



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

ФАКУЛЬТЕТ «Робототехники и комплексной автоматизации»

КАФЕДРА «Системы автоматизированного проектирования (РК-6)»

## РАСЧЁТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

к курсовому проекту

по дисциплине «Модели и методы анализа проектных решений»

на тему

«Разработка web-ориентированного редактора графовых моделей»

Студент РК6-72Б  
группа

Руководитель КП

Консультант

\_\_\_\_\_  
подпись, дата

2022.01.20

\_\_\_\_\_  
подпись, дата

Ершов В.А.  
ФИО

Соколов А.П.  
ФИО

Першин А.Ю.  
ФИО

**ПРОСМОТРЕНО**

**Соколов А.П. 22:14, 20/1/22**

**Рекомендованная  
оценка отлично!**

Москва, 2021

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

УТВЕРЖДАЮ

Заведующий кафедрой РК-6  
индекс

\_\_\_\_\_ А.П. Карпенко

«\_\_\_\_\_» \_\_\_\_\_ 2021 г.

## ЗАДАНИЕ

### на выполнение курсового проекта

Студент группы: РК6-72Б

Ершов Виталий Алексеевич

(фамилия, имя, отчество)

Тема курсового проекта : Разработка web-ориентированного редактора графовых моделей

Источник тематики (кафедра, предприятие, НИР): кафедра

Тема курсового проекта утверждена на заседании кафедры «Системы автоматизированного проектирования (РК-6)», Протокол № \_\_\_\_\_ от «\_\_\_\_\_» \_\_\_\_\_ 2021 г.

#### Техническое задание

**Часть 1.** Аналитический обзор литературы.

*В рамках аналитического обзора литературы необходимо проанализировать актуальность исследований в области оптимизации процесса реализации вычислительных методов, найти существующие программные инструменты, позволяющие оптимизировать процесс разработки. Должны быть определены перспективы использования графоориентированного подхода для реализации вычислительных методов.*

**Часть 2.** Математическая постановка задачи, разработка архитектуры программной реализации, программная реализация.

*Необходимо описать архитектуру разрабатываемого редактора графов, разработать web-ориентированное приложение, позволяющее создавать графовые модели вычислительных методов.*

**Часть 3.** Проведение вычислительных экспериментов, отладка и тестирование.

*В рамках тестирования необходимо представить ряд примеров, показывающих реализованные в редакторе функциональные возможности.*

Оформление курсового проекта :

Расчетно-пояснительная записка на 29 листах формата А4.

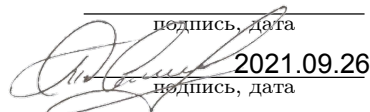
Перечень графического (иллюстративного) материала (чертежи, плакаты, слайды и т.п.):

<i>количество: 8 рис., 0 табл., 9 источн.</i>
<i>[здесь следует ввести количество чертежей, плакатов]</i>

Дата выдачи задания «26» сентября 2021 г.

**Студент**

**Руководитель    курсового проекта**

\_\_\_\_\_  
подпись, дата  
  
2021.09.26  
\_\_\_\_\_  
подпись, дата

Ершов В.А.  
ФИО

Соколов А.П.  
ФИО

Примечание: Задание оформляется в двух экземплярах: один выдается студенту, второй хранится на кафедре.

## РЕФЕРАТ

Данная работа посвящена разработке web-ориентированного редактора графов, который является очень полезным инструментом при использовании графоориентированного программного каркаса для реализации сложных вычислительных методов, представленного в работе [1]. Разрабатываемый редактор позволяет создавать графовые модели в соответствии с рассматриваемым графоориентированным подходом, экспортировать созданные графы в формате aDOT, а также загружать заранее подготовленные графовые модели из этого формата. В работе описана важность данной разработки, представлена архитектура разрабатываемого редактора и рассмотрены реализованные в редакторе функциональные возможности, а также приведен ряд примеров, демонстрирующих его работу.

**Тип работы:** курсовой проект .

**Тема работы:** *«Разработка web-ориентированного редактора графовых моделей».*

**Объект исследования:** Применение графоориентированного подхода для реализации сложных вычислительных методов.

# СОДЕРЖАНИЕ

<b>ВВЕДЕНИЕ</b> .....	6
Обзор источников литературы .....	6
<b>1. Постановка задачи</b> .....	9
1.1. Концептуальная постановка задачи .....	9
<b>2. Программная реализация</b> .....	10
2.1. Архитектура .....	10
2.2. Реализация .....	11
<b>3. Тестирование и отладка</b> .....	19
3.1. Экспорт графа в формате aDOT .....	19
3.2. Импорт графа из формата aDOT .....	20
<b>4. Анализ результатов</b> .....	24
<b>ЗАКЛЮЧЕНИЕ</b> .....	28
<b>Литература</b> .....	29

## ВВЕДЕНИЕ

При разработке любого программного обеспечения разработчики ставят перед собой несколько очень важных задач: эффективность и гибкость дальнейшей поддержки программного обеспечения. В современной разработке существует множество паттернов проектирования, архитектур и вспомогательных систем, которые позволяют быстрее создавать программное обеспечение и в дальнейшем упрощают его поддержку для новых разработчиков. Таким образом, даже в небольших проектах, над которыми работает небольшая команда разработчиков стараются использовать системы контроля версий, юнит-тестирование. Если приложение представляет из себя API при его разработке используют специальные системы для быстрой документации API. Такие подходы считаются стандартом при разработке программного обеспечения рассчитанное на удовлетворение потребностей конечных пользователей - разработка мобильных приложений, проектирование и реализация API и прочее.

Однако при появлении первых ЭВМ в 1930-х годах [2] программирование использовалось для решения научных задач. Научное программирование сильно отличается от других видов программирования, при разработке научного программного обеспечения очень важно получить корректный и стабильный конечный продукт, а также четко разделить интерфейсную и научную часть. Из этого следует, что разработчик научного программного обеспечения должен быть экспертом в предметной области, а сам разработчик обычно является конечным пользователем [3], в то время как в индустриальном программировании разработчик зачастую не является конечным пользователем и от него не требуется быть экспертом в предметной области разрабатываемого приложения.

Также научное программирование отличается от индустриального программирования тем, что стандарты проектирования научного программного обеспечения вырабатываются существенно медленнее или не вырабатываются вовсе, что приводит к отсутствию каких-либо системных подходов к разработке. Это приводит к сложностям при валидации и дальнейшей поддержке кода. Так, код, написанный эффективно и корректно, может оказаться бесполезным в том если поддержка кода оказывается затруднительной, это приводит к формированию и накоплению "best practices" при программировании численных методов. Вследствие чего стали появляться системы, которые позволяют минимизировать написание кода и снизить трудозатраты на его поддержку.

### Обзор источников литературы

В основном существующие платформы используют визуальное программирование [4]. Одними из самых популярных и успешных разработок в этой области являются Simulink и LabView. Simulink позволяет моделировать вычислительные методы с помощью графических блок-диаграмм и может быть интегрирован со средой MATLAB. Также Simunlink позволяет автоматически генерировать код на языке на C для реализации вычислительного метода в

режиме реального времени. LabView используется для аналогичных задач - моделирование технических систем и устройств [5]. Среда позволяет создавать виртуальные приборы с помощью графической блок-диаграммы, в которой каждый узел соответствует выполнению какой-либо функции. Представление программного кода в виде такой диаграммы делает его интуитивно понятным инженерам и позволяет осуществлять разработку системы более гибко и быстро. В составе LabView есть множество специализированных библиотек для моделирования систем из конкретных технических областей.

Существует множество других систем и языков программирования для реализации вычислительных методов, однако, каждый из них является узкоспециализированным и решает определенную задачу. Так, например, система визуального моделирования FEniCS [6] используется для решения задач с использованием метода конечных элементов. Система имеет открытый исходный код, а также предоставляет удобный интерфейс для работы с системой на языках Python или C++. FEniCS предоставляет механизмы для работы с конечно-элементными расчетными сетками и функциями решения систем нелинейных уравнений, а также позволяет вводить математические модели в исходной интегрально-дифференциальной форме.

Отдельного упоминания стоит система TensorFlow. TensorFlow представляет из себя библиотеку с открытым исходным кодом, которая используется для машинного обучения. Аналогично LabView и Simulink, TensorFlow позволяет строить программные реализации численных методов. Стоит обратить внимание, что в основе TensorFlow лежит такое понятие как граф потока данных. В самом графе ребра - тензоры, представляют из себя многомерные массивы данных, а узлы - математические операции над ними.

Применение ориентированных 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 (АП). Эта задача предполагает, как правило, постановку и решение некоторой обратной задачи, которая в свою очередь, часто, решается путём 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 графориентированный программный каркас для реализации сложных вычислительных методов, теоретические основы представлены в ра-

боте [1], а принципы применения графоориентированного подхода зафиксированы в патенте [7]. Заметим, что в отличие от TensorFlow, в представленном графоориентированном программном каркасе узлы определяют фиксированные состояния общих данных, а ребра определяют функции преобразования данных. Для описания графовых моделей был разработан формат aDOT, который расширяет формат описания графов DOT [8], входящий в пакет утилит визуализации графов Graphviz. В aDOT были введены дополнительные атрибуты и определения, которые описывают функции-предикаты, функции-обработчики и функции перехода в целом. Подробное описание формата aDOT приведено в [9].

В описанном в работе методе вводятся такие понятия как функции-обработчики и функции-предикаты, заметим, что 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 ускоряет процесс реализации вычислительного метода - функции-перехода могут разрабатываться параллельно несколькими независимыми разработчиками. Также, возможность параллелизации процесса разработки позволяет организовать модульное тестирование и документирование разрабатываемого кода.



# 1 Постановка задачи

## 1.1 Концептуальная постановка задачи

В разработанном А.П.Соколовым и А.Ю.Першиным графоориентированном программном каркасе, для описания графовых моделей был разработан формат aDOT, который является расширением формата описания графов DOT. Для визуализации графов, описанных с использованием формата DOT, используются специальные программы визуализации. Самой популярной из них является пакет утилит Graphviz, пакет позволяет визуализировать графы описанные в формате DOT и получать изображение графа в разных форматах: PNG, SVG и т.д. Формат aDOT расширяет формат DOT с помощью дополнительных атрибутов и определений, которые описывают функции-предикаты, функции-обработчики и функции-перехода в целом. Таким образом, становится очевидно, что для построения графовых моделей с использованием формата aDOT необходим графический редактор.

Разрабатываемый графический редактор должен удовлетворять следующим требованиям:

- Разрабатываемый редактор должен представлять из себя web-приложение - это позволяет сделать редактор независимым от операционной системы.
- Редактор должен предоставлять возможность создавать ориентированный граф с нуля
- Возможность экспортировать созданный граф в формат aDOT
- Возможность загрузить граф из формата aDOT

## 2 Программная реализация

### 2.1 Архитектура

Редактор графов является web-приложением, соответственно весь программный код написан на языке JavaScript. Также стоит заметить, что приложение должно работать быстро, пользователь должен сразу видеть результат своих действий, будь то простое добавление очередной вершины или сохранение графа в формате aDOT, следовательно вся бизнес-логика должна выполняться на стороне клиента, то есть через JavaScript. В программном коде приложения для хранения графа реализован класс Graph, но в JavaScript концепция объектно-ориентированного программирования немного отличается от традиционной, поскольку в JavaScript ключевое слово `class` является лишь синтаксическим сахаром и на самом деле класс представляет из себя объект. Также стоит обратить внимание, что приватные методы класса остаются доступными внешнему коду, а сама приватность является лишь пометкой для разработчика, что этот метод не планируется вызывать через объект класса во внешнем коде.

В программном коде реализован один класс Graph, который предоставляет все необходимые методы для создания и работы с графом, UML – диаграмма класса представлена на рисунке (1).

Graph	
+ vertices:	Object
+ predicates:	Object
+ functions:	Object
+ edges:	Object
+ createdVerticesLabels:	Set
+ createdVerticesPositions:	Array
+ vertexID:	Number
+ edgeID:	Number
+ radius:	Number
+ borderWidth:	Number
+ arrowheadSize:	Number
+ AddVertex(Number, Number):	Bool
+ DeleteVertex(String):	Void
+ AddEdge(String, String):	Bool
+ DeleteEdge(String):	Void
+ ExportADOT(String, String):	Bool
+ ImportADOT(String):	Void
+ FindCycles():	Void
+ Clear():	Void

Рисунок 1. UML – диаграмма класса Graph

В UML – диаграмме представлены только публичные методы класса Graph, которые вызываются из внешнего кода и реализуют всю бизнес-логику приложения. Для уменьшения объема диаграммы приватные методы были опущены.

## 2.2 Реализация

### 1 Создание примитивов на странице

Как было ранее сказано, редактор графов является web-приложением написанным на языке JavaScript. Для создания графа необходима возможность создавать простейшие примитивы: окружности, линии и прочее. JavaScript предоставляет несколько вариантов для решения этой задачи:

- Использование HTML – элемента canvas
- Работа с векторной графикой svg

Использование canvas не подходит поскольку после создания примитива невозможно получить его свойства, а следовательно и отредактировать их. Таким образом, необходимо работать с векторной графикой svg, использование svg позволяет создавать примитивы, которые создаются как HTML теги с набором свойств, что позволяет редактировать или удалять созданные примитивы. Однако работать с svg без использования сторонних библиотек достаточно затруднительно, программный код сильно увеличивается и разработка функций для отрисовки примитивов занимает существенно больше времени. Для решения этой задачи подходит библиотека d3.js. Библиотека предоставляет набор инструментов для визуализации данных на странице, который состоит из нескольких десятков небольших модулей, каждый из которых решает свою задачу. В программном коде возможности d3 используются для создания svg примитивов на странице, например создание вершины (листинг 2.1).

---

```

1  svg
2    .append("circle")
3    .attr("cx", x)
4    .attr("cy", y)
5    .attr("r", radius)
6    .attr("stroke-width", borderWidth)
7    .attr("id", id)
8    .attr("fill", "#FFFFFF")
9    .attr("stroke", "#000000")
10   .attr("class", "vertex");

```

---

Листинг 2.1. Пример создание вершины с использование библиотеки d3

### 2 Хранение информации о графовой модели

Для того, чтобы иметь возможность гибко редактировать графовую модель, а также иметь возможность экспортировать граф в формат aDOT, необходимо хранить множество свойств графа. Так, при удалении вершины, помимо удаления примитивов со страницы, необходимо удалить всю информацию о вершине и связанных с ней ребрах, в противном случае при

сохранении графа в формате aDOT в файле будет содержаться информация об уже удаленных вершинах или ребрах.

В разделе 2.1 описано, что вся бизнес-логика выполняется на стороне клиента. Это касается и хранения данных. Все свойства графа хранятся в оперативной памяти. Для хранения большей части свойств в программном коде используются объекты. Объект - это мощная структура данных в javascript - используется для хранения коллекций разных значений. Рассмотрим на примере как храниться информации о вершинах. В рассматриваемом примере (листинг 2.2) представлены хранимые в объекте свойства одной вершины из которой выходит одно ребро.

---

```

1  {
2      "edges": [
3          {
4              "direction": "from",
5              "function": "f1",
6              "predicate": "p1",
7              "label": "<p1, f1>",
8              "metadata": {
9                  "edgeID": "edge1",
10                 "pathID": "edge1_path",
11                 "labelID": "edge1_label",
12             }
13             "type": "straight",
14             "value": "vertex2",
15         },
16     ],
17     "metadata": {
18         "label": "s1",
19         "label_metadata": {
20             "pathID": "vertex1_path",
21             "labelID": "vertex1_label",
22         },
23         "position": {
24             "x": 227.609375,
25             "y": 211,
26         },
27     }
28 }
```

---

Листинг 2.2. Пример хранение вершины в оперативной памяти

Аналогичным образом с помощью объектов храниться информация о предикатах и функциях, которые пользователь вводит в процессе создания графовой модели. Для хранения более простых блоков данных используются массивы, например, хранение позиций уже созданных вершин, или хеш-таблицы, например, хранение уже созданных меток вершин.

### 3 Реализация основных функций бизнес-логики

В редакторе реализован весь необходимый функционал для создания ориентированного графа, а именно: создание и удаление вершин, создание и удаление ребер. Заметим, что это достаточно тривиальные задачи, в которых требуется получить данные от пользователя, а затем корректно их сохранить. Сохранение графовой модели в формате aDOT также является тривиальной задачей - требуется получить все необходимые данные, которые были сохранены в процессе создания графовой модели и сформировать из них корректное описание графовой модели на языке aDOT.

Более интересной задачей является загрузка графа из формата aDOT, сложность заключается в корректной человекопонятной визуализации графа. В первую очередь необходимо определить какие критерии визуализации необходимы для рассматриваемой задачи, существует несколько основных критериев исходя из которых выбирается алгоритм для визуализации:

- Пересечения - минимизация общего числа пересечений ребер
- Области - минимизация размеров областей
- Длина ребер - минимизация общей длины ребер
- Универсальная длина ребер - минимизация различий в длинах ребер
- Смежные вершины расположены рядом, несмежные далеко
- Группировка по кластерам - если существуют множества связанные друг с другом сильнее чем с остальным графом они образуют кластер
- Распределение вершин и/или ребер равномерно

Также стоит обратить внимание, что при визуализации графа необходимо получить его укладку. Укладка - это получение координат для каждой вершины, в основном речь идет о координатах на плоскости. Существует три основных метода укладок:

- Force-Directed and Energy-Based. В данных методах используется симуляция физических сил. Вершины - заряженные частицы, которые отталкиваются друг от друга, а ребра - упругие связи стягивающие смежные вершины. Минусом данного типа укладки является вычислительная сложность - для каждой вершины графа необходимо рассчитать силы, действующие на нее. Алгоритмы использующие данный тип укладки: Fruchterman-Reingold, Force Atlas, OpenOrd.
- Dimension Reduction. В данном методе решается задача снижения размерности. Граф описывается матрицей смежности к которой применяются алгоритмы снижения размерности такие как UMAP, PCA и прочие.
- Feature-Based Layout. Укладка вершин по определенному свойству, которое присутствует у всех вершин

В разрабатываемом редакторе требуется построить ориентированный граф в котором стартовая вершина будет располагаться левее всех остальных, а конечная вершина правее всех

остальных. Ни один из рассмотренных методов укладки не позволяет должным образом реализовать данное требование, поэтому для получения укладки загружаемого графа был разработан алгоритм укладки.

Алгоритм основан на разбиении графа по уровням. Рассмотрим следующее aDOT-определение графовой модели  $G$  (листинг 2.3).

---

```

1 digraph G {
2   // Parallelism
3   s1 [parallelism=threading]
4   // Graph definition
5   __BEGIN__ -> s1
6   s4 -> s6 [morphism=edge_1]
7   s5 -> s6 [morphism=edge_1]
8   s1 => s2 [morphism=edge_1]
9   s1 => s3 [morphism=edge_1]
10  s2 -> s4 [morphism=edge_1]
11  s3 -> s5 [morphism=edge_1]
12  s6 -> __END__
13 }
```

---

Листинг 2.3. Пример aDOT-определение простейшей графовой модели  $G$

Если представить такой граф на бумаге, то становится очевидно, что каждый набор вершин имеет одну координату по оси  $X$ , а связанные с ними вершины находятся правее по координате  $X$ , таким образом становится очевидным разбиение графа по уровням. На рисунке (2) представлен этот граф, разбитый на уровни.

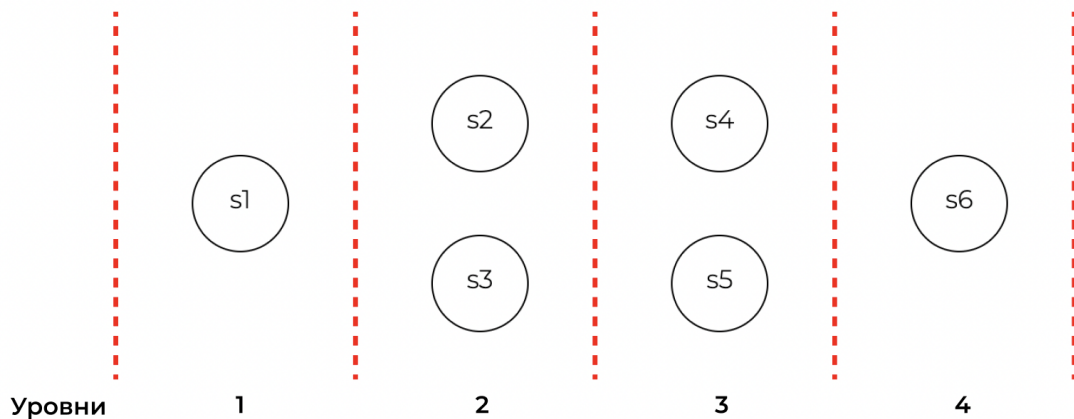


Рисунок 2. Узлы графовой модели  $G$ , представленные на разных "уровнях"

Разбиение графа по уровням является основой алгоритма визуализации. Рассмотрим более сложную графовую модель на языке aDOT (листинг 2.4).

---

```

1 digraph TEST
2 {
3   // Parallelism
4     s11 [parallelism=threading]
5     s4 [parallelism=threading]
6     s12 [parallelism=threading]
7     s15 [parallelism=threading]
8     s2 [parallelism=threading]
9     s8 [parallelism=threading]
10  // Functions
11    f1 [module=DEFAULT_VALUE, entry_func=DEFAULT_VALUE]
12  // Predicates
13    p1 [module=DEFAULT_VALUE, entry_func=DEFAULT_VALUE]
14  // Edges
15    edge_1 [predicate=p1, function=f1]
16  // Graph model description
17    __BEGIN__ -> s1
18    s6 -> s8 [morphism=edge_1]
19    s7 -> s8 [morphism=edge_1]
20    s10 -> s8 [morphism=edge_1]
21    s11 => s8 [morphism=edge_1]
22    s11 => s9 [morphism=edge_1]
23    s14 -> s9 [morphism=edge_1]
24    s3 -> s9 [morphism=edge_1]
25    s4 => s6 [morphism=edge_1]
26    s4 => s7 [morphism=edge_1]
27    s4 => s10 [morphism=edge_1]
28    s4 => s11 [morphism=edge_1]
29    s12 => s4 [morphism=edge_1]
30    s12 => s14 [morphism=edge_1]
31    s13 -> s3 [morphism=edge_1]
32    s15 => s14 [morphism=edge_1]
33    s15 => s14 [morphism=edge_1]
34    s2 => s12 [morphism=edge_1]
35    s2 => s13 [morphism=edge_1]
36    s2 => s15 [morphism=edge_1]
37    s1 -> s2 [morphism=edge_1]
38    s8 => s9 [morphism=edge_1]
39    s8 => s6 [morphism=edge_1]
40    s9 -> __END__
41 }
```

---

Листинг 2.4. Пример aDOT-определения графовой модели TEST

Полученный в результате граф представлен на рисунке (3).

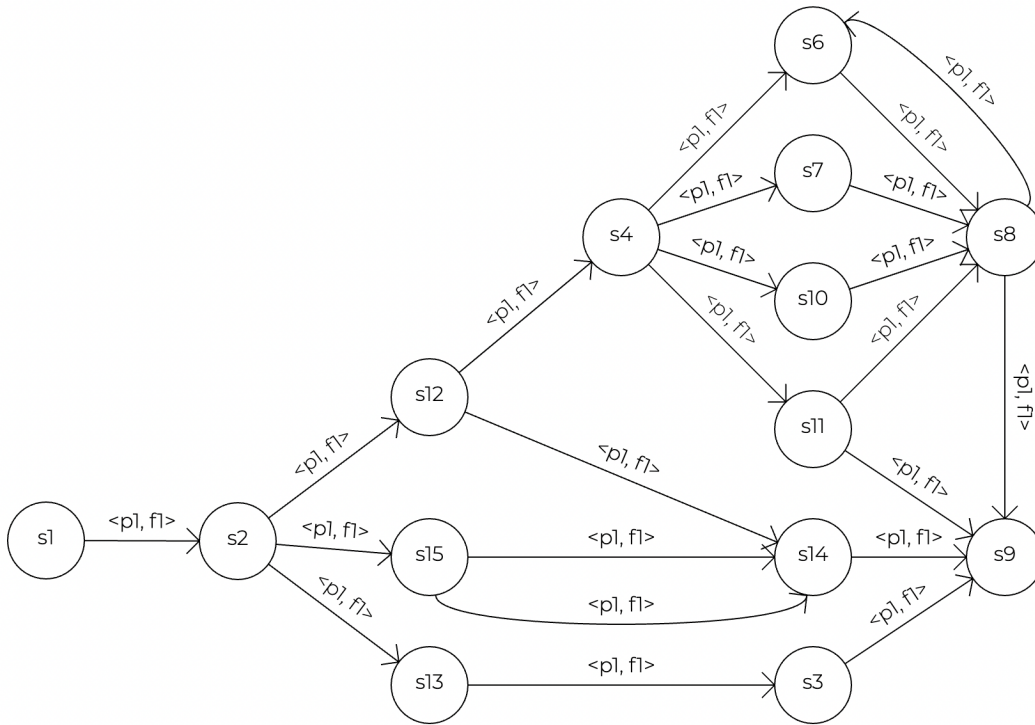


Рисунок 3. Визуализация графовой модели TEST

Алгоритм выполняется в два этапа: разбиение вершин по уровням, так, будет получено расположение вершин по координате X, затем в рамках каждого уровня распределение вершин по координате Y. Для хранения уровней вершин используется объект `levels`. Для рассматриваемого в примере графа после разбиения вершин объект `levels` содержит следующие данные (листинг 2.5)

---

```

1 {
2   "1": [{"s1": []}],
3   "2": [{"s2": ["s1"]}],
4   "3": [{"s12": ["s2"]}, {"s13": ["s2"]}, {"s15": ["s2"]}],
5   "4": [{"s4": ["s12"]}],
6   "5": [{"s9": ["s14", "s3"]}, {"s6": ["s4"]}, {"s7": ["s4"]}, {"s10": ["s4"]},
7   {"s11": ["s4"]}, {"s14": ["s12", "s15"]}, {"s3": ["s13"]}],
8   "6": [{"s8": ["s6", "s7", "s10", "s11"]}, {"s9": ["s11", "s14", "s3"]}
9 }
```

---

Листинг 2.5. Объект `levels` после разбиения вершин по уровням

Ключами объекта `levels` являются номера уровней, представленные в формате `string`. Свойствами являются массивы, на один уровень - один массив. Каждый элемент массива содержит информацию об одной вершине на этом уровне, следовательно количество вершин на уровне - это размер массива. Заметим, что элемент массива это не просто `string` с названием



вершины, а объект. Этот объект содержит один ключ - название вершины на этом уровне, а свойством является массив, который содержит список вершин из которых перешли в эту вершину (переход только по одному ребру). В качестве примера рассмотрим уровень 5 (листинг 2.6) объекта levels, который был представлен ранее (листинг 2.5).

```

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

```

Листинг 2.6. Пример уровень 5 объекта levels

Уровень 5 в графе содержит следующие вершины: s9, s6, s7, s10, s11, s14, s3. В вершину s6 приходит ребро из вершины s4, таким образом, по ключу s6 находится массив с один элементом s6. На рисунке (4) представлен граф с выделенным уровнем 5.

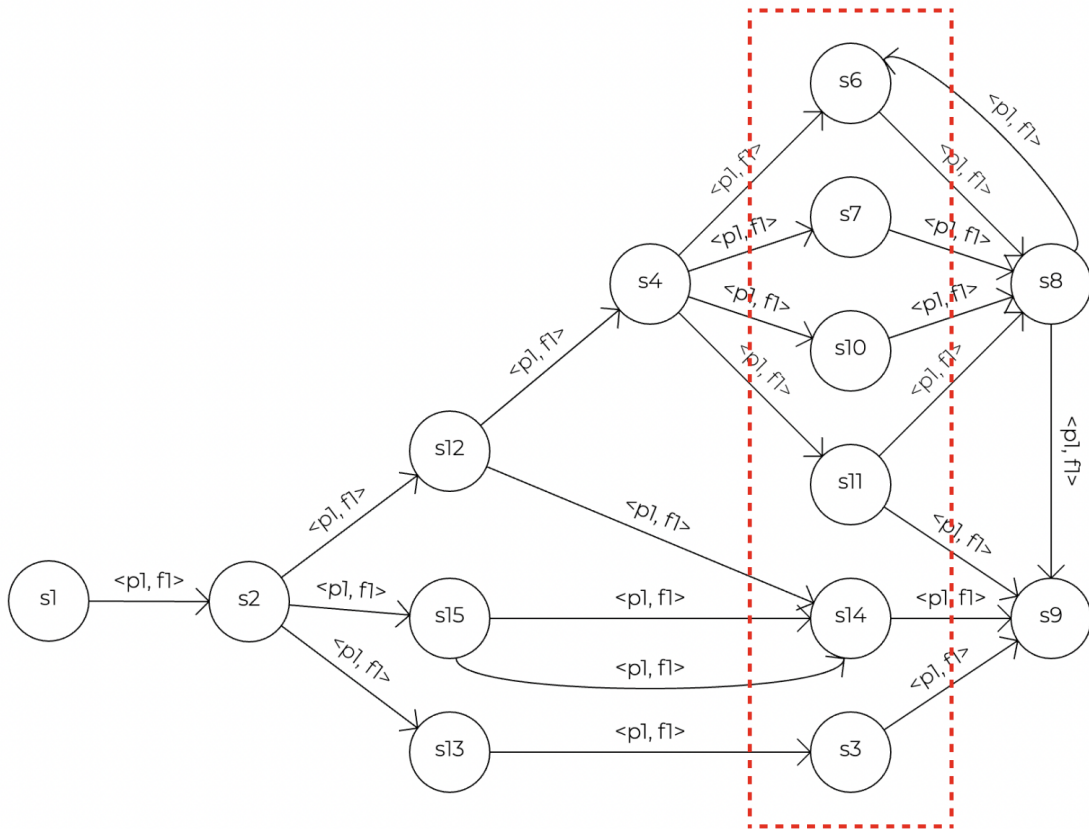


Рисунок 4. Выделенный уровень 5 на рассматриваемом графе

Обратим внимание на то, что в объекте levels для уровня 5 содержится вершина s9 которой нет на уровне 5, а она присутствует только на уровне 6. Заполнение объекта levels происходит слева-направо, то есть от меньшего уровня к большему, например, в графе представленном выше есть связь  $s13 \rightarrow s3$ , таким образом пока в объект levels не будет записана вершина s13, мы не сможем записать связанную с ней вершину s3.

Обратным образом происходит дублирование вершин, если обратиться к описанию графовой модели представленной выше, то можно заметить такую связь:  $s1 \rightarrow s2 \rightarrow s13 \rightarrow s3 \rightarrow s9$ . При начальной инициализации объекта `levels`  $s1$  будет находиться на уровне 1,  $s2$  будет находиться на уровне 2 и так далее. Таким образом, вершина  $s3$  будет находиться на уровне 4 - это некорректное расположение. Для разрешения подобных коллизий после начальной инициализации объекта `levels` "в лоб" предусмотрено множество дополнительных проверок. Таким образом, на выходе получается корректно сформированный объект `levels` и сформированный объект, который будет хранить информацию о вершинах которые находятся не на своем уровне, эти вершины не будут отрисовываться, но при этом они будут учитываться при размещении по оси  $Y$  связанных с ними вершин, следовательно просто удалить такие вершины из объекта `levels` нельзя.

После получения разбиения на уровни необходимо в рамках каждого уровня разместить вершины по координате  $Y$ . Размещение начинается с уровня, который содержит больше всего вершин, будем называть его самым большим. На данном уровне все вершины строятся с одинаковым вертикальным отступом. Затем необходимо построить вершины слева и справа от этого уровня. При построении вершин слева от самого большого уровня для каждой вершины необходимо получить список вершин в которые приходят ребра из рассматриваемой вершины, а затем из этого списка получить две координаты  $Y$ : координата  $Y$  самой высокой вершины в списке, координата  $Y$  самой низкой вершины в списке. Затем можно получить координату рассматриваемой вершины:

$$cy = firstY + \frac{(lastY - firstY)}{2}$$

Аналогичным образом строим вершины справа от самого большого уровня, только получаем список вершин из которых приходят ребра в рассматриваемую вершину.

## 3 Тестирование и отладка

### 3.1 Экспорт графа в формате aDOT

Рассмотрим работу редактора на примерах. Все примеры не имеют никакого математического смысла и приведены лишь для демонстрации работоспособности функционала редактора. Рассмотрим ориентированный граф созданный в редакторе с нуля, граф представлен на рисунке (5).

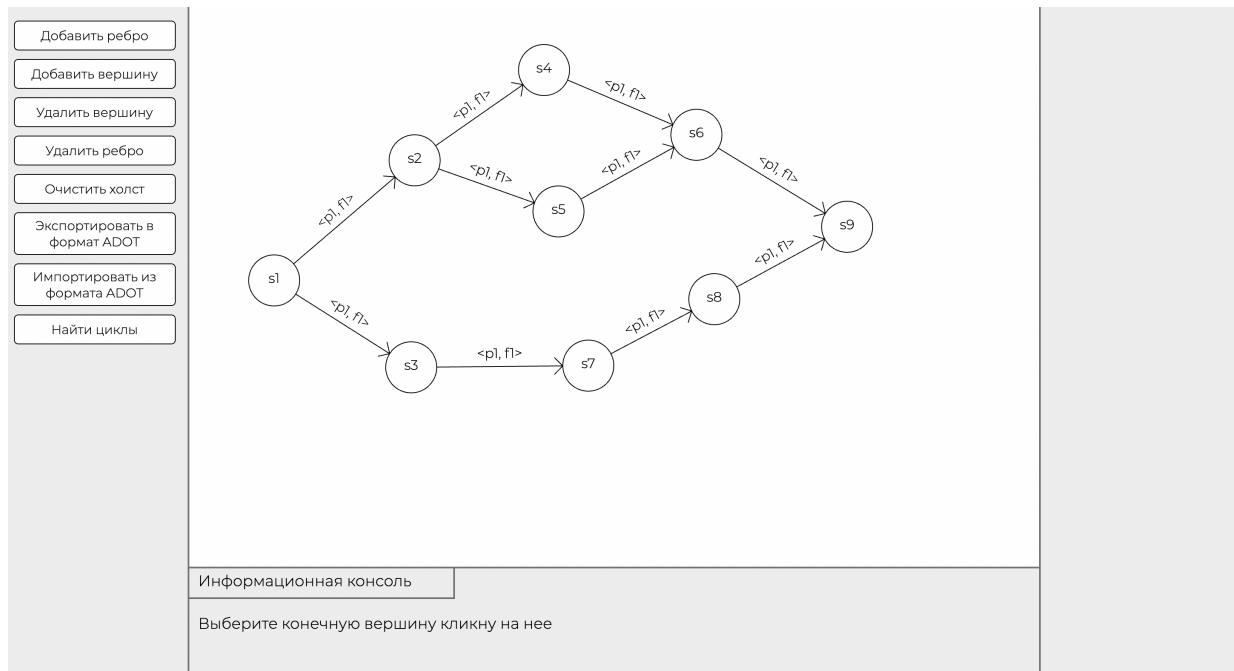


Рисунок 5. Ориентированный граф созданный с нуля в редакторе

Экспортируем построенный граф в формате aDOT. В результате будет получено следующее описание графовой модели в формате aDOT (листинг 3.1).

```

1 digraph TEST
2 {
3   // Parallelism
4   s1 [parallelism=threading]
5   s2 [parallelism=threading]
6   // Function
7   f1 [module=f1_module, entry_func=f1_function]
8   // Predicates
9   p1 [module=p1_module, entry_func=p1_function]
10  // Transition
11  edge_1 [predicate=p1, function=f1]
12  // Graph model
13  __BEGIN__ -> s1
14  s1 => s2 [morphism=edge_1]
15  s1 => s3 [morphism=edge_1]
16  s2 => s4 [morphism=edge_1]
17  s2 => s5 [morphism=edge_1]
```

```

18  s3 -> s7 [morphism=edge_1]
19  s4 -> s6 [morphism=edge_1]
20  s5 -> s6 [morphism=edge_1]
21  s6 -> s9 [morphism=edge_1]
22  s7 -> s8 [morphism=edge_1]
23  s8 -> s9 [morphism=edge_1]
24  s9 -> __END__
25  }

```

Листинг 3.1. Полученное описание графовой модели (5) в формате aDOT

Ранее было сказано, что задача экспорта графа является достаточно тривиальной, поэтому перейдем к рассмотрению примеров загрузки графа из формата aDOT.

### 3.2 Импорт графа из формата aDOT

В качестве первого примера рассмотрим описание графовой модели, которое было получено в предыдущем примере (листинг 3.1). После загрузки файла с описанием графовой модели получим граф представленный на рисунке (6). Полученный граф выглядит более строго нежели этот же граф, но созданный вручную. Все вершины выровнены по оси Y относительно связанных с ними вершин.

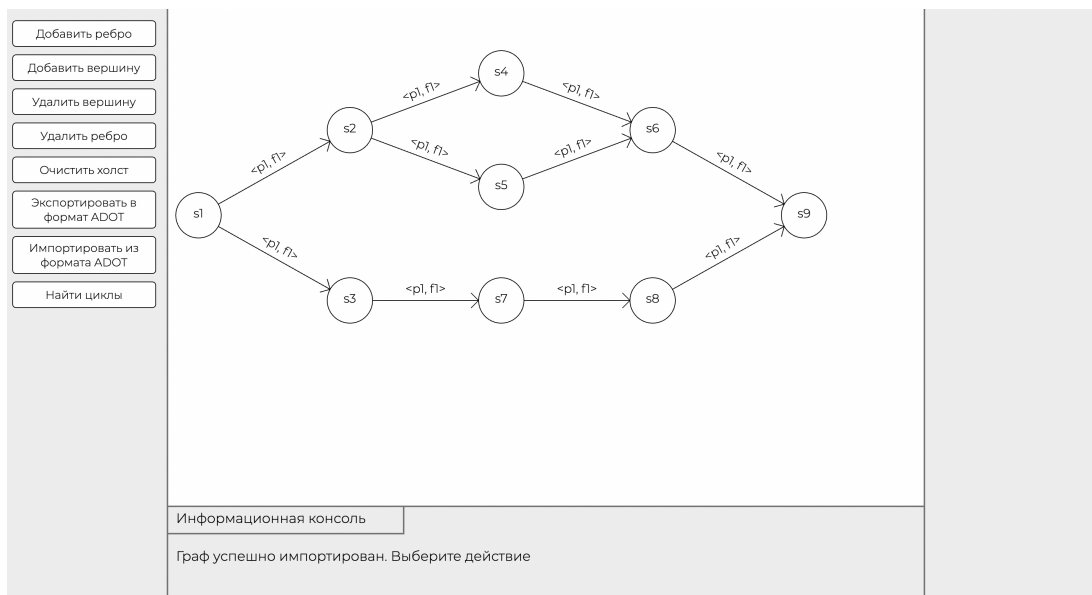


Рисунок 6. Загруженный в формате aDOT граф (листинг 3.1)

Рассмотрим несколько других примеров, сначала будет представлено описание графовой модели в формате aDOT, а затем изображение полученного в результате графа.

```

1  digraph TEST
2  {
3  // Parallelism
4  s2 [parallelism=threading]
5  s1 [parallelism=threading]

```

```

6  s10 [parallelism=threading]
7  // Function
8  f1 [module=f1_module, entry_func=f1_function]
9  // Predicates
10 p1 [module=p1_module, entry_func=p1_function]
11 // Transition
12 edge_1 [predicate=p1, function=f1]
13 // Graph model
14 __BEGIN__ -> s1
15 s4 -> s6 [morphism=edge_1]
16 s5 -> s6 [morphism=edge_1]
17 s2 => s4 [morphism=edge_1]
18 s2 => s5 [morphism=edge_1]
19 s2 => s7 [morphism=edge_1]
20 s1 => s2 [morphism=edge_1]
21 s1 => s10 [morphism=edge_1]
22 s6 -> s9 [morphism=edge_1]
23 s8 -> s9 [morphism=edge_1]
24 s7 -> s6 [morphism=edge_1]
25 s10 => s11 [morphism=edge_1]
26 s10 => s12 [morphism=edge_1]
27 s11 -> s8 [morphism=edge_1]
28 s12 -> s8 [morphism=edge_1]
29 s9 -> __END__
30 }

```

Листинг 3.2. Пример описание графовой модели в формате aDOT

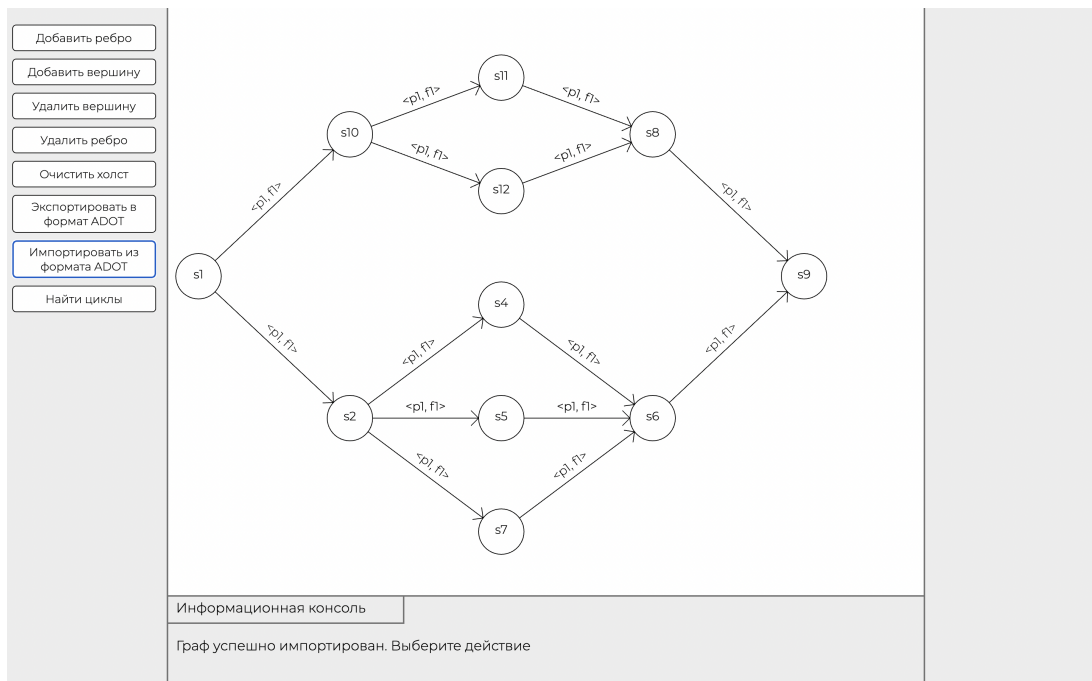


Рисунок 7. Загруженный в формате aDOT граф (листинг 3.2)

В описание графовой модели (листинг 3.2) добавим несколько циклов и обратных ребер. Обратными ребрами будет называть ребра, которые выходят из вершины и приходят в вершину на уровне расположенном левее, при этом не формируя цикл. Получим следующее описание графовой модели

---

```

1 digraph TEST
2 {
3 // Parallelism
4 s7 [parallelism=threading]
5 s11 [parallelism=threading]
6 s2 [parallelism=threading]
7 s10 [parallelism=threading]
8 s1 [parallelism=threading]
9 // Function
10 f1 [module=f1_module, entry_func=f1_function]
11 // Predicates
12 p1 [module=p1_module, entry_func=p1_function]
13 // Transition
14 edge_1 [predicate=p1, function=f1]
15 // Graph model
16 __BEGIN__ -> s1
17 s4 -> s6 [morphism=edge_1]
18 s5 -> s6 [morphism=edge_1]
19 s7 => s6 [morphism=edge_1]
20 s7 => s1 [morphism=edge_1]
21 s11 => s8 [morphism=edge_1]
22 s11 => s2 [morphism=edge_1]
23 s12 -> s8 [morphism=edge_1]
24 s2 => s4 [morphism=edge_1]
25 s2 => s5 [morphism=edge_1]
26 s2 => s7 [morphism=edge_1]
27 s10 => s11 [morphism=edge_1]
28 s10 => s12 [morphism=edge_1]
29 s1 => s2 [morphism=edge_1]
30 s1 => s10 [morphism=edge_1]
31 s6 -> s9 [morphism=edge_1]
32 s8 -> s9 [morphism=edge_1]
33 s9 -> s6 [morphism=edge_1]
34 s9 -> __END__
35 }
```

---

Листинг 3.3. Пример описание графовой модели в формате aDOT включающей циклы

В таком случае будет построен граф представленный на рисунке (8)

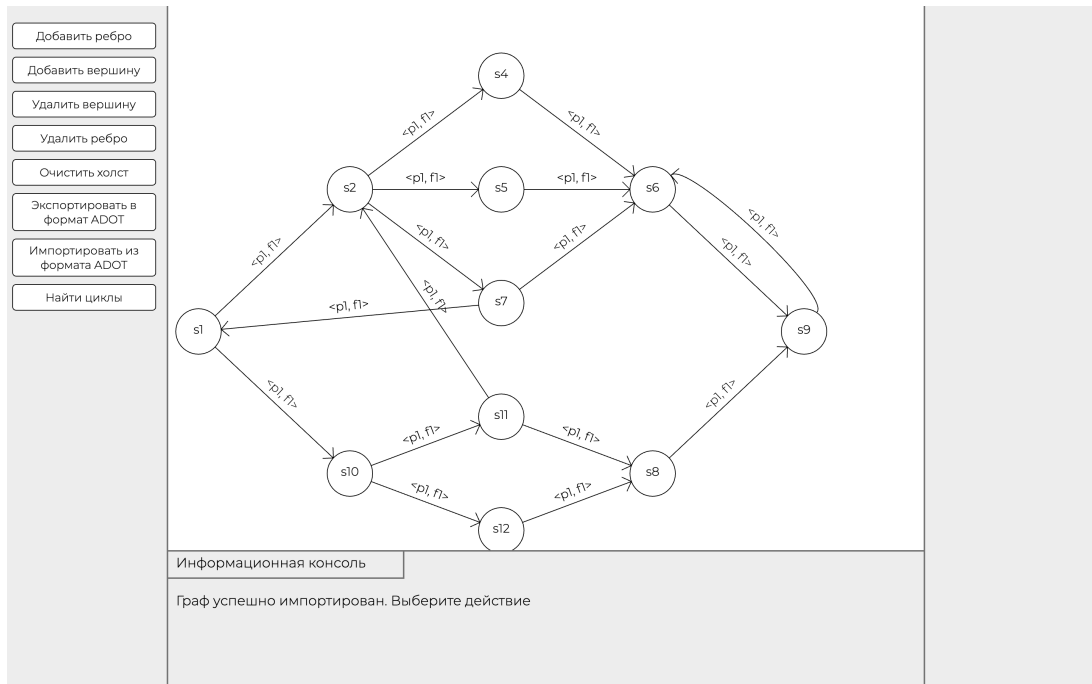


Рисунок 8. Загруженный в формате aDOT граф (листинг 3.3)

Из примеров видно, что граф, который загружается из формата aDOT получается человекочитаемым. Расстояния между вершинами по оси X и по оси Y задаются через константы, что позволяет гибко настраивать размеры загружаемого графа - сделать его меньше или наоборот больше.

## 4 Анализ результатов

В примерах было продемонстрировано, что разработанный редактор удовлетворяет всем поставленным требованиям, однако некоторый функционал требует дополнительной отладки с целью выявления необработанных пользовательских сценариев и других неточностей в реализации. Также в будущем планируется расширить функционал приложения, чтобы сделать его более удобным для конечного пользователя. Далее будет описан реализованный на 2021.12.31 функционал.

Для каждой из реализованных функций будет составлен следующий список:

1. Реализованный функционал
2. Сложности, возникшие при реализации
3. Возможное улучшение функционала

### 1 Добавление вершины

Для добавления вершины необходимо выбрать соответствующий пункт в меню, а затем кликнуть на холст. После этого потребуется уточнить название вершины, после ввода названия вершины построение будет полностью завершено.

Сложности, возникшие при реализации:

1. Валидация всех необходимых параметров в процессе создания вершины: положение вершины, название вершины;
2. Корректное сохранение всех данных о вершине для дальнейшего использования

Возможное улучшение функционала:

1. Возможность редактировать вершину после ее создания. На текущий момент в случае ошибки при создании вершины ее надо удалить, а затем построить еще раз.
2. Использование набора горячих клавиш для ускорения процесса создания вершины.
3. При вводе метки вершины предлагать пользователю автозаполненное название, в том случае если прошлые названия имеют инкрементирующийся для каждой вершины постфикс, например, уже созданы вершины  $s_1$ ,  $s_2$ , при создании новой вершины в поле ввода названия система должна предложить пользователю название  $s_3$ .

### 2 Добавление ребра

Для добавления ребра необходимо поочередно выбрать две вершины, которые будут соединены ребром. Затем система потребует ввода предиката и функции, в том случае если введенный предикат или функция являются уникальными в рамках графа, то система потребует ввода информации о module и entry func. После этого построение ребра будет завершен.



Сложности, возникшие при реализации:

1. Использование множества вспомогательных функций: подсчет количества ребер между вершинами, для определения типа построения ребра (прямое или кривые Безье), подсчет количества ребер выходящих из вершины для уточнения типа параллелизма, проверка полей ввода предиката и функции (предикат может быть неопределен, а функция должна быть определена), проверка введенных предиката и функции на уникальность в рамках графа.
2. Выбор типа построения ребра: прямое, кривые Безье. Обработка ситуации, когда пользователь создает цикл между этими вершинами.

Возможное улучшение функционала:

1. Возможность перевыбора вершин для построения ребра между ними. На данный момент нельзя допустить ошибку при выборе вершин, необходимо закончить процесс построения ребра, удалить его и создать новое, выбрав другие вершины.
2. Алгоритм для построения ребра с использованием кривых Безье. На данный момент ребро строится с помощью одной кривой Безье, что не позволяет строить ребра более сложного вида, тем самым разрешая коллизии, когда ребро проходит через другие вершины.

### 3 Удаление вершины

Для удаления вершины необходимо выбрать соответствующий пункт в меню, а затем кликнуть на вершину для удаления - вершина удаляется с холста вместе со всеми связанными с ней ребрами.

Сложности, возникшие при реализации:

1. Корректно удалить все связанные с вершиной ребра, а затем удалить информацию об этих ребрах и связанных вершин из объектов.

Возможное улучшение функционала:

1. Возможность одновременного выбора нескольких вершин для удаления.
2. Возможность откатить действия в том случае если была удалена лишняя вершина. В целом это касается всего приложения, на данный момент любое действие неотвратимо.

## 4 Удаление ребра

Для удаления ребра необходимо выбрать соответствующий пункт в меню, а затем кликнуть на ребро для удаления - ребро удалится с холста.

Сложности, возникшие при реализации:

1. Удалить информацию о ребре из связанных этим ребром вершин.

Возможное улучшение функционала:

1. Более удобный выбор ребра, на данный момент надо кликать ровно на ребро поскольку для перехвата события клика используется `event.target.closest`.
2. Возможность одновременного выбора нескольких ребер для удаления.

## 5 Экспорт графа в формате aDOT

Для экспорта в формат aDOT необходимо выбрать соответствующий пункт в меню, а затем поочередно выбрать две вершины, которые будут являться стартовой и конечной вершиной соответственно. Затем сформируется текстовый файл с описанием графа в формате aDOT и автоматически начнется его загрузка.

Сложности, возникшие при реализации:

1. Корректное формирование файла в формате aDOT.

## 6 Импорт графа из формата aDOT

Для импорта из формата aDOT необходимо выбрать соответствующий пункт в меню, а затем выбрать файл для импорта. Далее система сама построит граф, сохранив всю необходимую информацию.

Сложности, возникшие при реализации:

1. Разработка алгоритма визуализации графа.

Возможное улучшение функционала:

1. Оптимизация алгоритма, уменьшение асимптотической сложности.

Перейдем к функционалу, который планируется реализовать. В первую очередь стоит обратить внимание, что при построении графа, невозможно уменьшить масштаб или наоборот приблизить, также невозможно и перемещаться по холсту, что не позволяет строить графы, которые из-за своих размеров выходят за пределы холста. Еще одной полезной функциональной возможностью является сохранение текущего графа в базе данных, это позволит одновременно

работать с несколькими графами путем переключения между ними, а также работать над одним графом с разных устройств

## ЗАКЛЮЧЕНИЕ

В результате проведенной работы был реализован редактор графов, который является достаточно полезным инструментом, дополняющим предложенный А.П.Соколовым и А.Ю.Першиными графоориентированный программный каркас для реализации сложных вычислительных методов. Редактор позволяет создавать ориентированный граф с нуля, а затем экспортировать созданный граф в формате aDOT, также есть возможность загрузить граф из формата aDOT. С помощью редактора можно получить графовую модель вычислительно метода до его реализации, что позволяет заранее распределить задачи между разработчиками, а также, возможно, увидеть тонкие места в реализуемом методе и уделить им особое внимание.

## Список использованных источников

- 1 Соколов А.П., Першин А.Ю. Графоориентированный программный каркас для реализации сложных вычислительных методов. 2019.
- 2 Timothy Williamson. History of computers: A brief timeline. 2021.
- 3 J. E. Hannay C. MacLeod J. S. e. a. How do scientists develop and use scientific software? 2009.
- 4 Robert van Liere. CSE. A Modular Architecture for Computational Steering. 2015.
- 5 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.
- 6 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.
- 7 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.
- 8 Stephen C. North Eleftherios Koutsofios. Drawing Graphs With Dot. 1999.
- 9 Соколов А.П. Описание формата данных aDOT (advanced DOT) [Электронный ресурс]. Облачный сервис SA2 Systems. [Офиц. сайт]. 2020.