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

ОТЧЕТ

по учебной практике

Тема: Реализация модифицированного алгоритма Краскала на языке Kotlin с визуализацией

Студент гр. 2303 _____ Ильин Е.С.

Студент гр. 2303 _____ Ламашовский Д.В.

Студент гр. 2303 _____ Ефремова А.В.

Руководитель _____ Шестопалов Р.П.

Санкт-Петербург
2024

ЗАДАНИЕ

НА УЧЕБНУЮ ПРАКТИКУ

Студент Ильин Е.С. группы 2303

Студент Ламашовский Д.В. группы 2303

Студент Ефремова А.В. группы 2303

Тема практики: Командная итеративная разработка визуализатора алгоритма на Kotlin с графическим интерфейсом

Задание на практику:

Командная итеративная разработка визуализатора алгоритма на Kotlin с графическим интерфейсом.

Алгоритм: модифицированный алгоритм Краскала

Сроки прохождения практики: 26.06.2024 – 09.07.2024

Дата сдачи отчета: 05.07.2024

Дата защиты отчета: 05.07.2024

Студент _____ Ильин Е.С.

Студент _____ Ламашовский Д.В.

Студент _____ Ефремова А.В.

Руководитель _____ Шестопалов Р.П.

АННОТАЦИЯ

Целью проекта является получение навыков программирования на Kotlin и реализация алгоритма по поиску минимального остовного дерева во взвешенном графе с визуализацией.

SUMMARY

The aim of the project is to acquire programming skills in Kotlin and implement an algorithm for finding a minimum spanning tree in a weighted graph with visualization.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Основная цель работы – реализация модифицированного алгоритма Краскала для поиска минимального остовного дерева взвешенного графа и представить его в виде приложения с графическим интерфейсом. Для написания алгоритма необходимы структуры данных для представления графа. Используя информацию о рёбрах, хранящуюся в графе, создаём отсортированный по весу список рёбер с инцидентными им вершинами, которые будут рассмотрены по ходу алгоритма. На каждой итерации алгоритма выбираем первое в списке нерассмотренное ребро и сравниваем цвета инцидентных ему вершин (назовём эти вершины A и B): если цвета A и B одинаковые, пропускаем это ребро, если разные, то добавляем данное ребро в МОД и перекрашиваем все вершины того же цвета что и A в цвет вершины B. Данный алгоритм позволяет избегать проверки на наличие циклов. Ниже представлен псевдокод алгоритма:

```
fun findMSTWithKruskal(G: Graph): List<Edge> {
    G.vertices.forEach { it.color = getColorFromIndex(it.index) } // Присваиваем различные цвета вершинам

    val result: ArrayList<Edge> = arrayListOf()
    val edgeList = G.edges.sortedBy { it.weight } // Создаём отсортированный по весу список рёбер
    var i = 0
    while (i < edgeList.size) {
        val edge = edgeList[i] // Берём очередное ребро
        if (edge.first.color != edge.second.color) { // Если вершины на концах ребра разного цвета
            G.replaceVertexColor(oldColor = edge.first.color, newColor = edge.second.color) // Меняем цвет всех вершин одного из цветов на концах ребра в цвет вершины на другом конце ребра
            result.add(edge) // Добавляем в МОД
        }

        i += 1
    }

    return edgeList
}
```

1. ТРЕБОВАНИЯ К ПРОГРАММЕ

1.1. Исходные Требования к программе

1.1.1 – Требования к вводу исходных данных

Пользователю предоставляется несколько способов введения исходных данных:

1) Загрузка графа из файла в одном из поддерживаемых форматов:

1.1) Файл .tgf (Trivial Graph Format)

1.2) Файл .gml (GML - язык описания графов)

2) Создание графа непосредственно в среде приложения на холсте. Для этого сначала надо включить редактирование. Возможности при редактировании графа:

2.1) По холсту можно перемещаться: для приближения / отдаления используется колёсико мыши, для перемещения по холсту надо зажать ЛКМ по пустому месту и двигать ей.

2.2) Нажатие ЛКМ по пустому месту – создание новой вершины. При этом есть возможность дать вершине имя либо оставить имя, сгенерированное по-умолчанию.

2.3) Нажатие ПКМ по вершине позволяет удалить её вместе со всеми её рёбрами.

2.4) Зажав вершину ЛКМ, вы можете перемещать её по холсту.

2.5) Нажатие на вершину ЛКМ с зажатым Shift позволяет переименовать её.

2.6) Нажав на вершину ЛКМ, её можно выделить. Это позволяет производить операции над рёбрами:

2.6.1) Нажатие ЛКМ на другую вершину приводит к созданию нового ребра. При этом можно задать ребру вес в виде целого числа либо оставить 1 по-умолчанию

2.6.2) Нажатие ПКМ на другую вершину позволяет удалить ребро между ними.

2.6.3) Нажатие ЛКМ на другую вершину с зажатым Shift позволяет изменить вес ребра между ними.

2.7) Нажатие на кнопку B (английскую) - возврат к изначальному виду (убирает зум и сдвиг на поле).

3) Помимо редактирования на холсте, возможно использовать текстовый ввод (в котором можно вводить последовательность команд для редактирования графа, например, очистка графа, вставка вершины / ребра и т.д.).

1.1.2 – Требования к визуализации

Текстовое описание: Графический интерфейс программы содержит следующие составляющие:

В верхней части расположено меню:

1) Вкладка “File” - содержит кнопки “Load graph”, “Save graph”, “Exit”
2) Вкладка “Mode” - содержит радиокнопки “Edit mode” и “Algorithm mode”
3) Вкладка “Options” - кнопки “Graph render options” и “Algorithm options”
4) Вкладка “Info” - кнопки “Guide” и “About program”.

Остальное меню меняется в зависимости от режима:

1) Режим редактирования:

1.1) В холсте (Graph view) граф с возможностью редактирования в соответствии с пунктом 2 описания входных данных (помимо прочего, возможно приближать/отдалять граф и перемещаться по полю, зажав пустое место и двигая мышь).

1.2) Текстовое окошко (Console) работает в режиме условной консоли, в которую можно вводить команды для редактирования графа (подобно следующим: add node A, del edge A-B, clear (для очистки графа), и т.д.).

Справа от холста набор кнопок:

1.3) Кнопка для очистки графа (C).

1.4) Кнопка для вставки шаблонного графа (I) (открывает окошко в котором выбирается тип графа и его характеристики, например, полный и нуль-граф и их размер).

1.5) Кнопка для авторасположения графа (A) (программа автоматически распределяет граф, как если бы он был загружен из файла TGF без указания позиций вершин).

1.6) Кнопка для получения информации о графе (?) (например, число вершин, число ребер, число компонент связности).

Схема интерфейс в режиме редактирования представлен на рисунке 1:

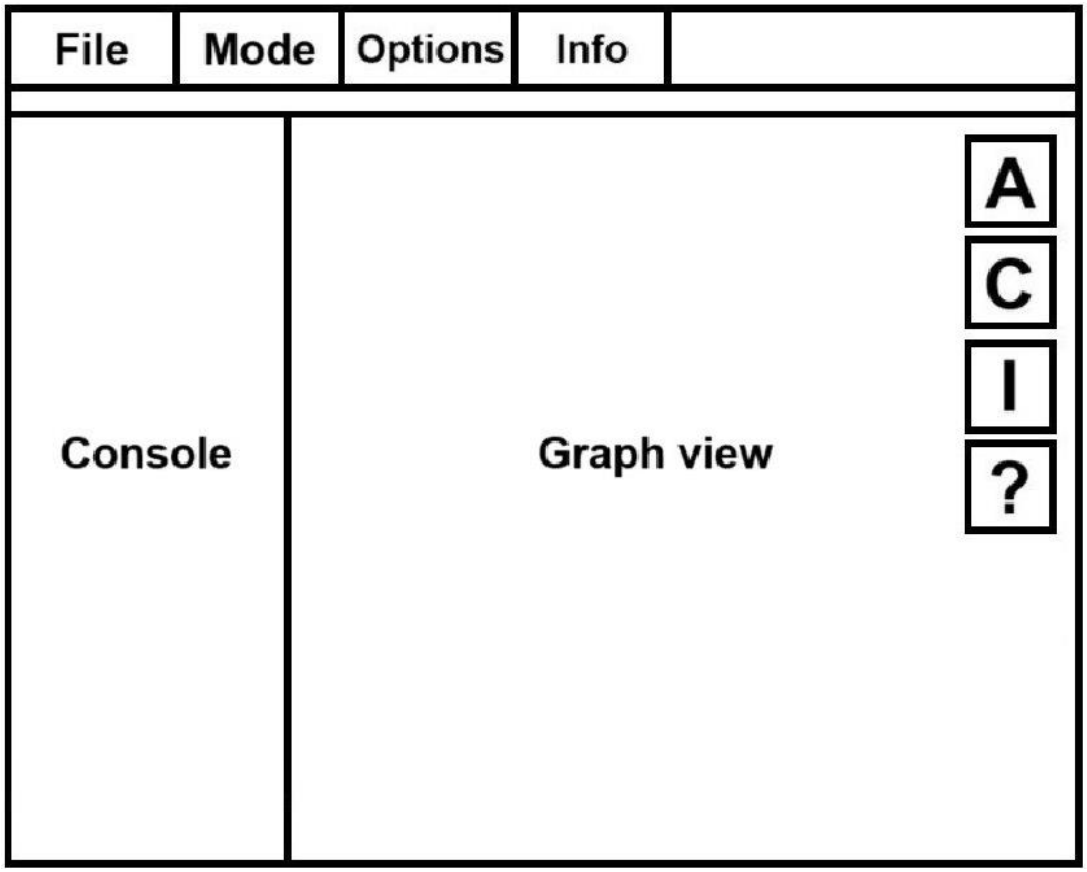

Рисунок 1 – интерфейс в режиме редактирования.

2) Режим алгоритма:

2.1) В холсте (Graph view) возможно только перемещать вершины (но не редактировать граф).

2.2) Текстовое окошко (Log) используется для вывода подробностей о работе алгоритма.

Справа от холста:

2.3) Кнопка для авторасположения графа (A) (программа автоматически распределяет граф, как если бы он был загружен из файла).

2.4) Кнопка для получения информации о графе (?).

Под консолью:

2.5) Кнопки “Шаг назад”, “Авторабота алгоритма”, “Шаг вперед”. При нажатии “Авторабота алгоритма” сама кнопка меняется на “Остановить автоработу”, соседние кнопки на “Быстрее” и “Медленнее”.

Схема интерфейса в режиме работы алгоритма представлена на рисунке 2:

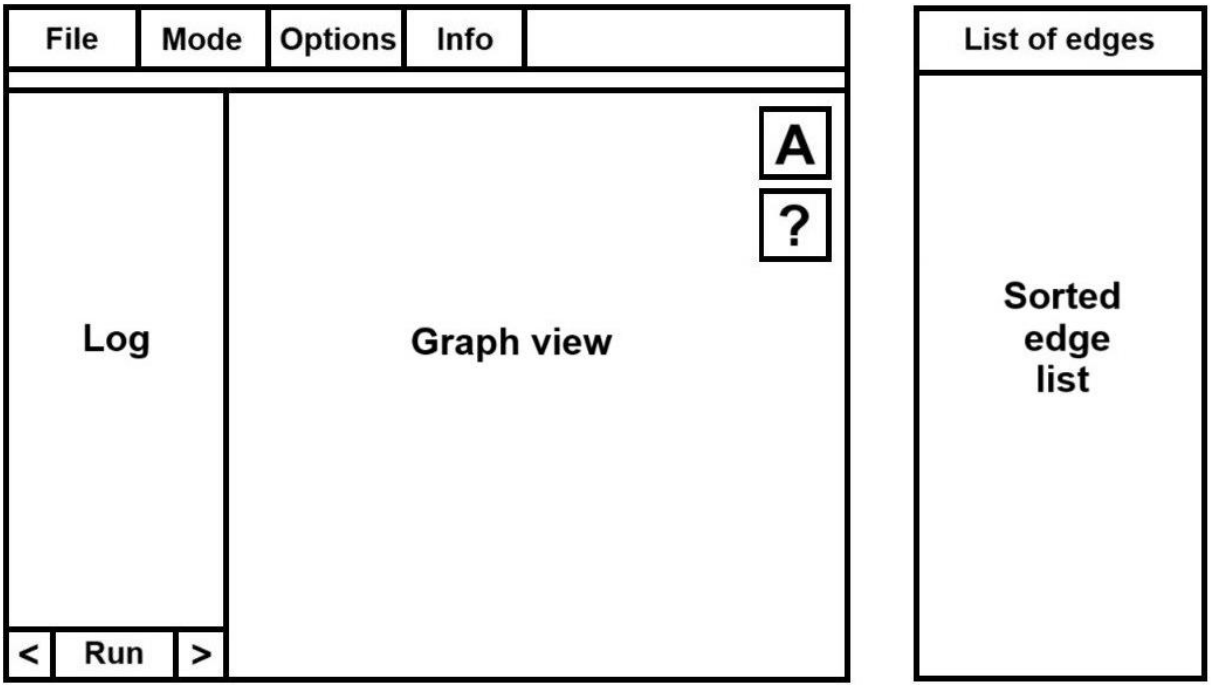

Рисунок 2 – интерфейс в режиме работы алгоритма.

Схема архитектуры приложения представлена в виде UML на рисунке 3:

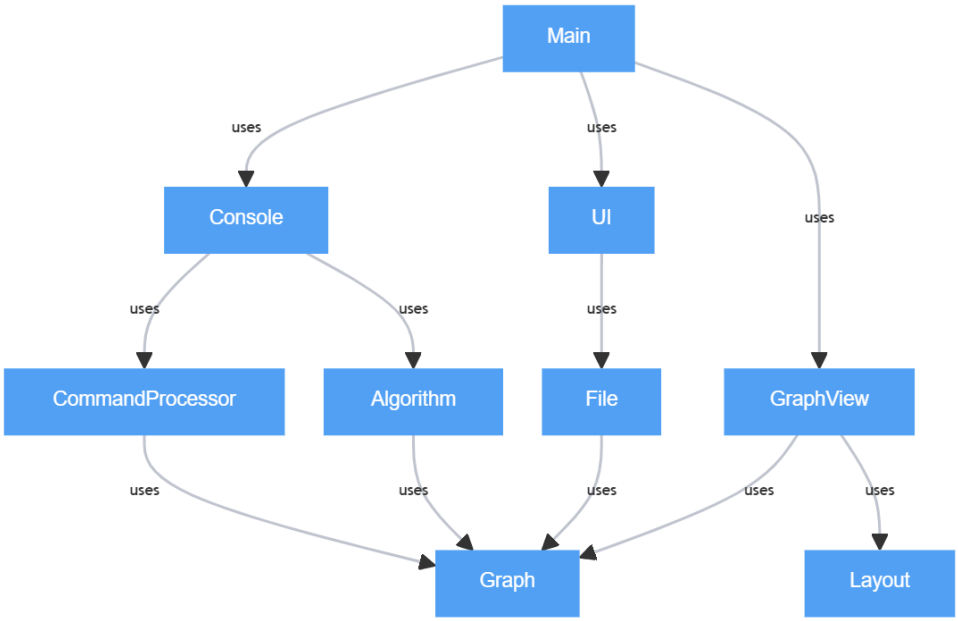

Рисунок 3 – схема архитектуры приложения.

1.1. Уточнение требований после сдачи бета версии

Добавить кнопку финиш, позволяющую сразу получить результат работы алгоритма. Добавить возможность сохранить результат.

2. ПЛАН РАЗРАБОТКИ И РАСПРЕДЕЛЕНИЕ РОЛЕЙ В БРИГАДЕ

2.1. План разработки

Приблизительный план разработки:

28.06.2024 - защита вводного задания, демонстрация спецификации и плана. К этому этапу планируется иметь план и спецификации, по которым в дальнейшем будет идти разработка приложения.

01.07.2024 - защита прототипа. В рамках прототипа планируется как минимум иметь интерфейс, местами без функционала, возможность открывать и просматривать граф, возможность запустить алгоритм и получить его результат (без пошагового исполнения).

01.07.2024 - бета версия. В рамках бета-версии планируется завершить разработку основного функционала, внесение правок по итогам защиты прототипа.

05.07.2024 - релиз. В рамках релиза планируется подправить интерфейс приложения, внести правки по результатам защиты бета-версии, отладить приложение.

05.07.2024 - отчёт.

2.2. Распределение ролей в бригаде

Ламашовский Денис – тест приложения, часть отчёта.

Ефремова Анна – реализация алгоритма, дизайн, отчёт

Ильин Егор – API графа, файлы, визуализация, интерфейс, часть отчёта со своим кодом.

3. ОСОБЕННОСТИ РЕАЛИЗАЦИИ

3.1 Основные структуры данных

typealias VertexId – тип идентификатора вершины. Идентификаторы вершин используются во внутренней работе графа. Гарантируется уникальность идентификаторов вершин. Нельзя задать идентификатор извне или как-либо изменить его, а также использовать из предположения о некотором распределении идентификаторов (не гарантируется, что они неотрицательны, расположены подряд и т.д.).

data class Vertex – класс, представляющий одну вершину графа. Вершина имеет ряд свойств: идентификатор, имя, положение на холсте, цвет и список исходящих рёбер. Имя, как и идентификатор, должно быть уникальным, и его можно задавать извне/проводить над ним произвольные операции. Положение вершины опционально (определено как nullable). Цвет обязателен.

data class OutcomingEdge – класс исходящего из вершины ребра. Используется внутри графа и не предназначен для внешнего использования. Хранит вершину, в которую ребро идёт, вес ребра (целочисленное значение) и цвет ребра. Как правило, ребро графа задаётся двумя такими исходящими рёбрами, поскольку граф ненаправленный, потому для каждого ребра из A в B есть аналогичное ребро из B в A.

data class Edge – класс ребра графа для внешнего по отношению к классу графа использования. Содержит первую и вторую вершину ребра, его вес и его цвет.

open class Graph – базовый класс графа. Содержит список вершин, к которому можно обратиться на чтение извне, а также свойство *edges*, позволяющее получить список рёбер графа типа *Edge*. Имеет ряд методов для работы с графом, но почти все из них объявлены как *protected* и предназначены для использования в наследнике.

class GraphColoring – класс, представляющий цвета вершин и рёбер графа. Позволяет сохранить раскраску графа и затем применить её.

open class GraphException – базовый класс исключения работы графа. Имеет ряд наследников, которые могут выбрасываться методами графа в разных ситуациях

(попытка обратиться к несуществующей вершину / ребру, попытка вставить уже существующее ребро / вершину, добавить петлю, и так далее).

enum class LayoutType – перечисление видов расположения графа (круговое расположение, расположение решёткой и т.д.)

abstract class Layout – базовый класс расположения графа. Имеет метод, который распределяет вершины графа по холсту.

class CircleLayout – класс, наследующий *Layout*, класс кругового расположения. Располагает вершины графа по кругу, учитывая компоненты связности и рёбра графа.

class NaiveGridLayout – класс, наследующий *Layout*, располагает вершины решёткой.

class SpringLayout – класс, наследующий Layout, использует CircleLayout для исходного расположения вершин и затем запускает симуляцию, где рёбра действуют как пружины и возникает сила отталкивания между вершинами.

class RenderableGraph – наследует класс *Graph*. Представляет собой граф, имеющий полноценный публичный интерфейс и возможность отрисовки. Имеет методы для отрисовки, для работы с графом через объекты вершин или через их имена, для работы с цветами вершин и рёбер и так далее.

Поля и методы класса *RenderableGraph*:

1. Поле *vertices: TreeSet<Vertex>* – список вершин. Вне класса доступно только на чтение, можно, например, проитерироваться по всем вершинам графа. Для всего остального – работа через методы класса, описанные ниже.
2. Свойство *edges: ArrayList<Edge>* – при получении значения создаёт и возвращает список всех рёбер в графе.
3. Свойство *graphColoring: GraphColoring* – позволяет получать объект раскраски графа и применять его к графу через присвоение этому свойству.
4. *renderGraph(drawScope: DrawScope, textMeasurer: TextMeasurer, offset: Offset, scale: Float, widgetScale: Float)* – производит отрисовку графа. Используется во модуле *UI*.

5. *fun positionVertices(layout: Layout)* – располагает вершины графа в соответствии с переданным *layout*. *Layout* – интерфейс с функцией *positionVertices*, которая каким-то образом присваивает всем вершинам некоторую позицию.

6. *fun resetColors()* – сбрасывает цвета в графе к стандартным.

7. *fun replaceVertexColor(oldColor: Color, newColor: Color): Int* – заменяет цвета всех вершин старого цвета на новый цвет. Возвращает число перекрашенных вершин. Если старый и новый цвета совпадают, возвращает 0 и ничего не делает.

8. *fun setVertexColor(vertex: Vertex, color: Color?)* – присваивает переданной вершине переданный цвет. Если в качестве цвета передано *null*, то устанавливает цвет по умолчанию. Если переданной вершины нет в графе, выбрасывает исключение *NoSuchVertexException*.

9. *fun setEdgeColor(edge: Edge, color: Color?)* – устанавливает переданный цвет переданному ребру. Аналогично предыдущему методу, бросает исключение при отсутствии одной из вершин / ребра в графе. Аналогично, если в качестве цвета передать null, устанавливает ребру цвет по-умолчанию.

10. fun setEdgeColor(from: Vertex, to: Vertex, color: Color?) - устанавливает переданный цвет ребру между переданными вершинами. Аналогично предыдущему методу, бросает исключение при отсутствии одной из вершин / ребра в графе. Аналогично, если в качестве цвета передать *null*, устанавливает ребру цвет по умолчанию.

11. *fun getVertexAtPosition(position: Offset): Vertex?* – возвращает вершину, находящуюся на указанной позиции. Если таких вершин несколько, возвращает первую из тех, чей центр ближе всего к переданной позиции. Если таких вершин нет, возвращает *null*.

12. *fun getVertexByName(name: String): Vertex*? – возвращает вершину с указанным именем. Если такой вершины в графе нет, возвращает null.

13. *fun addVertex(name: String, pos: Offset? = null): Vertex* – добавляет в граф новую вершину с указанным именем и (опционально) с указанной позицией и

возвращает её. Если вершина с таким именем уже есть в графе, выбрасывает *VertexAlreadyExistsException*.

14. *fun removeVertex(vertex: Vertex)* – удаляет вершину вместе со всеми её рёбрами из графа. Если переданной вершины в графе нет, выбрасывает *NoSuchVertexException*. Если вершины совпадают, то выбрасывает *SelfLoopException*, т.к. петли запрещены.

15. *fun addEdge(from: String, to: String, weight: Int)* – добавляет в граф ребро с указанным весом между вершинами с указанными именами. Если таких вершин в графе нет - добавляет их. Используется при создании графа (т.к. можно сразу добавлять ребра минуя вершины).

16. *fun addEdge(from: Vertex, to: Vertex, weight: Int)* – добавляет ребро с указанным весом между указанными вершинами. Если вершины совпадают, то выбрасывает *SelfLoopException*, т.к. петли запрещены. Если такое ребро уже существует, то выбрасывается *EdgeAlreadyExistsException*. Если одной из вершин нет в графе – выбрасывает *NoSuchVertexException*.

17. *fun removeEdge(edge: Edge)* – удаляет переданное ребро из графа. Если переданных вершин нет в графе – выбрасывает *NoSuchVertexException*, если удаляемого ребра нет в графе – выбрасывает *NoSuchEdgeException*.

18. *fun removeEdge(from: Vertex, to: Vertex)* – удаляет ребро между указанными вершинами из графа. Если переданных вершин нет в графе – выбрасывает *NoSuchVertexException*, если удаляемого ребра нет в графе – выбрасывает *NoSuchEdgeException*.

19. *fun makeUpVertexName(): String* – вспомогательная функция. Генерирует и возвращает уникальное имя для вершины, которого гарантировано нет ни у одной из вершин в графе.

20. *fun getEdge(from: Vertex, to: Vertex): Edge?* – возвращает ребро из первой вершины во вторую. Если ребра между переданными вершинами нет, возвращает *null*. Если одной из вершин нет в графе, выбрасывает *NoSuchVertexException*.

21. *fun getOutcomingEdge(from: Vertex, to: Vertex): OutcomingEdge?* – возвращает ребро из первой вершины во вторую (ребро одностороннее, каждое реальное ребро графа представлено двумя такими односторонними ребрами). Если ребра между переданными вершинами нет, возвращает *null*. Если одной из вершин нет в графе, выбрасывает *NoSuchVertexException*.

22. *fun splitIntoComponents(): Array<MutableSet<Vertex>>* – функция, наследуемая от *Graph* (и единственная доступная извне). Возвращает разбиение множества вершин графа на его компоненты связности.

3.2 Структуры данных, отвечающие за алгоритм

open class AlgorithmException(text: String) – базовый класс исключения для работы алгоритма. Его наследники:

class TooManyVertices(message: String) : AlgorithmException("Message: $message") – данное исключение выбрасывается, если в переданном алгоритму графе слишком много вершин (что мешает их корректной различимой окраске).

class TooManyComponents(message: String): AlgorithmException("Message: $message") – данное исключение выбрасывается, если в переданном алгоритму графе больше одной компоненты связности, так как в таком графе невозможно найти МОД.

class IsEmptyGraph(message: String): AlgorithmException("Message: $message") – данное исключение выбрасывается, если алгоритму передан пустой граф.

enum class AlgorithmTheme(val edgeInMST: Color, val skippedEdge:Color, val textEdgeInMST:Color, val textSkippedEdge:Color, val allConsoleText:Color, val consoleSkippedEdge:Color, val consoleEdgeInMST:Color, val consoleRecoloredVert:Color, val consoleMSTWeight:Color) – перечисление вариантов тем для визуализации алгоритма (тема определяет цвет шрифта в консоли и окне с ребрами, цвет рёбер вошедших в МОД и т.д.).

class AlgorithmOptions – содержит поля, отвечающие за настройку отображения работы алгоритма (выбранная тема, размеры текста консоли и окна с ребрами, необходимость отображать окно с рёбрами).

class Kruskal(val graph: RenderableGraph) – класс реализующий модифицированный алгоритм Краскала для графа *graph*. Поле *internal var result:ArrayList<Edge>* хранит список рёбер, вошедших в МОД данного графа, а свойство *val weightMST:Int* позволяет получить текущий (т.е. известный на данном шаге) вес МОД.

В классе *Kruskal* определены следующие методы:

fun printInformConsole(stepNum:Int, skiped:Int, added:Edge?, recolored:Int) – печатает информацию о текущем шаге в консоль (номер шага *stepNum*, число пропущенных на текущем шаге рёбер *skiped*, добавленное на текущем шаге ребро *added*, число перекрашенных на текущем шаге вершин *recolored*).

fun createTextEdges(ind:Int, edgeList:List<Edge>) – печатает информацию о текущем шаге в окно со списком рёбер (индекс текущего ребра из списка рёбер *ind*, список рёбер *edgeList*).

fun init() – проверяет граф на исключительные ситуации (слишком много вершин / подан пустой граф / у графа больше одной компоненты связности). Затем инициализирует поле с отсортированным списком рёбер и раскрашивает вершины графа в разные цвета.

fun findNextEdge(cur:Int, edgeList:List<Edge>): Int – ищет следующее по отсортированному списку ребро, вершины инцидентные которому имеют различные цвета.

fun step(): Boolean – совершает шаг модифицированного алгоритма Краскала, возвращает false, если МОД собран, true в противном случае.

4. ТЕСТИРОВАНИЕ

4.1. Тестирование интерфейса и обработки исключительных ситуаций.

1) Для выхода из приложения можно воспользоваться клавишей Escape или кнопкой File Exit (рисунок 4).

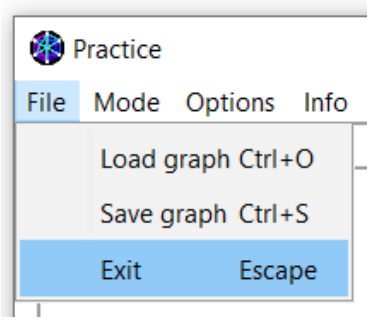

Рисунок 4 – Кнопка выхода в меню приложения.

Попробуем создать граф и выйти из приложения (рисунки 5-6).

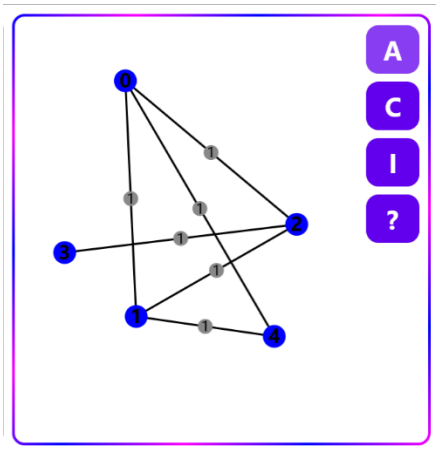

Рисунок 5 – Окно с графом до выхода из приложения.

Рисунок 6 – Окно с графом после повторного запуска.

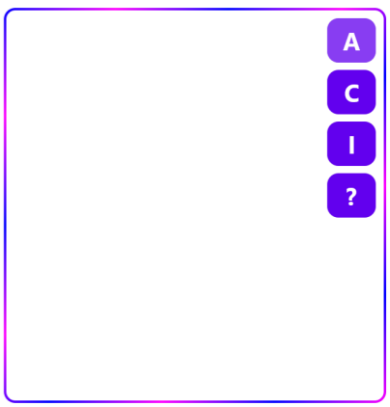

Рисунок 6 – Окно с графом после повторного запуска.

Видим, что граф не сохраняется при перезапуске программы. Для случаев, когда сохранить граф необходимо для дальнейшей работы, в программе предусмотрена возможность сохранения графа в одном из нескольких форматов.

2) Настройки отображения. В программе предусмотрена возможность настроить отображение элементов графа используя меню, открываемое с помощью кнопки Options -> Graph Render Options. Следующие характеристики могут быть изменены пользователем: размер вершин, ширина ребер, размер имен вершин, размер весов ребер, расположение веса на ребре (ближе к какой из вершин он будет находиться).

Попробуем изменить эти настройки и посмотреть, как изменится визуальное отображение графа (рисунки 7-9):

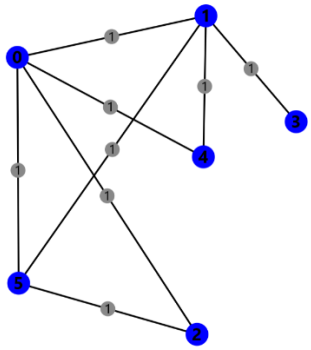

Рисунок 7 – Исходное отображение графа.

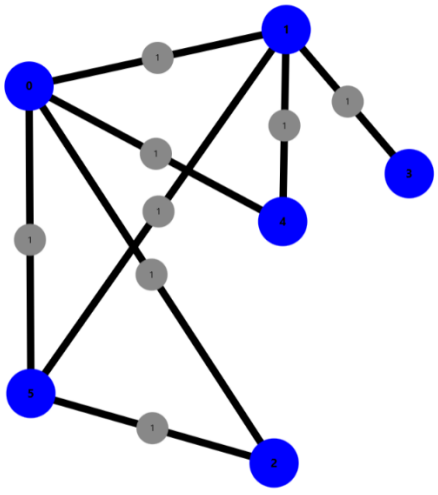

Рисунок 8 – Отображение графа с увеличенными шириной ребер и размером вершин, а также уменьшенными размерами имен вершин и весов ребер.

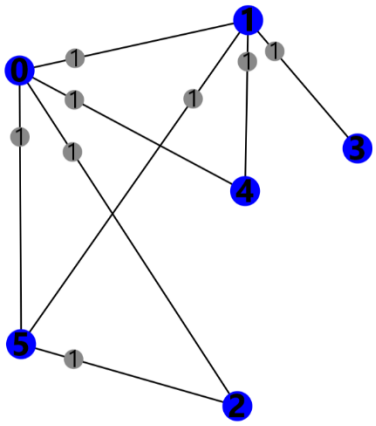

Рисунок 9 – отображение графа с увеличенными именами вершин и весами ребер, а также смещенными весами в сторону меньшей по номеру вершины.

Кроме того, пользователь может расположить вершины вручную (рисунок 7) или выбрать один из трех способов автоматического расположения вершин:

По кругу (рисунок 10). Предполагает возможность задания радиуса круга, а также дополнительных условий, связанных с плотностью ребер и вершин.

По прямоугольной сетке (рисунок 11). Предполагает возможность задания ширины шага сетки.

На основе физической симуляции (рисунок 12). Предполагает возможность задания разных настроек физической симуляции.

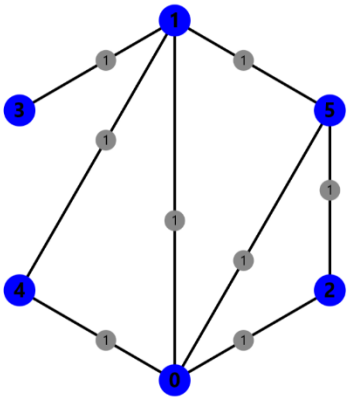

Рисунок 10 – отображение графа с расстановкой вершин по кругу.

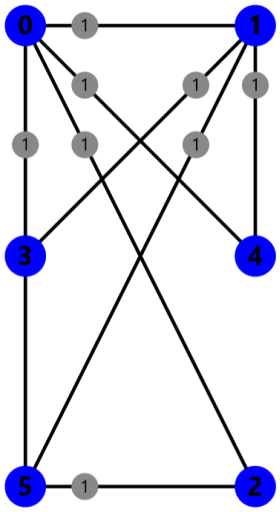

Рисунок 11 – отображение графа с расстановкой вершин по прямоугольной сетке.

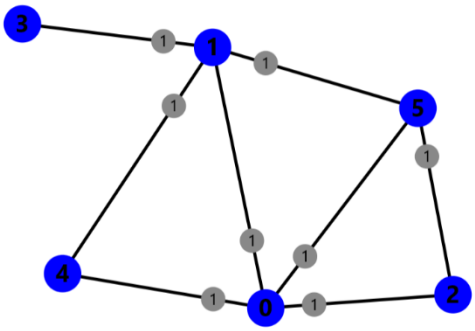

Рисунок 12 – Отображение графа с расстановкой вершин на основе физической симуляции.

Из приведенных примеров, в частности, видно, что настройки отображения и способы расположения вершин можно комбинировать, т.е. пользователю предоставляется гибкий инструмент для работы с отображением графа.

3) Кнопка A позволяет обновить расстановку вершин по заданному в настройках способу. Особо полезной она будет, когда пользователь немного изменил исходный граф (удалил или добавил вершины, переместил их). Кнопка позволит восстановить расстановку вершин. Добавим в исходный граф (рисунок 10) еще одну вершину (рисунок 13) и воспользуемся кнопкой (рисунок 14).

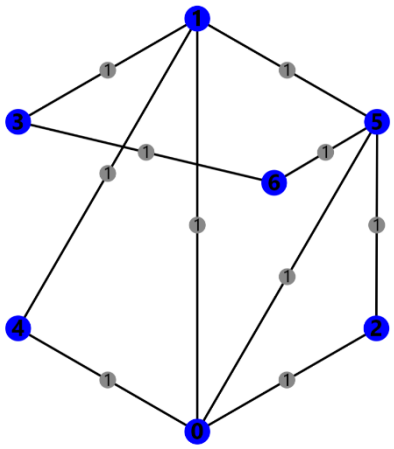

Рисунок 13 – Исходный граф с новой вершиной.

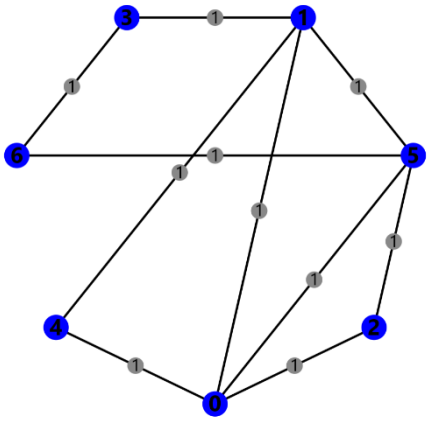

Рисунок 14 – Обновленный граф с расстановкой вершин по кругу.

Заметим, что использование кнопки A возможно и для других способов расстановки вершины.

4) Кнопка C позволяет очистить поле, т.е. удаляет все вершины и ребра (рисунки 15-16). Запуск возможен для любого графа.

Рисунок 15 – Подтверждение удаления.

Рисунок 16 – Очищенное поле.

5) Пользователь может вставить графы особой структуры (кнопка I). Программно предусмотрена возможность создания следующих графов:

а) Полный граф из N вершин (рисунок 17)

б) Нуль граф из N вершин (рисунок 18)

в) Цикл на N вершин (рисунок 19)

г) Полный двудольный граф на K вершин в первой доле и M вершин во второй доле (рисунок 20).

Попробуем создать такие графы:

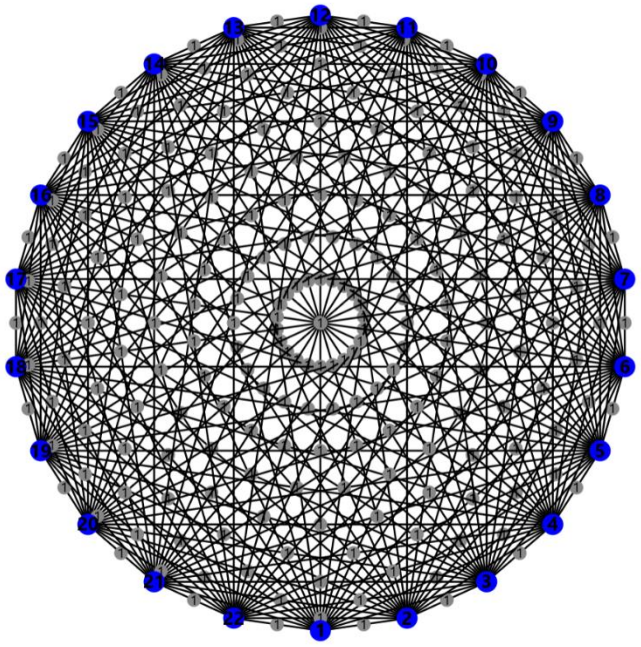

Рисунок 17 – Созданный с помощью кнопки I полный граф на 22 вершины.

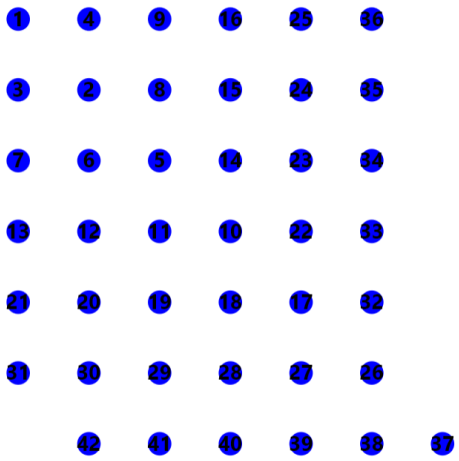

Рисунок 18 – Созданный с помощью кнопки I нуль граф на 42 вершины.

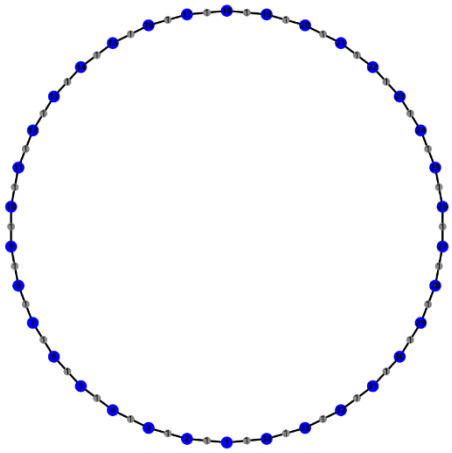

Рисунок 19 – Созданный с помощью кнопки I цикл на 34 вершины.

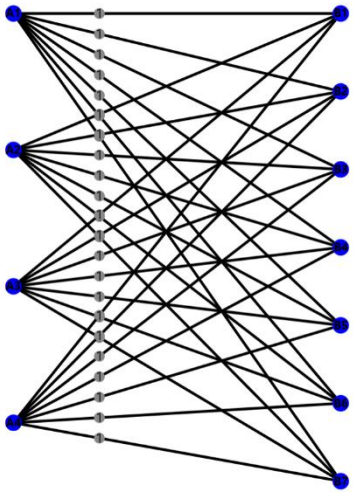

Рисунок 20 – Созданный с помощью кнопки I двудольный граф на 4 вершины в одной доле и 7 вершин в другой доле.

6) Пользователь может узнать следующие характеристики графа, воспользовавшись кнопкой ?: количество вершин, количество ребер, количество компонент связности, суммарный вес ребер графа.

Например, для двудольного графа (рисунок 20) будет выведена следующая информация (рисунок 21).

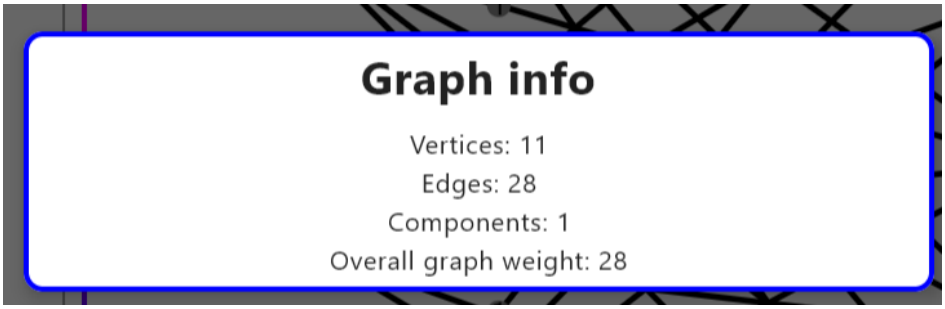

Рисунок 21 – Характеристики двудольного графа.

Попробуем создать граф с несколькими компонентами связности и разными весами ребер (рисунок 22) и узнаем его характеристики (рисунок 23).

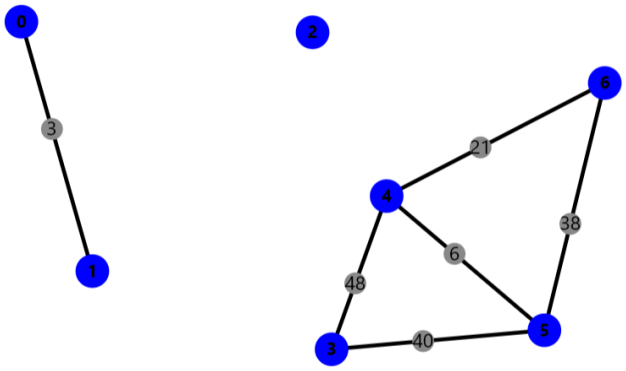

Рисунок 22 – Граф с несколькими компонентами связности.

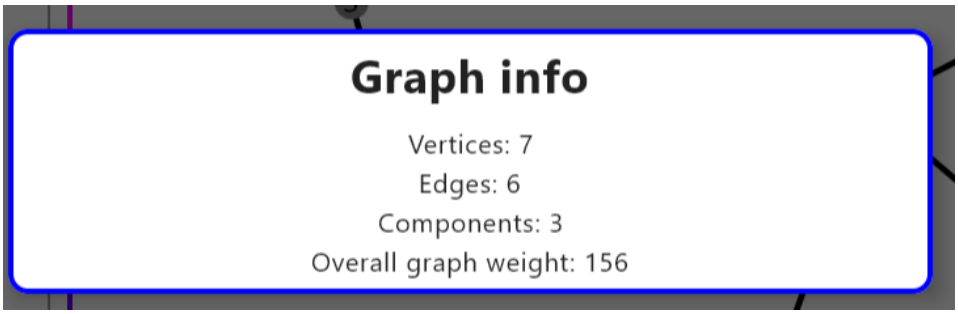

Рисунок 23 – Характеристики графа с несколькими компонентами связности.

7) Для перехода в режим алгоритма можно воспользоваться кнопкой Mode Algorithm. При этом автоматически происходит проверка графа, он должен

удовлетворять следующим условиям: быть связным, иметь как минимум одну вершину.

Попробуем перейти в режим алгоритма с различными графами (рисунки 24-26):

Рисунок 24 – Переход в режим алгоритма с несвязным графом.

Рисунок 25 – Переход в режим алгоритма без графа (пустое поле).

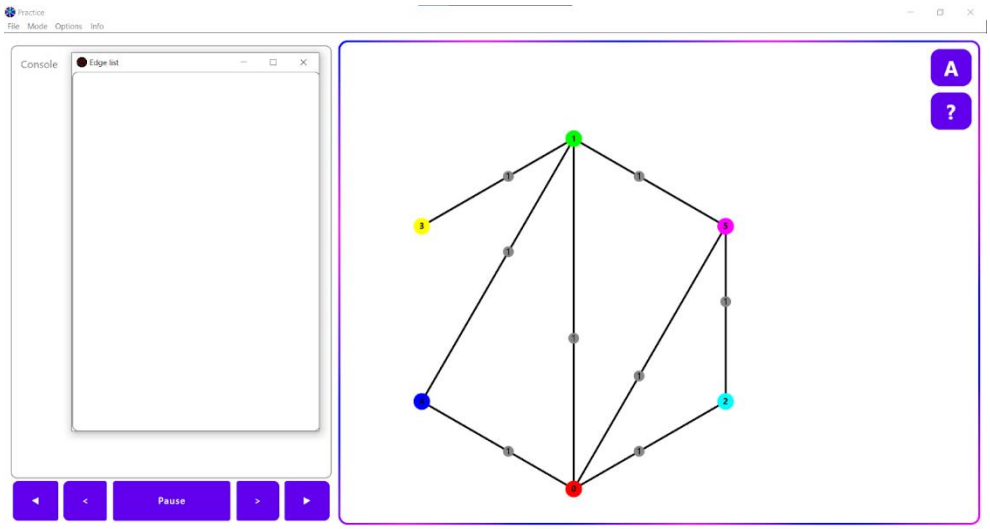

Рисунок 26 – Переход в режим алгоритма с корректным графом.

8) Различимость цветов. На рисунке 26 видим, что для корректного графа осуществляется раскраска, при этом алгоритм старается выбрать визуально различимые цвета. Убедимся, что на большем количестве вершин цвета остаются различимыми (рисунки 27-28):

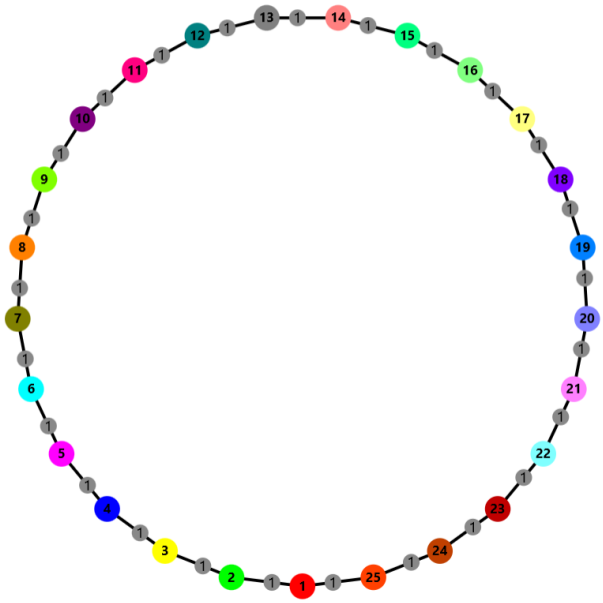

Рисунок 27 – Раскраска графа с 25 вершинами.

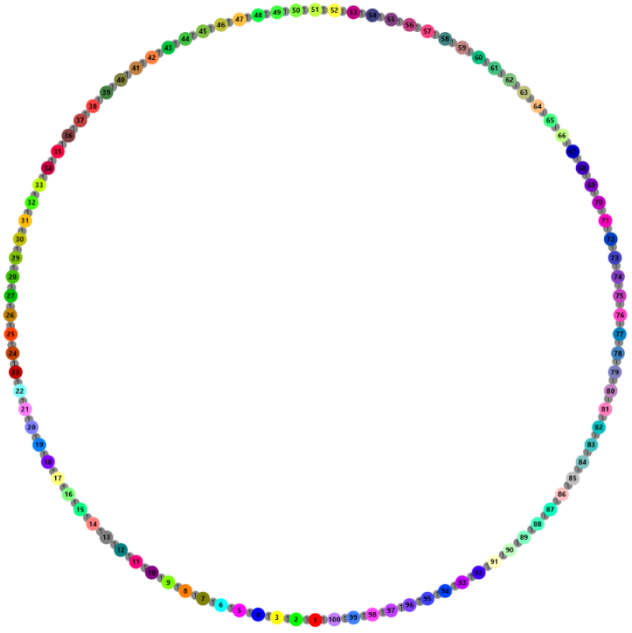

Рисунок 28 – Раскраска графа с 100 вершинами.

Видим, что при большом количестве вершин цвета становятся ближе к другу по оттенку. Это естественное ограничение, тем не менее программа старается подбирать максимально удаленные от предыдущих цвета.

9) Для теста пошаговой визуализации алгоритма создадим граф и будем переключать шаги алгоритма с помощью кнопок в левом нижнем углу (рисунки 29 - 31):

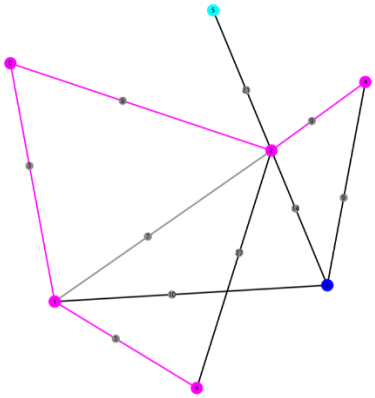

Рисунок 29 – Граф после 4 шага алгоритма.

0 – (w = 3) – 1 - in MST
1 – (w = 5) – 6 - in MST
0 – (w = 6) – 2 - in MST
1 – (w = 7) – 2 skipped
2 – (w = 9) – 4 - in MST
3 – (w = 9) – 4
1 – (w = 10) – 3
2 – (w = 14) – 3
2 – (w = 21) – 5
2 – (w = 27) – 6

Рисунок 30 – Список ребер после 4 шага алгоритма.

Console

_____ Step №1 _____
0 edges skipped
Added 0 --(weight = 3)-- 1
Recolored 1 vertices
Current MST weight: 3
_____ Step №2 _____
0 edges skipped
Added 1 --(weight = 5)-- 6
Recolored 2 vertices
Current MST weight: 8
_____ Step №3 _____
0 edges skipped
Added 0 --(weight = 6)-- 2
Recolored 3 vertices
Current MST weight: 14
_____ Step №4 _____
1 edges skipped
Added 2 --(weight = 9)-- 4
Recolored 4 vertices
Current MST weight: 23

Рисунок 31 – Консоль после 4 шага алгоритма.

Видим, что после применения нескольких шагов визуализация графа соответствует данным из списка ребер и консоли. Попробуем сделать несколько шагов назад (рисунки 32-34), чтобы убедиться, что взаимно однозначное соответствие сохраняется и при обратном ходе.

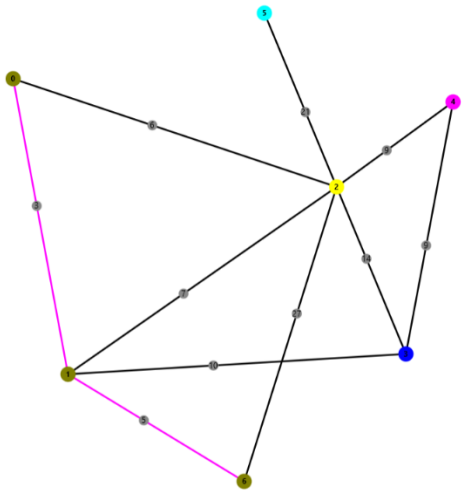

Рисунок 32 – Граф отката на 2 шага алгоритма.

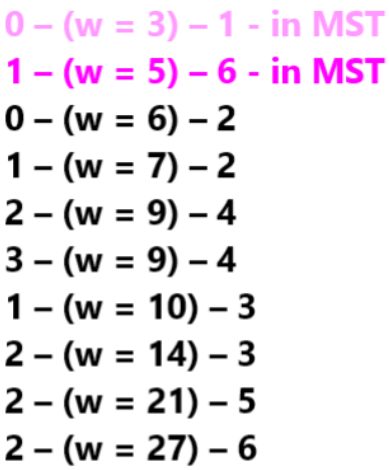

Рисунок 33 – Список ребер после отката на 2 шага алгоритма.

Рисунок 34 – Консоль после отката на 2 шага алгоритма.

10) Также для пользователя доступны кнопки мгновенного перехода в начало и конец алгоритма. Попробуем из середины выполнения алгоритма перейти в начало алгоритма (рисунок 35) и в конец алгоритма (рисунок 36), чтобы убедиться в работоспособности этих кнопок.

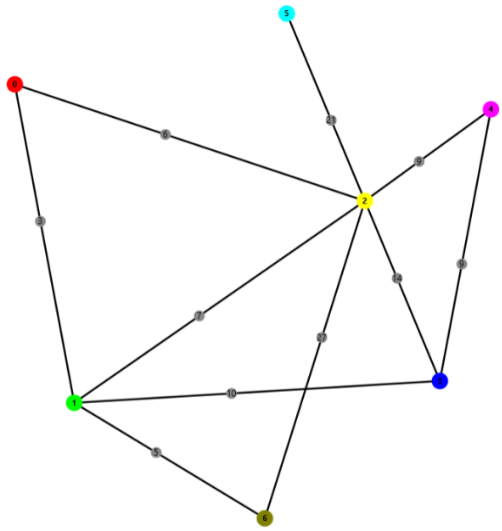

Рисунок 35 – Граф, после перехода в начало алгоритма.

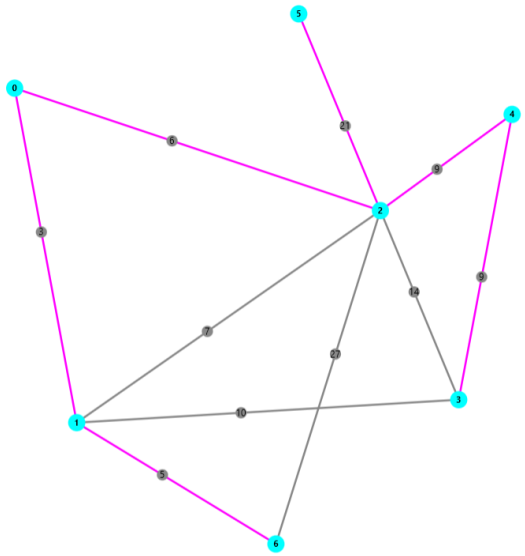

Рисунок 36 – Граф, после перехода в конец алгоритма.

11) Кроме пошаговой визуализации, когда пользователь для перехода на следующий шаг жмет кнопку, есть и автоматический режим, который выполняет шаги алгоритма самостоятельно. Попробуем включить автоматический режим, дойти до пятого шага алгоритма, а затем нажать паузу (рисунок 37).

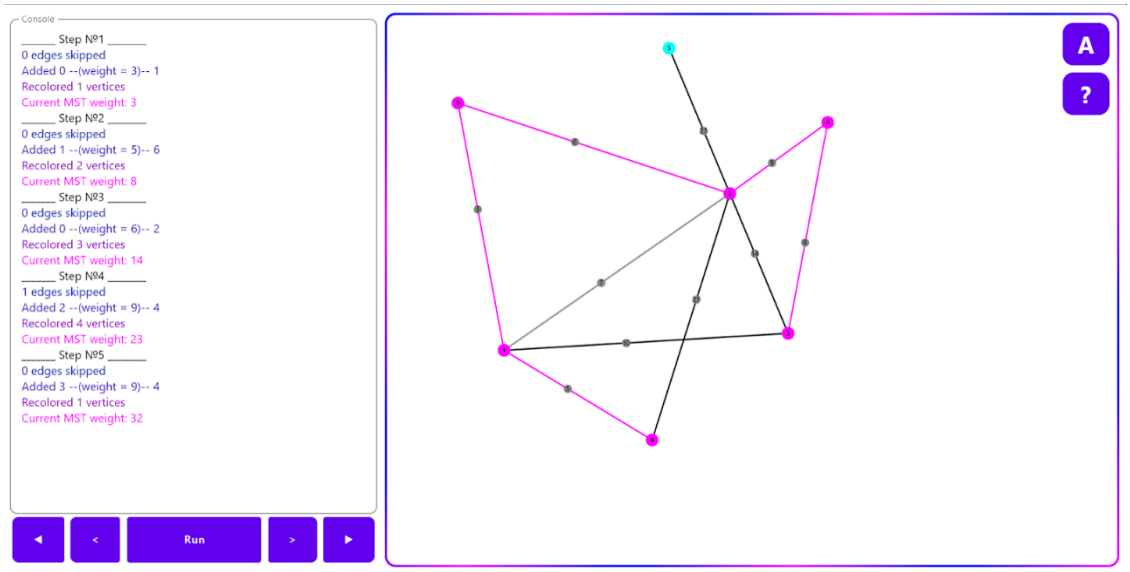

Рисунок 37 – Программа, после автоматического запуска алгоритма до пятого шага.

12) В программе предусмотрен ряд настроек работы алгоритма. Можно сменить тему приложения, размер текста в консоли, в окне ребер, также можно окно ребер отключить.

Для теста этих настроек попробуем на разных этапах алгоритма:

а) Сделать максимальный размер шрифта в консоли и на окне ребер, сменить тему на красную (рисунки 38-39).

б) Сделать минимальный размер шрифта в консоли, отключить окно ребер, сменить тему на пастельную (рисунок 40).

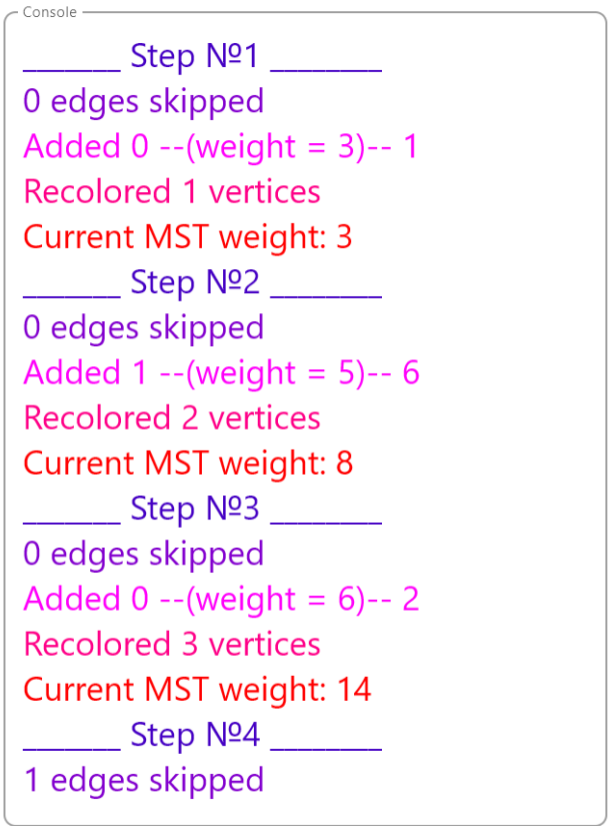

Рисунок 38 – Консоль, после увеличения шрифта и смены темы.

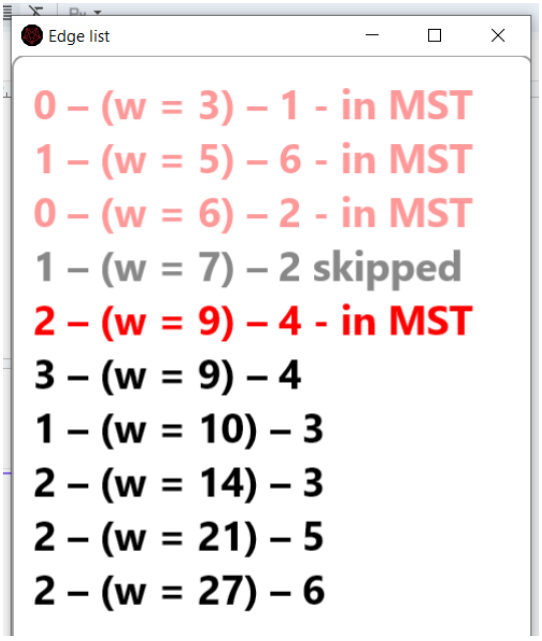

Рисунок 39 – Список ребер, после увеличения шрифта и смены темы.

Рисунок 40 – Консоль, после уменьшения шрифта и смены темы.

13) Убедимся в корректности работы алгоритма Краскала запустив его на разных графах (рисунки 41-43).

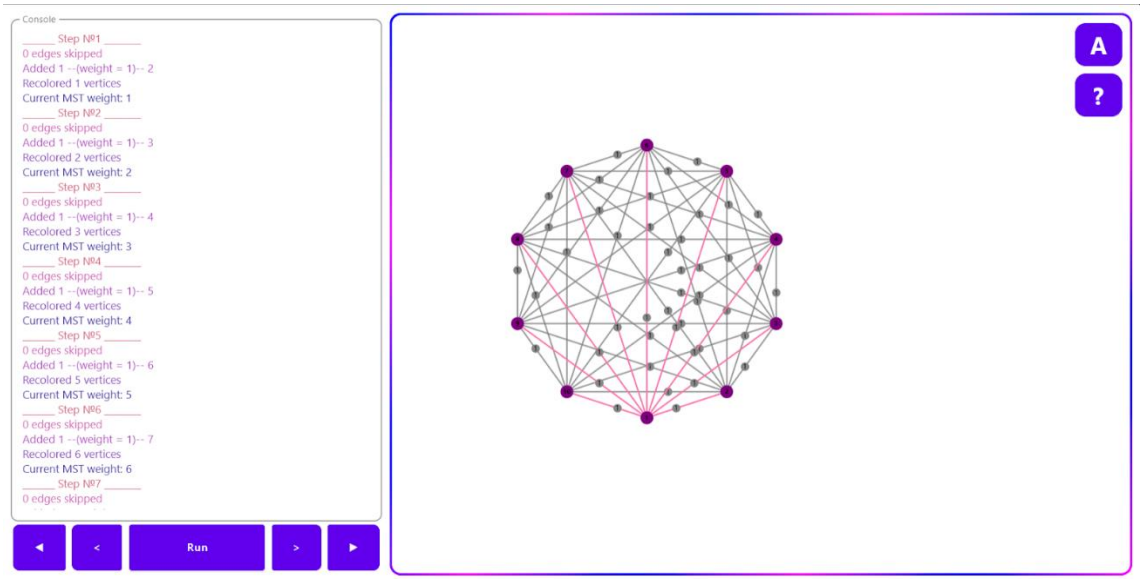

Рисунок 41 – Запуск алгоритма на графе с одинаковыми весами.

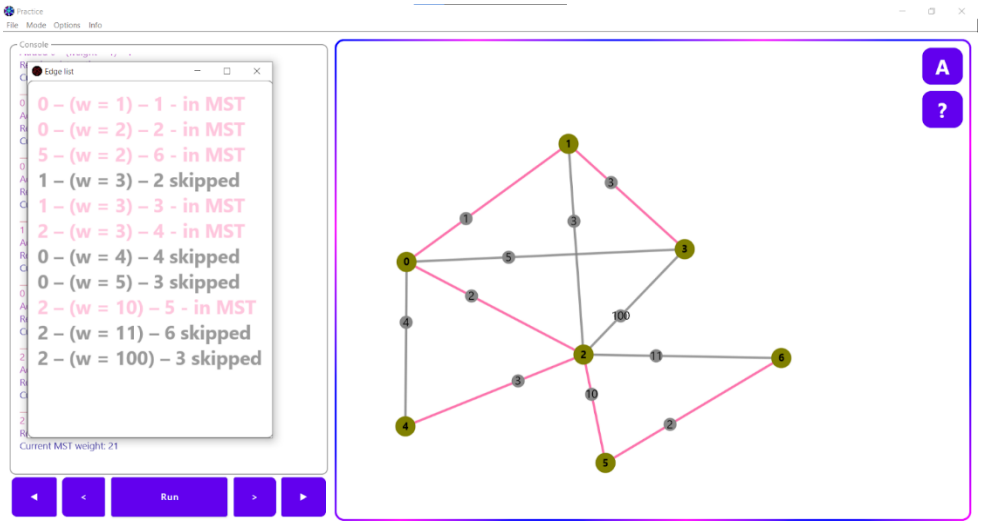

Рисунок 42 – Запуск алгоритма на произвольном графе.

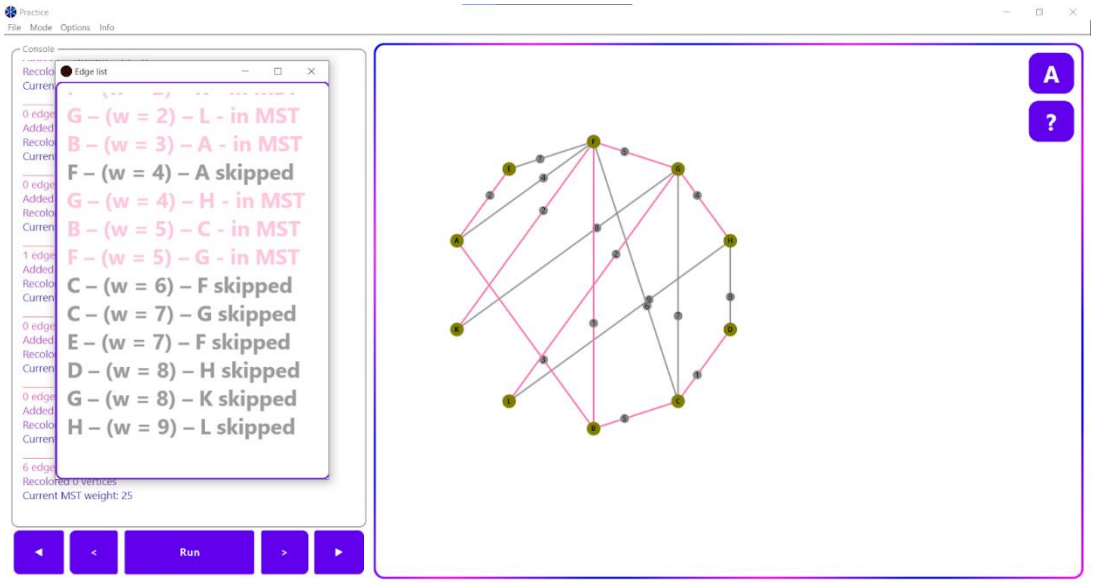

Рисунок 43 – Запуск алгоритма на произвольном графе.

14) Тест переименования вершин:

Нажатие на вершину ЛКМ с зажатым Shift позволяет переименовать выбранную вершину (представлено на рисунке 44):

Рисунок 44 – переименование вершины

Если в имени вершины есть $ то всё, начиная с этого символа, не отображается (представлено на рисунке 45):

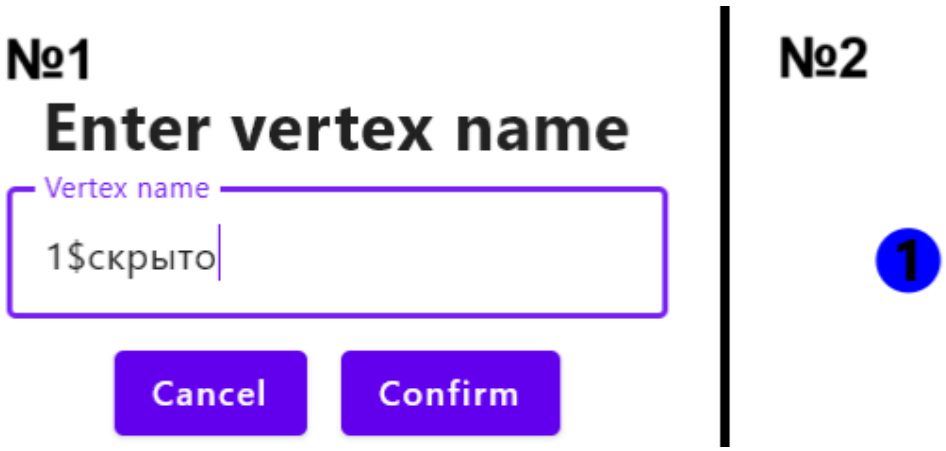

Рисунок 45 – скрытые имена вершин

15) изменения веса ребра.

Нажав на вершину ЛКМ, её можно выделить, затем нажав ЛКМ на другой вершине с зажатым Shift можно изменить вес ребра между ними. Приложение позволяет давать ребрам любой целочисленный вес в рамках типа Int (что представлено на рисунке 46):

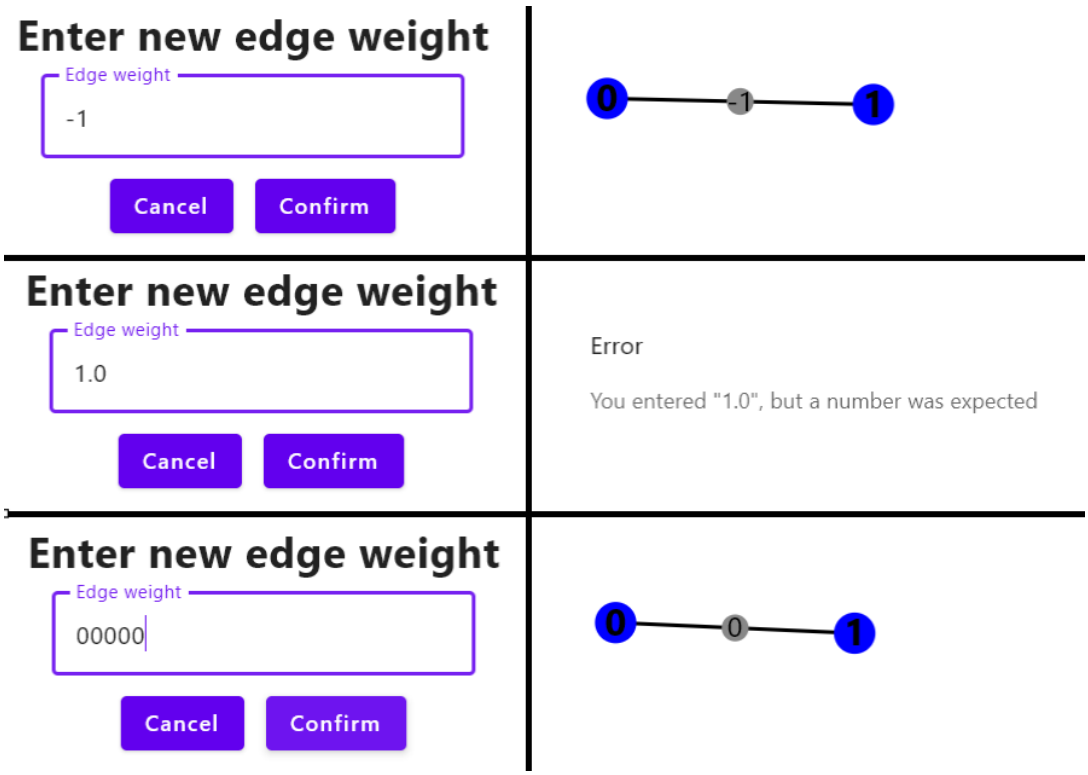

Рисунок 46 – изменение веса ребра

16) Возможно перемещение по холсту (зажатием ЛКМ и переносом холста куда-либо) и возможность вернуться к начальному виду нажатием B (продемонстрировано на рисунке 47):

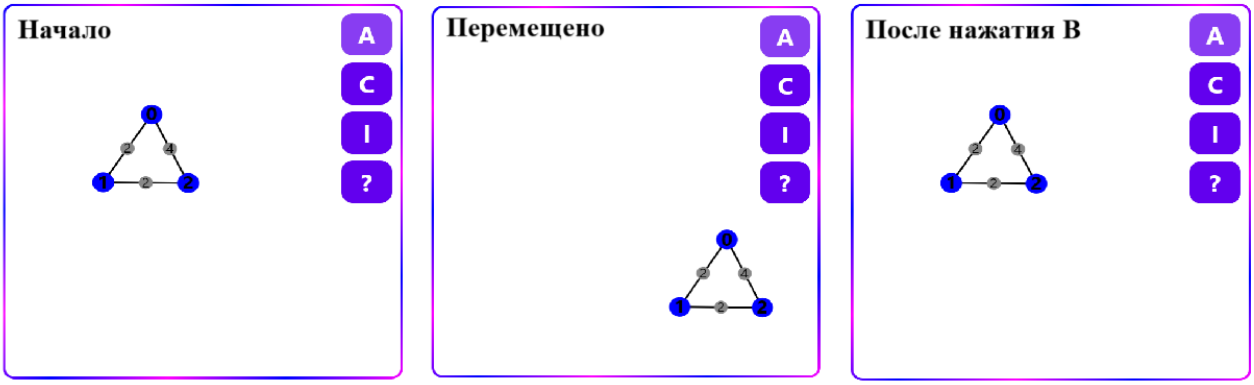

Рисунок 47 – перемещение по холсту.

17) Зажатие вершины ЛКМ позволяет передвинуть ее (рисунок 48)

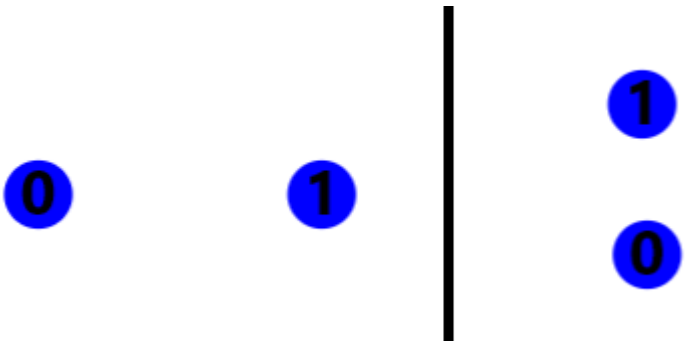

Рисунок 48 – перемещение вершины

ЗАКЛЮЧЕНИЕ

В ходе выполнения практической работы было реализовано приложение с графическим интерфейсом, содержащее графический редактор графов, демонстрирующее пошаговое выполнение модифицированного алгоритма Краскала. Закреплены навыки программирования на языке Kotlin.

Разработанное приложение соответствует требованиям, предъявленным в начале работы.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Репозиторий бригады:

URL: https://github.com/CoreJust/Practice-Kruskal-Visualization/tree/master

2. Руководство по Jetpack Compose:

URL: https://metanit.com/kotlin/jetpack/

3. Алгоритм SpringLayout

URL:

https://wwwmayr.in.tum.de/lehre/2012WS/algoprak/uebung/tutorial11.english.pdf

4. Сайт Android Developers:

URL: https://developer.android.com/

5. Сайт Kotlin Programming Language

URL: https://kotlinlang.org/

6. Спецификации формата GML:

URL: https://hwiegman.home.xs4all.nl/fileformats/G/gml/gml-tr.html

7. Использование ресурсов в Compose Mutliplarform

URL: https://www.jetbrains.com/help/kotlin-multiplatform-dev/compose-images-resources.html#189ecf6d

8. Статьи о технических аспектах работы с графами (файловые форматы, отрисовка)

URL: https://docs.yworks.com/yfiles/doc/developers-guide/

9. Формат TGF

URL: https://en.wikipedia.org/wiki/Trivial_Graph_Format