Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н.Э. Баумана (национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ *Робототехника и комплексная автоматизация*

КАФЕДРА *Системы автоматизированного проектирования (РК-6)*

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

К ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЕ

НА ТЕМУ

«Разработка web-ориентированного редактора графов»

Студент	РК6-82Б		В.А. Ершов
	(Группа)	(Подпись, дата)	(И.О.Фамилия)
Руководитель ВКР			А.П. Соколов
		(Подпись, дата)	(И.О.Фамилия)
Консультант			А.Ю. Першин
		(Подпись, дата)	(И.О.Фамилия)
Нормоконтролер			С.В. Грошев
		(Подпись, дата)	(И.О.Фамилия)

Москва, 2022 г.

РЕФЕРАТ

Данная работа посвящена описанию программной реализации web-ориентированного редактора графов. Разработанный редактор позволяет работать с форматом описания графов aDOT. Данный формат был предложен А.П. Соколовым и А.Ю. Першиным, как формат описания графов в разработанном графоориентированном программном каркасе для реализации сложных вычислительных методов [4]. В работе описана важность разработки подобного редактора, а также подробно рассмотрен весь функционал, который предоставляет разработанный редактор. Все программные решения, такие как выбор алгоритмов, использование тех или иных структур данных, разработка собственных моделей хранения данных, подробно аргументируются. Для каждого реализованного сценария бизнес логики приведены примеры, которые демонстрируют работу данного сценария.

Тип работы: выпускная квалификационная работа.

Тема работы (проект темы): разработка web-ориентированного редактора графов.

Объект исследований: методы автоматизации размещения графических элементов графовых моделей на плоскости.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Современная сфера разработки состоит из множества различных направлений: разработка web-приложений, разработка мобильных приложений, системное программирование, разработка систем баз данных, разработка игр и т.д. В каждом направлении используется свой уникальный стек технологий, но у всех них есть одно общее – это подходы к разработке и требования к разрабатываемому программному обеспечению. Основными требованиями к программному обеспечению являются: эффективность, надежность, практичность, гибкость и удобство сопровождения.

Удобство сопровождения – это легкость изменения разрабатываемой системы с целью реализации дополнительных возможностей, повышения быстродействия, исправления ошибок и так далее. Для увеличения удобства сопровождения разработчики тщательно проектируют архитектуру системы, используют популярные паттерны проектирования, а также следуют общепринятым подходам к разработке: использование методологий Agile, систем контроля версий git, а также написание unit-тестов. Однако, подобные подходы к разработке являются актуальными только при разработке программных систем, которые рассчитаны на использование множеством конечных пользователей. К таким системам относятся web-приложения, мобильные приложения, игры, базы данных и т.д. В области научного программирования ситуация обстоит иначе.

Научное программирование – это программирование вычислительных методов, которые затем используются учеными для работы. Отличительными особенностями научного программирования являются крайне строгие требования к корректности и надежности разрабатываемой системы, а также к четкому разделению интерфейсной и научной частей. Зачастую разработчик является конечным пользователем разрабатываемого программного обеспечения. Из этого следует, что разработчик должен быть экспертом в предметной области, что накладывает серьезные ограничения на удобство сопровождения.

Стандарты проектирования научного программного обеспечения вырабатываются существенно медленнее или не вырабатываются вовсе, что приводит к отсутствию каких-либо системных подходов к разработке. Из-за специфики научного программирования, новому разработчику требуется большой объем времени, чтобы разобраться в написанном коде и приступить к его поддержке. Так, корректно написанный код может оказаться бесполезным в том случае если его поддержка оказывается затруднительной. Вследствие чего стали появляться подходы и системы, которые позволяют минимизировать написание кода и снизить трудозатраты на его поддержку.

В основном существующие платформы используют визуальное программирование [1]. Популярными и успешными разработками в этой области являются LabVIEW и Simulink. Simulink позволяет моделировать вычислительные методы с помощью графических блок-диаграмм. Также Simulink может быть интегрирован со средой MATLAB. Помимо моделирования вычислительных методов Simulink позволяет автоматически генерировать код на языке C для реализации метода в режиме реального времени. LabVIEW используется для аналогичных задач – моделирование технических систем и устройств [2]. LabVIEW позволяет создавать виртуальные приборы с помощью графических блок-диаграмм, в которых каждый узел соответствует выполнению какой-либо функции. Программный код, представленный в таком виде, интуитивно понятен инженерам, что позволяет осуществлять разработку системы более гибко и быстро. Также LabVIEW предоставляет множество специализированных библиотек для моделирования систем из конкретных технических областей.

Также существуют и другие системы для моделирования вычислительных методов. Однако в отличие от Simulink и LabVIEW, они являются узкоспециализированными и созданы для решения определенных задач. Например, система визуального моделирования FEniCS [3] используется для решения задач с использованием метода конечных элементов. FEniCS предоставляет инструменты для работы с конечно-элементными расчетными сетками и функциями решения систем нелинейных уравнений, а также позволяет работать с матема-

тическими моделями с исходной интегрально-дифференциальной форме. Для работы с системой можно использовать языки Python и C++.

Отдельного упоминания стоит TensorFlow. TensorFlow представляет из себя библиотеку для машинного обучения с открытым исходным кодом и, аналогично LabVIEW и Simulink, позволяет моделировать вычислительные методы. В основе TensorFlow лежит такое понятие как граф потока данных. Ребра графа – это тензоры, которые представляют собой многомерные массивы данных, а узлы – это математические операции над ними.

Применение графов очень удобно для построения архитектур процессов обработки данных (как в автоматическом, так и в автоматизированном режимах). Вместе с тем многочисленные возникающие в инженерной практике задачи предполагают проведение повторяющихся в цикле операций. Самым очевидным примером является задача автоматизированного проектирования. Данная задача предполагает, как правило, постановку и решение некоторой обратной задачи, которая в свою очередь, часто, решается путем многократного решения прямых задач (простым примером являются задачи минимизации некоторого функционала, которые предполагают варьирование параметров объекта проектирования с последующим решением прямой задачи и сравнения результата с требуемым согласно заданному критерию оптимизации). Отметим, что прямые задачи (в различных областях) решаются одними методами, тогда как обратные – другими. Эти процессы могут быть очевидным образом отделены друг от друга за счет применения единого уровня абстракции, обеспечивающего определение интерпретируемых архитектур алгоритмом, реализующих методы решения как прямой, так и обратной задач. Очевидным способом реализации такого уровня абстракции стало использование ориентированных графов.

А.П. Соколов и А.Ю. Першин разработали графоориентированный программный каркас для реализации сложных вычислительных методов, теоретические основы представлены в работе [4], а принципы применения графоориентированного подхода зафиксированы в патенте [5]. В отличии от TensorFlow, в представленном графоориентированном программном каркасе узлы

определяют фиксированные состояния общих данных, а ребра определяют функции их преобразования. Для описания графовых моделей был разработан формат aDOT (advanced DOT), который является расширением известного формата описания графов DOT. В формате aDOT были введены дополнительные атрибуты и определения, которые описывают функции-предикаты, функции-обработчики и функции перехода в целом. Подробное описание формата aDOT приведено в [6].

В описанном графоориентированном программном каркасе вводятся такие понятия как функции-обработчики и функции-предикаты. Функции-предикаты позволяют проверить, что на вход функции-обработчика будут поданы корректные данные. Подобная семантика функций-предикатов предполагает, что функции-предикаты и функции-обработчики должны разрабатываться одновременно в рамках одной функции-перехода. Такой подход существенно упрощает процесс программной реализации вычислительного метода, поскольку функции-перехода (функция-обработчик и функция-предикат) могут разрабатываться параллельно несколькими независимыми разработчиками. Возможность параллельной разработки вычислительного метода позволяет организовать модульное тестирование, а также более качественное документирование разрабатываемого кода. Однако, использование формата aDOT, для описания вычислительного метода, позволяет ослабить требования к тщательной документации разрабатываемого метода, а наглядное представление метода в виде ориентированного графа и вовсе отказаться от документации, оставив только поясняющие комментарии в программном коде.

На данный момент существует множество систем визуализации графов. Большинство таких систем позволяют получить изображение графа представленного в виде матрицы или списка смежности, а более продвинутые системы, такие как Graphviz, предлагают собственный формат описания графов. Однако, ни одна из существующих систем визуализации не предоставляет инструменты, которые позволили бы работать с форматом aDOT. Это следует из того, что семантика формата aDOT требует наличия дополнительных атрибутов у графовой

модели. Таким образом, становится очевидным необходимость разработки визуализатора графов описанных с использованием формата aDOT.

1. ПОСТАНОВКА ЗАДАЧИ

В разработанном А.П. Соколовым и А.Ю. Першиным графоориентированном программном каркасе [4], для описания графовых моделей был разработан формат aDOT (advanced DOT), который является расширением известного формата описания графов DOT, который был разработан как формат описания графов в пакете утилит Graphviz. Graphviz позволяет визуализировать графы описанные в формате DOT и получать их изображение в форматах PNG, SVG и других. Формат aDOT расширяет формат DOT с помощью дополнительных атрибутов и определений, которые описывают функции-предикаты, функции-обработчики и функции-перехода в целом. Таким образом, для визуализации графов описанных в формате aDOT необходимо разработать редактор графов. Заметим, что графоориентированный программный каркас [4] допускает наличие циклов в графе. Более того наличие цикла в графе указывает на то, что функция-перехода должна быть запущена многократно. Такие функции перехода могут оказаться узким местом программы и даже вызывать ошибки. Следовательно, редактор должен предоставлять возможность поиска циклов в загружаемой и/или создаваемой в редакторе графовой модели тем самым позволяя разработчику быстро находить потенциально узкие места в программе. Таким образом, разрабатываемый редактор должен удовлетворять следующим требованиям:

1. Редактор позволяет создавать ориентированный граф с нуля;
2. Редактор позволяет экспортировать созданный граф в формат aDOT;
3. Редактор позволяет загрузить (импортировать) граф, описанный в формате aDOT;
4. Редактор предоставляет возможность поиска циклов;
5. Редактор должен быть кроссплатформенным.

2. АРХИТЕКТУРА ПРОГРАММНОЙ РЕАЛИЗАЦИИ

2.1. Выбор языка программирования

Разрабатываемый редактор графов должен представлять собой GUI-приложение. В наше время создание GUI-приложений возможно практически на всех современных языках программирования: C++, Python, Java, C# и так далее. Как правило, стандартные библиотеки языков программирования не предоставляют удобных инструментов для создания GUI-приложений. Для этого существует множество различных графических библиотек, которые позволяют разработчику работать с абстрактными сущностями, что серьезно упрощает и ускоряет процесс разработки GUI-приложения. Далее приведена таблица 1 с перечнем популярных GUI-библиотек.

Таблица 1 – Популярные GUI-библиотеки

Язык программирования	GUI-библиотеки
C++	• SFML – простая в освоении библиотека, которая в основном используется для создания простых GUI-приложений. • Qt – open source библиотека, которая предоставляет обширный набор инструментов для проектирования GUI-приложений. Библиотека доступна на всех популярных ОС: Windows, Linux, Mac OS.
Python	• Tkinter • Kivy – событийно-ориентированный фреймворк, который обычно используется для разработки игр. • PyQt – библиотека для работы с Qt на Python

Каждая GUI-библиотека имеет свои преимущества и недостатки, поэтому перед проектированием GUI необходимо тщательно исследовать все требования к интерфейсу и только потом переходить к выбору языка программирова-

ния и GUI-библиотеки. Однако, у всех GUI-библиотек есть общий недостаток – поддержка кроссплатформенности. Некоторые библиотеки позволяют разрабатывать GUI-приложения только для определенных платформ. В то время как другие библиотеки, предоставляющие возможность разработки кроссплатформенных GUI-приложений, требуют написания большого объема дополнительного кода.

Одно из требований к разрабатываемому редактору – кроссплатформенность. Поскольку разработка кроссплатформенных приложений с использованием GUI-библиотек требует большого объема времени и ресурсов, было принято решение разрабатывать web-приложение, которое будет доступно в интернете. Обычно web-приложения состоят из трех частей:

1. Фронтенд – клиентская сторона пользовательского интерфейса;
2. Бэкенд – программно-аппаратная часть сервиса (бизнес-логика приложения);
3. База данных.

Поскольку в разрабатываемом редакторе отсутствует сложная бизнес-логика, то можно отказаться от использования бэкенда и базы данных и реализовать всю бизнес-логику приложения на стороне клиента (пользователя), то есть на фронтенде. В таком случае, приложение будет доступно даже при отсутствии интернета, что, несомненно, является плюсом.

Таким образом, для разработки редактора использовался язык программирования JavaScript, поскольку только данный язык позволяет создавать интерактивные веб-страницы с бизнес логикой на стороне клиента (пользователя).

2.2. Архитектура приложения

Любая программная система имеет ряд общих проблем:

- Организация большого объема кода;

- Поддержка системы;
- Дублирование функциональности;
- Расширение функциональности.

Для решения данных проблем, перед написанием программного кода проводится тщательно проектирование архитектуры приложения. Основная задача архитектуры приложения - предоставить разработчику должный уровень абстракции. В корректно спроектированной архитектуре реализация абстрагирована от интерфейса, а также присутствует строгое разделение полномочий между блоками кода.

В наше время существует множество различных, широко известных, архитектур приложений, поэтому вместо разработки собственной архитектуры предпочтительнее использовать уже существующую архитектуру. Использование известной архитектуры, вместо проектирования собственной, позволит увеличить удобство сопровождения, поскольку новому разработчику не придется тратить время на изучение неизвестной для него архитектуры приложения. Для корректного выбора архитектуры приложения необходимо тщательно исследовать предметную область разрабатываемого приложения. После исследования предметной области необходимо рассмотреть все достоинства и недостатки существующих архитектур и выбрать подходящую архитектуру. Далее приведена таблица 2 с перечнем популярных архитектур фронтенда.

Таблица 2 – Популярные архитектуры фронтенд приложений

Архитектура	Описание
MVC (Model-View-Controller)	Существует модель, которая хранит данные. Также существует какой-то View, который показывает эти данные пользователю. Необходимо связать данные с их представлением, для этого используется Controller. Помимо MVC существуют архитектуры MVP и MVVM, несмотря на некоторые отличия, основная суть остается одинаковой.
Flux	Существует некоторый View, который создает структуру типа Action и передает ее в Dispatcher. Задача Dispatcher – вызывать

	коллбэки из Store. Store: • Исходя из полученных метаданных, Action выбирает метод обновления • Обновляет данные и триггерит событие изменения View берет данные из Store, а затем перерисовывается

Более подробное описание существующих архитектур фронтенда представлено в статье [7].

В достаточно простых web-приложениях, необходимость использования архитектуры отсутствует. Несмотря на все очевидные плюсы хорошо спроектированной архитектуры, использование архитектуры предполагает строгое соответствие программного кода выбранной архитектуре, что, несомненно, потребует дополнительного времени на разработку приложения.

Таким образом, в разработанном редакторе не используется никакой архитектуры. Вместо архитектуры, для увеличения удобства сопровождения, используется объектно-ориентированное программирование.

2.3. Инструменты для работы с графикой

Одно из требований к разрабатываемому редактору графов – возможность создавать ориентированный граф с нуля. Следовательно, необходимо реализовать возможность добавления и удаления вершин и ребер графа. Для работы с графикой в JavaScript существуют два различных подхода:

1. Использование HTML элемента $< canvas >$;
2. Использование HTML элемента $< svg >$.

Основное отличие между canvas и SVG (Scalable Vector Graphics) заключается в том, что canvas работает с растровой графикой, в то время как svg работает с векторной графикой. Canvas позволяет манипулировать каждым пикселем растрового изображения с помощью JavaScript API. В свою очередь SVG

позволяет создавать различные элементы (окружность, линия, квадрат и т.д.), которые будут доступы через DOM. Возможность доступа к созданным элементам через DOM позволяет достаточно гибко управлять примитивами на странице: удалять их или изменять их свойства. Подробное сравнение SVG и canvas представлено в статье [8].

В разрабатываемом редакторе необходимо работать только с двумя примитивами (окружность и прямая), следовательно, использование векторной графики SVG является более предпочтительным вариантом. Работать с SVG без использования сторонних библиотек достаточно затруднительно, поскольку для создания примитивов придется разрабатывать собственные функции.

Для работы с векторной графикой достаточно удобно использовать библиотеку D3.js. Библиотека D3.js предоставляет набор инструментов для визуализации данных на странице, который состоит из нескольких десятков небольших модулей, каждый из которых решает свою задачу. С помощью данной библиотеки можно легко создавать любой SVG элемент на странице. В листинге 1 приведен пример использования библиотеки D3 для создания окружности.

Листинг 1 – Создание окружности с помощью библиотеки D3

```
svg
    .append("circle")
    .attr("cx", x)
    .attr("cy", y)
    .attr("r", radius)
    .attr("stroke-width", borderWidth)
    .attr("id", id)
    .attr("fill", "#FFFFFF")
    .attr("stroke", "#000000")
    .attr("class", "vertex");
```

В разработанном редакторе графов возможности библиотеки D3.js используются для создания вершин и ребер графа.

2.4. Использование ООП

В разработанном редакторе графов используется ООП (Объектно-Ориентированное Программирование), чтобы инкапсулировать работу с графом. В JavaScript концепция ООП программирования отличается от традиционной концепции ООП, которая используется в объектно-ориентированных языках программирования (C++, Java). Ключевое слово class является лишь синтаксическим сахаром, поскольку класс на самом деле представляет собой объект (структура данных в JavaScript). Также, в JavaScript все методы класса имеют публичный модификатор доступа. Перед названием метода можно поставить символ "#" тем самым пометив метод как приватный, однако данный механизм не позволяет инкапсулировать метод и может использоваться лишь в качестве рекомендации для разработчика.

Разработанный редактор содержит класс $Graph$, который используется для хранения всех необходимых данных о графе, а также для реализации основных сценариев бизнес логики. UML-диаграмма данного класса представлена на рисунке 1.

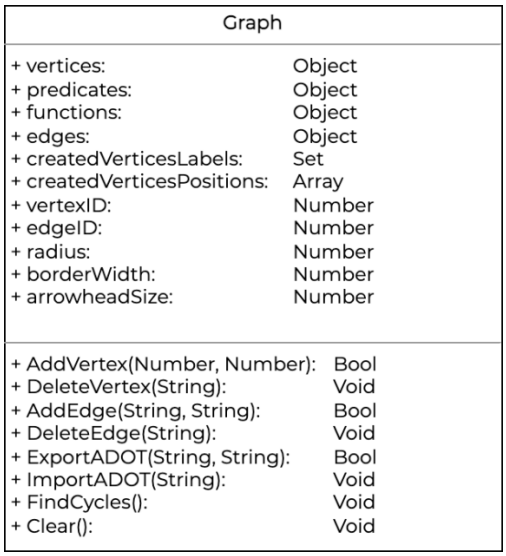


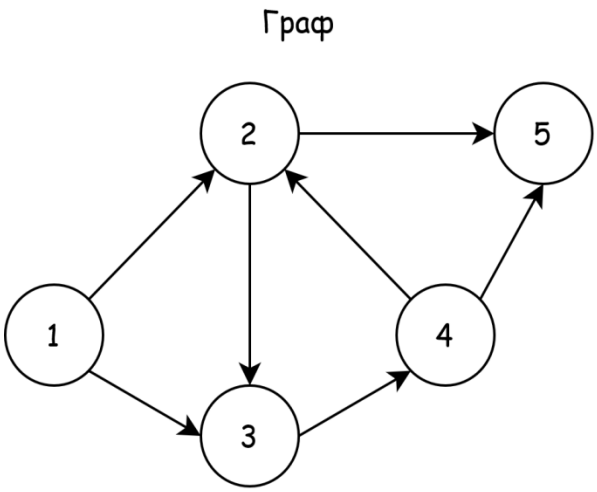
Рисунок 1 – UML-диаграмма класса $Graph$

Для уменьшения объема UML-диаграммы методы, имеющие приватный модификатор доступа, были опущены. Представленные в диаграмме публичные методы реализуют всю бизнес-логику приложения.

3. ПРОГРАММНАЯ РЕАЛИЗАЦИЯ

3.1. Модель данных для хранения графа

Для представления графовой модели обычно используются матрицы смежности или списка смежности. Матрица смежности представляет собой двумерный массив, содержащий n строк и n столбцов. Каждая ячейка матрицы имеет значение либо 1, либо 0. Если существует ребро E между вершиной i и вершной j, то ячейка $[i][j] = 1$, в противном случае $[i][j] = 0$. Пример матрицы смежности представлен на рисунке 2.



Матрица смежности

	1	2	3	4	5
1		1	1		
2			1		1
3				1	
4		1			1
5					

Рисунок 2 – Матрица смежности

Список смежности представляет собой массив односвязных списков. Индексом массива является вершина, а значением односвязный список, состоящий из соседей этой вершины. Пример списка смежности представлен на рисунке 3.

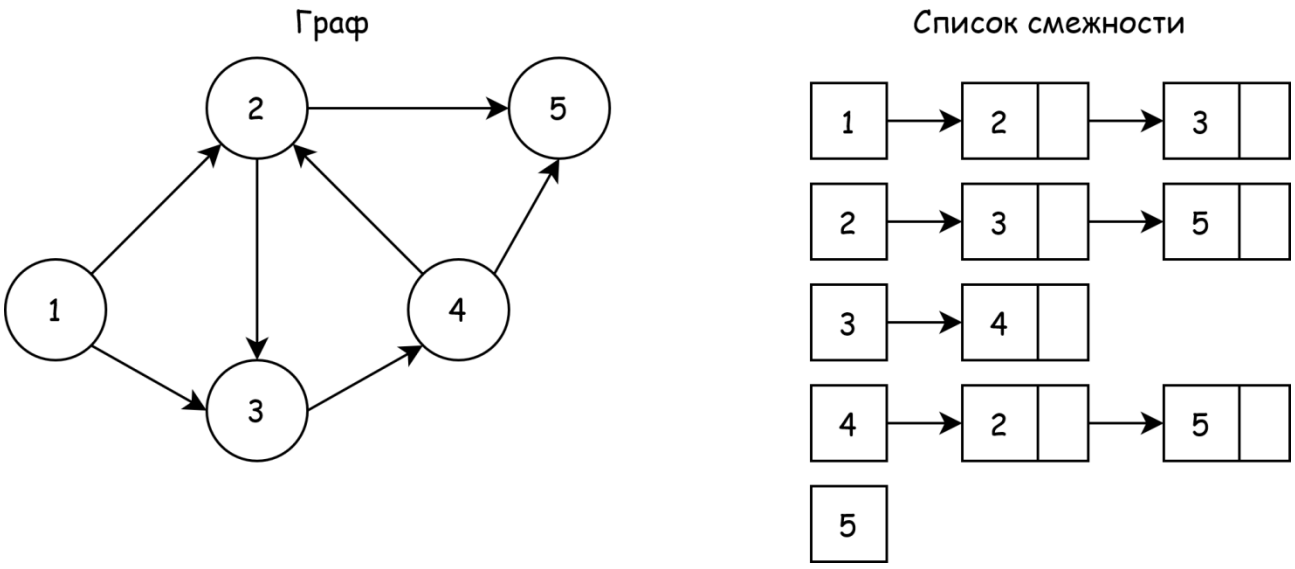


Рисунок 3 – Список смежности

Данные способы представления графовой модели содержат информацию только о вершинах графа и связях между ними, но в разрабатываемом редакторе необходимо хранить множество дополнительной метаинформации, поэтому для хранения графовой модели была разработана собственная модель данных.

Разработанная модель данных представляет собой массив вершин, где вершина будет представлена как объект. Объект – это тип данных в JavaScript, который позволяет хранить коллекцию значений разных типов. В листинге 2 приведен JavaScript объект содержащий информацию об одной вершине графа.

Листинг 2 – JavaScript объект содержащий информацию об одной вершине графа

```
{
    "edges": [ #массив связанных с вершиной ребер
        {
            "direction": "from", #направление from/to
            "function": "f1",    #функция-обработчик
            "predicate": "p1",   #функция-предикат
            "label": "<p1, f1>", #метка ребра
            "type": "straight",  #тип ребра (прямое/кривая Безье)
            "value": "vertex2",  #вершина связанная этим ребром
            "metadata": { #метаинформация для работы через DOM
                "edgeID": "edge1",          #ID элемента(ребра)
                "pathID": "edge1_path",     #ID пути по которому строилась метка
ребра
                "labelID": "edge1_label",  #ID метки ребра
            }
        },
    ],
```

```
    "metadata": { #метаинформация о вершине
        "label": "s1", #метка вершины
        "label_metadata": { #метаинформация о метке вершины
            "pathID": "vertex1_path", #ID пути по которому строилась метка вер-
шины
            "labelID": "vertex1_label", #ID метки вершины
        },
        "position": { #информация о положении вершины на странице
            "x": 227.609375, #положение по оси X
            "y": 211, #положение по оси Y
        },
    }
}
```

При использовании списка смежности для представления графовой модели, получение списка связанных вершин занимает константное время $O(1)$. В то время как, в разработанной модели, для получения списка связанных вершин, необходимо исследовать все ребра, выходящие из вершины, что потребует линейное время $O(E)$, где E – количество ребер, которые выходят из вершины. Однако подобная модель представления графа имеет и ряд преимуществ.

При удалении вершины необходимо удалить все связанные с вершиной ребра, а также удалить информацию об этих ребрах из связанных вершин. Разработанная модель хранения графа позволяет за константное время $O(1)$ получать массив всех связанных с вершиной ребер, в то время как в списке смежности подобная операция потребовала бы линейное время $O(E)$. Однако, основное преимущество разработанной модели хранения графа – возможность корректной визуализации загруженного из формата aDOT графа. Описание данного алгоритма будет приведено в разделе 2.6.

Помимо хранения графой модели необходимо хранить информацию о функциях-предикатах и функциях-обработчиках. Функции-предикаты и функции-обработчики хранятся в виде объектов. Для хранения остальных данных используются массивы и хэш-таблицы.

3.2. Основные сценарии бизнес логики

3.2.1. Создание вершины

Создание вершины является самым простым сценарием бизнес логики разработанного редактора. Для создания вершины необходимо выбрать пункт меню “Добавить вершину”, а затем кликнуть в любой точке рабочей области, после чего в указанном месте отрисуется вершина (окружность определенного радиуса, который определен как константа). Последним этапом создания вершины является ввод названия вершины. Заметим, что после того, как вершина отрисовалась, любые действия в приложении блокируются до того момента, пока не будет введено название вершины. Подобное ограничение не позволяет создавать вершины без названий, поскольку согласно формату aDOT все вершины должны иметь названия.

Несмотря на тривиальность рассматриваемого сценария, существует несколько ситуаций, требующих дополнительной обработки:

- Попытка создания вершины поверх существующей вершины;
- Выбор неуникального названия вершины.

Для того чтобы запретить создание вершины поверх существующей вершины необходимо проверить, что после отрисовки вершины в выбранной позиции она не будет пересекаться с существующими вершинами. Для этой проверки используется массив, который хранит информацию о расположении каждой вершины графа. Элементами массива являются массивы, которые содержат три элемента: ID вершины, позиция вершины по оси X, позиция вершины по оси Y. Таким образом, при попытке создания вершины поверх существующей вершины, можно легко определить ID этой вершины и изменить ее цвет, так пользователь сможет увидеть с какой вершиной происходит конфликт. На рисунке 4 представлена описанная ситуация.

Рисунок 4 – Создание вершины поверх существующей вершины

Для того чтобы запретить пользователю вводить неуникальное название вершины (название, которое уже присвоено существующей вершине) необходимо проверить, что введенное название вершины уникально в рамках графа. Для этой проверки используется set (структура данных, представляющая из себя коллекцию значений, в которой каждое значение встречается не более одного раза), в котором хранятся названия существующих вершин.

3.2.2. Создание ребра

Для создания ребра необходимо выбрать пункт меню “Добавить ребро”, а затем последовательно выбрать две вершины, между которыми нужно построить ребро, после чего между выбранными вершинами будет построено ребро. Последним этапом создания ребра является ввод необходимых данных: функция-предикат, функция-обработчик. Аналогично процессу создания вершины, после отрисовки ребра любые действия в приложении блокируются до того

момента, пока не будут введены необходимые данные. Подобное ограничение следует из требований формата aDOT: всем ребрам в графе должны быть присвоены некоторые функции-предикаты и некоторые функции-обработчики.

В данном сценарии существует ряд ситуаций, которые требуют дополнительной обработки:

- Попытка соединить вершину саму с собой;
- Построение ребра между вершинами, которые уже содержат ребра между собой;
- Существование заданной функции-предиката или функции-обработчика в рамках графовой модели.

При построении ребра между вершинами выполняется проверка, что выбраны разные вершины, в противном случае построение ребра отклоняется и выводится соответствующее сообщение в консоль. Также выполняется проверка существования ребер между выбранными вершинами. В случае наличия хотя бы одного ребра между выбранными вершинами, новое ребро будет построено с использованием кривых Безье. Высота кривой Безье прямо пропорционально зависит от количества существующих ребер. На рисунке 5 представлен граф имеющий ребра, которые построены с использованием кривых Безье.

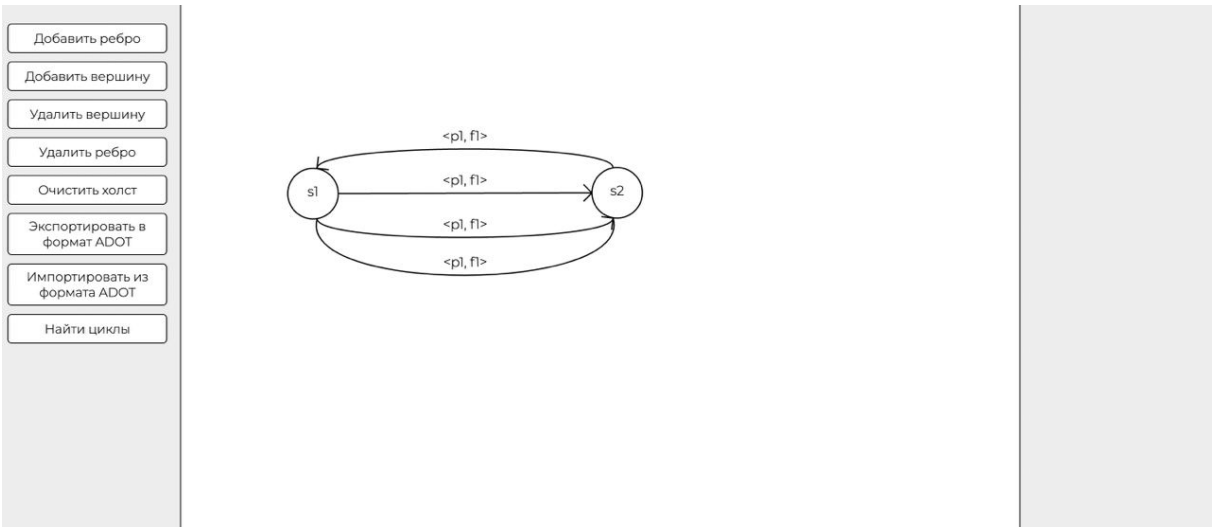


Рисунок 5 – Граф имеющий ребра, построенные с использованием кривых Безье

Для хранения информации о функциях-предикатах и функциях-обработчиках в классе $Graph$ определены два объекта: $predicates$ и $functions$. В процессе построения ребра, после ввода функции-предиката и функции-обработчика, необходимо проверить их уникальность в рамках графа. В случае уникальности необходимо сохранить информацию о новой функции-предикате или функции-обработчике в соответствующем объекте.

3.2.3. Удаление вершины

Для удаления вершины необходимо выбрать пункт меню “Удалить вершину”, а затем выбрать вершину, которую предполагается удалить. В данном сценарии присутствует несколько особенностей, которые необходимо учесть:

- При удалении вершины должны удалиться и все связанные с ней ребра;
- При удалении ребра одна из связанных вершин может “повиснуть”, то есть оказаться несвязанной ни с одной вершиной графа, такую вершину необходимо удалить.

Заметим, что при удалении вершины ее необходимо удалить со страницы, а также из объекта $vertices$ класса $Graph$. Для удаления вершины со страницы достаточно найти ее в DOM-дереве, а затем удалить. Именно поэтому, модель хранения графа содержит достаточно много дополнительной информации (ID вершины, ID текстовых элементов с названием вершины, ID ребер и т.д.). Удаление одного ребра происходит аналогичным образом.

Рассмотрим удаление вершины на примере. На рисунке 6 представлен некоторый ориентированный граф.

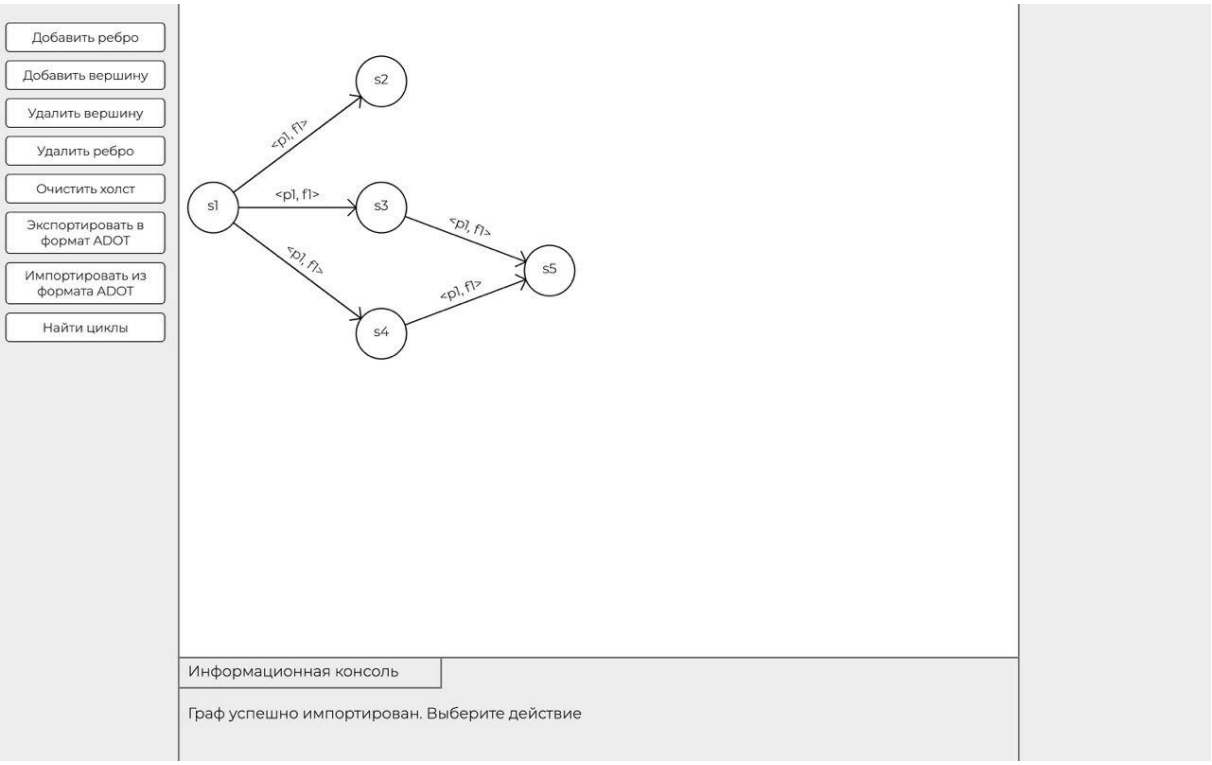


Рисунок 6 – Ориентированный граф

При удалении вершины $s1$ будут удалены все связанные с ней ребра, а также вершина $s2$, поскольку она не будет связана ни с одной вершиной графа. На рисунке 7 представлен граф после удаления вершины $s1$.

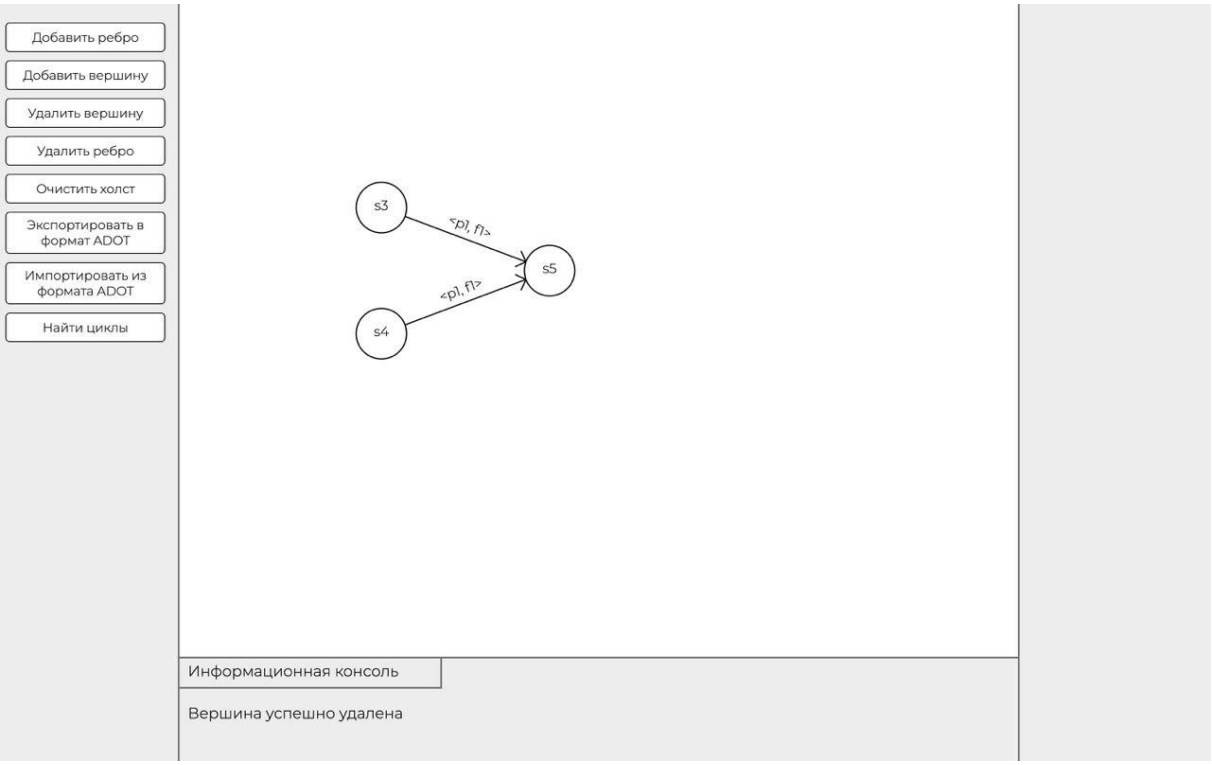


Рисунок 7 – Граф после удаления вершины $s1$

3.2.4. Экспорт графа в формат aDOT

Для экспорта графа в формат aDOT необходимо выбрать пункт меню “Экспортировать в формат aDOT”, а затем последовательно выбрать начальную и конечную вершины, после чего будет загружен текстовый файл с описанием графа в формате aDOT. Все данные, которые необходимы для экспорта графа в формат aDOT, сохраняются в различные структуры данных в процессе создания графа. Таким образом, в программном коде процесс формирования текстового описания графа в формате aDOT заключается в последовательном сборе всех необходимых данных из различных массивов, объектов и хэш-таблиц. В силу своей тривиальности данный сценарий не имеет никаких особенностей, которые требуют отдельного рассмотрения. Рассмотрим экспорт графа в формат aDOT на примере графа представленного на рисунке 6. В листинге 3 приведено описание рассматриваемого графа в формате aDOT.

Листинг 3 – описание графа в формате aDOT

```
digraph TEST
{
// В узле указана стратегия распараллеливания
      s1 [parallelism=threading]
// Определения функций обработчиков
      f1 [module=DEFAULT_VALUE, entry_func=DEFAULT_VALUE]
// Определения функций предикатов
      p1 [module=DEFAULT_VALUE, entry_func=DEFAULT_VALUE]
// Определения функций перехода
      edge_1 [predicate=p1, function=f1]
// Описание графовой модели
      __BEGIN__ -> s1
      s1 => s2 [morphism=edge_1]
      s1 => s3 [morphism=edge_1]
      s1 => s4 [morphism=edge_1]
      s3 -> s5 [morphism=edge_1]
      s4 -> s5 [morphism=edge_1]
      s5 -> __END__
}
```

Асимптотическая сложность экспорта графа в формат aDOT составляет $O(F + P + V + E)$, где

F – количество функций-обработчиков, которые определены в рамках графа

P – количество функций-предикатов, которые определены в рамках графа

V – количетсво вершин в графе

E – количество ребер в графе

3.2.5. Поиск циклов в графе

Для поиска циклов в графовой модели необходимо выбрать пункт меню “Найти циклы”, после чего все найденные в графе циклы будут подсвечены. В качестве примера рассмотрим граф, представленный на рисунке 8.

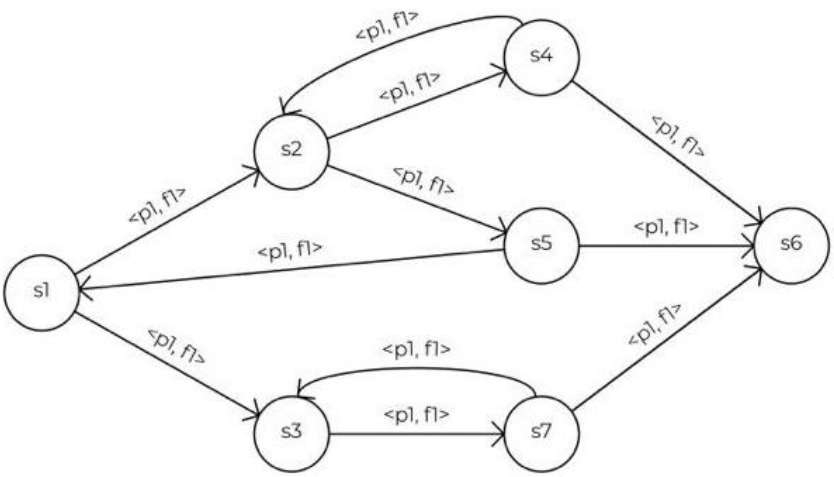


Рисунок 8 – Ориентированный граф

На рисунке 9 представлен граф с найденными циклами.

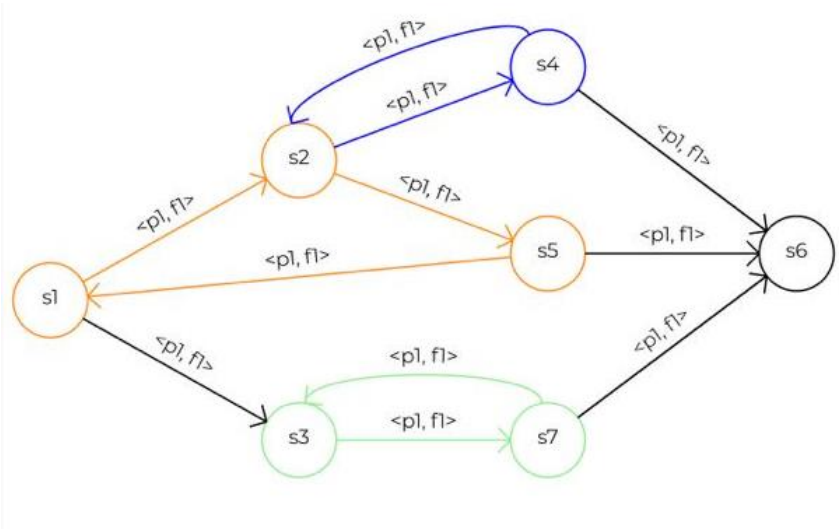


Рисунок 9 – Граф с подсвеченными циклами

Поиск циклов в графе предполагает исследование всех возможных путей из каждой вершины графа. Если в процессе следования по пути какая-либо вершина была посещена дважды, то это означает, что данная вершина является началом цикла. Для эффективного поиска циклов в графе необходимо определить корректный алгоритм обхода графа.

Обход графа – это переход от одной вершины графа к другой в поисках свойств связей этих вершин. Для обхода графов существуют два алгоритма: поиск в глубину (Depth-First Search, DFS) и поиск в ширину (Breadth-First Search, BFS). Оба алгоритма используются для решения одной задачи – эффективный обход графа, однако они имеют ряд отличительных особенностей и применяются для решения разных задач.

Суть алгоритма поиска в глубину (DFS) – погружаться вглубь графа следуя в определенном направлении. Движение начинается от некоторой начальной вершины по определенному пути до тех пор, пока не будет достигнута искомая вершина или конец пути. Если искомая вершина не была найдена и был достигнут конец пути, то необходимо вернуться в начальную вершину и пойти по другому пути. В отличие от алгоритма DFS, суть алгоритма поиска в ширину BFS состоит в том, чтобы на каждом шаге продвигаться вперед по одному соседу за раз. Более подробное описание алгоритма обхода графа DFS и BFS представлено в статье [9].

Оба алгоритма обхода имеют одинаковую асимптотическую сложность $O(V + E)$, где V – количество вершин в графе, а E – количество ребер в графе, а, следовательно, оба алгоритма одинаково эффективны.

С точки зрения программной реализации, алгоритм поиска в глубину (DFS) использует рекурсию для обхода всех возможных путей. В то время как, алгоритм поиска в ширину (BFS) использует очередь для хранения соседних вершин и итеративно продвигается по графу шаг за шагом. Таким образом, при работе с большими графовыми моделями поиск в глубину будет крайне неэффективен, а в худшем случае будет вызывать переполнение стека и аварийное завершение программы. Однако и поиск в ширину обладает рядом недостатков, которые могут очень серьезно снизить эффективность работы алгоритма. При работе с достаточно широкими графами (у вершин очень много соседей) все соседние вершины необходимо сохранить в очереди, что несомненно потребует большой объем памяти. Также, нередко возникает ситуация, когда решения расположены очень глубоко в графе и применение поиска в ширину для их нахождения будет просто непрактично.

Таким образом, для решения задачи поиска циклов в графе необходимо использовать поиск в глубину (DFS). Блок-схема алгоритма поиска циклов в графе представлена на рисунках 10, 11, 12.

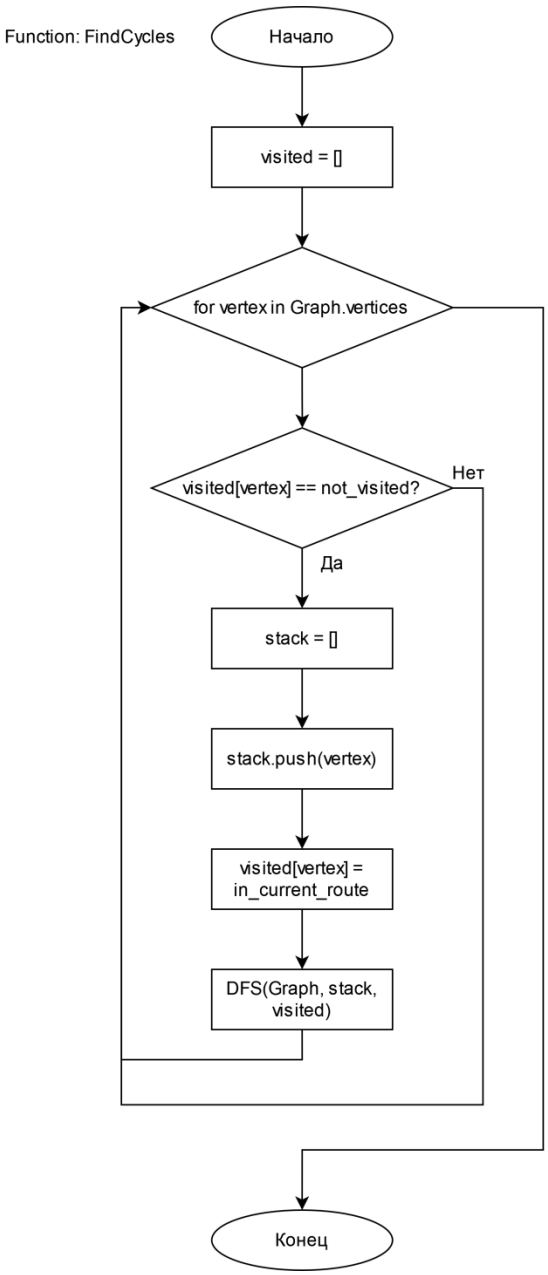


Рисунок 10 – Блок-схема поиска циклов в графе, функция FindCycles

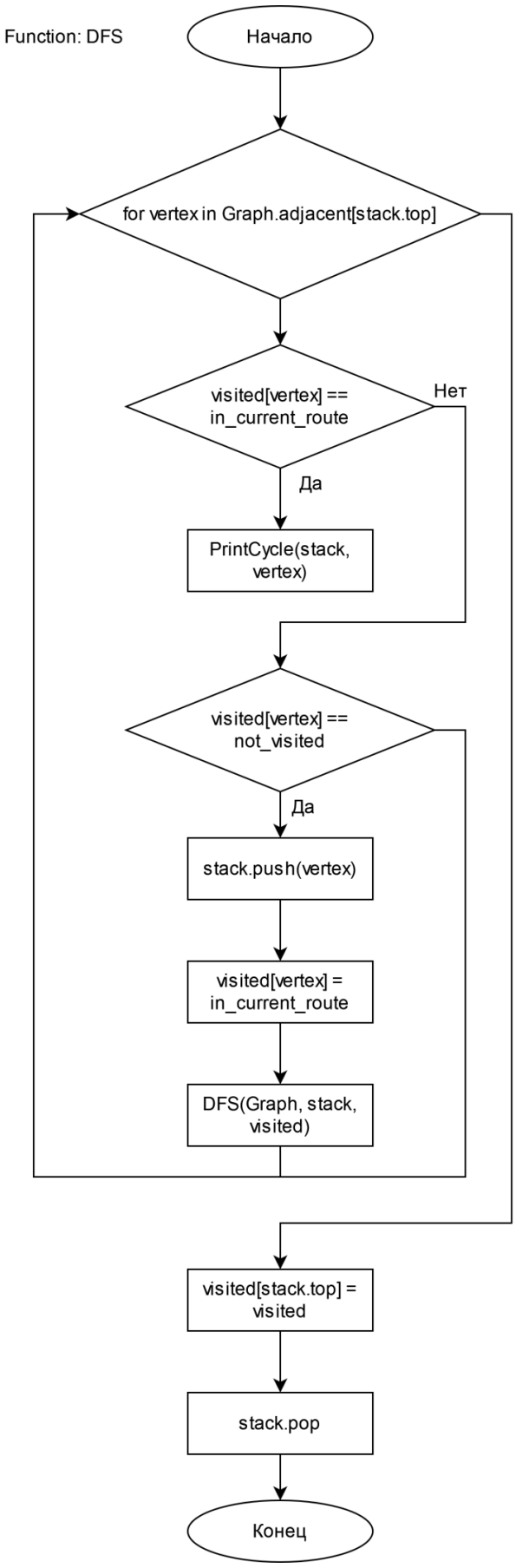


Рисунок 11 – Блок-схема поиска циклов в графе, функция DFS

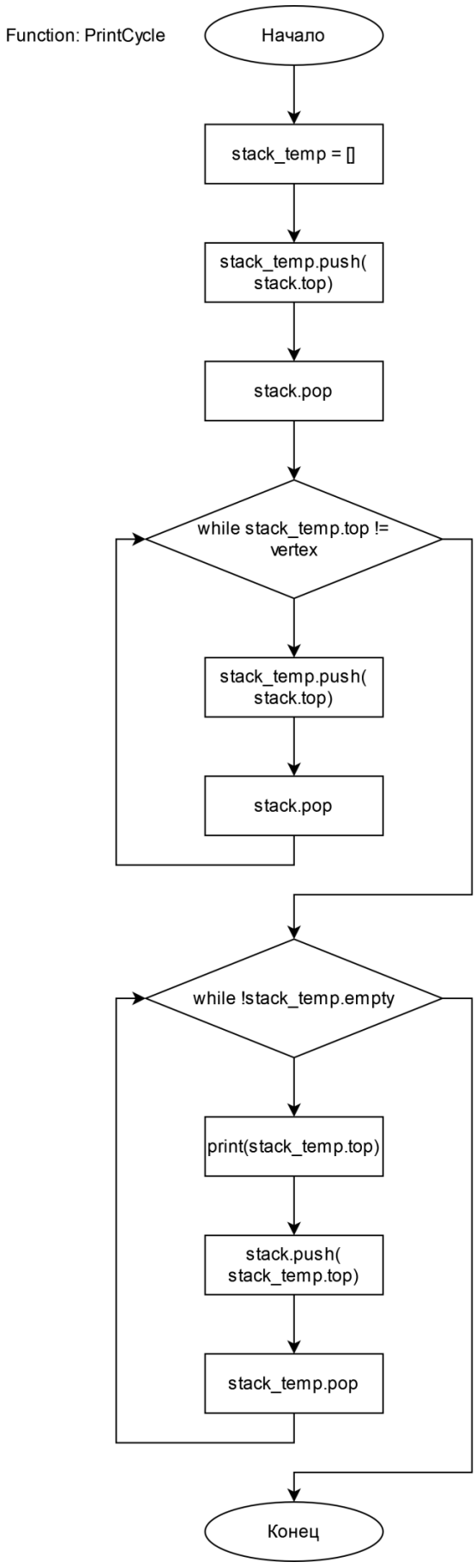


Рисунок 12 – Блок-схема поиска циклов в графе, функция PrintCycle

3.2.6. Импорт графа из формата aDOT

Для импорта графа из формата aDOT необходимо выбрать пункт меню “Импортировать из формата aDOT”, а затем загрузить текстовый файл с описание графа в формате aDOT. В результате будет построен ориентированный граф, который полностью соответствует описанию в формате aDOT. С точки зрения программной реализации данный сценарий является самым сложным, поскольку требует реализации алгоритма визуализации графа. Помимо алгоритма визуализации, необходимо реализовать сохранение всей информации о графовой модели в соответствующие объекты класса $Graph$.

Перед реализацией собственного алгоритма визуализации необходимо исследовать уже существующие решения. К любому алгоритму визуализации выдвигаются следующие требования:

- Граф должен быть представлен в человекопонятом виде;
- Минимизация расходов по памяти;
- Минимизация времени работы.

Основным требованием является представление графа в человекопонятном виде. Для увеличения наглядности построенного графа применяются эстетические критерии. Широко используются следующие эстетические критерии:

- Минимизация пересечений;
- Минимизация области размещения;
- Минимизация общей длины ребер;
- Минимизация длины ребра;
- Минимизация максимального различия между длинами ребер;
- Максимальная симметричность.

Визуализация графа – это решение задачи укладки. Укладка – это способ сопоставить каждой вершине графа координаты. Рассмотрим три основных вида укладок:

1. Force-Directed and Energy-Based. В данных методах используется симуляция физических сил. Вершины – это заряженные частицы, которые отталкиваются друг от друга, а ребра – это упругие связи, стягивающие смежные вершины. Минусом данного типа укладки является высокая вычислительная сложность, поскольку для каждой вершины графа необходимо рассчитать силы, которые на нее действуют. Данный вид укладки используется такими алгоритмами как: Fruchterman-Reingold, OpenOrd, ForceAtlas;
2. Dimension Reduction. Граф описывается матрицей смежности, к которой применяются алгоритмы снижения размерности. Основная идея – рассчитать расстояния между вершинами, а затем постепенно снижать размерность пространства;
3. Feature-Based Layout. Для укладки используется некоторое свойство, которое присутствует у всех вершин.

Согласно формату aDOT у графов должны быть определены стартовая (начальное состояние данных) и конечная (конечное состояние данных) вершины. Для того, чтобы оптимально представить граф в формате aDOT, необходимо размещать вершины друг за другом, постепенно двигаясь по оси абсцисс. Таким образом, стартовая вершина будет иметь минимальную координату по оси X, а конечная вершина - максимальную. Остальные вершины графа будут равномерно распределены между стартовой вершиной и конечной вершиной. К сожалению, ни один из рассмотренных вариантов укладки не позволяет разместить вершины подобным образом. В результате был разработан алгоритм визуализации графа, который позволяет получить оптимальное представление графа.

Описанное размещение представляет собой набор вершин размещенных по уровням, где каждый уровень имеет свою координату по оси X. На рисунке 13 представлено размещение вершин по уровням.

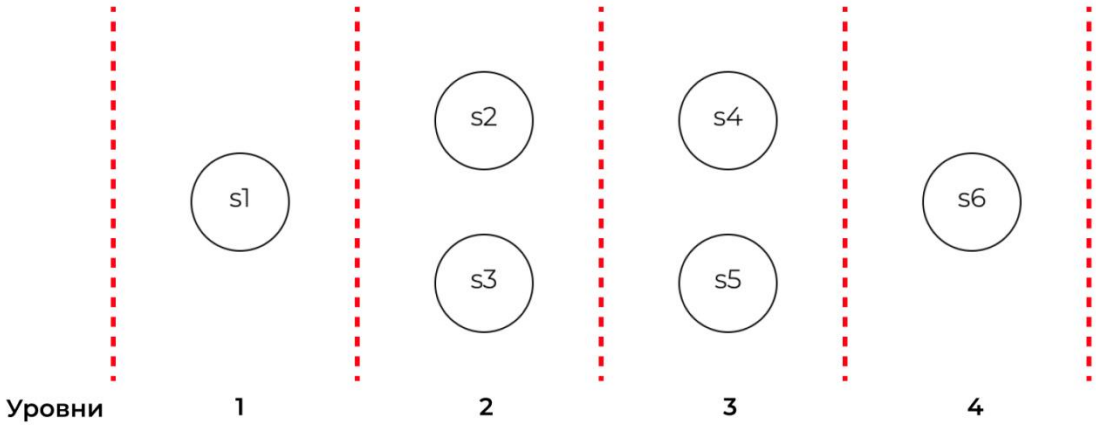


Рисунок 13 – размещение вершин по уровням, $s1$ – стартовая вершина, $s6$ – конечная вершина

Рассмотрим работу алгоритма на примере некоторого графа описанного в формате aDOT. В листинге 4 приведено описание графа в формате aDOT.

Листинг 4 – Описание графа в формате aDOT

```
digraph TEST
{
// Parallelism
    s11 [parallelism=threading]
    s4 [parallelism=threading]
    s12 [parallelism=threading]
    s15 [parallelism=threading]
    s2 [parallelism=threading]
    s8 [parallelism=threading]
// Functions
    f1 [module=DEFAULT_VALUE, entry_func=DEFAULT_VALUE]
// Predicates
    p1 [module=DEFAULT_VALUE, entry_func=DEFAULT_VALUE]
// Edges
    edge_1 [predicate=p1, function=f1]
// Graph model description
    __BEGIN__ -> s1
    s6 -> s8 [morphism=edge_1]
    s7 -> s8 [morphism=edge_1]
    s10 -> s8 [morphism=edge_1]
    s11 => s8 [morphism=edge_1]
    s11 => s9 [morphism=edge 1]
    s14 -> s9 [morphism=edge_1]
    s3 -> s9 [morphism=edge_1]
    s4 => s6 [morphism=edge_1]
    s4 => s7 [morphism=edge_1]
    s4 => s10 [morphism=edge_1]
    s4 => s11 [morphism=edge_1]
    s12 => s4 [morphism=edge_1]
    s12 => s14 [morphism=edge_1]
    s13 -> s3 [morphism=edge_1]
    s15 => s14 [morphism=edge_1]
```

```
    s15 => s14 [morphism=edge_1]
    s2 => s12 [morphism=edge_1]
    s2 => s13 [morphism=edge_1]
    s2 => s15 [morphism=edge_1]
    s1 -> s2 [morphism=edge_1]
    s8 => s9 [morphism=edge_1]
    s8 => s6 [morphism=edge_1]
    s9 -> __END__
}
```

В результате визуализации, с использованием разработанного алгоритма, будет получен граф, представленный на рисунке 14.

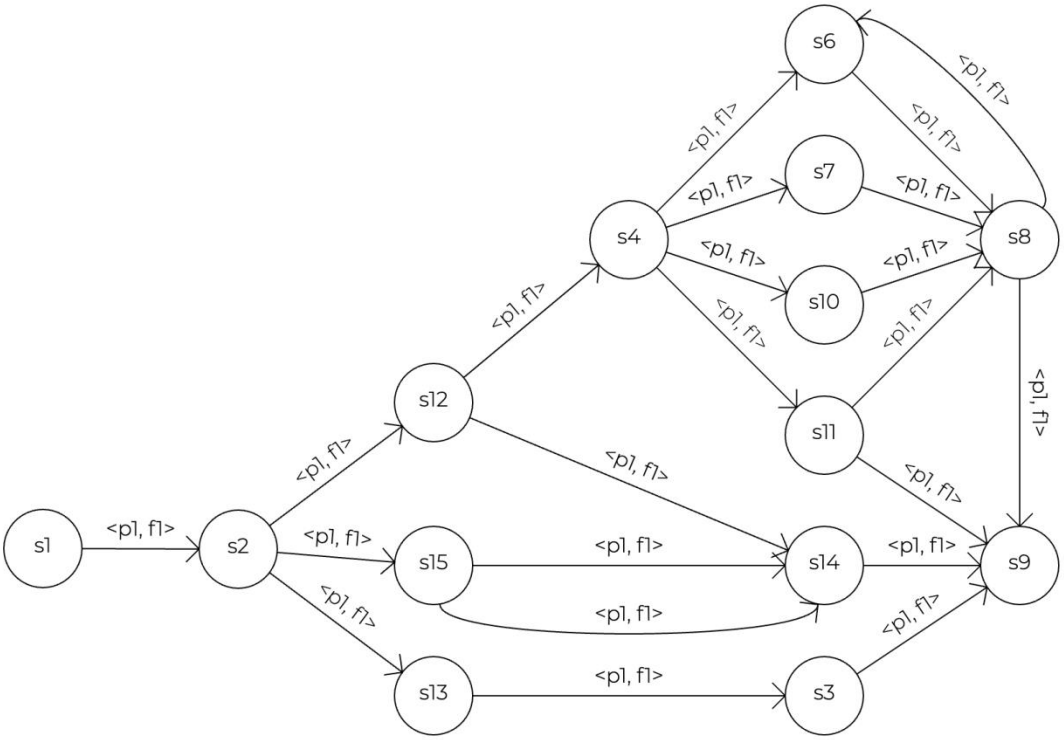


Рисунок 14 – Полученный в результате визуализации граф

Алгоритм визуализации содержит два основных этапа: размещение вершин по уровням, распределение вершин по оси ординат в рамках каждого уровня. В программной реализации для хранения уровней используется объект $levels$. В листинге 5 представлено содержимое объекта $levels$ для рассматриваемого (рисунок 14) графа.

Листинг 5 – Содержимое объекта $levels$ для рассматриваемого (рисунок 14) графа

```
{
    "1":[{"s1":[]}],
    "2":[{"s2":["s1"]}],
    "3":[{"s12":["s2"]}, {"s13":["s2"]}, {"s15":["s2"]}],
    "4":[{"s4":["s12"]}],
    "5":[{"s9":["s14","s3"]}, {"s6":["s4"]}, {"s7":["s4"]}, {"s10":["s4"]},
    {"s11":["s4"]}, {"s14":["s12","s15"]}, {"s3":["s13"]}],
    "6":[{"s8":["s6","s7","s10","s11"]}, {"s9":["s11","s14","s3"]}]
}
```

Ключами являются номера уровней, а значениями массивы, которые содержат список вершин принадлежащих данному уровню. Заметим, что для каждой вершины хранится список вершин, из которых можно перейти в данную вершину. Для ясности рассмотрим 5-й уровень объекта $levels$.

На 5-м уровне содержатся следующие вершины: $s9$, $s6$, $s7$, $s10$, $s11$, $s14$, $s3$. На рисунке 15 представлен граф с выделенным 5-м уровнем.

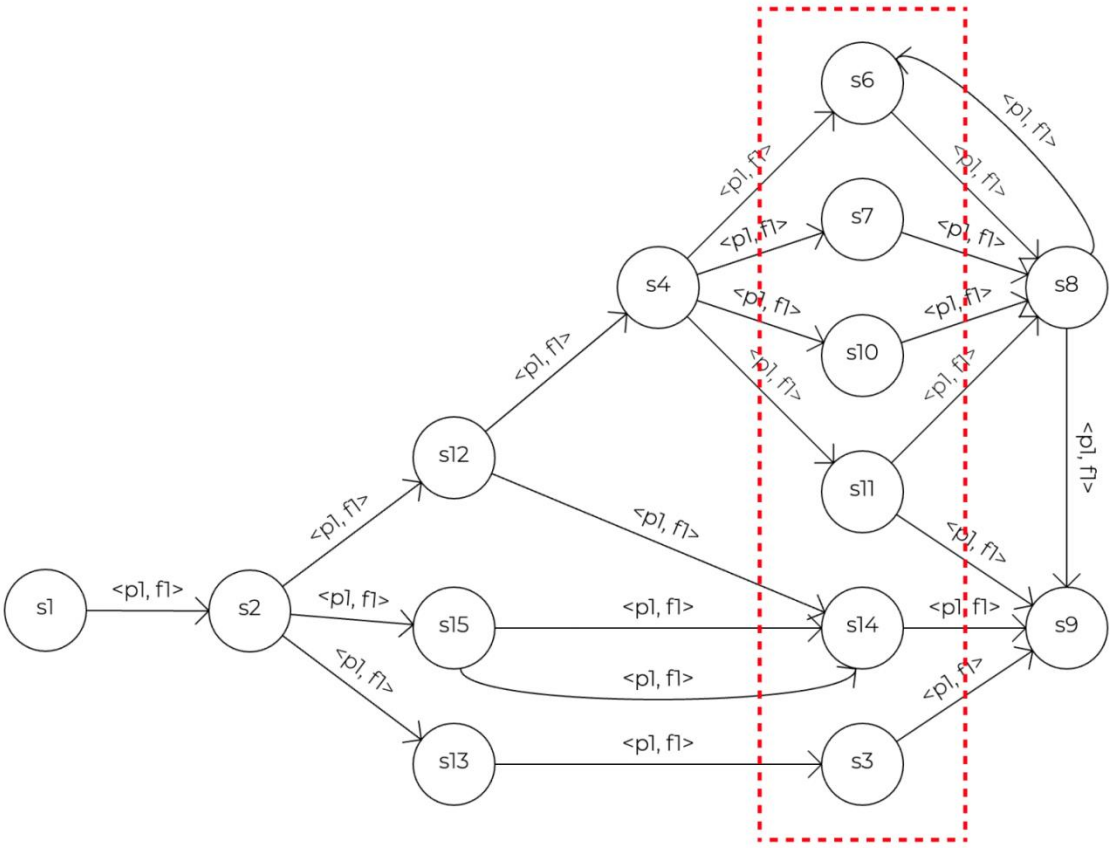


Рисунок 15 – Граф с выделенным 5-м уровнем

Заметим, что в объекте $levels$ на 5-м уровне содержится вершина $s9$, которая в действительности отсутствует на выделенном 5-м уровне графа. Вместо этого, вершина $s9$ находится на 6-м уровне. Подобная коллизия связана с тем, что формирование объекта $levels$ происходит слева направо, от меньшего уровня к большему. К примеру, вершина $s13$ располагается на 3-м уровне, таким образом, после рассмотрения связи $s13 \rightarrow s3$ вершина $s3$ будет расположена на 4-м уровне в объекте $levels$, что также несоответствует действительности – в графе вершина $s3$ расположена на 5-м уровне. Разрешение подобных коллизий происходит после полного формирования объекта $levels$. Все вершины, которые расположены не на своем уровне, помечаются как удаленные, в результате чего они не будут отрисовываться.

После разбиения вершин на уровни, в рамках каждого уровня необходимо корректно распределить вершины по оси ординат. Распределение вершин начинается с уровня, который содержит максимальное число вершин. Между всеми вершинами на данном уровне будет одинаковое расстояние (задается в виде константы). Затем выполняется распределение вершин на остальных уровнях. При распределении вершин, уровень которых меньше уровня с максимальным количеством вершин, для каждой вершины необходимо получить список смежных с ней вершин. В списке смежных вершин необходимо определить минимальную координату по оси Y (координата смежной вершины, которая расположена ниже остальных смежных вершин) и максимальную координату по оси Y (координата смежной вершины, которая расположена выше всех остальных смежных вершин). В результате чего для текущей вершины можно вычислить координату по оси Y:

$$y = minY + \frac{(maxY - minY)}{2}$$

Практически аналогичным образом распределяются вершины, уровень которых больше уровня с максимальным количеством вершин. Отличие заключается в том, что вместо получения списка смежных вершин, необходимо получить список вершин, для которых текущая вершина является смежной.

4. ТЕСТИРОВАНИЕ

4.1. Экспорт графа в формат aDOT

Рассмотрим созданный в редакторе ориентированный граф. Рассматриваемый граф представлен на рисунке 16.

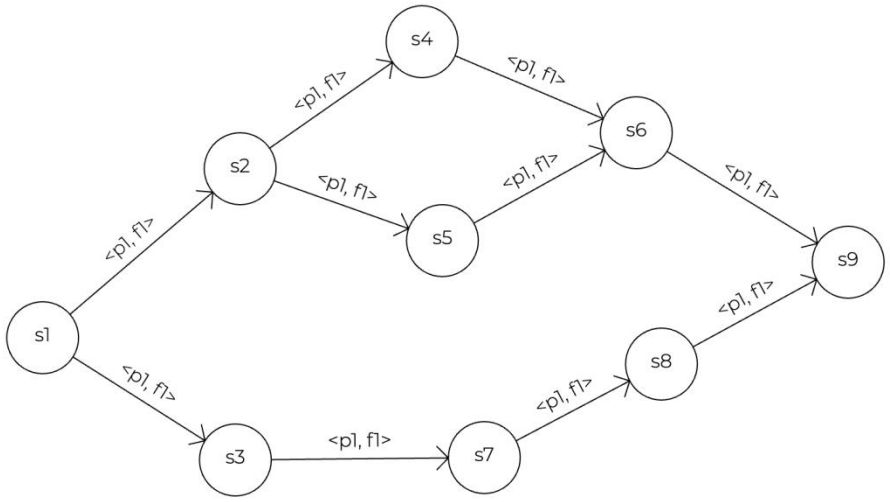


Рисунок 16 – Созданный в редакторе ориентированный граф

В результате операции экспорта в формат aDOT будет получено описание графа в формате aDOT представленное в листинге 6.

Листинг 6 – Полученное, в результате экспорта графа (рисунок 16), описание в формате aDOT

```
digraph TEST
{
// Parallelism
        s1 [parallelism=threading]
        s2 [parallelism=threading]
// Function
        f1 [module=f1_module, entry_func=f1_function]
// Predicates
        p1 [module=p1_module, entry_func=p1_function]
// Transition
        edge_1 [predicate=p1, function=f1]
// Graph model
        __BEGIN__ -> s1
        s1 => s2 [morphism=edge 1]
        s1 => s3 [morphism=edge_1]
```

```
        s2 => s4 [morphism=edge_1]
        s2 => s5 [morphism=edge_1]
        s3 -> s7 [morphism=edge_1]
        s4 -> s6 [morphism=edge_1]
        s5 -> s6 [morphism=edge_1]
        s6 -> s9 [morphism=edge_1]
        s7 -> s8 [morphism=edge_1]
        s8 -> s9 [morphism=edge_1]
        s9 -> __END__
}
```

Данное описание в формате aDOT в точности соответствует созданному в редакторе графу (рисунок 16).

4.2. Импорт графа из формата aDOT

В качестве примера, рассмотрим, полученное ранее, описание графа в формате aDOT (листинг 6). В результате выполнения операции импорта будет получен граф, представленный на рисунке 17.

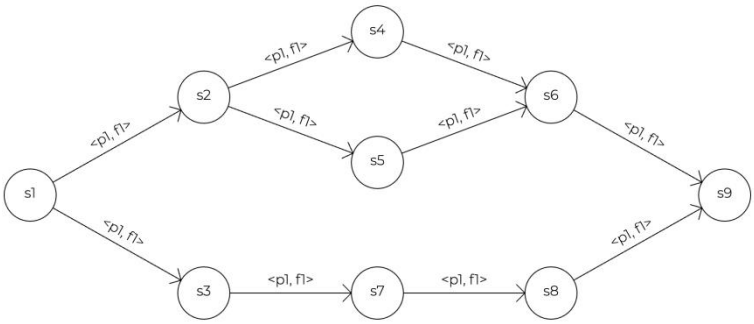


Рисунок 17 – Загруженный из формата aDOT (листинг 6) граф

Построенный граф в точности соответствует своему описанию в формате aDOT. Более того, загруженный из формата aDOT (листинг 6) граф выглядит более строго и эстетично, нежели вручную созданный в редакторе графе (рисунок 16).

Рассмотренный граф (листинг 6, рисунок 17) является достаточно простым, поскольку в нем отсутствуют циклы и обратные ребра. Для того, чтобы убедиться в корректности работы алгоритма, рассмотрим более сложный граф,

который имеет циклы и обратные ребра. Описание графа в формате aDOT представлено в листинге 7.

Листинг 7 – Описание графа в формате aDOT

```
digraph TEST
{
// Parallelism
        s7 [parallelism=threading]
        s11 [parallelism=threading]
        s2 [parallelism=threading]
        s10 [parallelism=threading]
        s1 [parallelism=threading]
// Function
        f1 [module=f1_module, entry_func=f1_function]
// Predicates
        p1 [module=p1_module, entry_func=p1_function]
// Transition
        edge_1 [predicate=p1, function=f1]
// Graph model
        __BEGIN__ -> s1
        s4 -> s6 [morphism=edge_1]
        s5 -> s6 [morphism=edge_1]
        s7 => s6 [morphism=edge_1]
        s7 => s1 [morphism=edge_1]
        s11 => s8 [morphism=edge_1]
        s11 => s2 [morphism=edge_1]
        s12 -> s8 [morphism=edge_1]
        s2 => s4 [morphism=edge_1]
        s2 => s5 [morphism=edge_1]
        s2 => s7 [morphism=edge_1]
        s10 => s11 [morphism=edge_1]
        s10 => s12 [morphism=edge_1]
        s1 => s2 [morphism=edge_1]
        s1 => s10 [morphism=edge_1]
        s6 -> s9 [morphism=edge_1]
        s8 -> s9 [morphism=edge_1]
        s9 -> s6 [morphism=edge_1]
        s9 -> __END__
}
```

Полученный в результате выполнения операции импорта граф представлен на рисунке 18.

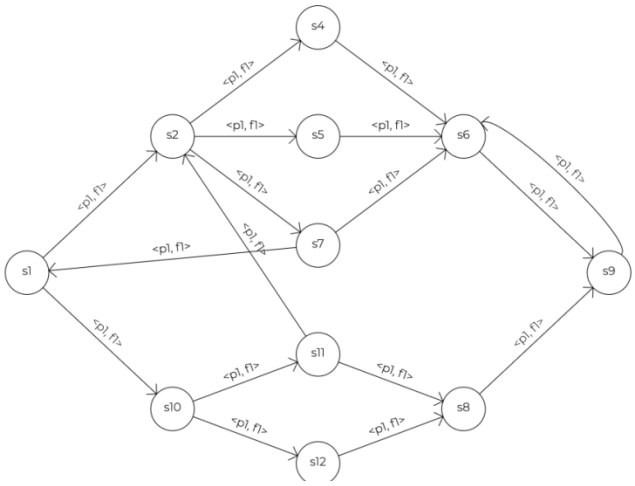


Рисунок 18 – Загруженный из формата aDOT (листинг 7) граф

Полученный граф в точности соответствует описанию в формате aDOT (листинг 7).

4.3. Поиск циклов в графе

В результате выполнения операции поиска циклов, все найденные в графе циклы будут подсвечены. На рисунке 19 представлен ориентированный граф, в котором были найдены и подсвечены все циклы.

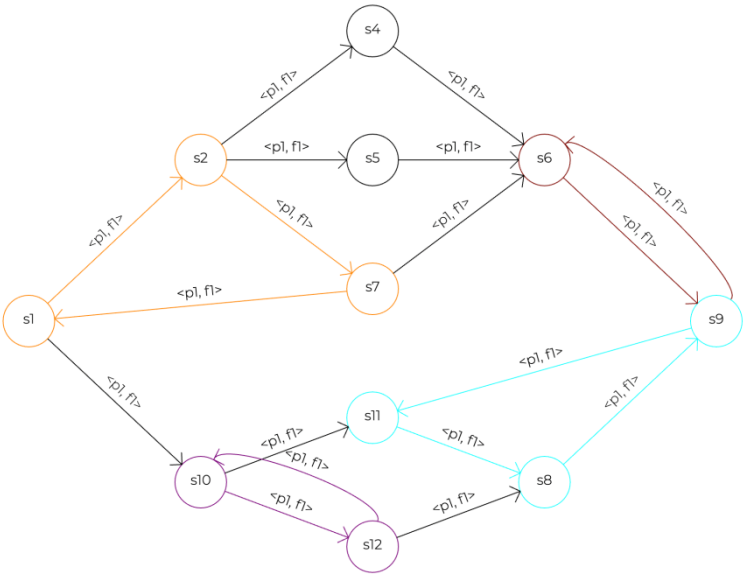


Рисунок 19 – Ориентированный граф с подсвеченными циклами

ЗАКЛЮЧЕНИЕ

В результате проведенной работы был разработан редактор графов, который ориентирован на работу с форматом описания графов aDOT. Разработанный редактор предоставляет возможность создавать и редактировать граф, сохранять созданный граф в формате aDOT, загружать граф из этого формата, а также находить циклы в графе.

В процессе подготовки к программной реализации были изучены популярные архитектуры фронтенд приложений, рассмотрены различные подходы для работы с графикой в JavaScript. Также была разработана собственная модель хранения графа, в основе которой лежит представление графа в виде списка смежности. Разработанная модель позволяет хранить дополнительные атрибуты необходимые для поддержки формата aDOT.

В процессе программной реализации, для решения той или иной задачи, были использованы различные популярные алгоритмы и структуры данных. Для решения некоторых задач, после проведения обзора существующих решений, было принято решение разработать собственные алгоритмы.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Robert van Liere. CSE. A Modular Architecture for Computational Steering. 2015.

2. А.В. Коргин М.В Емельянов В.А. Ермаков. Применение LabVIEW для решения задач сбора и обработки данных измерений при разработке систем мониторинга несущих конструкций. 2013.

3. Mortensen Mikael Langtangen Hans Petter Wells Garth N. A FeniCS-Based Programming Framework for Modeling Turbulent Flow by the Reynolds-Averaged Navier-Stokes Equations. 2011.

4. Соколов А.П., Першин А.Ю. Графоориентированный программный каркас для реализации сложных вычислительных методов. 2019.

5. Соколов А.П., Першин А.Ю. Патент на изобретение RU 2681408. Способ и система графо-ориентированного создания масштабируемых и сопровождаемых программных реализаций сложных вычислительных методов. 2019.

6. Соколов А.П. Описание формата данных aDOT (advanced DOT) [Электронный ресурс]. Облачный сервис SA2 Systems. [Офиц. Сайт]. 2020.

7. Верхнеуровневая архитектура фронтенда. Лекция Яндекса. URL: https://habr.com/ru/company/yandex/blog/425611/

8. SVG или canvas? URL: https://habr.com/ru/company/ruvds/blog/476292/

9. Обход графа: поиск в глубину и поиск в ширину простыми словами на примере JavaScript. URL: https://habr.com/ru/post/504374/

ПРИЛОЖЕНИЕ А

Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Московский государственный технический университет
имени Н.Э. Баумана (национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

АКТ
проверки выпускной квалификационной работы

Студент группы *РК6-82Б*

Ершов Виталий Алексеевич
(Фамилия, имя, отчество)

Тема выпускной квалификационной работы: *разработка web-ориентированного редактора графов*

Выпускная квалификационная работа проверена, размещена в ЭБС «Банк ВКР» в полном объеме и соответствует / не соответствует требованиям, изложенным в Положении о порядке
ненужное зачеркнуть
подготовки и защиты ВКР.
Объем заимствования составляет _____% текста, что с учетом корректного заимствования соответствует / не соответствует требованиям к ВКР
_____
ненужное зачеркнуть *бакалавра, специалиста, магистра*

Нормоконтролёр _____ С.В. Грошев
(подпись) (ФИО)

Согласен:
Студент _____ В.А. Ершов
(подпись) (ФИО)

Дата: