = 0; vertexNumber < graph.getVertexCount(); vertexNumber++)
      distTo[vertexNumber] = Double.POSITIVE_INFINITY;
    distTo[source] = 0.0;
    priorityQueue.offer(new Entry(distTo[source], source));
    steps.add(new MementoShortestWay(-1, visitedVertex, edgeTo, priorityQueue,
            new String[]{"Начальное состояние алгоритма. В очереди только одна вершина: %s",
Integer.toString(source)}));

    while (!priorityQueue.isEmpty()) {
      relax(graph, priorityQueue.poll().getValue());
    }

    return steps;
  }

  @Override
  public boolean hasPathTo(int v) {
    return distTo[v] < Double.POSITIVE_INFINITY;
  }

  @Override
  public Iterable<DirectedEdge> pathTo(int v) {
    if (!hasPathTo(v)) return null;
    Stack<DirectedEdge> path = new Stack<>();
    for (DirectedEdge e = edgeTo[v]; e != null; e = edgeTo[e.getFrom()])
```

```
            path.push(e);
        return path;
    }

}


package graph.shortest_way;

import graph.data_structures.DirectedEdge;
import graph.data_structures.WeightedDigraph;
import graph.data_structures.Entry;

import java.util.PriorityQueue;

public class MementoShortestWay {
    private final int currentVertex;
    private final boolean[] processedVertices;
    private final DirectedEdge[] currentWays;
    private final PriorityQueue<Entry> inQueueVertices;
    private final String[] log;

    public MementoShortestWay(int currentVertex,
                              boolean[] processedVertices,
                              DirectedEdge[] currentWays,
                              PriorityQueue<Entry> inQueueVertices,
                              String[] log) {
        this.currentVertex = currentVertex;
        this.processedVertices = processedVertices.clone();
        this.currentWays = currentWays.clone();
        this.inQueueVertices = new PriorityQueue<>(inQueueVertices);
        this.log = log;
    }

    public String getCurrentVertex(WeightedDigraph digraph, String separator) {
        if(currentVertex!=-1)
            return digraph.name(currentVertex)+separator+"\n";
        else
            return "";
    }

    public String getProcessedVertices(WeightedDigraph digraph,String separator) {
        StringBuilder stringBuilder = new StringBuilder();
        for(int i=0;i<processedVertices.length;i++){
            if(processedVertices[i]){
                stringBuilder.append(digraph.name(i)).append(separator).append("\n");
            }
        }
        return stringBuilder.toString();
    }
```

```
    public String getCurrentWays(WeightedDigraph digraph,String separator) {
        StringBuilder stringBuilder = new StringBuilder();
        for(DirectedEdge edge:currentWays){
            if(edge!=null) {
                stringBuilder.append(digraph.name(edge.getFrom()))
                        .append(separator)
                        .append(digraph.name(edge.getTo()))
                        .append(separator)
                        .append(edge.getWeight())
                        .append("\n");
            }
        }
        return stringBuilder.toString();
    }

    public String getInQueueVertices(WeightedDigraph digraph,String separator) {
        StringBuilder stringBuilder = new StringBuilder();
        for(Entry element: inQueueVertices){
            stringBuilder.append(digraph.name(element.getValue())).append(separator).append("\n");
        }
        return stringBuilder.toString();
    }

    public String[] getLog() {
        return log;
    }
}
```

```
package graph.shortest_way;

import graph.data_structures.Digraph;
import graph.data_structures.DirectedEdge;

import java.util.List;

public interface ShortestWayAlgorithm {
    List<MementoShortestWay> buildWay(Digraph G, int source, int target);

    boolean hasPathTo(int v);

    Iterable<DirectedEdge> pathTo(int v);
}
```

```
package com.company.controller;

import com.company.view.GraphCreatorView;

import java.awt.event.MouseEvent;

/**
 * Публичный интерфейс контроллера(Выступает контекстом для состояний)
```

```
 * void setStateOfMotion() - установить состояние перемещения вершин
 * void setStateOfAddingVertices() - установить состояние добавления вершин
 * void setStateOfConnectionVertices() - установить состояние добавления ребер
 * void setStateOfDelete() - установить состояние удаления элементов графа
 * void setStateOfAlgorithm(String algorithm) - установить состояние выбранного алгоритма
 * void nextStep() - следующий шаг алгоримта
 * void backStep() - предыдущий шаг алгоритма
 * void mousePressed(MouseEvent e, Object cell) - обработка нажатие на кнопку мыши
 * void mouseReleased(MouseEvent e, Object cell) - обработка отпускания кнопки мыши
 * void setView(GraphCreatorView view) - установить представление
 */
public interface GraphCreatorController {
    void saveGraph();

    void loadGraph();

    void setStateOfMotion();

    void setStateOfAddingVertices();

    void setStateOfConnectionVertices();

    void setStateOfDelete();

    void setStateOfAlgorithm();

    void nextStep();

    void backStep();

    void mousePressed(MouseEvent e, Object cell);

    void mouseReleased(MouseEvent e, Object cell);

    void setView(GraphCreatorView view);
}

package com.company.controller;

import com.company.model.GraphCreatorModel;
import com.company.model.adapter.GraphAdapter;
import com.company.model.states.*;
import com.company.view.GraphCreatorView;

import java.awt.event.MouseEvent;
import java.io.*;

public class GraphCreatorControllerImpl implements GraphCreatorController {

    private final GraphCreatorModel model;
    private final GraphAdapter adapter;
```

```
    private GraphCreatorView view;
    private State currentState;

    public GraphCreatorControllerImpl(GraphCreatorModel model,GraphAdapter adapter) {
        this.model = model;
        this.adapter = adapter;
    }

    @Override
    public void saveGraph() {
        File file = view.showFileChooserDialog("Сохранить граф");
        if (file != null) {
            try {
                model.saveGraph(file.getAbsoluteFile().toString());
            } catch (IOException e) {
                view.showErrorDialog("Ошибка", "Не удалось сохраненить граф. Попробуйте еще
раз!");
            }
        }
    }

    @Override
    public void loadGraph() {
        File file = view.showFileChooserDialog("Загрузить граф");
        if (file != null) {
            try {
                model.loadGraph(file.getAbsoluteFile().toString());
            } catch (Exception e) {
                view.showErrorDialog("Ошибка", "Не удалось загрузить граф. Попробуйте еще
раз!");
            }
        }
    }

    @Override
    public void setStateOfMotion() {
        currentState.close();
        currentState = new MoveState(model);
        view.setLabelHelp(currentState.getStatus());
    }

    @Override
    public void setStateOfAddingVertices() {
        currentState.close();
        currentState = new AddVertexState(model, view);
        view.setLabelHelp(currentState.getStatus());
    }

    @Override
    public void setStateOfConnectionVertices() {
        currentState.close();
```

```
    currentState = new ConnectionVertexState(model, view);
    view.setLabelHelp(currentState.getStatus());
}

@Override
public void setStateOfDelete() {
    currentState.close();
    currentState = new DeleteState(model);
    view.setLabelHelp(currentState.getStatus());
}

@Override
public void setStateOfAlgorithm() {
    currentState.close();
    if(!(currentState instanceof AlgorithmShortestWayState)) {
        currentState = new AlgorithmShortestWayState(model, view, adapter);
    }
    view.setLabelHelp(currentState.getStatus());
}

@Override
public void nextStep() {
    currentState.nextStep();
    view.setLabelHelp(currentState.getStatus());
}

@Override
public void backStep() {
    currentState.backStep();
    view.setLabelHelp(currentState.getStatus());
}

@Override
public void startAlgorithm() {
    currentState.startAlgorithm();
    view.setLabelHelp(currentState.getStatus());
}

@Override
public void finishAlgorithm() {
    currentState.finishAlgorithm();
    view.setLabelHelp(currentState.getStatus());
}


@Override
public void mousePressed(MouseEvent e, Object cell) {
    currentState.mousePressed(e.getX(), e.getY(), cell);
    view.setLabelHelp(currentState.getStatus());
}
```

```
    @Override
    public void mouseReleased(MouseEvent e, Object cell) {
        if (e != null)
            currentState.mouseReleased(e.getX(), e.getY(), cell);
        else
            currentState.mouseReleased(-1, -1, cell);
        view.setLabelHelp(currentState.getStatus());
    }

    @Override
    public void setView(GraphCreatorView view) {
        this.view = view;
        currentState = new AddVertexState(model, view);
        view.setLabelHelp(currentState.getStatus());
    }
}

package com.company.model.algorithms;

import com.mxgraph.model.mxCell;

import java.awt.*;

public class MementoShortestWayView {
    private final Object currentVertex;
    private final Object[] processedVertices;
    private final Object[] currentWays;
    private final Object[] inQueueVertices;
    private final Object[] answer;

    public MementoShortestWayView(Object currentVertex, Object[] processedVertices, Object[]
currentWays, Object[] inQueueVertices, Object[] answer) {
        this.currentVertex = currentVertex;
        this.processedVertices = processedVertices;
        this.currentWays = currentWays;
        this.inQueueVertices = inQueueVertices;
        this.answer = answer;
    }

    public Object getCurrentVertex() {
        return currentVertex;
    }

    public Object[] getProcessedVertices() {
        return processedVertices;
    }

    public Object[] getCurrentWays() {
        return currentWays;
    }
```

```
    public Object[] getInQueueVertices() {
        return inQueueVertices;
    }

    public Object[] getAnswer() {
        return answer;
    }

}

package com.company.model.adapter;
import com.company.model.algorithms.MementoShortestWayView;
import com.mxgraph.view.mxGraph;

import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.util.List;
import java.util.function.Consumer;

public interface GraphAdapter {
    void shortestWay(String alg, Object gr, Object s, Object t,
Consumer<List<MementoShortestWayView>> callbackEnd);
}

package com.company.model.adapter;


import com.company.model.algorithms.MementoShortestWayView;
import com.mxgraph.model.mxCell;
import com.mxgraph.view.mxGraph;
import graph.data_structures.DirectedEdge;
import graph.data_structures.WeightedDigraph;
import graph.shortest_way.Dijkstra;
import graph.shortest_way.MementoShortestWay;
import graph.shortest_way.ShortestWayAlgorithm;

import java.util.*;
import java.util.function.Consumer;

import static com.company.Constants.SEPARATOR;
import static com.company.Constants.Algorithms.*;

public class MxGraphAdapter implements GraphAdapter {

    private String convertFromMxGraphToGraph(mxGraph graph) {
        Set<String> vertex = new HashSet<>();
        StringBuilder sb = new StringBuilder();
        for (Object e : graph.getChildEdges(graph.getDefaultParent())) {
            mxCell edge = (mxCell) e;
            String source = (String) edge.getSource().getValue();
```

```
            String target = (String) edge.getTarget().getValue();
            vertex.add(source);
            vertex.add(target);

sb.append(source).append(SEPARATOR).append(target).append(SEPARATOR).append(edge.getVa
lue()).append("\n");
        }
        for (Object v : graph.getChildVertices(graph.getDefaultParent())) {
            mxCell ver = (mxCell) v;
            if (!vertex.contains(ver.getValue())) {
                vertex.add((String) ver.getValue());
                sb.append((String) ver.getValue()).append(SEPARATOR).append("\n");
            }
        }
        return sb.toString();
    }

    private Object[] convertFromGraphToMxGraph(mxGraph graph, String subgraph) {
        List<mxCell> cells = new ArrayList<>();
        Scanner scanner = new Scanner(subgraph);
        while (scanner.hasNextLine()) {
            String[] a = scanner.nextLine().split(String.format("[\\%s]", SEPARATOR));
            if (a.length == 1) {
                for (Object v : graph.getChildVertices(graph.getDefaultParent())) {
                    if (((mxCell) v).getValue().equals(a[0])) {
                        cells.add((mxCell) v);
                    }
                }
            } else {
                for (Object e : graph.getChildEdges(graph.getDefaultParent())) {
                    String source = (String) ((mxCell) e).getSource().getValue();
                    String target = (String) ((mxCell) e).getTarget().getValue();
                    if (source.equals(a[0]) && target.equals(a[1])) {
                        cells.add((mxCell) e);
                        cells.add((mxCell) ((mxCell) e).getSource());
                        cells.add((mxCell) ((mxCell) e).getTarget());
                    }
                }
            }
        }
        return cells.isEmpty() ? null : cells.toArray();
    }

    private List<MementoShortestWayView> convertToView(mxGraph graph, WeightedDigraph
digraph, List<MementoShortestWay> mementos) {
        List<MementoShortestWayView> mementosView = new ArrayList<>();
        for (MementoShortestWay memento : mementos) {
            Object[] currentVertex = convertFromGraphToMxGraph(graph,
memento.getCurrentVertex(digraph, SEPARATOR));
            Object[] processedVertices = convertFromGraphToMxGraph(graph,
memento.getProcessedVertices(digraph, SEPARATOR));
```

```
            Object[] currentWays = convertFromGraphToMxGraph(graph,
memento.getCurrentWays(digraph, SEPARATOR));
            Object[] inQueueVertices = convertFromGraphToMxGraph(graph,
memento.getInQueueVertices(digraph, SEPARATOR));
            String log = String.format(memento.getLog()[0],
digraph.name(Integer.parseInt(memento.getLog()[1])));
            if (currentVertex != null)
               mementosView.add(new MementoShortestWayView(currentVertex[0], processedVertices,
currentWays, inQueueVertices, null, log));
            else
               mementosView.add(new MementoShortestWayView(null, processedVertices,
currentWays, inQueueVertices, null, log));
         }
         return mementosView;
      }

      @Override
      public void shortestWay(String alg, Object gr, Object s, Object t,
Consumer<List<MementoShortestWayView>> callbackEnd) {
         new Thread(() -> {
            WeightedDigraph digraph = new
WeightedDigraph(convertFromMxGraphToGraph((mxGraph) gr), SEPARATOR);
            List<MementoShortestWayView> mementosView = null;
            ShortestWayAlgorithm algorithm = null;
            String sourceStr = (String) ((mxCell) s).getValue();
            String targetStr = (String) ((mxCell) t).getValue();
            switch (alg) {
               case DIJKSTRA:
                  algorithm = new Dijkstra();
                  mementosView = convertToView((mxGraph) gr, digraph,
digraph.shortestWay(sourceStr, targetStr, algorithm));
                  break;
               case FORD_BELLMAN:

                  break;
               case ASTAR:

                  break;
            }

            if (algorithm != null) {
               StringBuilder sb = new StringBuilder();
               Object[] answer;
               String log;
               if (algorithm.hasPathTo(digraph.index(targetStr))) {
                  for (DirectedEdge e : algorithm.pathTo(digraph.index(targetStr))) {
                     sb.append(digraph.name(e.getFrom()))
                           .append(SEPARATOR)
                           .append(digraph.name(e.getTo()))
                           .append(SEPARATOR).append(e.getWeight())
                           .append("\n");
```

```
                }
                log = "Путь из вершины " + sourceStr + " в вершину " + targetStr + " найден \n" +
sb.toString();
                answer = convertFromGraphToMxGraph((mxGraph) gr, sb.toString());
            } else {
                answer = null;
                log = "Пути из вершины " + sourceStr + " в вершину " + targetStr + " не существует
\n";
            }
            mementosView.add(new MementoShortestWayView(null, null, null, null,
                    answer, log));
        }
        callbackEnd.accept(mementosView);
    }).start();
  }
}

package com.company.model.algorithms;

import com.mxgraph.model.mxCell;

import java.awt.*;

public class MementoShortestWayView {
  private final Object currentVertex;
  private final Object[] processedVertices;
  private final Object[] currentWays;
  private final Object[] inQueueVertices;
  private final Object[] answer;
  private final String log;

  public MementoShortestWayView(Object currentVertex,
                                Object[] processedVertices,
                                Object[] currentWays,
                                Object[] inQueueVertices,
                                Object[] answer,
                                String log) {
    this.currentVertex = currentVertex;
    this.processedVertices = processedVertices;
    this.currentWays = currentWays;
    this.inQueueVertices = inQueueVertices;
    this.answer = answer;
    this.log = log;
  }

  public Object getCurrentVertex() {
    return currentVertex;
  }

  public Object[] getProcessedVertices() {
    return processedVertices;
```

```
    }

    public Object[] getCurrentWays() {
        return currentWays;
    }

    public Object[] getInQueueVertices() {
        return inQueueVertices;
    }

    public Object[] getAnswer() {
        return answer;
    }

    public String getLog() {
        return log;
    }
}

package com.company.model;

import java.io.IOException;

/**
 * Публичный интерфейс для модели графа
 * boolean addVertex(String name, double posX, double posY, double width, double height) - метод добавляет вершину в граф
 * boolean addEdge(String weight, Object vertex1, Object vertex2) - метод добавляет ребро между двумя вершинами
 * void delete(Object[] cells) - метод удаляет вершину и прилегающие ребра или просто ребро
 * boolean checkExistEdge(Object s, Object t) - метод проверяет, если ли ребро между двумя вершинами
 * void setNormalStyle() - устанавливает для всего графа нормальный стиль
 * void setStyleSelected(boolean flag, Object[] cells) - устанавливает или убирает выделенный стиль для ребер и вершин
 * Object getGraph() - возвращает граф
 * GraphAdapter getAdapter() - возвращает адаптер (для работы с алгоритмами)
 * void saveGraph(String fileName) throws IOException - сохранить граф
 * void loadGraph(String fileName) throws IOException, ClassNotFoundException - загрузить граф
 *
 */

public interface GraphCreatorModel {
    boolean addVertex(String name, double posX, double posY, double width, double height);

    boolean addEdge(String weight, Object vertex1, Object vertex2);

    void delete(Object[] cells);

    int checkExistEdge(Object s, Object t);
```

```
    void setNormalStyle();

    void setStyleSelected(boolean flag, Object[] cells);

    void setStyle(String style, Object[] cells);

    void saveGraph(String fileName) throws IOException;

    void loadGraph(String fileName) throws IOException, ClassNotFoundException;

    Object getGraph();
}
```

```
package com.company.model;

import com.mxgraph.model.mxCell;
import com.mxgraph.util.mxConstants;
import com.mxgraph.view.mxEdgeStyle;
import com.mxgraph.view.mxGraph;
import com.mxgraph.view.mxStylesheet;
import org.jgraph.event.GraphModelEvent;
import org.jgraph.event.GraphModelListener;

import java.io.*;
import java.util.HashMap;
import java.util.Hashtable;
import java.util.Map;

import static com.company.Constants.Size.MAX_SIZE_VERTEX_NAME;
import static com.company.Constants.StyleGraph.*;

public class GraphCreatorModelImpl implements GraphCreatorModel {

    private final mxGraph graph;
    private final Object parent;

    public GraphCreatorModelImpl() {
        this.graph = new mxGraph();
        this.parent = graph.getDefaultParent();
        graph.setCellsEditable(false);//Нельзя редактировать
        graph.setCellsResizable(false);//Нельзя изменять текст
        graph.setDisconnectOnMove(false);//Нельзя двигать ребро
        graph.setCellsDisconnectable(false);//Нельзя отрывать ребро от вершины
        graph.setEdgeLabelsMovable(false);//Нельзя двигать именную метку ребра
        graph.setKeepEdgesInBackground(true);//Ребра на заднем плане
        graph.setCellsSelectable(false);
        graph.setCellsMovable(false);
        StyleManager.initMyCustomEdgeNormalStyle(graph);
        StyleManager.initMyCustomEdgeSelectedStyle(graph);
        StyleManager.initMyCustomVertexNormalStyle(graph);
```

```
        StyleManager.initMyCustomVertexSelectedStyle(graph);
        StyleManager.initMyCustomCurrentVertexStyle(graph);
        StyleManager.initMyCustomInQueueVertexStyle(graph);
    }


  @Override
    public boolean addVertex(String name, double posX, double posY, double width, double height)
{
        for (Object v : graph.getChildVertices(parent)) {
            if (((mxCell) v).getValue().equals(name))
                return false;
        }
        if (name.length() > MAX_SIZE_VERTEX_NAME)
            name = name.substring(0, MAX_SIZE_VERTEX_NAME) + "...";
        graph.getModel().beginUpdate();
        graph.insertVertex(parent, null, name, posX, posY, width, height,
MY_CUSTOM_VERTEX_NORMAL_STYLE);
        graph.getModel().endUpdate();
        return true;
    }

    @Override
    public boolean addEdge(String weight, Object v1, Object v2) {
        try {
            Double.parseDouble(weight);
            graph.getModel().beginUpdate();
            graph.insertEdge(parent, null, weight, v1, v2, MY_CUSTOM_EDGE_NORMAL_STYLE);
            graph.getModel().endUpdate();
        } catch (NumberFormatException e) {
            return false;
        }
        return true;
    }

    @Override
    public void delete(Object[] cells) {
        graph.getModel().beginUpdate();
        graph.removeCells(cells);
        graph.getModel().endUpdate();
    }

    @Override
    public int checkExistEdge(Object s, Object t) {
        mxCell sourceVertex = (mxCell) s;
        mxCell targetVertex = (mxCell) t;
        for (int i = 0; i < sourceVertex.getEdgeCount(); i++) {
            mxCell target = (mxCell) ((mxCell) (sourceVertex.getEdgeAt(i))).getTarget();
            if (target == targetVertex)
                return i;
        }
```

```
        return -1;
    }

    @Override
    public void setNormalStyle() {
        graph.setCellStyle(MY_CUSTOM_EDGE_NORMAL_STYLE,
graph.getChildEdges(graph.getDefaultParent()));
        graph.setCellStyle(MY_CUSTOM_VERTEX_NORMAL_STYLE,
graph.getChildVertices(graph.getDefaultParent()));
    }

    @Override
    public void setStyleSelected(boolean flag, Object[] cells) {
        for (Object c : cells) {
            mxCell cell = (mxCell) c;
            if (flag) {
                if (cell.isVertex()) {
                    graph.setCellStyle(MY_CUSTOM_VERTEX_SELECTED_STYLE, new Object[]
{cell});
                } else if (cell.isEdge())
                    graph.setCellStyle(MY_CUSTOM_EDGE_SELECTED_STYLE, new Object[]{cell});
            } else {
                if (cell.isVertex())
                    graph.setCellStyle(MY_CUSTOM_VERTEX_NORMAL_STYLE, new Object[]
{cell});
                else if (cell.isEdge())
                    graph.setCellStyle(MY_CUSTOM_EDGE_NORMAL_STYLE, new Object[]{cell});
            }
        }
    }

    @Override
    public void setStyle(String style, Object[] cells) {
        graph.setCellStyle(style, cells);
    }

    @Override
    public void saveGraph(String fileName) throws IOException {
        try (ObjectOutputStream outputStream = new ObjectOutputStream(new
FileOutputStream(fileName))) {
            outputStream.writeObject(graph.getChildCells(parent));
        }

    }

    @Override
    public void loadGraph(String fileName) throws IOException, ClassNotFoundException {
        try (ObjectInputStream inputStream = new ObjectInputStream(new
FileInputStream(fileName))) {
            graph.removeCells(graph.getChildCells(parent));
            graph.addCells((Object[]) inputStream.readObject());
```

```
        }
    }

    @Override
    public Object getGraph() {
        return graph;
    }
}

package com.company.model;

import com.mxgraph.util.mxConstants;
import com.mxgraph.view.mxGraph;
import com.mxgraph.view.mxStylesheet;

import java.util.HashMap;
import java.util.Hashtable;
import java.util.Map;

import static com.company.Constants.StyleGraph.*;
import static com.company.Constants.StyleGraph.MY_CUSTOM_EDGE_SELECTED_STYLE;

class StyleManager {
    //Установить стиль для вершин в нормальном состоянии
    public static void initMyCustomVertexNormalStyle(mxGraph graph) {
        mxStylesheet stylesheet = graph.getStylesheet();
        Map<String, Object> vertexStyle = new Hashtable<>();
        vertexStyle.put(mxConstants.STYLE_FILLCOLOR, FILL_COLOR_NORMAL);
        vertexStyle.put(mxConstants.STYLE_FONTCOLOR, FONT_COLOR_NORMAL);
        vertexStyle.put(mxConstants.STYLE_STROKEWIDTH, STROKE_VERTEX_SIZE);
        vertexStyle.put(mxConstants.STYLE_STROKECOLOR, STROKE_COLOR_NORMAL);
        vertexStyle.put(mxConstants.STYLE_FONTSIZE, FONT_SIZE);
        vertexStyle.put(mxConstants.STYLE_PERIMETER, mxConstants.PERIMETER_ELLIPSE);
        vertexStyle.put(mxConstants.STYLE_FONTSTYLE, mxConstants.FONT_BOLD);
        vertexStyle.put(mxConstants.STYLE_SHAPE, mxConstants.SHAPE_ELLIPSE);
        vertexStyle.put(mxConstants.STYLE_VERTICAL_ALIGN, mxConstants.ALIGN_MIDDLE);
        stylesheet.putCellStyle(MY_CUSTOM_VERTEX_NORMAL_STYLE, vertexStyle);
    }

    //Установить стиль для ребер в нормальном состоянии
    public static void initMyCustomEdgeNormalStyle(mxGraph graph) {
        mxStylesheet stylesheet = graph.getStylesheet();
        Map<String, Object> edgeStyle = new HashMap<>();
        edgeStyle.put(mxConstants.STYLE_ROUNDED, true);
        edgeStyle.put(mxConstants.STYLE_SHAPE, mxConstants.SHAPE_CONNECTOR);
        edgeStyle.put(mxConstants.STYLE_ENDARROW, mxConstants.ARROW_CLASSIC);
        edgeStyle.put(mxConstants.STYLE_VERTICAL_ALIGN, mxConstants.ALIGN_MIDDLE);
        edgeStyle.put(mxConstants.STYLE_ALIGN, mxConstants.ALIGN_CENTER);
        edgeStyle.put(mxConstants.STYLE_STROKECOLOR, STROKE_COLOR_NORMAL);
        edgeStyle.put(mxConstants.STYLE_FILLCOLOR, STROKE_COLOR_NORMAL);
        edgeStyle.put(mxConstants.STYLE_STROKEWIDTH, STROKE_EDGE_SIZE);
```

```
        edgeStyle.put(mxConstants.STYLE_FONTCOLOR, FONT_COLOR_NORMAL);
        edgeStyle.put(mxConstants.STYLE_FONTSIZE, FONT_SIZE);
        edgeStyle.put(mxConstants.STYLE_FONTSTYLE, mxConstants.FONT_BOLD);
        edgeStyle.put(mxConstants.STYLE_LABEL_BACKGROUNDCOLOR,
FILL_COLOR_NORMAL);
        edgeStyle.put(mxConstants.STYLE_LABEL_BORDERCOLOR,
STROKE_COLOR_NORMAL);
        stylesheet.putCellStyle(MY_CUSTOM_EDGE_NORMAL_STYLE, edgeStyle);
    }

    //Установить стиль для вершин в выделеном состоянии
    public static void initMyCustomVertexSelectedStyle(mxGraph graph) {
        mxStylesheet stylesheet = graph.getStylesheet();
        Map<String, Object> vertexStyle = new Hashtable<>();
        vertexStyle.put(mxConstants.STYLE_FILLCOLOR, FILL_COLOR_SELECTED);
        vertexStyle.put(mxConstants.STYLE_FONTCOLOR, FONT_COLOR_SELECTED);
        vertexStyle.put(mxConstants.STYLE_STROKEWIDTH, STROKE_VERTEX_SIZE);
        vertexStyle.put(mxConstants.STYLE_STROKECOLOR, STROKE_COLOR_SELECTED);
        vertexStyle.put(mxConstants.STYLE_FONTSIZE, FONT_SIZE);
        vertexStyle.put(mxConstants.STYLE_PERIMETER, mxConstants.PERIMETER_ELLIPSE);
        vertexStyle.put(mxConstants.STYLE_FONTSTYLE, mxConstants.FONT_BOLD);
        vertexStyle.put(mxConstants.STYLE_SHAPE, mxConstants.SHAPE_ELLIPSE);
        vertexStyle.put(mxConstants.STYLE_VERTICAL_ALIGN, mxConstants.ALIGN_MIDDLE);
        stylesheet.putCellStyle(MY_CUSTOM_VERTEX_SELECTED_STYLE, vertexStyle);
    }

    //Установить стиль для ребер в выделеном состоянии
    public static void initMyCustomEdgeSelectedStyle(mxGraph graph) {
        mxStylesheet stylesheet = graph.getStylesheet();
        Map<String, Object> edgeStyle = new Hashtable<>();
        edgeStyle.put(mxConstants.STYLE_ROUNDED, true);
        edgeStyle.put(mxConstants.STYLE_SHAPE, mxConstants.SHAPE_CONNECTOR);
        edgeStyle.put(mxConstants.STYLE_ENDARROW, mxConstants.ARROW_CLASSIC);
        edgeStyle.put(mxConstants.STYLE_VERTICAL_ALIGN, mxConstants.ALIGN_MIDDLE);
        edgeStyle.put(mxConstants.STYLE_ALIGN, mxConstants.ALIGN_CENTER);
        edgeStyle.put(mxConstants.STYLE_STROKECOLOR, STROKE_COLOR_SELECTED);
        edgeStyle.put(mxConstants.STYLE_FILLCOLOR, STROKE_COLOR_SELECTED);
        edgeStyle.put(mxConstants.STYLE_STROKEWIDTH, STROKE_EDGE_SIZE);
        edgeStyle.put(mxConstants.STYLE_FONTCOLOR, FONT_COLOR_SELECTED);
        edgeStyle.put(mxConstants.STYLE_FONTSIZE, FONT_SIZE);
        edgeStyle.put(mxConstants.STYLE_FONTSTYLE, mxConstants.FONT_BOLD);
        edgeStyle.put(mxConstants.STYLE_LABEL_BACKGROUNDCOLOR,
FILL_COLOR_SELECTED);
        edgeStyle.put(mxConstants.STYLE_LABEL_BORDERCOLOR,
STROKE_COLOR_SELECTED);
        edgeStyle.put(mxConstants.STYLE_VERTICAL_LABEL_POSITION,
mxConstants.ALIGN_BOTTOM);
        stylesheet.putCellStyle(MY_CUSTOM_EDGE_SELECTED_STYLE, edgeStyle);
    }

    public static void initMyCustomCurrentVertexStyle(mxGraph graph) {
```

```
        mxStylesheet stylesheet = graph.getStylesheet();
        Map<String, Object> vertexStyle = new Hashtable<>();
        vertexStyle.put(mxConstants.STYLE_FILLCOLOR, GREEN_COLOR);
        vertexStyle.put(mxConstants.STYLE_FONTCOLOR, FONT_COLOR_SELECTED);
        vertexStyle.put(mxConstants.STYLE_STROKEWIDTH, STROKE_VERTEX_SIZE);
        vertexStyle.put(mxConstants.STYLE_STROKECOLOR, GREEN_COLOR);
        vertexStyle.put(mxConstants.STYLE_FONTSIZE, FONT_SIZE);
        vertexStyle.put(mxConstants.STYLE_PERIMETER, mxConstants.PERIMETER_ELLIPSE);
        vertexStyle.put(mxConstants.STYLE_FONTSTYLE, mxConstants.FONT_BOLD);
        vertexStyle.put(mxConstants.STYLE_SHAPE, mxConstants.SHAPE_ELLIPSE);
        vertexStyle.put(mxConstants.STYLE_VERTICAL_ALIGN, mxConstants.ALIGN_MIDDLE);
        stylesheet.putCellStyle(MY_CUSTOM_CURRENT_VERTEX_STYLE, vertexStyle);
    }

    public static void initMyCustomInQueueVertexStyle(mxGraph graph) {
        mxStylesheet stylesheet = graph.getStylesheet();
        Map<String, Object> vertexStyle = new Hashtable<>();
        vertexStyle.put(mxConstants.STYLE_FILLCOLOR, FILL_COLOR_NORMAL);
        vertexStyle.put(mxConstants.STYLE_FONTCOLOR, FONT_COLOR_SELECTED);
        vertexStyle.put(mxConstants.STYLE_STROKEWIDTH, "3");
        vertexStyle.put(mxConstants.STYLE_STROKECOLOR, STROKE_COLOR_SELECTED);
        vertexStyle.put(mxConstants.STYLE_FONTSIZE, FONT_SIZE);
        vertexStyle.put(mxConstants.STYLE_PERIMETER, mxConstants.PERIMETER_ELLIPSE);
        vertexStyle.put(mxConstants.STYLE_FONTSTYLE, mxConstants.FONT_BOLD);
        vertexStyle.put(mxConstants.STYLE_SHAPE, mxConstants.SHAPE_ELLIPSE);
        vertexStyle.put(mxConstants.STYLE_VERTICAL_ALIGN, mxConstants.ALIGN_MIDDLE);
        stylesheet.putCellStyle(MY_CUSTOM_IN_QUEUE_VERTEX_STYLE, vertexStyle);
    }
}
```

```
package com.company;

public interface Constants {

    interface Algorithms {
        String DIJKSTRA = "Алгоритм Дейкстры";
        String FORD_BELLMAN = "Алгоритм Форда-Беллмана";
        String ASTAR = "Алгоритм A*";
    }

    interface Size {
        double WIDTH_VERTEX = 40;
        double HEIGHT_VERTEX = 40;
        int MAX_SIZE_VERTEX_NAME = 15;
        int WIDTH = 1000;
        int HEIGHT = 800;
        int INTEND = 5;
    }

    interface StyleGraph {
```

```
        String MY_CUSTOM_VERTEX_NORMAL_STYLE =
"MY_CUSTOM_VERTEX_NORMAL_STYLE";
        String MY_CUSTOM_CURRENT_VERTEX_STYLE =
"MY_CUSTOM_CURRENT_VERTEX_STYLE";
        String MY_CUSTOM_IN_QUEUE_VERTEX_STYLE =
"MY_CUSTOM_IN_QUEUE_VERTEX_STYLE";
        String MY_CUSTOM_EDGE_NORMAL_STYLE =
"MY_CUSTOM_EDGE_NORMAL_STYLE";
        String MY_CUSTOM_VERTEX_SELECTED_STYLE =
"MY_CUSTOM_VERTEX_SELECTED_STYLE";
        String MY_CUSTOM_EDGE_SELECTED_STYLE =
"MY_CUSTOM_EDGE_SELECTED_STYLE";
        String STROKE_COLOR_NORMAL = "#cdc4d1";
        String FILL_COLOR_NORMAL = "#67adb9";
        String FONT_COLOR_NORMAL = "#fecb16";
        String STROKE_COLOR_SELECTED = "#fecb16";
        String FILL_COLOR_SELECTED = "#c66179";
        String FONT_COLOR_SELECTED = "#fecb16";
        String STROKE_VERTEX_SIZE = "1";
        String STROKE_EDGE_SIZE = "3";
        String FONT_SIZE = "14";
        String GREEN_COLOR = "#228B22";
    }

    interface NameButton{
        String SAVE = "Сохранить";
        String LOAD = "Загрузить";
        String MOVE = "Перемещение";
        String ADD_VERTEX = "Добавить вершину";
        String CONNECT_VERTEX = "Соединить вершины";
        String DELETE = "Удалить";
        String NEXT = "Вперед";
        String BACK = "Назад";
    }

    String SEPARATOR = "$";
}

package com.company;

import com.company.controller.GraphCreatorControllerImpl;
import com.company.model.GraphCreatorModel;
import com.company.model.GraphCreatorModelImpl;
import com.company.model.adapter.MxGraphAdapter;
import com.company.view.GraphCreatorViewImpl;

public class Main {

    public static void main(String[] args) {
        GraphCreatorModel graphCreatorModel = new GraphCreatorModelImpl();
```

```
        GraphCreatorControllerImpl graphCreatorController = new
GraphCreatorControllerImpl(graphCreatorModel,new MxGraphAdapter());
        new GraphCreatorViewImpl(graphCreatorController, graphCreatorModel);
    }
}
```