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

ОТЧЕТ

по учебной практике

Тема: Кратчайшие пути в графе. Алгоритм Дейкстры.

Студентка гр. 1304 _____ Чернякова В.А.

Студентка гр. 1304 _____ Ярусова Т.В.

Студент гр. 1303 _____ Беззубов Д.В.

Руководитель _____ Шестопалов Р.П.

Санкт-Петербург
2023

ЗАДАНИЕ
НА УЧЕБНУЮ ПРАКТИКУ

Студентка Чернякова В.А. группы 1304

Студентка Ярусова Т.В. группы 1304

Студент Беззубов Д.В. группы 1303

Тема практики: Кратчайшие пути в графе. Алгоритм Дейкстры.

Задание на практику:

Командная итеративная разработка визуализатора алгоритма на Kotlin с графическим интерфейсом.

Алгоритм: Дейкстры.

Сроки прохождения практики: 30.06.2020 – 13.07.2020

Дата сдачи отчета: 13.07.2020

Дата защиты отчета: 13.07.2020

Студентка	_____	Чернякова В.А.
Студентка	_____	Ярусова Т.В.
Студент	_____	Беззубов Д.В.
Руководитель	_____	Шестопалов Р.П.

АННОТАЦИЯ

Создание программы с поддержкой графического интерфейса для нахождения кратчайшего пути в графе с помощью алгоритма Дейкстры. Главное преимущество алгоритма Дейкстры – значительно низкая сложность, которая является почти линейной. Реализация алгоритма основана на использовании очереди с приоритетом для перехода в близлежащие вершины. Графический интерфейс дает возможность как пошагово отследить нахождение кратчайшего пути, так и сразу же получить результат. Предусмотрена возможность загрузки графа из файла, а также его создание в рабочей области по щелчку мыши. Приложение поддерживает обработку некорректно введенных данных пользователем - вывод диалогового окна с сообщением об ошибке, что предотвращает возможность некорректной работы. Программа написана на языке программирования Kotlin в среде разработки IntelliJ IDEA с использованием инструментария с графическим интерфейсом swing.

SUMMARY

Creating a program with GUI support for finding the shortest path in a graph using Dijkstra's algorithm. The main advantage of Dijkstra's algorithm is a significantly low complexity, which is almost linear. The implementation of the algorithm is based on using a priority queue to go to nearby vertices. The graphical interface makes it possible to track the finding of the shortest path step by step, and immediately get the result. It is possible to load a graph from a file, as well as create it in the workspace at the click of the mouse. The application supports the processing of incorrectly entered data by the user - the output of a dialog box with an error message, which prevents the possibility of incorrect operation. The program is written in the Kotlin programming language in the IntelliJ IDEA development environment using swing GUI toolkit.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Kotlin — это язык программирования, который позволяет писать программы для разных платформ. Сейчас на Kotlin пишут приложения для Android, кроссплатформенные и веб-приложения.

У языка Kotlin есть особенности, которые делают его привлекательным для разработчиков. Защита от ошибок: в Kotlin есть несколько инструментов, которые предотвращают случайные ошибки при написании кода. Поддержка ООП и функционального программирования: язык Kotlin умеет работать как с объектами, так и с функциями — в зависимости от того, что требуется для проекта или конкретной задачи. И другие особенности.

С помощью данного языка программирования также можно реализовывать алгоритмы поиска кратчайших путей в графе. Задача о поиске кратчайшего пути на графе может быть интерпретирована по-разному и применяться в различных областях, например, картографические сервисы, недетерминированные машины, сети дорог и так далее. Разнообразие задач определяет актуальность практики.

Алгоритм Дейкстры – один из способов решения задачи поиска кратчайшего пути в графе. Алгоритм работает за счёт переоценивания расстояния каждой вершины от начальной вершины. Используется «жадный» подход в том смысле, что находится следующее лучшее решение, в надежде, что конечный результат является лучшим решением для всей задачи.

Алгоритм Дейкстры имеет несколько реальных вариантов использования, некоторые из которых следующие: цифровые картографические сервисы в Google Maps, приложения для социальных сетей, телефонная сеть, IP-маршрутизация для поиска открытого кратчайшего пути, расписание рейсов, назначение файлового сервера, роботизированный путь.

Целью данной практики является овладение новым языком программирования, Kotlin, и реализация алгоритма поиска кратчайшего пути на графе Дейкстры с визуализацией.

Для достижения цели были поставлены следующие задачи:

- Изучение нового языка программирования Kotlin.
- Разработка графического интерфейса. Создание приложения с поддержкой методов визуализации графа, таких как создание и удаление вершин, ребер, отрисовка шагов алгоритма и результата.
- Реализация алгоритма Дейкстры с использованием очереди с приоритетом для перехода в близлежащие вершины.
- Разработка классов графического и физического представления графа. Первые классы необходимы для отображения элементов в приложении, вторые же для запуска алгоритма. Для удобного интерпретирования графа из одного представления в другое удобно использовать адаптер.

1. ТРЕБОВАНИЯ К ПРОГРАММЕ

1.1. Исходные требования к программе

1.1.1. Требования к вводу исходных данных

Алгоритм должен получать на вход взвешенный ориентированный граф с положительными весами ребер. Именем вершины может быть следующее: символ, строка, число. Данные могут задаваться, как и в рабочем пространстве программы в соответствии с нажатием соответствующих кнопок, так и в виде файла.

1.1.2. Требования к визуализации

Пользовательский интерфейс должен представлять собой диалоговое окно, которое содержит набор кнопок, предназначенных для управления программой пользователем.

Диалоговое окно должно состоять из:

- Рабочей области. Она предназначена для построения графа и отображения пошаговых действий в процессе работы алгоритма или его результата.
- Окна логирования. Позволяет отслеживать работу алгоритма на каждом шаге.
- Кнопки «Добавить вершину». Создание вершины графа в рабочей области после клика мышью. При создании пользователь задает вершине имя, соответствующее требованиям к входным данным программы.
- Кнопки «Удалить». Удаление выбранного щелчком мыши элемента графа: ребра или вершины с инцидентными ей ребрами.
- Кнопки «Соединить вершины». Проведение направленного ребра. Задаются параметры начальной и конечной вершины, а также вес ребра.

- Кнопки «Перемещение». Перемещение объектов мышью внутри рабочей области.
- Кнопки «Сохранить». Сохранение созданного в рабочей области графа.
- Кнопки «Загрузить». Загрузить в рабочую область граф.
- Кнопки «Старт». Запуск работы алгоритма Дейкстры на графе. Выбор стартовой вершины.
- Кнопки «Вперед». Отображение следующей итерации алгоритма.
- Кнопки «Назад». Отображение предыдущей итерации алгоритма.
- Кнопки «Старт». Запуск работы алгоритма на графе.
- Кнопки «Сброс». Очищение рабочей области.
- Кнопки «Запустить алгоритм». Выбор стартовой и конечной вершины перед стартом алгоритма.

1.2. Шаблон архитектуры

Для реализации приложения было принято решение использовать архитектуру MVC. Это позволяет разбить реализацию на следующие модули: модель, визуализация и контроллер. Модели соответствует представление графа, контроллер отвечает за взаимодействие пользователя с графическим интерфейсом и реализацию работы алгоритма, визуализация, соответственно, графический интерфейс. Соответствующая схема MVC представлена на рисунке 1.

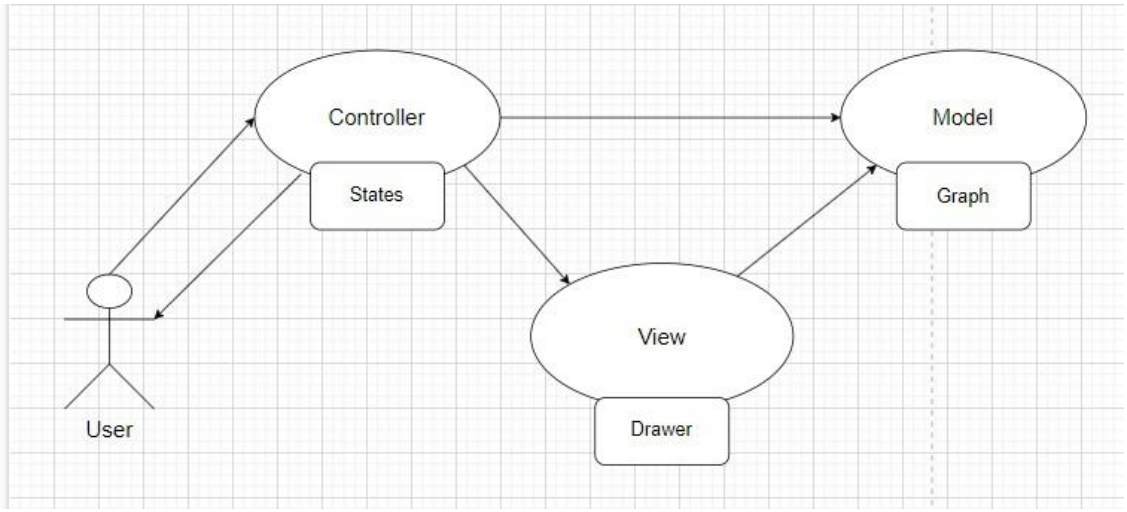

Рисунок 1 – схема архитектуры приложения.

На рисунке 2 представлена UML диаграмма с подробным описанием архитектуры MVC на основе классов, используемых в программе.

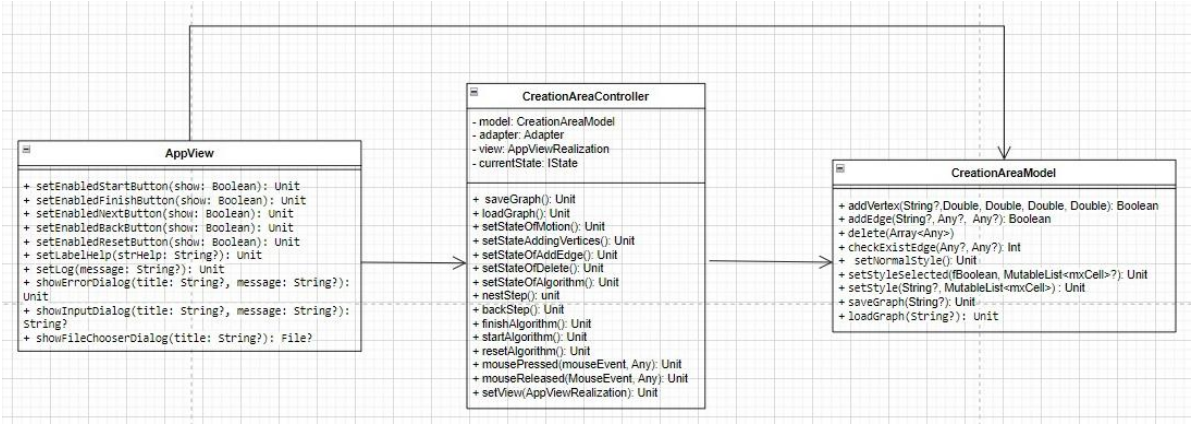

Рисунок 2 – UML диаграмма архитектуры MVC.

В зависимости от выбираемых пользователем действий, объект созданной программы меняет состояние. Для этого используется паттерн проектирования State. На рисунке 3 представлена UML работы приложения с графом на основе паттерна State.

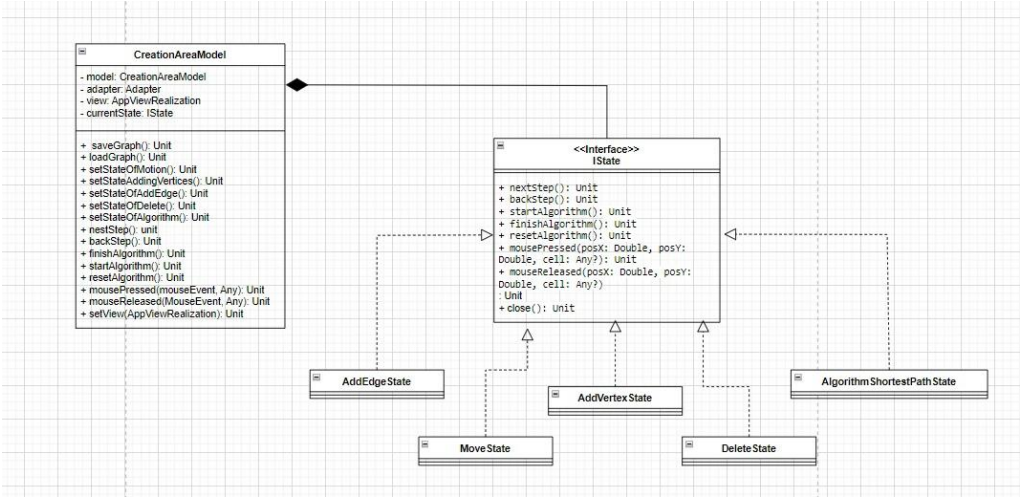

Рисунок 3 – UML диаграмма применения паттерна State.

Для отображения графа используется библиотека mxGraph. Представление графа таким образом тяжело совместимо с алгоритмом, для чего становится необходимым возможность переводить граф из одного типа в другой. Для данной цели используется паттерн проектирования Adapter. На рисунке 4 представлена UML диаграмма реализации алгоритма с использованием паттерна Adapter.

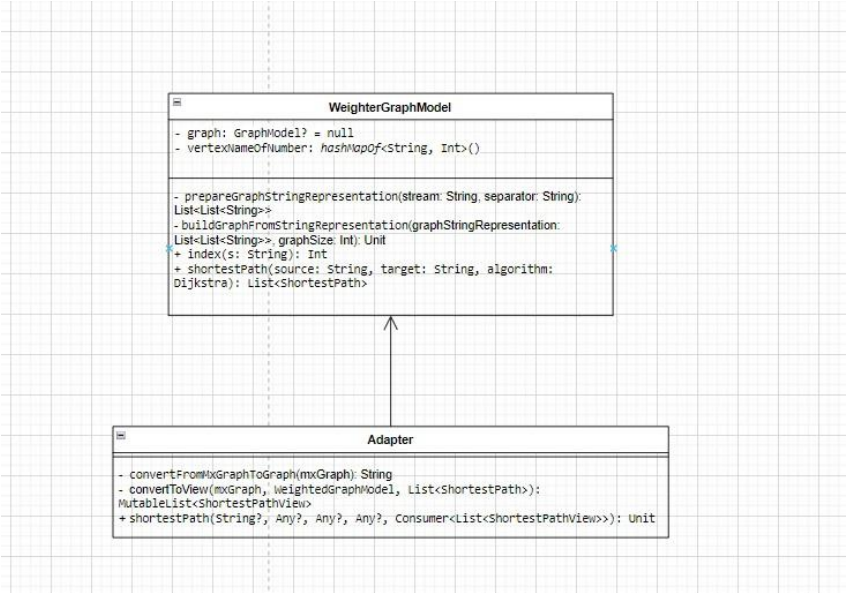

Рисунок 4 – UML диаграмма применения паттерна Adapter.

В работе реализован алгоритм Дейкстры для поиска кратчайших путей в графе. За основу взят следующий псевдокод:

```
func dijkstra(s):
    for v∈V
       d[v] = ∞
       used[v] = false
```

```
    d[s] = 0
    for i∈V
       v = null
       for j∈V    // найдем вершину с минимальным расстоянием
          if !used[j] and (v == null or d[j] < d[v])
             v = j
       if d[v] == ∞
          break
       used[v] = true
       for e : исходящие из v ребра   // произведем релаксацию по
всем ребрам, исходящим из v
          if d[v] + e.len < d[e.to]
             d[e.to] = d[v] + e.len
```

На рисунке 5 представлена диаграмма, отражающая сценарии взаимодействия пользователя с программой.

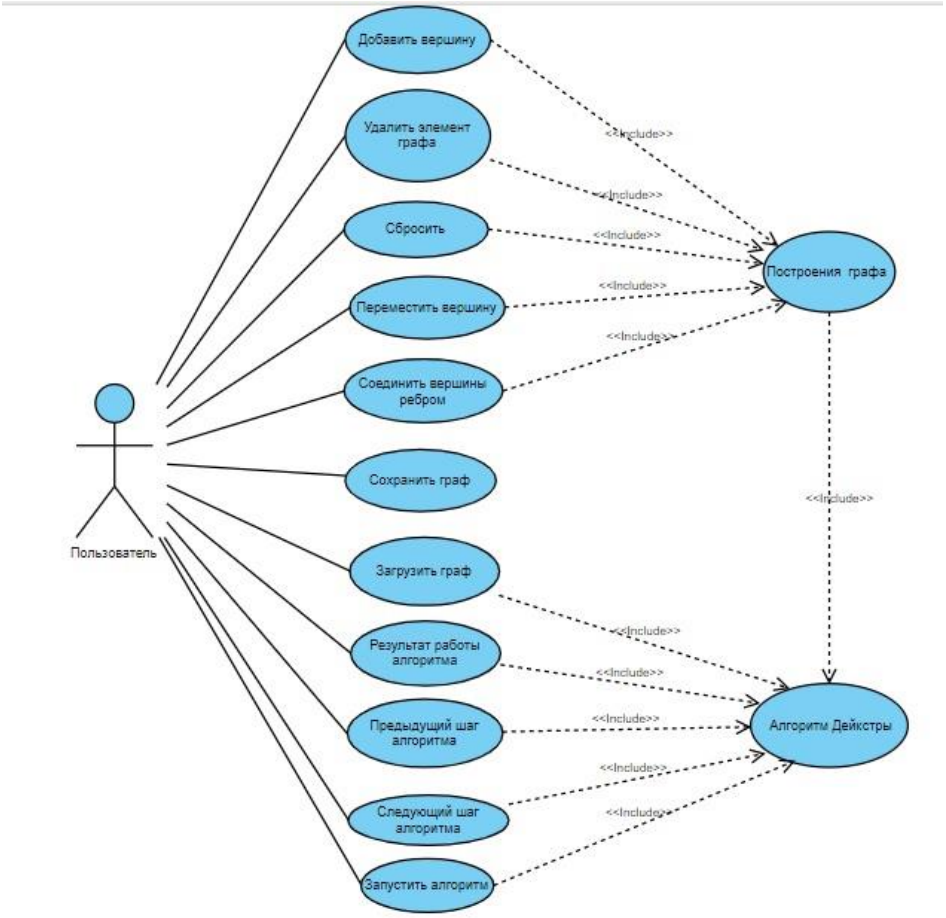

Рисунок 5 – user case диаграмма.

1.3. Требования к тестированию

1.3.1. Тестирование работы алгоритма

Проведение тестов, позволяющих проверить корректность работы алгоритма как на стандартных, так и на исключительных

случаях, таких как: отсутствие пути в графе, граф не является связным.

1.3.2. Тестирование визуализации

Разработка Unit-тестов, которые позволяют проверить корректность работы конкретных состояний, смены между состояниями, работу с графом, а именно добавление и удаление вершин, ребер.

2. ПЛАН РАЗРАБОТКИ И РАСПРЕДЕЛЕНИЕ РОЛЕЙ В БРИГАДЕ

2.1. План разработки

Для выполнения поставленных задач был прописан примерный план разработки проекта, представленный на таблице 1.

Таблица 1 - План разработки

№ п/п	Наименование работ	Срок выполнения
1	Создание спецификации и плана разработки. Построение UML диаграмм с архитектурными решениями.	01.07-03.07
2	Написание кода алгоритма, разработка обработчиков кнопок интерфейса.	03.07-05.07
3	Написание кода для демонстрации пошаговой работы алгоритма.	05.07-07.07
4	Исправление недочетов проекта.	07.07-10.07
5	Написание тестов к проекту.	10.07-12.07

2.2. Распределение ролей в бригаде

Беззубов Д.В. – архитектура программы.

Ярусова Т.В. – реализация алгоритма.

Чернякова В.А. – визуализация работы алгоритма.

3. ОСОБЕННОСТИ РЕАЛИЗАЦИИ

3.1. Архитектура программы

Архитектура приложения основана на наборе архитектурных идей и принципов для построения сложных систем с пользовательским интерфейсом MVC. Основная цель применения этой концепции состоит в отделении модели от её визуализации.

Для переключения между состояниями – действиями, совершаемыми над графом, используется паттерн проектирования State. Это поведенческий шаблон проектирования. Используется в тех случаях, когда во время выполнения программы объект должен менять своё поведение в зависимости от своего состояния. Основная идея в том, что программа может находиться в одном из нескольких состояний, которые всё время сменяют друг друга. Набор этих состояний, а также переходов между ними, предопределён и конечен. Находясь в разных состояниях, программа может по-разному реагировать на одни и те же события, которые происходят с ней.

Для графического отображения графа используется библиотека mxGraph. Для работы алгоритма граф должен быть представлен в виде списков вершин, ребер и так далее, так как это упрощает работу алгоритма. Для удобного перехода между двумя способами хранения графа в программе, используется паттерн Adapter. Данный паттерн предназначен для преобразования интерфейса одного класса в интерфейс другого. Благодаря реализации данного паттерна можно использовать вместе классы с несовместимыми интерфейсами.

Для поддерживания приложением возможности пошагового отображения алгоритма, необходимо сохранять внутренне состояние объекта на каждом шаге алгоритма. Для этого используется поведенческий шаблон проектирования Memento. Данный паттерн позволяет сохранять и восстанавливать прошлые состояния объектов, не раскрывая подробностей их реализации.

3.2. Структуры данных

Чтобы реализовать алгоритм Дейкстры были созданы следующие структуры данных.

- Класс *DirectedEdge*. Хранение информации о конкретном ребре, а именно стартовая вершина, финишная вершина и вес ребра между.

Поля класса:

val from: Int. Индекс стартовой вершины.

val to: Int. Индекс финишной вершины.

val weight: Double. Вес ребра между вершинами.

Методы класса:

override fun toString(): String. Переопределение функции приведение к строковому типу для корректного использования внутри алгоритма.

- Класс *GraphModel*. Хранение информации о количестве вершин в графе и всех ребрах.

Поля класса:

private val vertexCount: Int. Количество вершин в графе.

private val edges = mutableListOf<MutableSet<DirectedEdge>>(). Список всех ребер в графе.

Методы класса:

fun getVertexCount(): Int. Получение количества вершин в графе.

fun addEdge(edge: DirectedEdge). Добавление нового ребра в граф.

fun getEdgesForVertex(vertex: Int): Iterable<DirectedEdge>. Получение смежных ребер с входной вершиной.

- Класс *Vertexes*. Создан для работы с очередью приоритетов. Хранение следующей пары – ключ, расстояние от стартовой вершины, значение, индекс вершины в графе.

Поля класса:

private var key = 0.0. Хранение метки до текущей вершины от стартовой.

var value: Int. Индекс вершины в графе.

Методы класса:

override operator fun compareTo(other: Vertexes?): Int. Перегрузка сравнения.

override fun equals(other: Any?): Boolean. Проверка на равенство.

override fun hashCode(): Int. Перевод вершины в хэш-код.

override fun toString(): String. Перегрузка метода преобразования в строковый тип.

- Класс *WeightedGraphModel*. Хранение информации об обрабатываемом графа в виде специальной модели для корректной работы алгоритма.

Поля класса:

private var graph: GraphModel? = null. Объект класса GraphModel.

private var vertexNameOfNumber = hashMapOf<String, Int>(). Сопоставление по индексу названия вершины в графе.

Методы класса:

private fun prepareGraphStringRepresentation(stream: String, separator: String): List<List<String>>. Представление обрабатываемого графа в виде строки для создания модели для работы алгоритма.

private fun buildGraphFromStringRepresentation(graphStringRepresentation: List<List<String>>, graphSize: Int). Создание модели графа по полученной в методе выше строке.

fun index(s: String): Int. Получение индекса вершины в графе по названию.

fun name(vertex: Int): String. Получение имени вершины в графе по индексу.

fun shortestPath(source: String, target: String, algorithm: Dijkstra): List<ShortestPath>. Запуск работы алгоритма на данном графе.

3.3. Алгоритм

- Класс *Dijkstra*. Работа алгоритма Дейкстры.

Поля класса:

private var edgeTo = MutableList<DirectedEdge?>. Список обработанных ребер, включенных в путь.

private var distTo = MutableList<Double>. Список текущих расстояний до вершин.

private var visitedVertex = MutableList<Boolean>. Обработанные вершины.

private val priorityQueue = PriorityQueue<Vertexes>(). Приоритетная очередь

private var steps = mutableListOf<ShortestPath>(). Список состояний программы.

private val digraph = graphModel. Модель графа. Данный класс описан раннее.

Методы класса:

private fun relaxation(graph: GraphModel, vertex: Int). Пересчитывание меток вершин. Производится релаксация.

fun buildWay(graph: GraphModel, source: Int): MutableList<ShortestPath>. Построение кратчайших расстояний до вершин от стартовой.

fun hasPathTo(v: Int): Boolean. Проверка на существование пути.

fun pathTo(v: Int): Iterable<DirectedEdge>. Составление пути, проверка существования которого осуществляется в методе выше.

- Класс *ShortestPath*. Сохранение текущего состояния алгоритма. Реализация паттерна Memento.

Поля класса:

private val currentVertex: Int. Текущая вершина.

private val processedVertices: MutableList<Boolean>. Пометка обработанных на данном шаге вершин.

private val currentWays: MutableList<DirectedEdge?>. Текущие рассматриваемые ребра.

private val inQueueVertices: PriorityQueue<Vertexes>. Копия состояния очереди на момент вызова.

Методы класса:

fun getCurrentVertex(digraph: WeightedGraphModel, separator: String): String. Получение имени текущей вершины в строковом виде.

fun getProcessedVertices(digraph: WeightedGraphModel, separator: String?): String. Получение обрабатываемых вершин на текущем шаге.

fun getCurrentWays(digraph: WeightedGraphModel): String. Получение путей, найденных на данном шаге алгоритма.

fun getInQueueVertices(digraph: WeightedGraphModel, separator: String?): String. Получение состояния очереди на текущем шаге алгоритма.

3.4. Контроллер

- Класс *CreationAreaController*. Связь между программой и графическим интерфейсом – действиями, совершаемыми пользователем.

Поля класса:

private var model: CreationAreaModel. Модель графа в виде mxGraph.

private var adapter: Adapter. Адаптер для перевода графа из хранения вида mxGraph в удобный для использования алгоритма.

private var view: AppViewRealization? = null. Класс пользовательский интерфейс.

private var currentState: IState? = null. Текущие состояние программы.

Методы класса:

fun saveGraph(). Сохранение графа в файл.

fun loadGraph(). Загрузка графа из файла.

fun setStateOfMotion(). Перемещение объектов графа.

fun setStateAddingVertices(). Добавление вершин.

fun setStateOfAddEdge(). Добавление ребер.

fun setStateOfDelete(). Удаление элементов графа.

fun setStateOfAlgorithm(). Запуск алгоритма. Выбор вершин для поиска пути.

fun nextStep(). Следующий шаг алгоритма.

fun backStep(). Предыдущий шаг алгоритма.

fun startAlgorithm(). Работа алгоритма на графе.

fun finishAlgorithm(). Завершение алгоритма, получение результата.

fun resetAlgorithm(). Сброс алгоритма на графе.

fun mousePressed(mouseEvent: MouseEvent?, cell: Any?). Щелчок мыши в рабочей области.

fun mouseReleased(mouseEvent: MouseEvent?, cell: Any?). Отсутствие щелчка – мышь отпущена.

fun setView(view: AppViewRealization?). Отслеживание действий пользователя в графическом интерфейсе.

3.5. Модель

- Класс *Adapter*. Адаптация графа, создаваемого пользователем в формате mxGraph, к способу хранения, используемому при реализации алгоритма Дейкстры.

Методы класса:

private fun convertFromMxGraphToGraph(graph: mxGraph): String. Хранение графа в виде строки при переводе из mxGraph.

private fun convertFromGraphToMxGraph(graph: mxGraph, subgraph: String): MutableList<mxCell>. Хранение графа в виде mxGraph при переводе из строки.

private fun convertToView(graph: mxGraph, digraph: WeightedGraphModel, mementos: List<ShortestPath>). Отображение на экране промежуточных шагов алгоритма.

fun shortestPath(currentAlgorithm: String?, graph: Any?, s: Any?, t: Any?, callbackEnd: Consumer<List<ShortestPathView>>). Получение итого пути в виде mxGraph.

- Класс *ShortestPathView*. Отображение конкретного шага алгоритма на основе паттерна Memento.

Поля класса:

private val currentVertex: Any?. Текущая вершина.

private val processedVertices: MutableList<mxCell>?. Вершины, обрабатываемые на данном шаге.

private val currentWays: MutableList<mxCell>?. Пути, найденные на текущем шаге алгоритма.

private val inQueueVertices: MutableList<mxCell>?. Состояние очереди на текущем шаге.

private val answer: MutableList<mxCell>?. Состояние ответа, кратчайшего пути, на текущем шаге.

private val log: String?. Сообщение логирующего окна на текущем шаге.

Методы класса:

fun getCurrentVertex(): Any?.

fun getProcessedVertices(): MutableList<mxCell>?.

fun getCurrentWays(): MutableList<mxCell>?.

fun getInQueueVertices(): MutableList<mxCell>?.

fun getAnswer(): MutableList<mxCell>?.

fun getLog(): String?.

Так как поля класса объявлены с спецификатором доступа *private,* были написаны соответствующие геттеры.

- Класс *CreationAreaModel*. Модель графа на графическом интерфейсе, с которым взаимодействует пользователь.

Поля класса:

val graph: mxGraph. Граф в виде объекта типа mxGraph.

private val parent: Any. Родители вершин.

Методы класса:

fun addVertex(name: String?, posX: Double, posY: Double, width: Double, height: Double): Boolean. Добавление новой вершины графа, если название не повторяется.

fun addEdge(weight: String?, vertexFirst: Any?, vertexSecond: Any?): Boolean. Добавление ребра с допустимым весом.

fun delete(cells: Array<Any>). Удаление элементов графа.

fun checkExistEdge(s: Any?, t: Any?): Int. Проверка на существования ребра.

fun setNormalStyle(). Установка стандартного стиля вершин и ребер.

fun setStyleSelected(flag: Boolean, cells: MutableList<mxCell>?). Установка стиля для выбранных объектов.

fun setStyle(style: String?, cells: MutableList<mxCell>). Добавления стиля.

fun saveGraph(fileName: String?). Сохранение графа. Сереализация всех элементов графа.

fun loadGraph(fileName: String?). Загрузка графа в файл. Десеарилизация всех элементов графа.

3.6. Состояния

- Интерфейс *IState*. Состояние редактора, которому делегируются вызовы методов контроллера.

Методы класса:

fun nextStep(). Состояние – следующий шаг алгоритма.

fun backStep(). Состояние – предыдущий шаг алгоритма.

fun startAlgorithm(). Состояние – запуск алгоритма.

fun finishAlgorithm(). Состояние – завершение работы алгоритма.

fun resetAlgorithm(). Состояние – сброс алгоритма.

fun mousePressed(posX: Double, posY: Double, cell: Any?). Состояние – щелчок мыши.

fun mouseReleased(posX: Double, posY: Double, cell: Any?) Состояние – отсутствие щелчка мыши.

val status: String?. Название состояния.

fun close(). Завершение состояния.

В следующих классах в соответствии с контролируемым ими состоянием переопределены те или иные методы интерфейса.

- *AddEdgeState().* Добавление ребра.
- *AddVertexState().* Добавление вершины.
- *AlgorithmShortestPathState().* Запуск алгоритма.
- *DeleteState().* Удаление элементов графа.
- *MoveState().* Перемещение элементов графа.

3.7. Графический интерфейс

- Интерфейс *AppView*. Интерфейс соответствующий UI. Все методы переопределены в классе, который описан далее.
- Класс *AppViewRealization*. Внешний вид графического интерфейса.

Поля класса:

private val controller: CreationAreaController. Объект соответствующего класса.

private val model: CreationAreaModel. Объект соответствующего класса.

private var topBar: TopAppBar? = null. Объект соответствующего класса.

private var bottomBar: BottomAppBar? = null. Объект соответствующего класса.

private var textArea: JTextArea? = null. Объект соответствующего класса.

private val frame = JFrame("Поиск кратчайшего пути в графе. Алгоритм Дейсктры."). Объект соответствующего класса.

Методы класса:

private fun initGUI(). Инициализация графического интерфейса. Создание объектов, отображаемых на экране классов.

override fun setEnabledStartButton(show: Boolean), override fun setEnabledFinishButton(show: Boolean), override fun setEnabledNextButton(show: Boolean), override fun setEnabledBackButton(show: Boolean), override fun setEnabledResetButton(show: Boolean) – установка соответствующих кнопок.

override fun setLabelHelp(strHelp: String?). Значение строки вспомогательной панели.

override fun setLog(message: String?). Значение текста логирующего окна.

override fun showErrorDialog(title: String?, message: String?). Создание окна с информацией об ошибке.

override fun showInputDialog(title: String?, message: String?): String?. Создание окна, считывающего информацию с клавиатуры.

override fun showFileChooserDialog(title: String?): File?. Создание диалогового окна с возможностью выбора файла.

- Класс BottomAppBar. Нижняя панель графического интерфейса.

Поля класса:

private val controller: CreationAreaController. Объект соответствующего класса.

private var backStepButton: JButton, nextStepButton: JButton, private var startButton: JButton, private var finishButton, private var resetButton, private var ranAlgorithm. Кнопки нижней панели.

Методы класса:

private fun initGUI(). Создание области нижней панели.

private fun initButtonControlAlgorithms(). Создание кнопок, отвечающих за алгоритм.

private fun initStartAlgorithm(). Создание области с кнопкой запуска алгоритма.

private fun createButton(name: String, listener: ActionListener, container: MutableList<JButton>, enabled: Boolean): JButton. Создание кнопки.

private fun createRanButton(listener: ActionListener, container: MutableList<JButton>, enabled: Boolean): JButton. Создание кнопки запуска алгоритма.

Для каждой кнопки также создан метод, в зависимости от действий пользователя устанавливающий значение параметра доступности *isEnabled*. Так, пока кнопка запуска алгоритма не нажата, кнопки действия с алгоритмом не доступны.

- Класс *TopAppBar*. Верхняя панель графического интерфейса.

Поля класса:

private val controller: CreationAreaController. Объект соответствующего класса.

private var labelHelp: JLabel. Объект соответствующего класса. Панель подсказок.

private var moveButton: JButton, private var addVertexButton: JButton, private var connectVertexButton: JButton, private var deleteButton: JButton. Кнопки верхней панели.

Методы класса:

private fun initGUI(). Создание верхней панели в окне приложения.

private fun initButtons(). Инициализация кнопок панели.

private fun initLabelHelp(). Создание панели подсказок.

private fun createButton(name: String, listener: ActionListener, container: MutableList<JButton>, enabled: Boolean): JButton. Создание кнопки.

fun setLabelHelp(text: String?). Установка текста в окно подсказок.

- Класс *CreationArea.* Создание рабочей области, где редактируется граф.

Методы класса:

private fun initGUI(). Создание рабочей области для редактирования и создания графа.

На рисунке 6 представлен пользовательский интерфейс, получаемый при создании соответствующих объектов описанных классов.

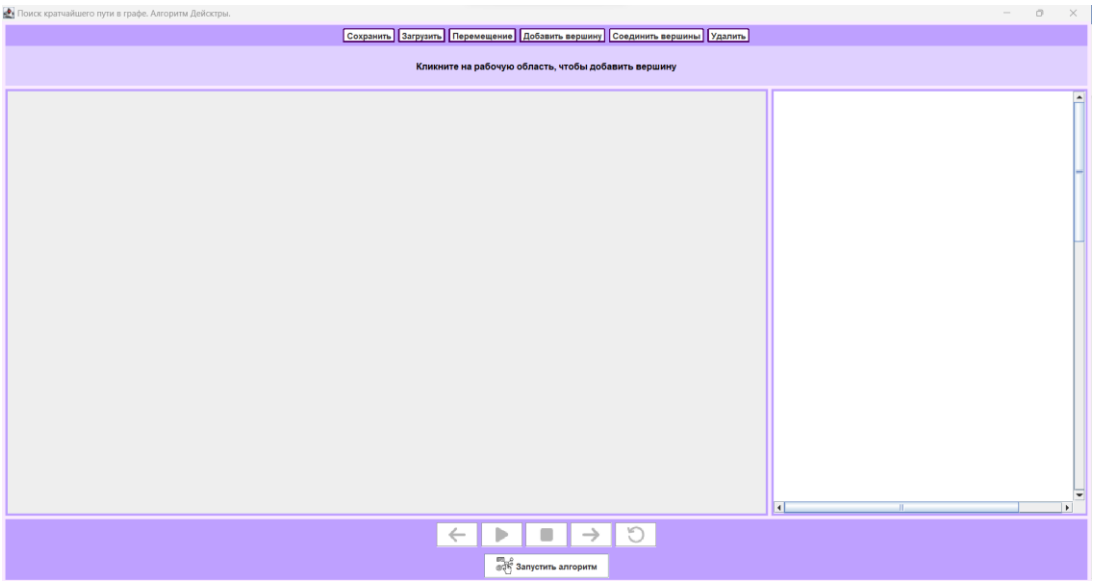

Рисунок 6 – графический интерфейс программы.

3.8. Специальные объекты

- *StyleManager*. Внутри созданы методы, отвечающие за стиль графа: *customVertexNormalStyle, customEdgeNormalStyle, customVertexSelectedStyle, customEdgeSelectedStyle, customCurrentVertexStyle, customInQueueVertexStyle*.
- *Constants*. Созданы различные константы, вызываемые в методах объекта, описанного выше. Такие как высота и ширина вершины, цвет обычный и при выборе и тому подобное.

4. ТЕСТИРОВАНИЕ

4.1. Тестирование работы алгоритма

Проверена работа алгоритма на стандартном случае – задан взвешенный ориентированный граф. Результат теста представлен на рисунке 7.

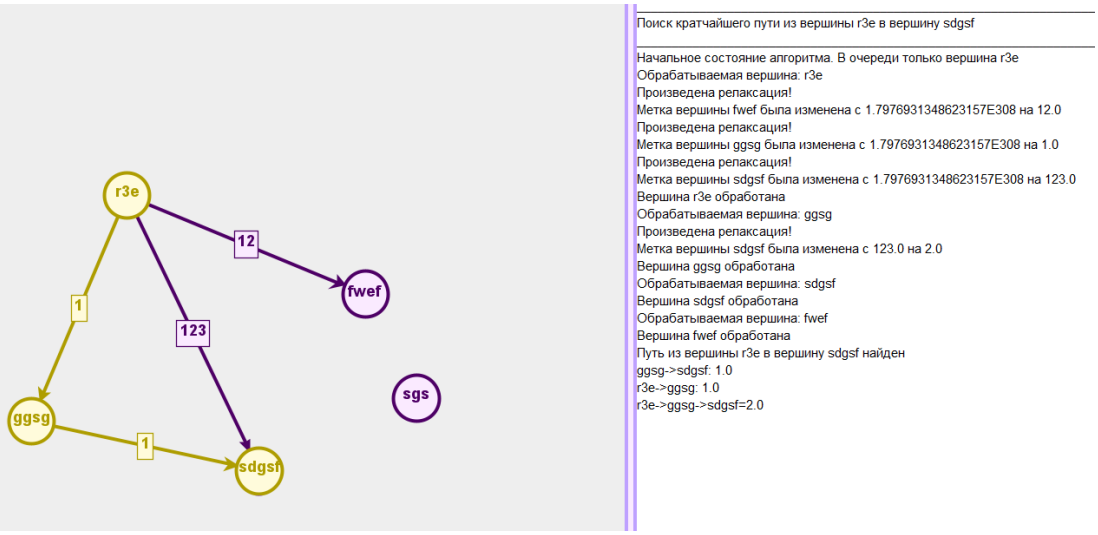

Рисунок 7 – тестирование работы алгоритма на стандартном графе.

Работа алгоритма в данном случае корректна.

Проверка работы алгоритма на графе, где путь отсутствует. Результат теста представлен на рисунке 8.

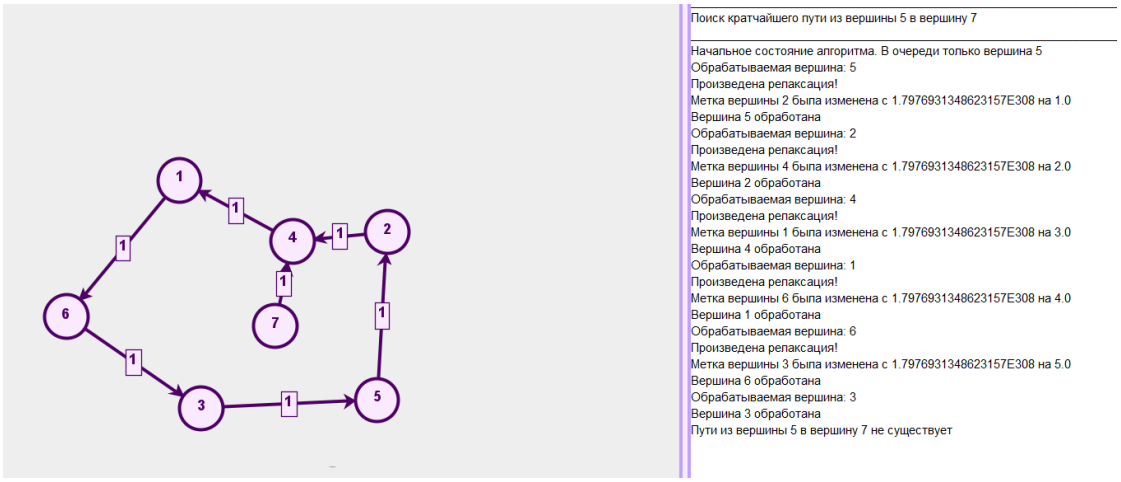

Рисунок 8 – тестирование работы алгоритма на графе из одной вершины.

Работа алгоритма в данном случае корректна.

Проверка работы алгоритма на несвязном графе. Результат теста представлен на рисунке 9.

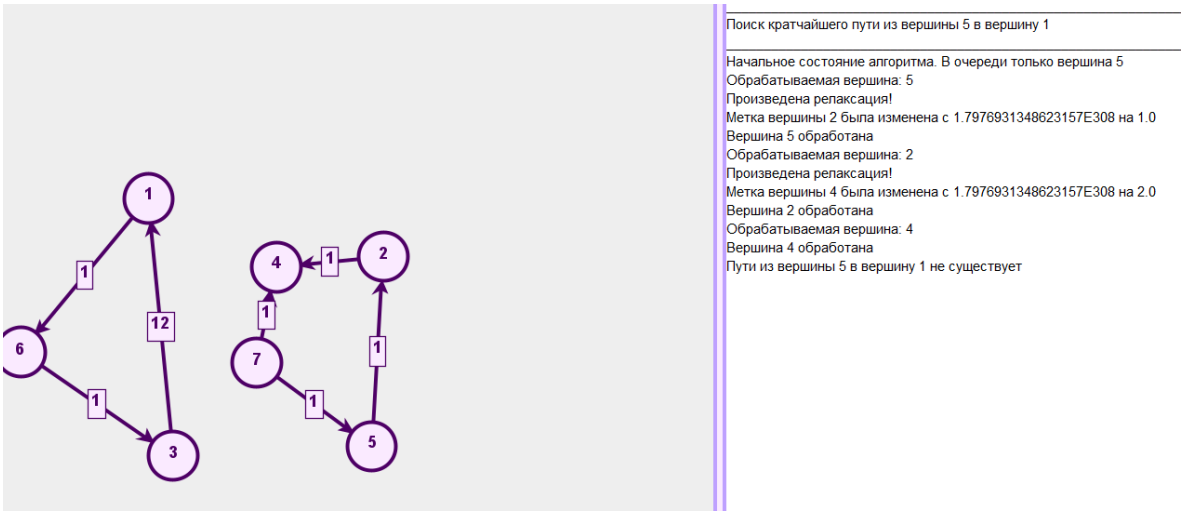

Рисунок 9 – тестирование работы алгоритма на графе из одной вершины.

Работа алгоритма в данном случае корректна.

4.2. Тестирование визуализации

Созданы Unit-тесты, позволяющие проверить корректность визуализации работы программы.

- Класс *CreationAreaControllerTest.* Тестирование состояний: перемещение, удаление, добавить вершину, соединить вершины. Создается экземпляр конкретного *State* и происходит сравнение поля *status: String*, хранящего текущее состояние, экземпляра класса Status со значением, которое должно быть у строки. Результат тестирования представлен на рисунке 10.

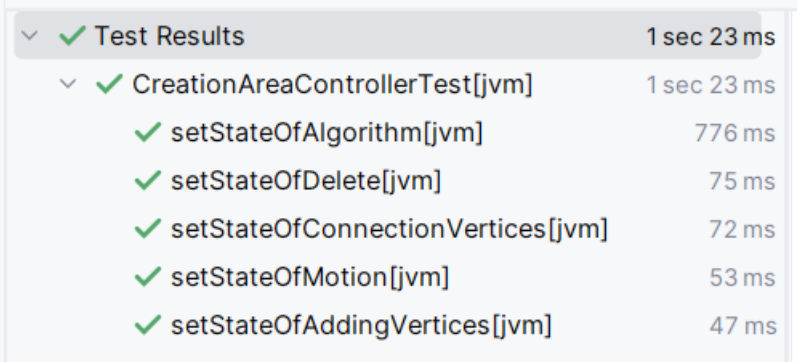

Рисунок 10 – тестирование состояний.

- Класс *CreationAreaModelTest.* Проверка работы с графом: добавить вершину, добавить ребро, удалить элемент графа. Проверка на наличие в графе создаваемой вершины, ребра, проверка на отсутствие элемента графа, ребра или вершины, при удалении. Результат тестирования представлен на рисунке 11.

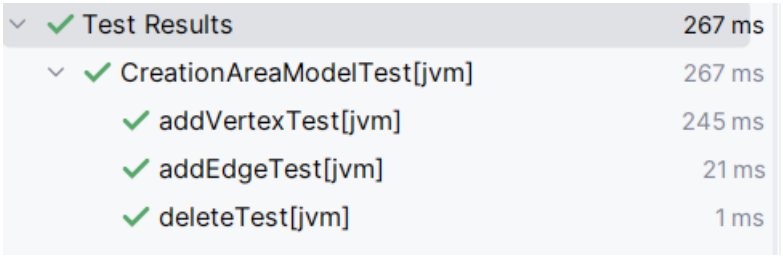

Рисунок 11 – тестирование работы с графом.

- Класс *CreationAreaViewTest.* Проверка смены состояний при нажатии на них. Метод *doClick()* инициирует нажатие соответствующей кнопки, с помощью методов классов получается значение области подсказок, так как значение текста зависит от нажимаемой кнопки, и сравнивается с корректным. Результат тестирования представлен на рисунке 12.

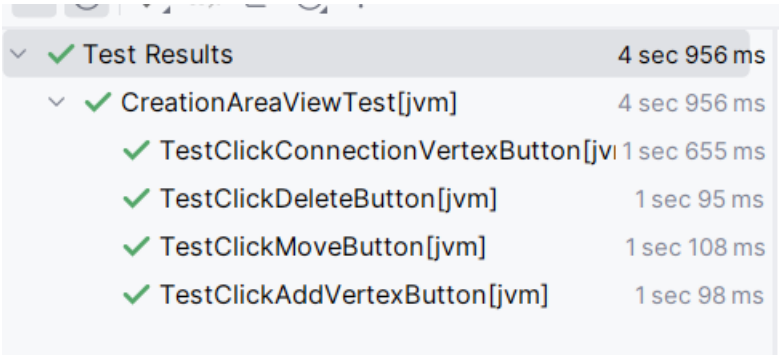

Рисунок 12 – тестирование смены кнопок состояния по щелчку.

ЗАКЛЮЧЕНИЕ

В ходе выполнения задания учебной практики были изучены основы программирования на новом языке Kotlin. Изучена библиотека swing для создания графически интерфейсов на данном языке. На основе полученных знаний было разработано приложение с графическим интерфейсом, реализующее алгоритм поиска кратчайшего пути в графе – алгоритм Дейкстры.

Были разработаны требования к входным данным, чтобы любой пользователь мог использовать созданное приложение правильно. Учитывая требования к визуализации, создан понятный и доступный графический интерфейс.

Заявленный шаблон архитектуры делает программу удовлетворяющей основным принципам ООП.

Таким образом, итоговая программа соответствует требованиям, предъявленным в начале работы.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Репозиторий бригады. URL: DaniiLBez/stud_practice: the program for visualizing the work of the Prima algorithm on Kotlin (github.com)
2. Руководство по языку Kotlin. URL: Kotlin | Руководство (metanit.com).
3. Библиотека swing. URL: Библиотека Swing (java-online.ru).
4. Алгоритм Дейкстры. URL: Алгоритм Дейкстры — Викиконспекты (ifmo.ru).
5. Основы паттернов проектирования. URL: Основы паттернов проектирования | C# и .NET (metanit.com).
6. Томас Х. Кормен, Чарльз И. Лейзерсон, Рональд Л. Ривест, Клиффорд Штайн Алгоритмы: построение и анализ — 2-е изд. — М.: «Вильямс», 2007.