РЕФЕРАТ

**Пояснительная записка к курсовому проекту:** 50 c., 11 рисунков, 1 приложение.

Цель работы – разработка программы визуализации графов.

Краткое описание результатов работы: Разработана программа «Визуализация графов» на Python с использованием библиотеки PyQt5. Реализованы функции для создания и редактирования узлов и рёбер, управления их параметрами, а также перемещения узлов по графу. Реализована возможность автоматического создания графа с помощью матрицы весов. Кроме того, реализованы алгоритмы Дейкстры, Прима, Камада-Кавай, силового метода и пружинного алгоритма для визуализации графов. Поддерживаются функции сохранения и загрузки графов в формате JSON, а также очистка графа. Программа разработана в среде Visual Studio и предоставляет удобный интерфейс для взаимодействия с графами, включая обработку ошибок и уведомления для пользователя.

Содержание

[1 МЕТОДЫ, АЛГОРИТМЫ И БИБЛИОТЕКИ 6](#_Toc186234222)

[1.1 Анализ методов 6](#_Toc186234223)

[1.2 Описание методов 6](#_Toc186234224)

[1.3 Используемые библиотеки 7](#_Toc186234225)

[1.4 Разработка алгоритмов 7](#_Toc186234226)

[1.5 Выводы и результаты 9](#_Toc186234227)

[2 ОПИСАНИЕ РАЗРАБОТАННОГО ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ 11](#_Toc186234228)

[2.1 Выбор средств разработки и системных программных средств 11](#_Toc186234229)

[2.2 Требования к разрабатываемой программе 11](#_Toc186234230)

[2.3 Структура программы 12](#_Toc186234231)

[2.3.1 Основные модули 12](#_Toc186234232)

[2.3.2 Описание основных классов 13](#_Toc186234233)

[2.3.3 Описание основных функций и процедур 17](#_Toc186234234)

[2.4 Описание интерфейса пользователя 17](#_Toc186234235)

[2.5 Контрольный пример и результаты тестирования 18](#_Toc186234236)

[2.6 Выводы и результаты 22](#_Toc186234237)

[ЗАКЛЮЧЕНИЕ 23](#_Toc186234238)

[ПРИЛОЖЕНИЕ А 24](#_Toc186234239)

ВВЕДЕНИЕ

### Обоснование актуальности

Визуализация графов и алгоритмы, связанные с ними, играют важную роль в различных областях, таких как компьютерные сети, транспортные системы и социальные сети. Современные инструменты визуализации часто сталкиваются с проблемами, связанными с эффективностью размещения узлов и рёбер, а также с интуитивностью пользовательского интерфейса. Существующие решения могут быть громоздкими и сложными для понимания, что затрудняет их использование.

### Анализ существующих решений

Существует множество программных средств для визуализации графов, таких как Gephi и Cytoscape. Эти инструменты предлагают широкий функционал, включая различные алгоритмы укладки графов и возможности для анализа. Однако многие из них требуют значительных ресурсов и имеют сложный интерфейс, что может затруднить работу пользователей. Применяемые алгоритмы укладки, такие как силовая и пружинная укладка, часто не оптимизированы для больших графов, что приводит к снижению производительности.

### Цель работы

Целью данной работы является разработка программы для визуализации графов, использующей алгоритмы силовой и пружинной укладки, а также алгоритм Камада-Кавай, с интуитивно понятным интерфейсом для пользователей.

### Задачи:

1. Изучение существующих методов визуализации графов.
2. Разработка архитектуры программы.
3. Разработка алгоритмов силовой и пружинной укладки графов.
4. Реализация алгоритма Камада-Кавай.
5. Создание пользовательского интерфейса с использованием PyQt5.
6. Добавление возможностей сохранения и загрузки.
7. Проведение тестирования и анализ результатов.

МЕТОДЫ, АЛГОРИТМЫ И БИБЛИОТЕКИ

### 1.1 Анализ методов

Для визуализации графов могут быть использованы различные структуры данных и алгоритмы укладки, но основа реализации общая и состоит из следующих функций:

* Чтение и обработка графов из файлов в указанной пользователем директории, добавление графов в визуализацию.
* Применение алгоритмов укладки для оптимального размещения узлов и рёбер.
* Сохранение состояния графа в формате, обеспечивающем дальнейшую загрузку и редактирование.

Для представления графов могут быть использованы различные структуры данных, такие как списки смежности или матрицы весов. Матрицы весов проще в реализации и более эффективны для неориентированных графов, поэтому они будут использованы в данной работе.

### 1.2 Описание методов

1. **Методы чтения графов:**

Программа принимает на вход путь к директории, в которой находятся файлы с описанием графов в формате .json. Файлы обрабатываются с использованием встроенных средств Python. Для работы с графами используется библиотека NetworkX, которая позволяет легко создавать и манипулировать графами.

1. **Методы обработки графов:**

Для оптимизации визуализации и улучшения восприятия графа используются алгоритмы укладки, такие как силовая и пружинная укладка, а также алгоритм Камада-Кавай. Эти алгоритмы обеспечивают равномерное распределение узлов и минимизацию перекрытий рёбер, что делает граф более наглядным.

1. **Алгоритм укладки графов:**

Программа применяет выбранные алгоритмы укладки для размещения узлов. Алгоритмы принимают на вход граф и возвращают координаты для каждого узла, которые затем используются для визуализации. Алгоритм Камада-Кавай, например, стремится минимизировать потенциальную энергию системы, что приводит к более сбалансированному расположению узлов.

1. **Методы создания графа:**

Программа предоставляет интерфейс для создания графов пользователем. Пользователь может добавлять узлы и рёбра, задавая их атрибуты, такие как вес рёбер и метки узлов. Для создания графа используется метод «add\_node» для добавления узлов и метод «add\_edge» для добавления рёбер в граф. Это позволяет пользователю настраивать граф в соответствии с его потребностями и визуализировать его сразу после создания. Также предусмотрена возможность создания графа с помощью матрицы весов: пользователю предоставляется квадратная матрица для заполнения, что упрощает процесс создания графа с заранее заданными связями и весами.

### 1.3 Используемые библиотеки

При реализации программы для визуализации графов были использованы следующие библиотеки:

* **PyQt5 — библиотека для создания графического пользовательского интерфейса (GUI), которая позволяет создавать сложные интерфейсы с использованием виджетов и диалоговых окон.**
* **NetworkX — библиотека для создания, манипуляции и изучения структуры, динамики и функций сложных сетей. Она предоставляет мощные инструменты для работы с графами, включая алгоритмы поиска кратчайших путей и построения минимальных остовных деревьев.**
* **NumPy — библиотека для работы с многомерными массивами и матрицами, а также для выполнения математических операций над ними. Используется для реализации различных алгоритмов расположения узлов графа.**
* **JSON — встроенная библиотека для работы с форматом JSON, которая используется для сериализации и десериализации графов, что позволяет сохранять и загружать данные графа в удобном формате.**
* **random — стандартная библиотека Python для генерации случайных чисел, которая применяется для создания случайных цветов узлов и расположения узлов графа.**

### 1.4 Разработка алгоритмов

Программа для визуализации графов использует три основных алгоритма, реализованных непосредственно:

* 1. **Силовой метод для расположения узлов графа:**

Данный алгоритм применяется для вычисления расположения узлов на холсте. Он основывается на физических моделях, где узлы рассматриваются как частицы, взаимодействующие друг с другом.

Алгоритм включает следующие шаги:

* На вход принимается граф, представленный в виде экземпляра класса Graph.
* Каждый узел и его соседи получают начальные случайные позиции.
* В цикле выполняются итерации, в каждой из которых рассчитываются силы взаимодействия между узлами:
  + Силы притяжения между связанными узлами.
  + Силы отталкивания между всеми узлами.
* Обновляются позиции узлов на основе рассчитанных сил, с учетом гравитации и ограничений по максимальной силе.
* После завершения итераций полученные позиции узлов обновляются на холсте.
  1. **Пружинный алгоритм для расположения узлов графа:**

Этот алгоритм также основан на физических принципах и использует модель пружин для расположения узлов.

Процесс включает:

* Инициализацию узлов и их случайных позиций.
* Рассмотрение узлов как тел с определенными силами, которые стремятся расположить их в равновесии.
* Обновление позиций узлов на основе взаимодействий между ними в течение заданного количества итераций.
* Корректировка позиций узлов с учетом границ холста.
  1. **Алгоритм Камада-Кавай для расположения узлов графа:**

Этот алгоритм предназначен для вычисления оптимальных позиций узлов в графе, чтобы минимизировать перекрытие и визуализировать структуру графа.

Алгоритм работает следующим образом:

* На вход принимается граф в формате NetworkX.
* Рассчитываются силы, действующие на узлы, с учетом расстояний между ними.
* Узлы перемещаются в соответствии с этими силами до тех пор, пока не будет достигнуто состояние равновесия.
* Результатом работы алгоритма является набор координат для каждого узла, которые затем используются для их визуализации на холсте.

## Выводы и результаты

1. **Интерфейс и удобство:** разработан интуитивно понятный пользовательский интерфейс с инструментальной панелью и цветовой палитрой, что позволяет пользователям легко взаимодействовать с редактором графов. Простота навигации и доступность основных функций делают программу удобной для различных категорий пользователей.
2. **Графические объекты:** программа поддерживает создание и перемещение основных графических объектов, таких как узлы и рёбра. Пользователи могут настраивать атрибуты этих объектов, включая вес рёбер и метки узлов, что способствует более точной визуализации данных.
3. **Форматы файлов:** добавлена поддержка форматов JSON для импорта, экспорта и сохранения графов.
4. **Пользовательские события:** реализована обработка событий мыши, что обеспечивает отзывчивое и эффективное управление графическими объектами. Пользователи могут легко взаимодействовать с графами, перемещая узлы и изменяя их свойства в реальном времени.
5. **Результат:** проект успешно соответствует поставленным задачам, предоставляя удобный инструмент для работы с визуализацией графов. Реализованные алгоритмы укладки, такие как силовая, пружинная и Камада-Кавай, обеспечивают оптимальное размещение узлов и рёбер, что делает графы более наглядными и понятными. Программа также имеет потенциал для дальнейшего расширения функционала, включая добавление новых алгоритмов и возможностей для анализа графов.

# ОПИСАНИЕ РАЗРАБОТАННОГО ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

## Выбор средств разработки и системных программных средств

Для реализации векторного графического редактора был выбран язык Python, так как он предоставляет широкий выбор библиотек для работы с графами и их визуализацией. В частности, PyQt5 была выбрана за:

* Широкие возможности для создания пользовательского интерфейса: PyQt5 позволяет создавать сложные и интуитивно понятные интерфейсы, что важно для пользователей, работающих с графами.
* Удобную работу с графическими объектами через QGraphicsView и QGraphicsScene: Эти компоненты обеспечивают эффективное отображение и взаимодействие с графами, что делает процесс визуализации более наглядным.
* Кроссплатформенность и богатую документацию: PyQt5 поддерживает работу на различных операционных системах, что позволяет пользователям легко интегрировать программу в свою рабочую среду.

## Требования к разрабатываемой программе

#### Функциональные требования:

1. Пользователи должны иметь возможность добавлять, изменять и удалять узлы и рёбра графа.
2. Возможность настройки атрибутов объектов, таких как цвет рёбер и метки узлов.
3. Поддержка сохранения текущего состояния графа в формате JSON и его последующей загрузки.
4. Пользователи должны иметь возможность изменять масштаб отображения графа для более детального просмотра.

#### Нефункциональные требования:

* 1. Интерфейс должен быть простым и понятным, чтобы пользователи могли легко осваивать программу.
  2. Программа должна быть совместима с операционной системой Windows для обеспечения широкой доступности.
  3. Программа должна обеспечивать быструю обработку и визуализацию графов, даже при работе с большими данными.

## Структура программы

### Основные модули

Проект состоит из следующих основных модулей:

Директория core:

* **graph.py** – Основной класс графа, реализующий структуру данных для хранения узлов и рёбер.
* **algorithms.py** – Реализация алгоритмов на графах, включая Дейкстра и Прима.
* **layout.py** – Алгоритмы для изменения расположения узлов, такие как Камада-Кавай, пружинный и силовой алгоритмы.
* **data\_storage.py** – Модуль сериализации и десериализации графов JSON.

Директория ui/:

* **main\_window.py** – Основное окно приложения, реализованное с использованием PyQt.
* **canvas.py** – Логика отрисовки графа.
* **dialogs/**:
  + **node\_dialog.py** – Диалог для добавления и изменения узлов.
  + **edge\_dialog.py** – Диалог для добавления и изменения рёбер.
  + **matrix\_dialog.py** – Диалог для ввода матрицы весов.
* **dialog\_handler.py** – Связка действий интерфейса с бизнес-логикой.
* **menu\_bar.py -** Класс для создания строки меню приложения с использованием PyQt.
* **Tool\_bar.py –** Класс для создания панели инструментов с использованием PyQt.

Директория utils/:

* **file\_operations.py** – Функции для сохранения и загрузки файлов в формате JSON.

Дополнительные элементы:

* **main.py** – Точка входа в приложение.

### Описание основных классов

**Graph:**

* Поля:
  + **Graph:** объект типа nx.Graph, который хранит узлы и рёбра графа
* Методы:
  + **add\_node** –Добавление узла с атрибутами
  + **add\_edge** –Добавление ребра между узлами
  + **remove\_node** – Удаление узла и всех связанных с ним рёбер
  + **remove\_edge** – Удаление ребра между узлами
  + **from\_weight\_matrix** – Создание графа из матрицы весов

**EdgeDialog:**

* Поля:
  + **start\_node:** объект QComboBox, позволяющий выбрать начальный узел для ребра.
  + **end\_node:** объект QComboBox, позволяющий выбрать конечный узел для ребра.
  + **weight\_input:** объект QSpinBox, позволяющий задать вес ребра (в диапазоне от 1 до 100).
  + **ok\_button:** кнопка для подтверждения ввода данных.
  + **cancel\_button:** кнопка для отмены операции.
* Методы:
  + **\_\_init\_\_ –** Инициализирует диалоговое окно для добавления или редактирования ребра.
  + **get\_data –** Возвращает данные, введенные в форму, в виде словаря.

**MatrixDialog:**

* Поля:
  + **node\_count:** количество узлов в графе, определяющее размерность матрицы.
  + **matrix:** двумерный список, представляющий матрицу весов, инициализированный нулями.
  + **table:** объект QTableWidget, отображающий матрицу в виде таблицы для ввода данных.
  + **ok\_button:** кнопка для подтверждения ввода данных.
* Методы:
  + **\_\_init\_\_ –** Инициализирует окно для ввода матрицы весов.
  + **process\_input –** Обрабатывает ввод данных из таблицы и сохраняет их в матрице.
  + **get\_matrix ­–** Возвращает матрицу весов после подтверждения ввода.

**NodeDialog:**

* Поля:
  + **label\_input:** объект QLineEdit, позволяющий вводить метку узла.
  + **color\_button:** кнопка для выбора цвета узла.
  + **color:** строка, представляющая цвет узла (по умолчанию "blue").
  + **random\_color\_button:** кнопка для генерации случайного цвета.
  + **ok\_button:** кнопка для подтверждения ввода данных.
  + **cancel\_button:** кнопка для отмены операции.
* Методы:
  + **\_\_init\_\_ -** Инициализирует окно для добавления или редактирования узла.
  + **select\_color -** Открывает диалог выбора цвета и обновляет стиль кнопки для выбранного цвета.
  + **set\_random\_color -** Генерирует случайный цвет и обновляет стиль кнопки для нового цвета.
  + **get\_data -** Возвращает данные из формы.

**Canvas:**

* Поля:
  + **node\_color:** цвет узлов (по умолчанию серый).
  + **edge\_color:** цвет рёбер (по умолчанию черный).
  + **edge\_thickness:** толщина рёбер (по умолчанию 2).
  + **scale\_factor:** фактор масштабирования (по умолчанию 1.1).
  + **mst\_edge\_color:** цвет рёбер минимального остовного дерева (по умолчанию синий).
  + **shortest\_path\_color:** цвет кратчайших путей (по умолчанию зеленый).
  + **update\_timer:** таймер для обновления графа каждые 20 мс.
  + **scene:** объект QGraphicsScene, представляющий сцену для рисования.
  + **graph:** объект nx.Graph, представляющий граф.
  + **nodes:** словарь для хранения узлов графа.
  + **edges:** словарь для хранения рёбер графа.
  + **edge\_labels:** словарь для хранения меток рёбер.
  + **selected\_node:** выбранный узел (по умолчанию None).
  + **offset:** смещение для перемещения узлов.
* Методы:
  + **\_\_init\_\_ -** Инициализирует холст для визуализации графа.
  + **create\_node -** Создаёт новый узел на холсте.
  + **delete\_node -** Удаляет узел и все связанные с ним рёбра.
  + **update\_node\_position -** Обновляет позицию узла в графе после его перемещения.
  + **create\_edge -** Создаёт новое ребро между двумя узлами.
  + **delete\_edge -** Удаляет ребро между двумя узлами.
  + **update\_edge\_position -** Обновляет позицию ребра.
  + **update\_edge\_label\_position -** Обновляет позицию метки ребра.
  + **update\_edges -** Обновляет позиции всех рёбер.
  + **update\_graph -** Метод, вызываемый таймером для обновления графа.
  + **mouseMoveEvent -** Обрабатывает перемещение мыши для перемещения узлов и обновления рёбер.
  + **mouseReleaseEvent -** Обрабатывает отпускание кнопки мыши, завершает перемещение узла.
  + **sync\_all\_node\_positions -** Синхронизирует позиции всех узлов в графе.
  + **highlight\_mst -** Метод для выделения рёбер минимального остовного дерева.
  + **highlight\_shortest\_paths -** Выделяет кратчайшие пути на графе.
  + **clear\_highlighted\_paths -** Сбрасывает выделение рёбер и узлов.
  + **clear\_graph -** Очищает весь граф и связанные элементы с холста.

**DialogHandler:**

* Поля:
  + **canvas:** объект Canvas, представляющий холст для визуализации графа.
  + **parent:** родительский виджет, к которому привязан диалог.
* Методы:
  + **\_\_init\_\_ -** Инициализирует обработчик диалогов.
  + **create\_action -** Создаёт объект QAction.
  + **add\_node -** Добавляет новый узел в граф.
  + **remove\_node -** Удаляет узел из графа.
  + **add\_edge -** Добавляет новое ребро между узлами.
  + **remove\_edge -** Удаляет ребро из графа.
  + **reset\_color -** Сбрасывает выделение и цвет узлов и рёбер.
  + **add\_graph\_from\_matrix -** Создаёт граф из матрицы смежности.
  + **sync\_all\_node\_positions -** Синхронизирует позиции всех узлов в графе.
  + **save\_graph -** Сохраняет граф в файл.
  + **load\_graph -** Очищает текущий граф, загружает данные и десериализует граф.
  + **delete\_graph -** Очищает граф на холсте.
  + **run\_dijkstra -** Запускает алгоритм Дейкстры.
  + **run\_prim -** Запускает алгоритм Прима для нахождения минимального остовного дерева.
  + **run\_kamada\_kawai -** Запускает алгоритм Камада-Кавай для расположения узлов.
  + **run\_force\_directed -** Запускает силовой метод для расположения узлов.
  + **run\_spring\_layout -** Запускает пружинный алгоритм для расположения узлов.

**MainWindow:**

* Поля:
  + **canvas:** объект Canvas, отображающий граф на центральной панели.
  + **dialog\_handler:** объект DialogHandler, управляющий взаимодействием с пользователем.
  + **menu\_bar:** объект MenuBarCreator, создающий меню приложения.
  + **tool\_bar:** объект ToolBarCreator, создающий панель инструментов.
* Методы:
  + **\_\_init\_\_ -** Инициализирует главное окно приложения.

**MenuBarCreator:**

* Поля:
  + **main\_window:** объект MainWindow, к которому привязан меню.
  + **dialog\_handler:** объект DialogHandler, управляющий действиями в меню.
* Методы:
  + **\_\_init\_\_ -** Инициализирует создатель меню.
  + **create\_menu\_bar -** Создаёт строку меню с основными функциями.

**ToolBarCreator:**

* Поля:
  + **main\_window:** объект MainWindow, к которому привязана панель инструментов.
  + **dialog\_handler:** объект DialogHandler, управляющий действиями на панели инструментов.
* Методы:
  + **\_\_init\_\_ -** Инициализирует создателя панели инструментов.
  + **create\_tool\_bar -** Создаёт панель инструментов для быстрого доступа к функциям.

### Описание основных функций и процедур

* **kamada\_kawai\_layout()** - Рассчитывает расположение узлов по методу Камада-Кавай.
* **force\_directed\_layout() -**  Рассчитывает расположение узлов по силовому методу.
* **spring\_layout() -** Реализует пружинный алгоритм для расположения узлов графа.
* **dijkstra() -** Реализует алгоритм Дейкстры для нахождения кратчайших путей.
* **prim\_mst() -** Реализует алгоритм Прима для нахождения минимального остовного дерева.
* **save\_to\_file() -** Сохраняет данные графа в файл.
* **load\_from\_file() -** Загружает данные графа из файла.

## Описание интерфейса пользователя

Программа предоставляет графический интерфейс с:

1. Строкой меню, содержащей вкладки "Файл", "Алгоритмы" и "Правка".
2. Панелью инструментов, включающей кнопки для добавления/удаления узлов и ребер, а также сброса цвета.
3. Областью для создания и визуализации графов.

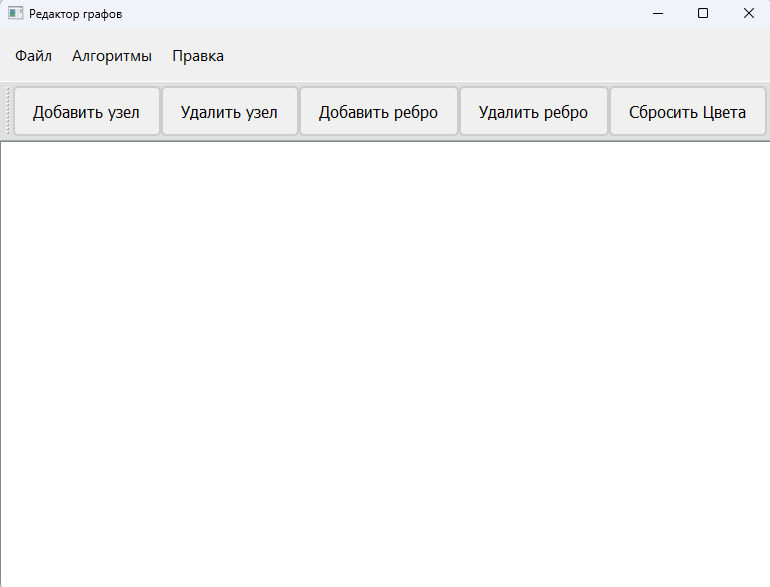


Рисунок 1 - Окно программы

## Контрольный пример и результаты тестирования

Программа протестирована на следующих примерах:

1. Создание графа вручную.

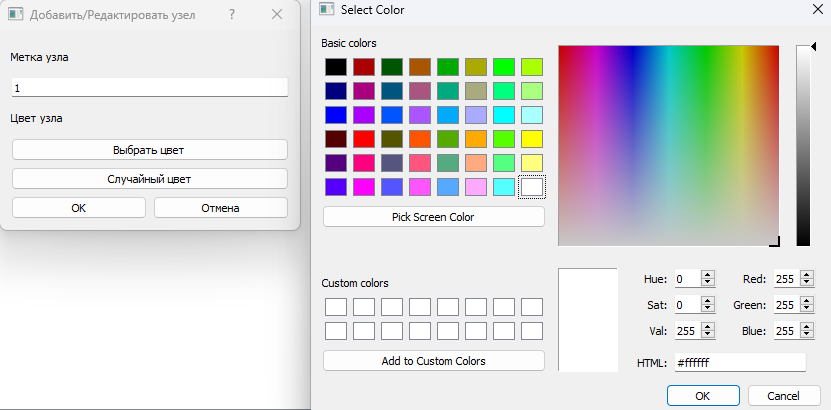


Рисунок 2 - Создание узла

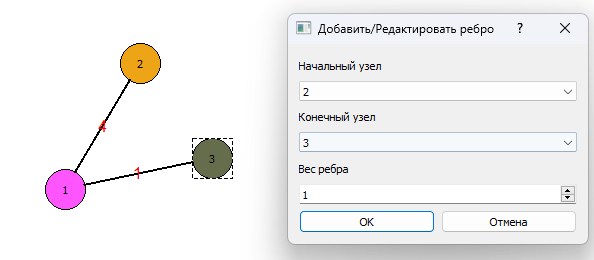


Рисунок 3 – Создание ребра

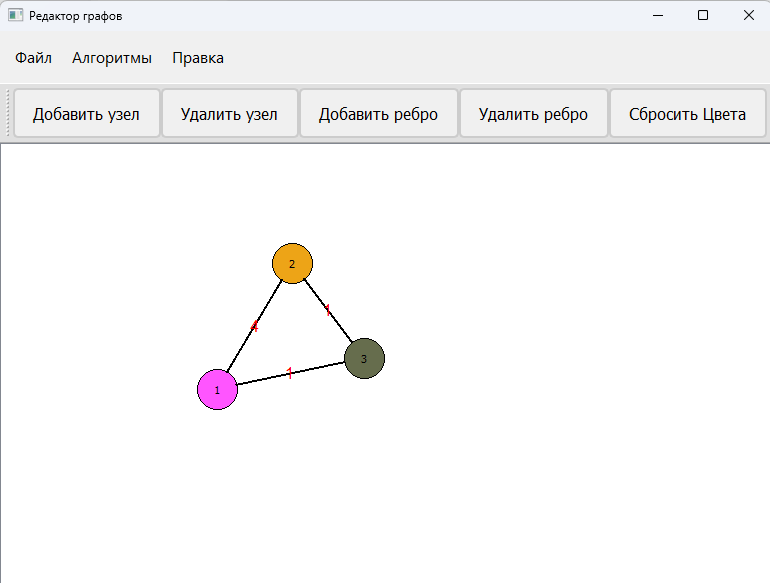


Рисунок 4 – Граф, созданный вручную

1. Создание графа с помощью матрицы весов.

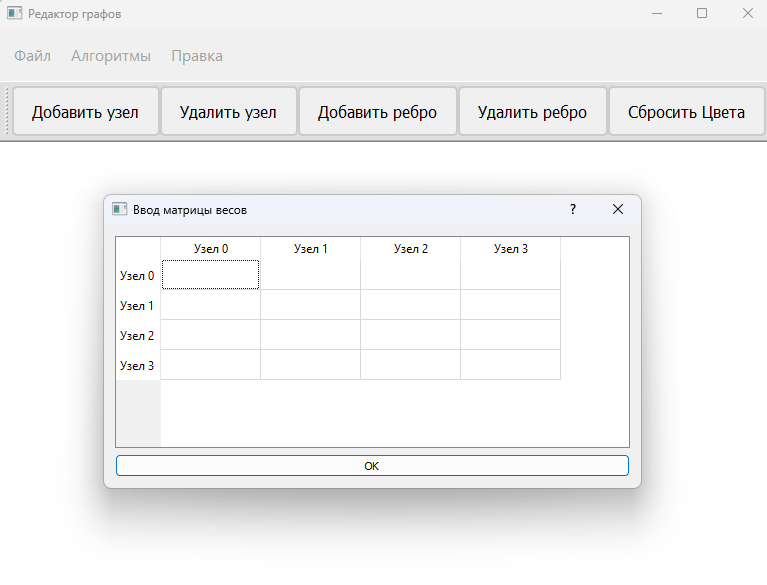


Рисунок 5 – Матрица

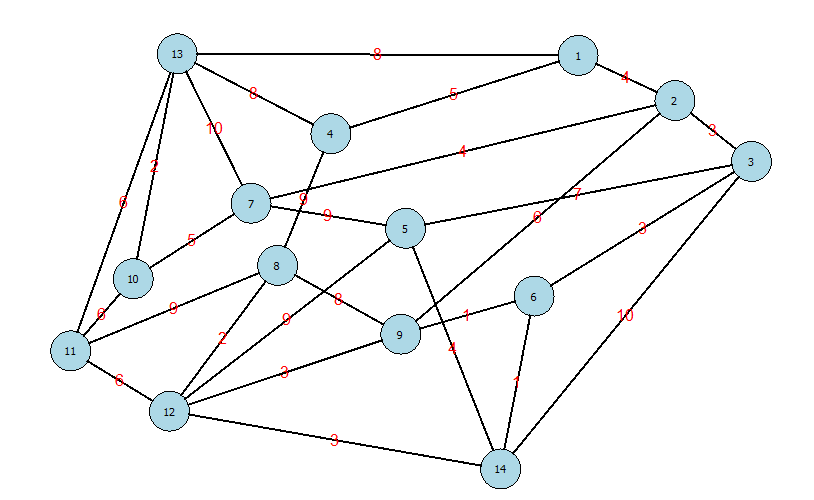


Рисунок 6 – Граф, созданный с помощью матрицы весов

1. Силовой алгоритм.

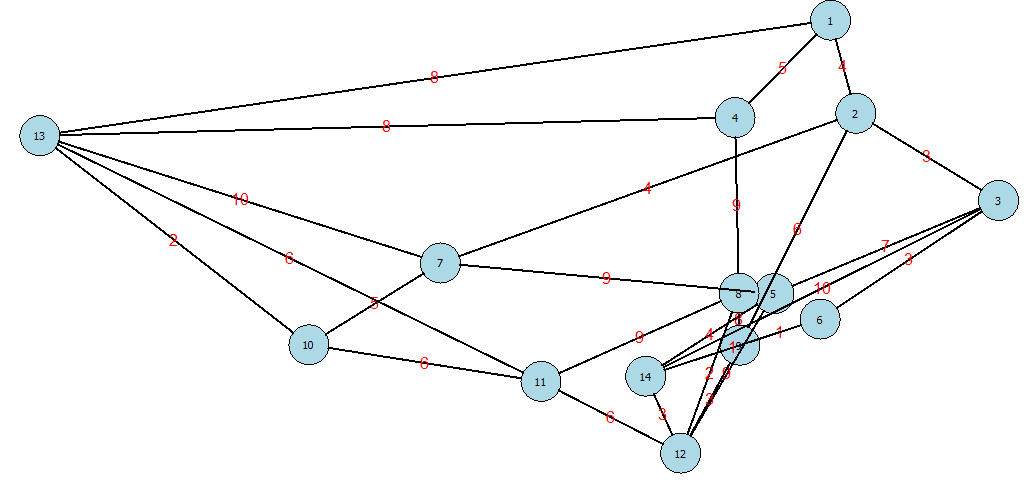


Рисунок 7 – Силовой алгоритм

1. Пружинный алгоритм.

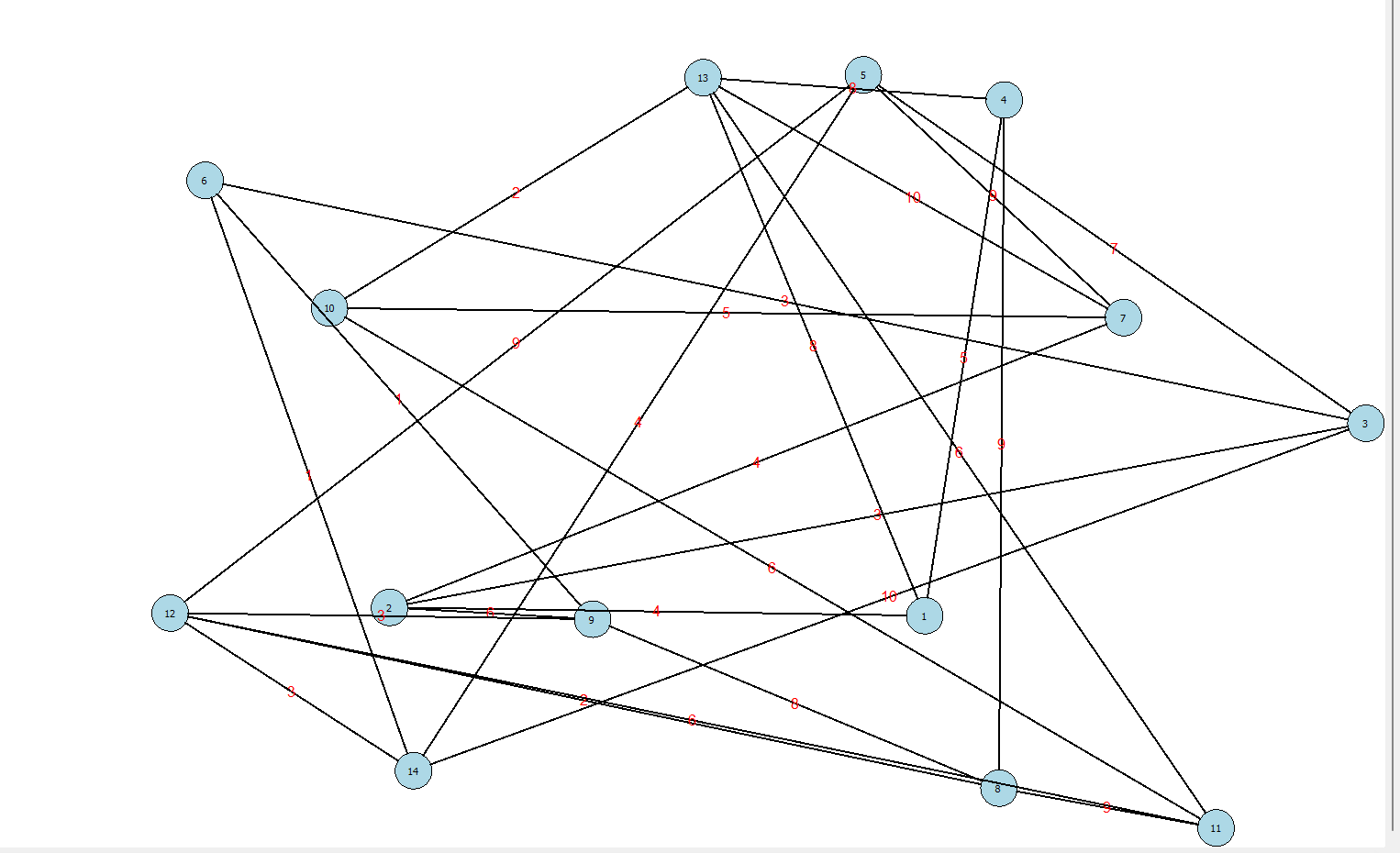


Рисунок 8 – Пружинный алгоритм

1. Алгоритм Камада-Кавай.

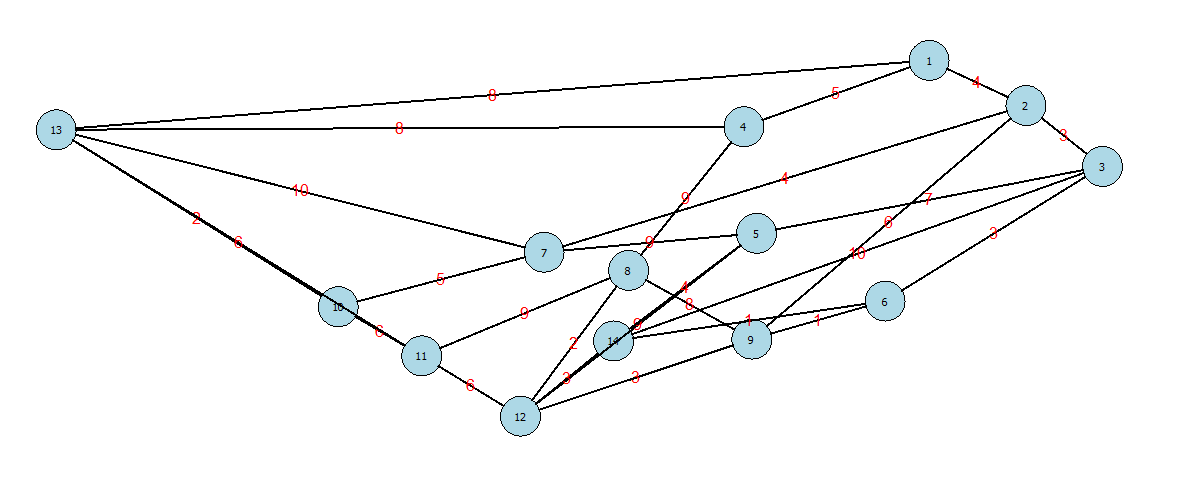


Рисунок 9 – Алгоритм Камада-Кавай

1. Алгоритм Дейкстры.

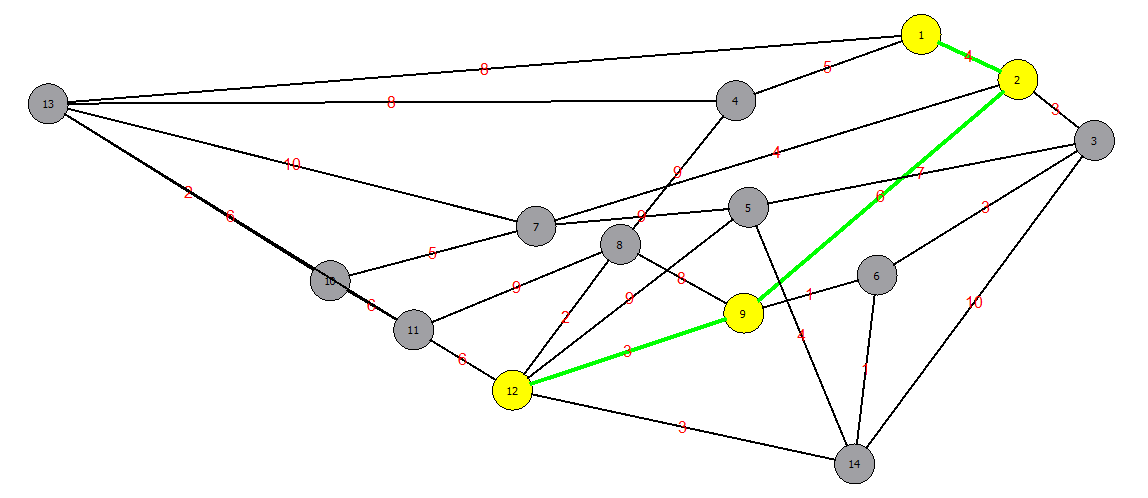


Рисунок 10 – Алгоритм Дейкстры

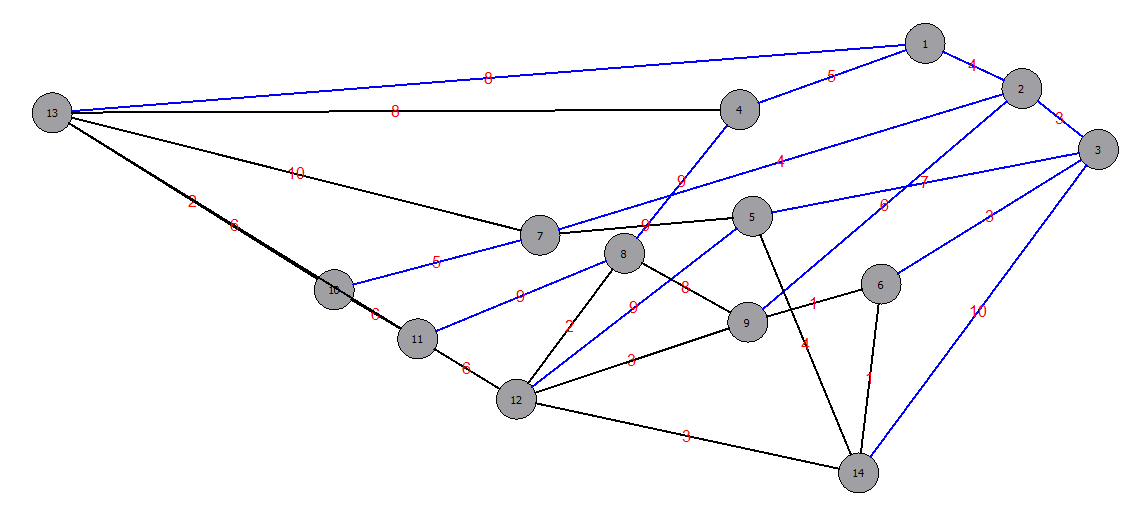
1. Алгоритм Прима.  
   

Рисунок 11 – Алгоритм Прима

## Выводы и результаты

Программа успешно реализована, все заявленные функциональные и нефункциональные требования выполнены.

ЗАКЛЮЧЕНИЕ

В ходе разработки программы для визуализации графов были успешно реализованы основные функциональные возможности, необходимые для эффективной работы с графовыми структурами данных. Программа предоставляет интуитивно понятный графический интерфейс пользователя и поддерживает взаимодействие с графическими объектами. Для оптимизации визуализации графов были реализованы различные алгоритмы укладки, обеспечивающие равномерное распределение узлов и минимизацию перекрытий ребер. Программа также поддерживает импорт, экспорт и сохранение графов в формате JSON, что позволяет пользователям легко обмениваться данными. В целом, разработанная программа представляет собой эффективный инструмент для визуализации и анализа графовых структур данных, который может быть полезен для широкого круга пользователей.

ПРИЛОЖЕНИЕ А

**# core/\_\_init\_\_.py**

from .algorithms import dijkstra, prim\_mst

from .data\_storage import deserialize\_graph, serialize\_graph

from .graph import Graph

from .layout import (

    force\_directed\_layout,

    kamada\_kawai\_layout,

    spring\_layout,

)

**# core/algorithms.py**

import networkx as nx

def dijkstra(graph: nx.Graph, start: str, end: str = None) -> tuple:

    """

    Реализует алгоритм Дейкстры для нахождения кратчайших путей.

    """

    try:

        lengths, paths = nx.single\_source\_dijkstra(graph.graph, source=start)

        if end is not None:

            if end in paths:

                path = paths[end]

                edges = [(path[i], path[i + 1]) for i in range(len(path) - 1)]

                return lengths[end], edges

            else:

                raise ValueError(f"Нет пути до узла {end}.")

        else:

            edges\_dict = {}

            for target, path in paths.items():

                edges = [(path[i], path[i + 1]) for i in range(len(path) - 1)]

                edges\_dict[target] = edges

            return lengths, edges\_dict

    except nx.NetworkXError as e:

        raise ValueError(f"Ошибка алгоритма Дейкстры: {e}")

def prim\_mst(graph: nx.Graph) -> list:

    """

    Реализует алгоритм Прима для нахождения минимального остовного дерева.

    """

    try:

        mst = nx.minimum\_spanning\_tree(graph.graph)

        return list(mst.edges(data=True))

    except nx.NetworkXError as e:

        raise ValueError(f"Ошибка алгоритма Прима: {e}")

**# core/data\_storage.py**

import json

def serialize\_graph(graph):

    """

    Сериализует граф в JSON-формат.

    """

    nodes = []

    for node\_id, data in graph.graph.nodes(data=True):

        position = data.get('position', (0, 0))

        nodes.append({

            "id": node\_id,

            "label": data['label'],

            "color": data['color'],

            "position": position

        })

    edges = []

    for start, end, data in graph.graph.edges(data=True):

        edges.append({

            "start": start,

            "end": end,

            "weight": data['weight']

        })

    return json.dumps({"nodes": nodes, "edges": edges}, indent=4)

def deserialize\_graph(graph, json\_data: str) -> None:

    """

    Восстанавливает граф из JSON-данных.

    """

    data = json.loads(json\_data)

    graph.clear\_graph()

    for node in data["nodes"]:

        graph.create\_node(

            node\_id=node["id"],

            label=node["label"],

            color=node["color"],

            position=tuple(node["position"])

        )

    for edge in data["edges"]:

        graph.create\_edge(

            start=edge["start"],

            end=edge["end"],

            weight=edge["weight"]

        )

**# core/graph.py**

import networkx as nx

import logging

logging.basicConfig(level=logging.INFO)

class Graph:

    def \_\_init\_\_(self) -> None:

        """

        Инициализация графа на основе NetworkX.

        """

        self.graph = nx.Graph()

    def add\_node(self, node\_id: str, label: str = "",

                 color: str = "blue") -> None:

        """

        Добавление узла с атрибутами.

        """

        if node\_id in self.graph.nodes:

            raise ValueError(f"Узел с идентификатором {node\_id} уже существует.")

        self.graph.add\_node(node\_id, label=label, color=color)

        logging.info(f"Узел {node\_id} добавлен: метка={label}, цвет={color}")

    def add\_edge(self, start: str, end: str, weight: float = 1.0, color: str = "black") -> None:

        """

        Добавление ребра между узлами.

        """

        if start not in self.graph.nodes or end not in self.graph.nodes:

            raise ValueError("Оба узла должны существовать в графе.")

        if self.graph.has\_edge(start, end):

            raise ValueError(f"Ребро между {start} и {end} уже существует.")

        self.graph.add\_edge(start, end, weight=weight, color=color)

        logging.info(f"Ребро добавлено: {start} -> {end}, вес={weight}, цвет={color}")

    def remove\_node(self, node\_id: str) -> None:

        """

        Удаление узла и всех связанных с ним рёбер.

        """

        if node\_id not in self.graph.nodes:

            raise ValueError(f"Узел {node\_id} не существует.")

        self.graph.remove\_node(node\_id)

        logging.info(f"Узел {node\_id} и все связанные с ним рёбра удалены.")

    def remove\_edge(self, start: str, end: str) -> None:

        """

        Удаление ребра между узлами.

        """

        if not self.graph.has\_edge(start, end):

            raise ValueError(f"Ребро между {start} и {end} не существует.")

        self.graph.remove\_edge(start, end)

        logging.info(f"Ребро {start} -> {end} удалено.")

    def from\_weight\_matrix(self, matrix: list) -> None:

        """

        Создание графа из матрицы весов.

        """

        self.graph.clear()

        logging.info("Граф очищен перед построением из матрицы.")

        for i in range(len(matrix)):

            self.graph.add\_node(i, label=f"Узел {i}")

        for i, row in enumerate(matrix):

            for j, value in enumerate(row):

                if value not in [0, "-", "0"]:

                    try:

                        weight = float(value)

                        self.graph.add\_edge(i, j, weight=weight)

                        logging.debug(f"Ребро {i} -> {j} добавлено с весом {weight}.")

                    except ValueError:

                        logging.error(f"Некорректное значение матрицы: {value} в позиции ({i}, {j}).")

                        raise ValueError(f"Некорректное значение матрицы: {value} в позиции ({i}, {j})")

**# core/layout.py**

import networkx as nx

import numpy as np

def kamada\_kawai\_layout(graph: nx.Graph) -> dict:

    """

    Рассчитывает расположение узлов по методу Камада-Кавай.

    """

    if not isinstance(graph, nx.Graph):

        raise ValueError("Ожидался объект NetworkX Graph.")

    if not graph.nodes:

        raise ValueError("Граф не содержит узлов.")

    if not graph.edges:

        raise ValueError("Граф не содержит рёбер.")

    layout = nx.kamada\_kawai\_layout(graph)

    if not layout:

        raise ValueError("Алгоритм Камада-Кавай не вернул расположение для узлов.")

    return layout

def force\_directed\_layout(graph: nx.Graph, iterations: int = 50, k: float = 1.0, gravity: float = 0.1) -> dict:

    """

    Рассчитывает расположение узлов по силовому методу.

    """

    if not isinstance(graph, nx.Graph):

        raise ValueError("Ожидался объект NetworkX Graph.")

    if not graph.nodes:

        raise ValueError("Граф не содержит узлов.")

    if not graph.edges:

        raise ValueError("Граф не содержит рёбер.")

    pos = {node: np.random.rand(2) for node in graph.nodes}

    for \_ in range(iterations):

        new\_pos = {node: pos[node].copy() for node in graph.nodes}

        for node in graph.nodes:

            force = np.zeros(2)

            for neighbor in graph.neighbors(node):

                diff = pos[neighbor] - pos[node]

                distance = max(np.linalg.norm(diff), 1e-9)

                force += diff / distance

            for other\_node in graph.nodes:

                if other\_node == node:

                    continue

                diff = pos[node] - pos[other\_node]

                distance = max(np.linalg.norm(diff), 1e-9)

                force += diff / (distance \*\* 2)

            force -= pos[node] \* gravity

            max\_force = 10.0

            if np.linalg.norm(force) > max\_force:

                force = force / np.linalg.norm(force) \* max\_force

            new\_pos[node] += force \* k

        center = np.mean(list(new\_pos.values()), axis=0)

        for node in new\_pos:

            new\_pos[node] -= center

        pos = new\_pos

    return pos

def spring\_layout(graph: nx.Graph, canvas) -> None:

    """

    Реализует пружинный алгоритм для расположения узлов графа.

    """

    screen\_size = canvas.scene.width(), canvas.scene.height()

    screen\_w, screen\_h = screen\_size

    const\_charge = 100000

    const\_spring = 0.001

    spring\_equ\_len = 10

    max\_iterations = 200

    dt = 0.1

    bodies = []

    for node\_id in graph.nodes():

        pos = np.random.random(2) \* 0.8 \* np.array([screen\_w, screen\_h]) + 0.1 \* np.array([screen\_w, screen\_h])

        bodies.append({'id': node\_id, 'pos': pos, 'vel': np.zeros(2)})

    def edge\_exists(body\_a, body\_b) -> bool:

        return graph.has\_edge(body\_a['id'], body\_b['id'])

    def unit(v: np.ndarray) -> np.ndarray:

        norm = np.linalg.norm(v)

        return v / norm if norm > 1e-9 else np.zeros\_like(v)

    def update\_body\_physics(bods, dt: float) -> None:

        for body\_a in bods:

            f\_net = np.zeros(2)

            for body\_b in bods:

                if body\_a['id'] == body\_b['id']:

                    continue

                ab = body\_b['pos'] - body\_a['pos']

                \_ab = max(np.linalg.norm(ab), 1e-9)

                f\_s = const\_spring \* unit(ab) \* (\_ab - spring\_equ\_len) if edge\_exists(body\_a, body\_b) else 0

                f\_c = const\_charge \* unit(ab) / \_ab\*\*2

                f\_net += f\_s - f\_c

            body\_a['vel'] += f\_net \* dt

            body\_a['vel'] \*= 0.99

            body\_a['pos'] += body\_a['vel'] \* dt

            body\_a['pos'] = np.clip(body\_a['pos'], 0, [screen\_w, screen\_h])

    for \_ in range(max\_iterations):

        update\_body\_physics(bodies, dt)

    p\_avg = np.mean([body['pos'] for body in bodies], axis=0)

    max\_pos = np.max([body['pos'] for body in bodies], axis=0)

    min\_pos = np.min([body['pos'] for body in bodies], axis=0)

    scale = min(screen\_w / (max\_pos[0] - min\_pos[0] + 1e-9), screen\_h / (max\_pos[1] - min\_pos[1] + 1e-9))

    for body in bodies:

        node\_item = canvas.nodes[body['id']][0]

        pos = (body['pos'] - p\_avg) \* scale + np.array(screen\_size) / 2

        pos[1] = screen\_h - pos[1]

        node\_item.setPos(pos[0], pos[1])

**# ui/dialogs/\_\_init\_\_.py**

from .node\_dialog import NodeDialog

from .edge\_dialog import EdgeDialog

from .matrix\_dialog import MatrixDialog

**# ui/dialogs/edge\_dialog.py**

from PyQt5.QtWidgets import (

    QDialog, QVBoxLayout, QHBoxLayout, QLabel, QPushButton,

    QSpinBox, QComboBox

)

class EdgeDialog(QDialog):

    """

    Диалоговое окно для добавления или редактирования ребра графа.

    Атрибуты:

        start\_node (QComboBox): Поле выбора начального узла.

        end\_node (QComboBox): Поле выбора конечного узла.

        weight\_input (QSpinBox): Поле ввода веса ребра.

        ok\_button (QPushButton): Кнопка подтверждения.

        cancel\_button (QPushButton): Кнопка отмены.

    """

    def \_\_init\_\_(self, nodes: list, parent=None):

        """

        Инициализация диалогового окна.

        :param nodes: Список узлов графа для выбора.

        :param parent: Родительский виджет.

        """

        super().\_\_init\_\_(parent)

        self.setWindowTitle("Добавить/Редактировать ребро")

        self.setFixedSize(300, 200)

        layout = QVBoxLayout()

        layout.addWidget(QLabel("Начальный узел"))

        self.start\_node = QComboBox()

        self.start\_node.addItems(nodes)

        layout.addWidget(self.start\_node)

        layout.addWidget(QLabel("Конечный узел"))

        self.end\_node = QComboBox()

        self.end\_node.addItems(nodes)

        layout.addWidget(self.end\_node)

        self.weight\_input = QSpinBox()

        self.weight\_input.setRange(1, 100)

        layout.addWidget(QLabel("Вес ребра"))

        layout.addWidget(self.weight\_input)

        button\_layout = QHBoxLayout()

        self.ok\_button = QPushButton("OK")

        self.ok\_button.clicked.connect(self.accept)

        self.cancel\_button = QPushButton("Отмена")

        self.cancel\_button.clicked.connect(self.reject)

        button\_layout.addWidget(self.ok\_button)

        button\_layout.addWidget(self.cancel\_button)

        layout.addLayout(button\_layout)

        self.setLayout(layout)

    def get\_data(self) -> dict:

        """

        Возвращает данные, введенные в форму.

        :return: Словарь с данными:

            - start: Начальный узел.

            - end: Конечный узел.

            - weight: Вес ребра.

        """

        return {

            "start": self.start\_node.currentText().strip(),

            "end": self.end\_node.currentText().strip(),

            "weight": self.weight\_input.value(),

        }

from PyQt5.QtWidgets import (

    QDialog, QVBoxLayout, QTableWidget, QPushButton,

    QMessageBox

)

class MatrixDialog(QDialog):

    """

    Диалоговое окно для ввода матрицы весов графа.

    """

    def \_\_init\_\_(self, node\_count: int, parent=None):

        """

        Инициализация окна для ввода матрицы весов.

        """

        super().\_\_init\_\_(parent)

        self.node\_count = node\_count

        self.matrix = [[0] \* node\_count for \_ in range(node\_count)]

        self.setWindowTitle("Ввод матрицы весов")

        layout = QVBoxLayout(self)

        self.table = QTableWidget(node\_count, node\_count, self)

        self.table.setHorizontalHeaderLabels([f"Узел {i}" for i in range(node\_count)])

        self.table.setVerticalHeaderLabels([f"Узел {i}" for i in range(node\_count)])

        layout.addWidget(self.table)

        self.ok\_button = QPushButton("OK", self)

        self.ok\_button.clicked.connect(self.process\_input)

        layout.addWidget(self.ok\_button)

    def process\_input(self):

        """

        Обрабатывает ввод данных из таблицы и сохраняет в матрице.

        """

        try:

            for i in range(self.node\_count):

                for j in range(self.node\_count):

                    item = self.table.item(i, j)

                    if item is None or item.text().strip() in ["", "-"]:

                        self.matrix[i][j] = 0

                    else:

                        self.matrix[i][j] = int(item.text().strip())

            self.accept()

        except ValueError as e:

            QMessageBox.critical(self, "Ошибка", f"Ошибка ввода данных: {e}")

    def get\_matrix(self) -> list:

        """

        Возвращает матрицу весов после подтверждения.

        """

        return self.matrix

**# ui/dialogs/node\_dialog.py**

import random

from PyQt5.QtWidgets import (

    QDialog, QVBoxLayout, QHBoxLayout, QLabel, QLineEdit, QPushButton,

    QColorDialog

)

from PyQt5.QtGui import QColor

class NodeDialog(QDialog):

    """

    Диалоговое окно для добавления или редактирования узла графа.

    """

    def \_\_init\_\_(self, parent=None):

        """

        Инициализация окна для добавления/редактирования узла.

        """

        super().\_\_init\_\_(parent)

        self.setWindowTitle("Добавить/Редактировать узел")

        self.setFixedSize(300, 200)

        layout = QVBoxLayout()

        self.label\_input = QLineEdit()

        layout.addWidget(QLabel("Метка узла"))

        layout.addWidget(self.label\_input)

        self.color\_button = QPushButton("Выбрать цвет")

        self.color\_button.clicked.connect(self.select\_color)

        self.color = "blue"

        layout.addWidget(QLabel("Цвет узла"))

        layout.addWidget(self.color\_button)

        self.random\_color\_button = QPushButton("Случайный цвет")

        self.random\_color\_button.clicked.connect(self.set\_random\_color)

        layout.addWidget(self.random\_color\_button)

        button\_layout = QHBoxLayout()

        self.ok\_button = QPushButton("OK")

        self.ok\_button.clicked.connect(self.accept)

        self.cancel\_button = QPushButton("Отмена")

        self.cancel\_button.clicked.connect(self.reject)

        button\_layout.addWidget(self.ok\_button)

        button\_layout.addWidget(self.cancel\_button)

        layout.addLayout(button\_layout)

        self.setLayout(layout)

    def select\_color(self):

        """

        Открывает диалог выбора цвета и обновляет стиль кнопки для выбранного цвета.

        """

        color = QColorDialog.getColor()

        if color.isValid():

            self.color = color.name()

            self.color\_button.setStyleSheet(f"background-color: {self.color}")

    def set\_random\_color(self):

        """

        Генерирует случайный цвет и обновляет стиль кнопки для нового цвета.

        """

        random\_color = QColor(random.randint(0, 255), random.randint(0, 255), random.randint(0, 255))

        self.color = random\_color.name()

        self.color\_button.setStyleSheet(f"background-color: {self.color}")

    def get\_data(self) -> dict:

        """

        Возвращает данные из формы.

        """

        label = self.label\_input.text().strip()

        return {

            "id": label,

            "label": label,

            "color": self.color,

        }

**# ui/canvas.py**

from PyQt5.QtWidgets import (

    QGraphicsView, QGraphicsScene, QGraphicsEllipseItem,

    QGraphicsLineItem, QGraphicsTextItem

)

from PyQt5.QtGui import QBrush, QPen, QColor, QFont

from PyQt5.QtCore import Qt, QPointF, QTimer

from math import atan2, cos, sin

import networkx as nx

class Canvas(QGraphicsView):

    """Класс Canvas для визуализации и взаимодействия с графом."""

    def \_\_init\_\_(self, parent=None):

        super().\_\_init\_\_(parent)

        self.node\_color = Qt.gray

        self.edge\_color = Qt.black

        self.edge\_thickness = 2

        self.scale\_factor = 1.1

        self.mst\_edge\_color = Qt.blue

        self.shortest\_path\_color = QColor(0, 255, 0)

        self.update\_timer = QTimer(self)

        self.update\_timer.timeout.connect(self.update\_graph)

        self.update\_timer.start(20)

        self.scene = QGraphicsScene(self)

        self.setScene(self.scene)

        self.graph = nx.Graph()

        self.nodes = {}

        self.edges = {}

        self.edge\_labels = {}

        self.selected\_node = None

        self.offset = QPointF()

    def create\_node(self, node\_id: str, label: str, color: str = "blue", position: tuple[float, float] = None):

        """Создаёт новый узел на холсте."""

        if node\_id in self.nodes:

            raise ValueError(f"Node with ID {node\_id} already exists.")

        if position is None:

            position = (50 \* len(self.nodes), 50)

        self.graph.add\_node(node\_id, label=label, color=color, position=position)

        radius = 20

        x, y = position

        ellipse = QGraphicsEllipseItem(x, y, radius \* 2, radius \* 2)

        ellipse.setBrush(QBrush(QColor(color)))

        ellipse.setFlag(QGraphicsEllipseItem.ItemIsMovable)

        ellipse.setData(0, node\_id)

        ellipse.setAcceptHoverEvents(True)

        ellipse.setFlag(QGraphicsEllipseItem.ItemIsSelectable)

        text = QGraphicsTextItem(label)

        text.setParentItem(ellipse)

        text.setDefaultTextColor(Qt.black)

        text.setPos(

            ellipse.rect().center().x() - text.boundingRect().width() / 2,

            ellipse.rect().center().y() - text.boundingRect().height() / 2

        )

        self.scene.addItem(ellipse)

        self.nodes[node\_id] = (ellipse, text)

        print(f"Node {node\_id} created at position {position}")

    def delete\_node(self, node\_id: str):

        """Удаляет узел и все связанные с ним рёбра."""

        if node\_id not in self.nodes:

            return

        edges\_to\_delete = [edge\_key for edge\_key in self.edges if node\_id in edge\_key]

        for edge\_key in edges\_to\_delete:

            self.delete\_edge(\*edge\_key)

        self.graph.remove\_node(node\_id)

        node, label = self.nodes.pop(node\_id)

        self.scene.removeItem(node)

    def update\_node\_position(self, node\_id: str):

        """Обновляет позицию узла в графе после его перемещения."""

        if node\_id not in self.nodes:

            raise ValueError(f"Node {node\_id} does not exist.")

        node = self.nodes[node\_id][0]

        position = node.pos()

        self.graph.nodes[node\_id]['position'] = (position.x(), position.y())

        print(f"Позиция узла {node\_id} обновлена на ({position.x()}, {position.y()})")

    def create\_edge(self, start: str, end: str, weight: int = 1):

        """Создаёт новое ребро между двумя узлами."""

        if not (start in self.nodes and end in self.nodes):

            raise ValueError("Both nodes must exist to create an edge.")

        if self.graph.has\_edge(start, end) or self.graph.has\_edge(end, start):

            raise ValueError(f"Edge between {start} and {end} already exists.")

        self.graph.add\_edge(start, end, weight=weight)

        start\_node = self.nodes[start][0]

        end\_node = self.nodes[end][0]

        if start == end:

            rect = start\_node.rect().adjusted(15, 15, -15, -15)

            edge = QGraphicsEllipseItem(rect)

            edge.setPen(QPen(self.edge\_color, self.edge\_thickness))

        else:

            edge = QGraphicsLineItem()

            self.update\_edge\_position(edge, start\_node, end\_node)

            edge.setPen(QPen(self.edge\_color, self.edge\_thickness))

        edge.setData(0, start)

        edge.setData(1, end)

        self.scene.addItem(edge)

        self.edges[(start, end)] = edge

        label = QGraphicsTextItem(str(weight))

        label.setDefaultTextColor(Qt.red)

        label.setFont(QFont("Arial", 12))

        self.scene.addItem(label)

        self.edge\_labels[(start, end)] = label

        self.update\_edge\_label\_position(edge, label, start\_node, end\_node)

        print(f"Ребро между {start} и {end} с весом {weight} добавлено.")

    def delete\_edge(self, start: str, end: str):

        """Удаляет ребро между двумя узлами."""

        if not self.graph.has\_edge(start, end):

            return

        self.graph.remove\_edge(start, end)

        edge = self.edges.pop((start, end), None)

        if edge:

            self.scene.removeItem(edge)

        label = self.edge\_labels.pop((start, end), None)

        if label:

            self.scene.removeItem(label)

    def update\_edge\_position(self, edge: QGraphicsLineItem, start\_node: QGraphicsEllipseItem, end\_node: QGraphicsEllipseItem):

        """Обновляет позицию ребра."""

        if isinstance(edge, QGraphicsLineItem):

            start\_center = start\_node.scenePos() + start\_node.rect().center()

            end\_center = end\_node.scenePos() + end\_node.rect().center()

            angle = atan2(end\_center.y() - start\_center.y(), end\_center.x() - start\_center.x())

            start\_offset = QPointF(cos(angle) \* start\_node.rect().width() / 2, sin(angle) \* start\_node.rect().height() / 2)

            end\_offset = QPointF(cos(angle + 3.14) \* end\_node.rect().width() / 2, sin(angle + 3.14) \* end\_node.rect().height() / 2)

            edge.setLine(

                start\_center.x() + start\_offset.x(),

                start\_center.y() + start\_offset.y(),

                end\_center.x() + end\_offset.x(),

                end\_center.y() + end\_offset.y()

            )

    def update\_edge\_label\_position(self, edge: QGraphicsLineItem, label: QGraphicsTextItem, start\_node: QGraphicsEllipseItem, end\_node: QGraphicsEllipseItem):

        """Обновляет позицию метки ребра."""

        if isinstance(edge, QGraphicsLineItem):

            line = edge.line()

            midpoint = QPointF(

                (line.x1() + line.x2()) / 2,

                (line.y1() + line.y2()) / 2

            )

            label.setPos(midpoint - QPointF(label.boundingRect().width() / 2, label.boundingRect().height() / 2))

    def update\_edges(self):

        """Обновляет позиции всех рёбер."""

        for (start, end), edge in self.edges.items():

            start\_node = self.nodes[start][0]

            end\_node = self.nodes[end][0]

            self.update\_edge\_position(edge, start\_node, end\_node)

            self.update\_edge\_label\_position(edge, self.edge\_labels[(start, end)], start\_node, end\_node)

    def update\_graph(self):

        """Метод, вызываемый таймером для обновления графа."""

        self.update\_edges()

        self.scene.update()

    def mouseMoveEvent(self, event):

        """

        Обрабатывает перемещение мыши.

        Перемещает захваченный узел и обновляет связанные рёбра.

        """

        if self.selected\_node:

            new\_pos = self.mapToScene(event.pos()) - self.offset

            self.selected\_node.setPos(new\_pos)

            node\_id = self.selected\_node.data(0)

            for (start, end), edge in self.edges.items():

                if start == node\_id or end == node\_id:

                    start\_node = self.nodes[start][0]

                    end\_node = self.nodes[end][0]

                    self.update\_edge\_position(edge, start\_node, end\_node)

                    self.update\_edge\_label\_position(edge, self.edge\_labels[(start, end)], start\_node, end\_node)

            self.scene.update()

        super().mouseMoveEvent(event)

    def mouseReleaseEvent(self, event):

        """

        Обрабатывает отпускание ЛКМ.

        Завершает перемещение узла.

        """

        self.selected\_node = None

        super().mouseReleaseEvent(event)

    def sync\_all\_node\_positions(self):

        """Синхронизирует позиции всех узлов в графе."""

        for node\_id, (node\_item, \_) in self.nodes.items():

            self.update\_node\_position(node\_id)

    def highlight\_mst(self, mst\_edges):

        """

        Метод для выделения рёбер минимального остовного дерева (MST).

        :param mst\_edges: Список рёбер минимального остовного дерева.

        """

        self.clear\_highlighted\_paths()

        for edge in mst\_edges:

            if isinstance(edge, tuple) and len(edge) == 3:

                start, end, \_ = edge

                if (start, end) in self.edges:

                    edge\_item = self.edges[(start, end)]

                    edge\_item.setPen(QPen(self.mst\_edge\_color, self.edge\_thickness))

                elif (end, start) in self.edges:

                    edge\_item = self.edges[(end, start)]

                    edge\_item.setPen(QPen(self.mst\_edge\_color, self.edge\_thickness))

            else:

                print(f"Некорректный элемент в mst\_edges: {edge}")

        self.scene.update()

    def highlight\_shortest\_paths(self, distances, paths):

        """Выделяет кратчайшие пути на графе."""

        self.clear\_highlighted\_paths()

        for path in paths:

            for i in range(len(path) - 1):

                start = path[i]

                end = path[i + 1]

                edge = self.edges.get((start, end)) or self.edges.get((end, start))

                if edge:

                    edge.setPen(QPen(self.shortest\_path\_color, self.edge\_thickness \* 2))

                start\_node, \_ = self.nodes.get(start, (None, None))

                end\_node, \_ = self.nodes.get(end, (None, None))

                if start\_node:

                    start\_node.setBrush(QBrush(Qt.yellow))

                if end\_node:

                    end\_node.setBrush(QBrush(Qt.yellow))

        self.repaint()

    def clear\_highlighted\_paths(self):

        """Сбрасывает выделение рёбер и узлов."""

        for edge in self.edges.values():

            edge.setPen(QPen(self.edge\_color, self.edge\_thickness))

        for node\_id, (ellipse, \_) in self.nodes.items():

            ellipse.setBrush(QBrush(self.node\_color))

        self.repaint()

    def clear\_graph(self):

        """Очищает весь граф и связанные элементы с холста."""

        for edge in list(self.edges.values()):

            self.scene.removeItem(edge)

        for label in self.edge\_labels.values():

            self.scene.removeItem(label)

        for node, (ellipse, \_) in self.nodes.items():

            self.scene.removeItem(ellipse)

        self.graph.clear()

        self.nodes.clear()

        self.edges.clear()

        self.edge\_labels.clear()

        self.scene.update()

**#ui/dialog\_handler.py**

import time

from PyQt5.QtWidgets import (

    QInputDialog, QMessageBox, QAction, QDialog

)

import networkx as nx

from core import (

    dijkstra, prim\_mst, kamada\_kawai\_layout,

    serialize\_graph, deserialize\_graph,

    force\_directed\_layout, spring\_layout

)

from utils.file\_operations import save\_to\_file, load\_from\_file

from .dialogs import NodeDialog, EdgeDialog, MatrixDialog

class DialogHandler:

    """

    Класс для обработки диалогов и взаимодействия с пользователем.

    """

    def \_\_init\_\_(self, canvas, parent):

        self.canvas = canvas

        self.parent = parent

    def create\_action(self, name, callback):

        """

        Создаёт объект QAction.

        """

        action = QAction(name, self.parent)

        action.triggered.connect(callback)

        return action

    def add\_node(self):

        """Добавляет новый узел в граф."""

        dialog = NodeDialog(self.parent)

        if dialog.exec\_() == QDialog.Accepted:

            node\_data = dialog.get\_data()

            try:

                self.canvas.create\_node(node\_data["id"], node\_data["label"], node\_data["color"])

            except ValueError as e:

                QMessageBox.warning(self.parent, "Ошибка", str(e))

    def remove\_node(self):

        """Удаляет узел из графа."""

        node\_id, ok = QInputDialog.getText(self.parent, "Удаление узла", "Введите ID узла для удаления:")

        if ok:

            try:

                self.canvas.delete\_node(node\_id)

            except ValueError as e:

                QMessageBox.warning(self.parent, "Ошибка", str(e))

    def add\_edge(self):

        """Добавляет новое ребро между узлами."""

        node\_ids = list(self.canvas.nodes.keys())

        if not node\_ids:

            QMessageBox.warning(self.parent, "Ошибка", "Сначала добавьте хотя бы два узла.")

            return

        dialog = EdgeDialog(node\_ids, self.parent)

        if dialog.exec\_() == QDialog.Accepted:

            edge\_data = dialog.get\_data()

            try:

                self.canvas.create\_edge(edge\_data["start"], edge\_data["end"], edge\_data["weight"])

            except ValueError as e:

                QMessageBox.warning(self.parent, "Ошибка", str(e))

    def remove\_edge(self):

        """Удаляет ребро из графа."""

        start, ok1 = QInputDialog.getText(self.parent, "Удаление ребра", "Введите ID начального узла:")

        if ok1:

            end, ok2 = QInputDialog.getText(self.parent, "Удаление ребра", "Введите ID конечного узла:")

            if ok2:

                try:

                    self.canvas.delete\_edge(start, end)

                except ValueError as e:

                    QMessageBox.warning(self.parent, "Ошибка", str(e))

    def reset\_color(self):

        """Сбрасывает выделение и цвет узлов и рёбер."""

        self.canvas.clear\_highlighted\_paths()

        self.canvas.update()

    def add\_graph\_from\_matrix(self):

        """Создаёт граф из матрицы смежности."""

        try:

            node\_count, ok = QInputDialog.getInt(self.parent, "Количество узлов", "Введите количество узлов:")

            if not ok or node\_count <= 0:

                return

            dialog = MatrixDialog(node\_count, self.parent)

            if dialog.exec() == QDialog.Accepted:

                matrix = dialog.matrix

                self.canvas.clear\_graph()

                for i in range(node\_count):

                    self.canvas.create\_node(str(i + 1), f"{i + 1}", "#ADD8E6")

                for i in range(node\_count):

                    for j in range(i + 1, node\_count):

                        if matrix[i][j] != 0:

                            if self.canvas.graph.has\_edge(str(i + 1), str(j + 1)):

                                QMessageBox.warning(self.parent, "Ошибка", f"Ребро между {i + 1} и {j + 1} уже существует.")

                            else:

                                try:

                                    weight = int(matrix[i][j])

                                    self.canvas.create\_edge(str(i + 1), str(j + 1), weight)

                                except ValueError as e:

                                    QMessageBox.warning(self.parent, "Ошибка", f"Некорректное значение ребра ({i + 1}, {j + 1}): {e}")

                QMessageBox.information(self.parent, "Матрица весов", "Граф успешно создан.")

        except Exception as e:

            QMessageBox.critical(self.parent, "Ошибка", f"Ошибка создания графа: {e}")

    def sync\_all\_node\_positions(self):

        """Синхронизирует позиции всех узлов в графе."""

        for node\_id, (node\_item, \_) in self.canvas.nodes.items():

            self.canvas.update\_node\_position(node\_id)

    def save\_graph(self):

        """Сохраняет граф в файл."""

        self.sync\_all\_node\_positions()

        graph\_data = serialize\_graph(self.canvas)

        save\_to\_file(self.parent, graph\_data)

    def load\_graph(self):

        """Загружает граф из файла."""

        self.delete\_graph()

        graph\_data = load\_from\_file(self.parent)

        if not graph\_data:

            return

        try:

            deserialize\_graph(self.canvas, graph\_data)

            self.canvas.update()

            QMessageBox.information(self.parent, "Загрузка", "Граф успешно загружен.")

        except Exception as e:

            QMessageBox.critical(self.parent, "Ошибка", f"Ошибка загрузки графа: {e}")

    def delete\_graph(self):

        """Очищает граф на холсте."""

        self.canvas.clear\_graph()

        self.canvas.update()

    def run\_dijkstra(self):

        """Запускает алгоритм Дейкстры."""

        start\_node, ok = QInputDialog.getText(self.parent, "Алгоритм Дейкстры", "Введите начальный узел:")

        if ok and start\_node in self.canvas.nodes:

            end\_node, ok = QInputDialog.getText(self.parent, "Алгоритм Дейкстры", "Введите конечный узел:")

            if ok and end\_node in self.canvas.nodes:

                distance, path = dijkstra(self.canvas, start\_node, end\_node)

                print(f"Distance: {distance}")

                print(f"Path: {path}")

                self.canvas.highlight\_shortest\_paths(distance, path)

            else:

                QMessageBox.warning(self.parent, "Ошибка", "Конечный узел не существует или не был введён.")

    def run\_prim(self):

        """Запускает алгоритм Прима для нахождения минимального остовного дерева."""

        mst\_edges = prim\_mst(self.canvas)

        self.canvas.highlight\_mst(mst\_edges)

    def run\_kamada\_kawai(self):

        """Запускает алгоритм Камада-Кавай для расположения узлов."""

        try:

            if not self.canvas.graph or not nx.is\_connected(self.canvas.graph):

                QMessageBox.warning(self.parent, "Ошибка", "Граф должен быть связным для выполнения алгоритма Камада-Кавай.")

                return

            self.canvas.graph.clear()

            for node\_id in self.canvas.nodes:

                self.canvas.graph.add\_node(node\_id)

            for (start, end), edge in self.canvas.edges.items():

                self.canvas.graph.add\_edge(start, end)

            start\_time = time.time()

            layout = kamada\_kawai\_layout(self.canvas.graph)

            if not layout:

                QMessageBox.warning(self.parent, "Ошибка", "Алгоритм Камада-Кавай не вернул расположение для узлов.")

                return

            end\_time = time.time()

            elapsed\_time = end\_time - start\_time

            max\_x = max(x for x, \_ in layout.values())

            max\_y = max(y for \_, y in layout.values())

            for node\_id, (x, y) in layout.items():

                scaled\_x = (x / max\_x) \* 100

                scaled\_y = (y / max\_y) \* 100

                self.canvas.update\_node\_position(node\_id)

                node\_item = self.canvas.nodes[node\_id][0]

                node\_item.setPos(scaled\_x, scaled\_y)

            self.canvas.scene.update()

            QMessageBox.information(self.parent, "Камада-Кавай", f"Расположение узлов выполнено за {elapsed\_time:.4f} секунд.")

        except Exception as e:

            QMessageBox.critical(self.parent, "Ошибка", f"Ошибка алгоритма Камада-Кавай: {e}")

    def run\_force\_directed(self):

        """Запускает силовой метод для расположения узлов."""

        try:

            if not self.canvas.graph or not nx.is\_connected(self.canvas.graph):

                QMessageBox.warning(self.parent, "Ошибка", "Граф должен быть связным для выполнения силового метода.")

                return

            self.canvas.graph.clear()

            for node\_id in self.canvas.nodes:

                self.canvas.graph.add\_node(node\_id)

            for (start, end), edge in self.canvas.edges.items():

                self.canvas.graph.add\_edge(start, end)

            start\_time = time.time()

            layout = force\_directed\_layout(self.canvas.graph)

            if not layout:

                QMessageBox.warning(self.parent, "Ошибка", "Силовой метод не вернул расположение для узлов.")

                return

            end\_time = time.time()

            elapsed\_time = end\_time - start\_time

            max\_x = max(x for x, \_ in layout.values())

            max\_y = max(y for \_, y in layout.values())

            for node\_id, (x, y) in layout.items():

                scaled\_x = (x / max\_x) \* 100

                scaled\_y = (y / max\_y) \* 100

                self.canvas.update\_node\_position(node\_id)

                node\_item = self.canvas.nodes[node\_id][0]

                node\_item.setPos(scaled\_x, scaled\_y)

            self.canvas.scene.update()

            QMessageBox.information(self.parent, "Силовой метод", f"Расположение узлов выполнено за {elapsed\_time:.4f} секунд.")

        except Exception as e:

            QMessageBox.critical(self.parent, "Ошибка", f"Ошибка силового метода: {e}")

    def run\_spring\_layout(self):

        """Запускает пружинный алгоритм для расположения узлов."""

        try:

            if not self.canvas.graph or not nx.is\_connected(self.canvas.graph):

                QMessageBox.warning(self.parent, "Ошибка", "Граф должен быть связным для выполнения пружинного алгоритма.")

                return

            self.canvas.graph.clear()

            for node\_id in self.canvas.nodes:

                self.canvas.graph.add\_node(node\_id)

            for (start, end), edge in self.canvas.edges.items():

                self.canvas.graph.add\_edge(start, end)

            start\_time = time.time()

            spring\_layout(self.canvas.graph, self.canvas)

            end\_time = time.time()

            elapsed\_time = end\_time - start\_time

            self.canvas.scene.update()

            QMessageBox.information(self.parent, "Пружинный алгоритм", f"Расположение узлов выполнено за {elapsed\_time:.4f} секунд.")

        except Exception as e:

            QMessageBox.critical(self.parent, "Ошибка", f"Ошибка пружинного алгоритма: {e}")

**# ui/main\_window.py**

from PyQt5.QtWidgets import QMainWindow

from ui.menu\_bar import MenuBarCreator

from ui.tool\_bar import ToolBarCreator

from ui.dialog\_handler import DialogHandler

from ui.canvas import Canvas

class MainWindow(QMainWindow):

    """Главное окно приложения для работы с графами."""

    def \_\_init\_\_(self):

        super().\_\_init\_\_()

        self.setWindowTitle("Редактор графов")

        self.setGeometry(100, 100, 900, 700)

        self.canvas = Canvas()

        self.setCentralWidget(self.canvas)

        self.dialog\_handler = DialogHandler(self.canvas, self)

        self.menu\_bar = MenuBarCreator(self, self.dialog\_handler)

        self.setMenuBar(self.menu\_bar.create\_menu\_bar())

        self.tool\_bar = ToolBarCreator(self, self.dialog\_handler)

        self.addToolBar(self.tool\_bar.create\_tool\_bar())

**# ui/menu\_bar.py**

from PyQt5.QtWidgets import QMenuBar

class MenuBarCreator:

    """Класс для создания строки меню приложения."""

    def \_\_init\_\_(self, main\_window, dialog\_handler):

        self.main\_window = main\_window

        self.dialog\_handler = dialog\_handler

    def create\_menu\_bar(self):

        """Создаёт строку меню с основными функциями."""

        menu\_bar = QMenuBar(self.main\_window)

        file\_menu = menu\_bar.addMenu("Файл")

        file\_menu.addAction(self.dialog\_handler.create\_action("Сохранить", self.dialog\_handler.save\_graph))

        file\_menu.addAction(self.dialog\_handler.create\_action("Загрузить", self.dialog\_handler.load\_graph))

        file\_menu.addAction(self.dialog\_handler.create\_action("Очистить", self.dialog\_handler.delete\_graph))

        file\_menu.addAction(self.dialog\_handler.create\_action("Выход", self.main\_window.close))

        algorithms\_menu = menu\_bar.addMenu("Алгоритмы")

        algorithms\_menu.addAction(self.dialog\_handler.create\_action("Поиск кратчайшего пути", self.dialog\_handler.run\_dijkstra))

        algorithms\_menu.addAction(self.dialog\_handler.create\_action("Минимальное остовное дерево", self.dialog\_handler.run\_prim))

        algorithms\_menu.addAction(self.dialog\_handler.create\_action("Расположение Камада-Кавай", self.dialog\_handler.run\_kamada\_kawai))

        algorithms\_menu.addAction(self.dialog\_handler.create\_action("Силовой метод", self.dialog\_handler.run\_force\_directed))

        algorithms\_menu.addAction(self.dialog\_handler.create\_action("Пружинный алгоритм", self.dialog\_handler.run\_spring\_layout))

        edit\_menu = menu\_bar.addMenu("Правка")

        edit\_menu.addAction(self.dialog\_handler.create\_action("Добавить узел", self.dialog\_handler.add\_node))

        edit\_menu.addAction(self.dialog\_handler.create\_action("Удалить узел", self.dialog\_handler.remove\_node))

        edit\_menu.addAction(self.dialog\_handler.create\_action("Добавить ребро", self.dialog\_handler.add\_edge))

        edit\_menu.addAction(self.dialog\_handler.create\_action("Удалить ребро", self.dialog\_handler.remove\_edge))

        edit\_menu.addAction(self.dialog\_handler.create\_action("Добавить граф по матрице весов", self.dialog\_handler.add\_graph\_from\_matrix))

        menu\_bar.setStyleSheet("""

            QMenuBar {

                background-color: #f0f0f0;

                font-size: 16px;

                padding: 5px;

            }

            QMenuBar::item {

                padding: 10px;

            }

            QMenuBar::item:selected {

                background-color: #d0d0d0;

            }

            QMenu {

                background-color: #ffffff;

                font-size: 14px;

                border: 1px solid #cccccc;

            }

            QMenu::item:selected {

                background-color: #b0b0b0;

            }

        """)

        return menu\_bar

**# ui/tool\_bar.py**

from PyQt5.QtWidgets import QToolBar, QToolButton

from PyQt5.QtCore import QSize, Qt

class ToolBarCreator:

    """Класс для создания панели инструментов."""

    def \_\_init\_\_(self, main\_window, dialog\_handler):

        self.main\_window = main\_window

        self.dialog\_handler = dialog\_handler

    def create\_tool\_bar(self):

        """Создаёт панель инструментов для быстрого доступа к функциям."""

        tool\_bar = QToolBar("Инструменты", self.main\_window)

        self.main\_window.addToolBar(Qt.TopToolBarArea, tool\_bar)

        add\_node\_action = self.dialog\_handler.create\_action("Добавить узел", self.dialog\_handler.add\_node)

        remove\_node\_action = self.dialog\_handler.create\_action("Удалить узел", self.dialog\_handler.remove\_node)

        add\_edge\_action = self.dialog\_handler.create\_action("Добавить ребро", self.dialog\_handler.add\_edge)

        remove\_edge\_action = self.dialog\_handler.create\_action("Удалить ребро", self.dialog\_handler.remove\_edge)

        reset\_color\_action = self.dialog\_handler.create\_action("Сбросить Цвета", self.dialog\_handler.reset\_color)

        add\_node\_button = QToolButton()

        add\_node\_button.setDefaultAction(add\_node\_action)

        add\_node\_button.setIconSize(QSize(32, 32))

        add\_node\_button.setText("Добавить узел")

        remove\_node\_button = QToolButton()

        remove\_node\_button.setDefaultAction(remove\_node\_action)

        remove\_node\_button.setIconSize(QSize(32, 32))

        remove\_node\_button.setText("Удалить узел")

        add\_edge\_button = QToolButton()

        add\_edge\_button.setDefaultAction(add\_edge\_action)

        add\_edge\_button.setIconSize(QSize(32, 32))

        add\_edge\_button.setText("Добавить ребро")

        remove\_edge\_button = QToolButton()

        remove\_edge\_button.setDefaultAction(remove\_edge\_action)

        remove\_edge\_button.setIconSize(QSize(32, 32))

        remove\_edge\_button.setText("Удалить ребро")

        reset\_color\_button = QToolButton()

        reset\_color\_button.setDefaultAction(reset\_color\_action)

        reset\_color\_button.setIconSize(QSize(32, 32))

        reset\_color\_button.setText("Сбросить Цвета")

        tool\_bar.addWidget(add\_node\_button)

        tool\_bar.addWidget(remove\_node\_button)

        tool\_bar.addWidget(add\_edge\_button)

        tool\_bar.addWidget(remove\_edge\_button)

        tool\_bar.addWidget(reset\_color\_button)

        tool\_bar.setStyleSheet("""

            QToolButton {

                background-color: #f0f0f0;

                border: 2px solid #cccccc;

                border-radius: 5px;

                padding: 12px;

                font-size: 16px;

                min-width: 100px;

            }

            QToolButton:hover {

                background-color: #d0d0d0;

            }

            QToolBar {

                background-color: #e0e0e0;

                padding: 5px;

            }

        """)

        return tool\_bar

**# utils/file\_operations.py**

from PyQt5.QtWidgets import QFileDialog, QMessageBox

def save\_to\_file(parent, graph\_data):

    """Сохраняет данные графа в файл."""

    file\_path, \_ = QFileDialog.getSaveFileName(parent, "Сохранить граф", "", "JSON Files (\*.json)")

    if not file\_path:

        return

    try:

        with open(file\_path, "w") as file:

            file.write(graph\_data)

        QMessageBox.information(parent, "Сохранение", "Граф успешно сохранён.")

    except Exception as e:

        QMessageBox.critical(parent, "Ошибка", f"Не удалось сохранить файл: {e}")

def load\_from\_file(parent):

    """Загружает данные графа из файла."""

    file\_path, \_ = QFileDialog.getOpenFileName(parent, "Загрузить граф", "", "JSON Files (\*.json)")

    if not file\_path:

        return None

    try:

        with open(file\_path, "r") as file:

            return file.read()

    except Exception as e:

        QMessageBox.critical(parent, "Ошибка", f"Не удалось загрузить файл: {e}")

        return None

**# main.py**

import sys

from PyQt5.QtWidgets import QApplication

from ui.main\_window import MainWindow

if \_\_name\_\_ == "\_\_main\_\_":

    """Запускает приложение и отображает главное окно."""

    app = QApplication(sys.argv)

    window = MainWindow()

    window.show()

    sys.exit(app.exec\_())