

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)

УТВЕРЖДАЮ

Заведующий кафедрой ИУ6

А.В. Пролетарский
«27» февраля 2020 г.

З А Д А Н И Е
на выполнение выпускной квалификационной работы магистра

Студент группы ИУ6-41М

Кирияненко Александр Владиславович
(Фамилия, имя, отчество)

Тема квалификационной работы Библиотека элементов программного интерфейса для
обработки графов

Источник тематики (НИР кафедры, заказ организаций и т.п.)

инициативная НИР кафедры

Тема квалификационной работы утверждена распоряжением по факультету ИУ № 03.02.01-04.03/27 от « 13 » ноября 2019 г.

Часть 1. Исследовательская

Провести анализ и исследование программной модели процессора с набором команд дискретной математики. Провести анализ программного интерфейса процессора обработки структур. Провести анализ технологий обработки графов. Исследовать принципы обработки графовых моделей для системы с дискретным набором команд, а также разработать модели графового представления для данных систем.

Часть 2. Конструкторская

Разработать симулятор для программного интерфейса процессора обработки структур. Выполнить проектирование и разработать библиотеку элементов программного интерфейса для обработки графов. Выполнить отладку и тестирование симулятора для программного интерфейса процессора обработки структур. Выполнить оценочное тестирование симулятора для программного интерфейса процессора обработки структур. Выполнить отладку и тестирование библиотеки элементов программного интерфейса для обработки графов. Выполнить комплексное тестирование библиотеки элементов программного интерфейса для обработки графов. Выполнить оценочное тестирование библиотеки элементов программного интерфейса для обработки графов.

Часть 3. Технологическая

Разработать технологию тестирования библиотеки элементов программного интерфейса для обработки графов и привести примеры тестов.

Оформление квалификационной работы:

Расчетно-пояснительная записка на 95–105 листах формата А4.


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

1. Цели и задачи разработки – 1 лист формата А1.
2. Анализ технологий обработки графов – 1 лист формата А1.
3. Схема структурная системы с аппаратной поддержкой операций дискретной математики – 1 лист формата А1.
4. Программная модель процессора с набором команд дискретной математики – 1 лист формата А1.
5. Модели хранения графов для систем с дискретным набором команд – 1 лист формата А1.
6. Диаграмма классов программного интерфейса процессора обработки структур – 1 лист формата А1.
7. Концепции представления графов – 1 лист формата А1.
8. Диаграмма классов библиотеки элементов программного интерфейса для обработки графов – 1 лист формата А1.
9. Схемы алгоритмов – 1 лист формата А1
10. Тестирование библиотеки элементов программного интерфейса для обработки графов – 1 лист формата А1

Дата выдачи задания « 4 » сентября 2019 г.


В соответствии с учебным планом выпускную квалификационную работу выполнить в полном объеме в срок до « 1 » июня 2020 г.

Руководитель квалификационной работы

 27.02.2020
(Подпись, дата)

А.Ю. Попов
(И.О. Фамилия)

Студент

 27.02.2020
(Подпись, дата)

А.В. Кирьяненко
(И.О. Фамилия)

Примечание: 1. Задание оформляется в двух экземплярах: один выдается студенту, второй хранится на кафедре.

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 Информатика и системы управления

УТВЕРЖДАЮ

КАФЕДРА Компьютерные системы и сети

Заведующий кафедрой **ИУ6**

ГРУППА ИУ6-41М

_____ А.В. Пролетарский
 «27» февраля 2020 г.

КАЛЕНДАРНЫЙ ПЛАН

выполнения выпускной квалификационной работы магистра
 студента: _____ Кирияненко Александр Владиславович

(фамилия, имя, отчество)

Тема квалификационной работы _____ Библиотека элементов программного интерфейса для
обработки графов

№ п/п	Наименование этапов выпускной квалификационной работы	Сроки выполнения этапов		Отметка о выполнении	
		план	факт	Должность	ФИО, подпись
1.	Задание на выполнение работы. Формулирование проблемы, цели и задач работы	<u>09.2019</u> <small>Планируемая дата</small>	<u>05.09.2019</u>	Руководитель ВКР	А.Ю. Попов
2.	1 часть <u>Исследовательская</u>	<u>12.2019</u> <small>Планируемая дата</small>	<u>20.12.2019</u>	Руководитель ВКР	А.Ю. Попов
3.	Утверждение окончательных формулировок решаемой проблемы, цели работы и перечня задач	<u>02.2020</u> <small>Планируемая дата</small>	<u>27.02.2020</u>	Заведующий кафедрой	А.В. Пролетарский
4.	2 часть <u>Конструкторская</u>	<u>05.2020</u> <small>Планируемая дата</small>	<u>06.05.2020</u>	Руководитель ВКР	А.Ю. Попов
5.	3 часть <u>Технологическая</u>	<u>05.2020</u> <small>Планируемая дата</small>	<u>13.05.2020</u>	Руководитель ВКР	А.Ю. Попов
6.	1-я редакция работы	<u>05.2020</u> <small>Планируемая дата</small>	<u>25.05.2020</u>	Руководитель ВКР	А.Ю. Попов
7.	Подготовка доклада и презентации	<u>06.2020</u> <small>Планируемая дата</small>	<u>29.05.2020</u>	Руководитель ВКР	А.Ю. Попов
8.	Заключение руководителя	<u>06.2020</u> <small>Планируемая дата</small>	<u>30.05.2020</u>	Руководитель ВКР	А.Ю. Попов
9.	Нормоконтроль	<u>06.2020</u> <small>Планируемая дата</small>	<u>29.05.2020</u>	Нормоконтролер	О.Ю. Еремин
10.	Внешняя рецензия	<u>06.2020</u> <small>Планируемая дата</small>	<u>29.05.2020</u>	Руководитель ВКР	А.Ю. Попов
11.	Защита работы на ГЭК	<u>06.2020</u> <small>Планируемая дата</small>	<u>02.06.2020</u>	Руководитель ВКР	А.Ю. Попов

Студент _____ Кирияненко Александр 27.02.2020
 (подпись, дата)

Руководитель работы _____ 27.02.2020
 (подпись, дата)

АННОТАЦИЯ

Дипломный проект посвящен разработке библиотеки элементов программного интерфейса для обработки графов. Основное предназначение библиотеки заключается в построении, изменении и обработки ультраграфов для систем с дискретным набором команд.

При выполнении дипломного проекта, было проведено исследование программной модели процессора с дискретным набором команд. Также проведен анализ технологий хранения и обработки графов. Было проведено исследование принципов обработки графовых моделей для систем с дискретным набором команд, а также разработаны модели графового представления для данных систем.

Также был доработан и отлажен программный интерфейс взаимодействия с процессором обработки структур. Для тестирования на локальной машине был разработан симулятор процессора обработки структур.

ABSTRACT

The graduation project is dedicated to the development of a library of program interface elements for processing graphs. The main purpose of the library is to build, modify and process ultragraphs for systems with a discrete set of instructions.

When completing a graduation project, a study was conducted of the software model of the processor with a discrete set of instructions. An analysis of graph storage and processing technologies was also carried out. A study was carried out of the principles of processing graph models for systems with a discrete set of instructions, and graph representation models for these systems were developed.

The software interface for interacting with the processor for processing structures was also finalized and debugged. For testing on a local machine, a simulator of a processor for processing structures was developed.

РЕФЕРАТ

Записка 165 с., 32 рис., 37 табл., 14 лист., 11 источников, 4 прил.

МИКРОПРОЦЕССОРНЫЕ СИСТЕМЫ, ПРОЦЕССОР ОБРАБОТКИ СТРУКТУР, ПРОЦЕССОР С НАБОРОМ КОМАНД ДИСКРЕТНОЙ МАТЕМАТИКИ, ПРОГРАММНЫЙ ИНТЕРФЕЙС, ГРАФЫ, МОДЕЛИ ПРЕДСТАВЛЕНИЯ ГРАФОВ.

Разработана библиотека элементов программного интерфейса для обработки графов для систем с дискретным набором команд. Основное предназначение библиотеки заключается в построении, изменении и обработке ультраграфов. Данная библиотека реализовывает интерфейс графа, предоставляемый библиотекой Boost Graph Library, которая в себя включает множество алгоритмов для работы с графами.

Процессор с набором команд дискретной математики, реализующее набор команд дискретной математики высокого уровня над множествами и структурами данных. Новая архитектура позволяет более эффективно решать задачи дискретной оптимизации, основанные на моделях множеств, графов и отношений.

Для локального тестирования ПО был разработан симулятор для программного интерфейса процессора обработки структур.

Материалы по дипломному проекту представлены в виде графической части, приложения с отлаженным программным кодом и расчетно-пояснительной записки.

ОПРЕДЕЛЕНИЯ, ОБОЗНАЧЕНИЯ И СОКРАЩЕНИЯ

ПО – программное обеспечение

ПИ – программный интерфейс

СП – процессор обработки структур

BGL – Boost Graph Library

DISC – Discrete Mathematic Instruction Set computer (команды операций дискретной математики)

МКОД – система с многими потоками команд и одним потоком данных

СУБД – система управления базами данных

ОЗУ – оперативно запоминающее устройство

ОП – обобщенное программирование

ООП – объектно-ориентированное программирование

СОДЕРЖАНИЕ

ВВЕДЕНИЕ	8
1 Анализ процессора обработки структур.....	10
1.1 Сравнение микропроцессора Leonhard с универсальными микропроцессорами	10
1.2 Основные характеристики микроархитектуры СП	15
1.3 Набор команд дискретной математики.....	17
2 Анализ технологий хранения и обработки графов.....	20
2.1 Библиотека Boost Graph Library	20
2.1.1 Введение в библиотеку Boost Graph Library	20
2.1.2 Графовые концепции	21
2.1.3 Графовые представления	27
2.2 Библиотека LEDA	29
2.3 Графовая СУБД Neo4j	30
2.4 Заключение по анализу технологий обработки графов	31
3 Исследование принципов обработки графовых моделей для систем с дискретным набором команд	33
3.1 Анализ требований, предъявляемых к графовым моделям для систем с набором дискретной математики	33
3.2 Моделирование с помощью ультраграфов.....	33
3.3 Графовые модели для системы с набором дискретной математики	35
3.3.1 Модель для ориентированного остовного графа.....	35
3.3.2 Модель для взвешенного ориентированного остовного графа.....	35
3.3.3 Модель для ориентированного мультиграфа.....	36
3.3.4 Модель для взвешенного ориентированного мультиграфа.....	37

3.3.5	Модель для гиперграфа	38
3.3.6	Модель для ультраграфа	40
3.3.7	Модель для взвешенного ультраграфа	41
3.3.8	Модель ультраграфа с атрибутами для ребер	43
3.3.9	Модель для хранения множества графов	45
4	Проектирование и разработка библиотеки.....	47
4.1	Анализ требований.....	47
4.2	Модель жизненного цикла разработки	50
4.3	Выбор методологии проектирования.....	51
4.4	Программный интерфейс взаимодействия с СП	52
4.4.1	Принципы построения библиотеки.....	52
4.4.2	Описание структур СП программного интерфейса	53
4.4.3	Разметка данных на поля	57
4.4.4	Хранение крупных структур данных.....	58
4.4.5	Разработка симулятора СП	61
4.5	Модель хранения графов.....	62
4.6	Концепции ультраграфов	64
4.7	Основные классы обработки графов.....	66
4.8	Алгоритм поиска свободного домена	69
4.9	Руководство по установке библиотеки	72
4.10	Примеры работы с библиотекой.....	74
5	Обеспечение качества библиотеки.....	80
5.1	Особенности работы с ультраграфами	80
5.2	Тестирование программного интерфейса взаимодействия СП.....	80

5.3 Тестирование библиотеки элементов программного интерфейса для обработки графов	82
5.3.1 Функциональное тестирование	82
5.3.2 Тестирование производительности.....	83
5.4 Документирование исходного кода библиотеки	90
ЗАКЛЮЧЕНИЕ	91
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ	92
Приложение А. Техническое задание.....	94
Приложение Б. Руководство системного программиста.....	104
Приложение В. Листинг программного кода.....	113
Приложение Г. Графическая документация.....	154

ВВЕДЕНИЕ

Теория графов активно применяется в программировании в силу удобного выражения задач обработки информации на теоретико-графовом языке. Появление суперкомпьютеров и сетей и возникшая при этом проблема эффективной организации параллельных и распределенных вычислений над информационными массивами большого объема подтвердили тенденцию использования графов как наиболее эффективного средства автоматизации программирования. Практические задачи могут быть смоделированы в виде графов для различных областей, например таких, как маршрутизации пакетов в Интернете, проектировании телефонной сети, систем сборки программного обеспечения, поисковых машин, молекулярная биология, систем автоматизированного планирования дорожного маршрута, научных вычислений и т. п. Достоинство графовых абстракций является, что найденное решение проблем теории графов может быть использовано для решения проблем в широком диапазоне областей.

В качестве основного обрабатывающего блока современных ЭВМ выступает арифметико-логическое устройство. Вместе с тем, при решении практических задач используется существенно большее количество математических операций, включая операции над множествами в дискретной математике. В МГТУ им. Н.Э. Баумана проведен полный цикл создания принципиально новой универсальной вычислительной системы, начиная от создания принципов и моделей и заканчивая созданием опытного образца, проведения тестов и испытаний. Разработано принципиально новое вычислительное устройство: Процессор с набором команд дискретной математики, реализующее набор команд дискретной математики высокого уровня над множествами и структурами данных. Новая архитектура позволяет более эффективно решать задачи дискретной оптимизации, основанные на моделях множеств, графов и отношений. [1]

Для таковых систем была спроектирована и разработана библиотека элементов программного интерфейса для обработки графов. Данная библиотека реализовывает интерфейс графа, предоставляемый библиотекой Boost Graph Library, которая в себя включает множество алгоритмов для работы с графами. Библиотека позволяет строить и обрабатывать ультраграфы.

Для тестирования на локальной машине был разработан симулятор для программного интерфейса процессора обработки структур.

1 Анализ процессора обработки структур

1.1 Сравнение микропроцессора Leonhard с универсальными микропроцессорами

В МГТУ им. Баумана разработан микропроцессор Lenhard, реализующий набор команд DISC (Discrete Mathematics Instruction Set computer, операций дискретной математики). При низкой тактовой частоте (100МГц) производительность Lenhard сравнима с производительностью микропроцессоров семейства Intel Pentium (2 ГГц). Это достигается за счет параллелизма при обработки сложных моделей данных. Микропроцессор Leonhard занимает в 400 раз меньше ресурсов кристалла, чем одно ядро семейства Intel Core. Также микропроцессор Leonhard потребляет в 35 раз меньше энергии чем одно ядро семейства Intel Core. Благодаря использованию микропроцессора Leonhard впервые в истории разработана универсальная вычислительная система с многими потоками команд и одним потоком данных (МКОД, MISD). [2]

На рисунке 1 показана структурная схема универсальных микропроцессоров. У данных микропроцессоров есть следующие недостатки:

- обработка списковых структур данных, таких как деревья, приводит к большому количеству кэш-промахов;
- страничная организация памяти и пакетный режим работы современных ОЗУ увеличивает время выполнения команды и ожидания данных из памяти;
- длинные конвейеры современных микропроцессоров и высокая латентность при доступе к памяти снижают эффективность списковых структур данных.

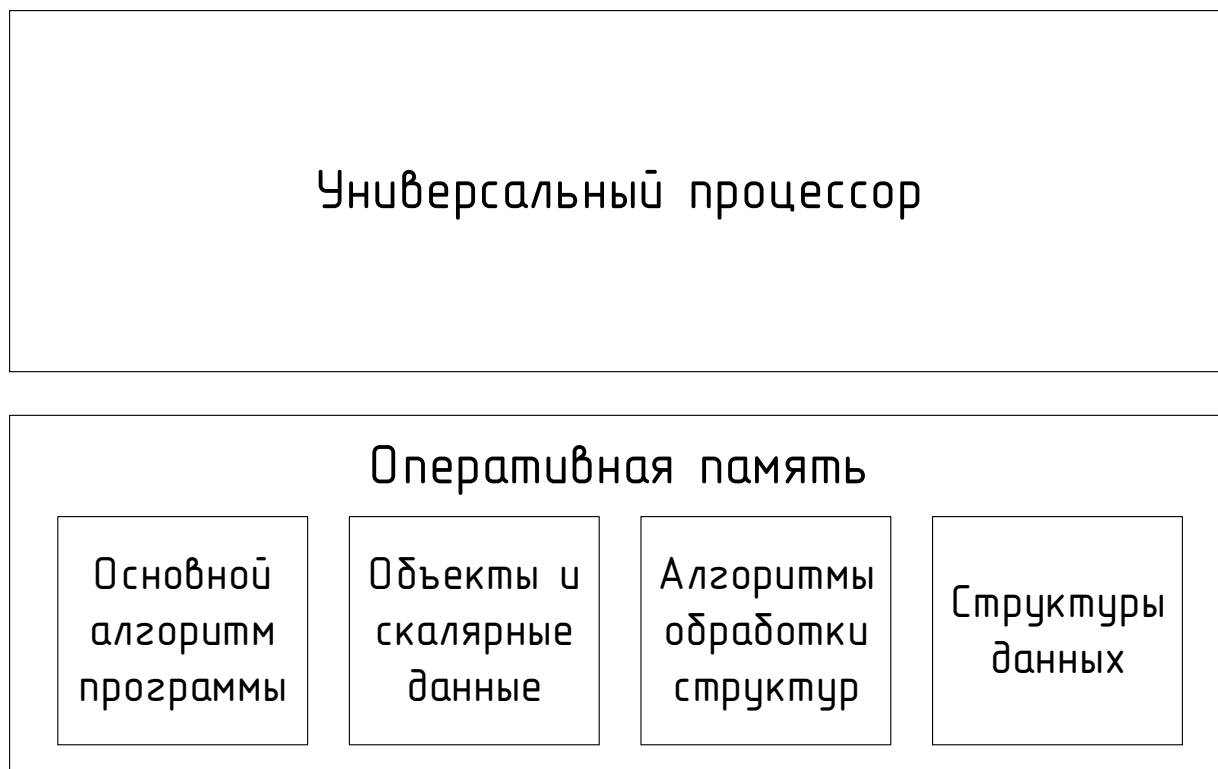


Рисунок 1 – Структурная схема универсальных микропроцессоров

В отличие от универсальных микропроцессоров процессор обработки структур Leonhard (Рисунок 2) структуры данных обрабатывает аппаратно, и обработка происходит параллельно с общей вычислительной нагрузкой. Помимо этого, структуры данных хранятся в независимой локальной памяти структур (SPU LM).

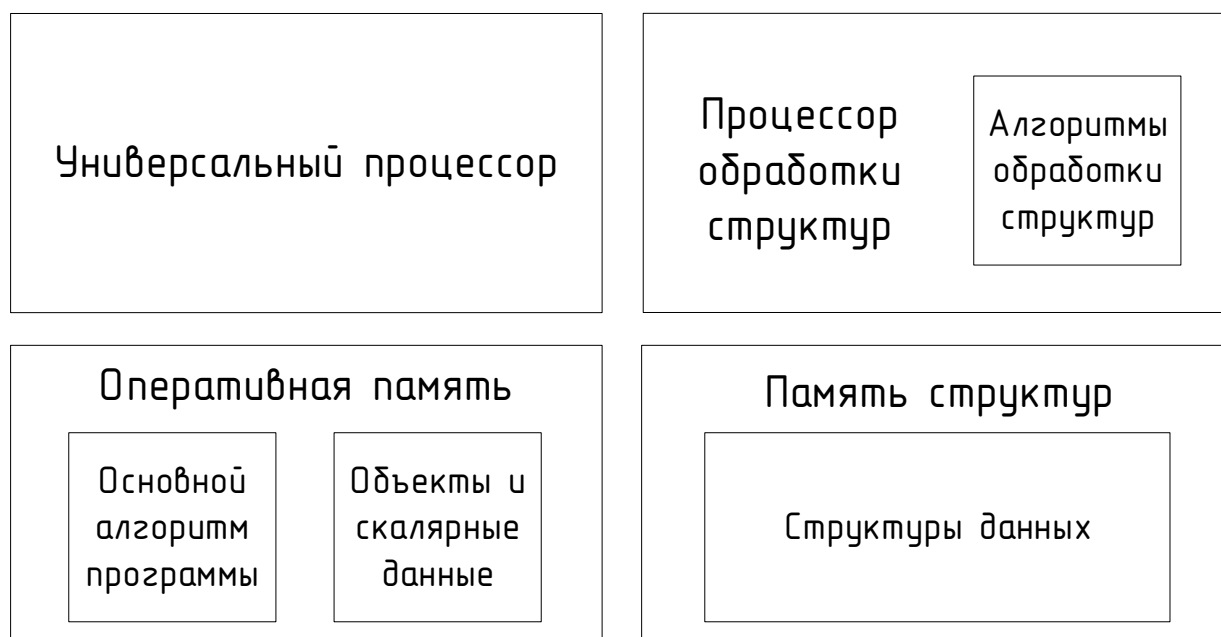


Рисунок 2 – Структурная схема систем с аппаратной поддержкой операций дискретной математики

Ниже в таблице 1 приводится сравнение универсальных микропроцессорных систем с микропроцессорами обработки структур Leonhard.

Таблица 1 – Сравнение универсального микропроцессора с микропроцессором Leonhard

Универсальный микропроцессор	Микропроцессор Leonhard
Обрабатывает числа.	Обрабатывает множества.
Аппаратно реализует арифметическую и логическую обработку.	Аппаратно реализует математический аппарат дискретной математики.
Обработка сложных моделей данных (деревьев, графов) выполняется на конвейере микропроцессора последовательно.	Микроархитектура Leonhard обеспечивает параллельную обработку множеств, структур данных, графов.
Использует ОЗУ для хранения программ, данных структур данных, множеств и т.д.	Использует независимую память для хранения структур данных, множеств, графов.
Распределение памяти для хранения структур данных осуществляется программно.	Распределение памяти осуществляется аппаратными механизмами микропроцессора Leonhard.

На рисунках 3, 4 и 5 показаны графики сравнения производительности при выполнении команд добавления, удаления и поиска соответственно для процессора обработки структур и других универсальных процессоров.

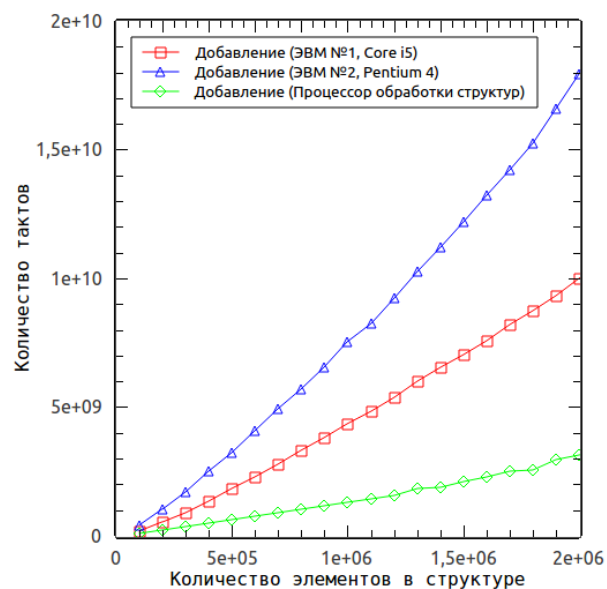
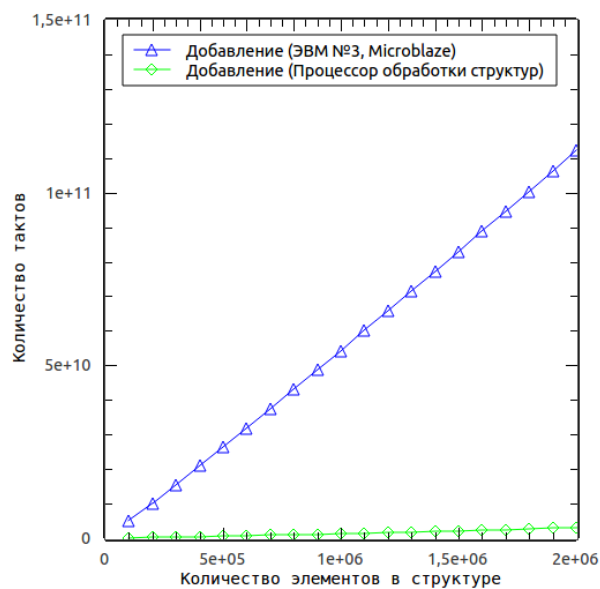


Рисунок 3 – График производительности при выполнении команды добавления [1]

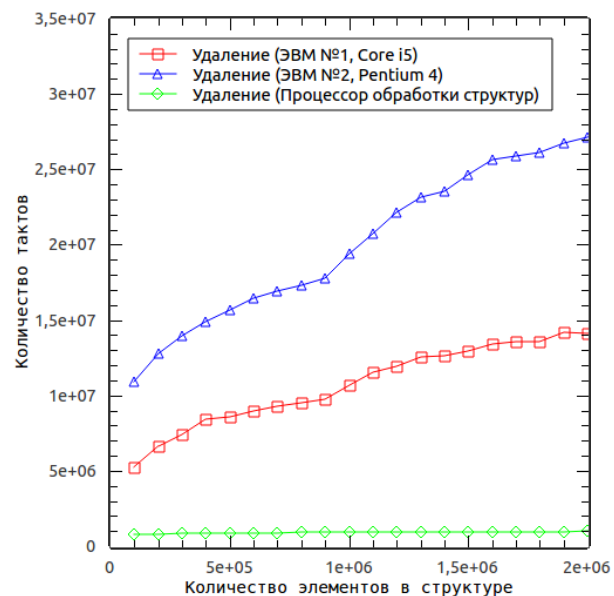
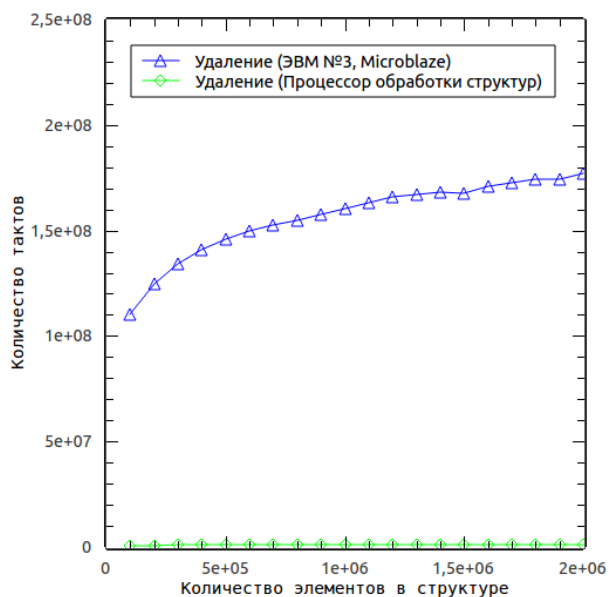


Рисунок 4 – График производительности при выполнении команды удаления [1]

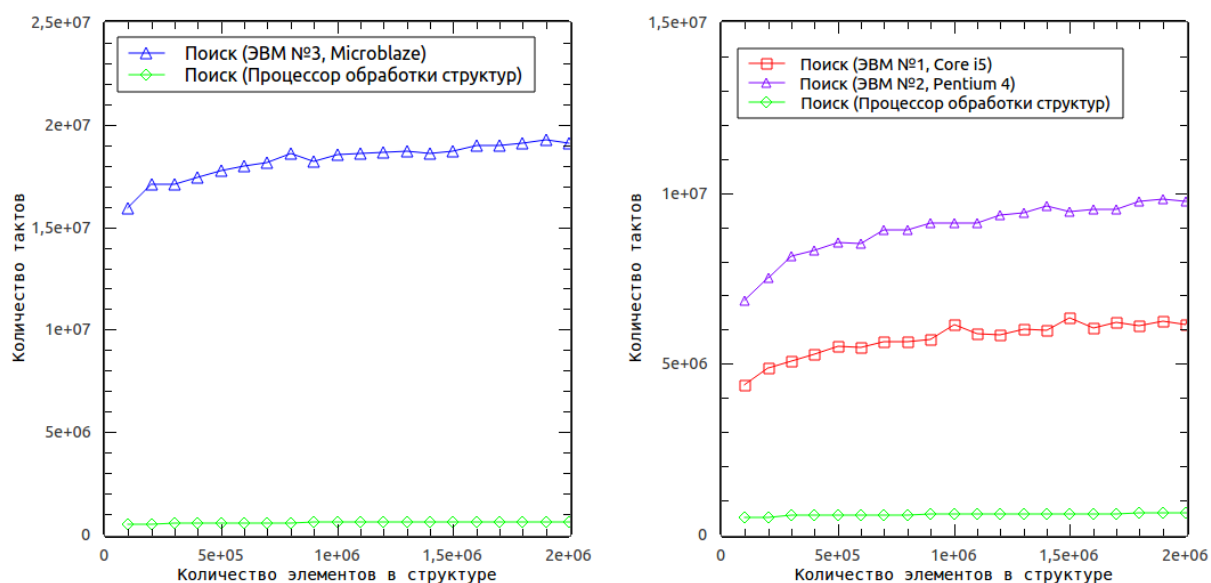


Рисунок 5 - График производительности при выполнении команды поиска [1]

В таблице 2 приводятся результаты сравнения производительности МКОД системы с универсальными ЭВМ.

Таблица 2 – Результаты ускорения в МКОД системе [1]

Эксперимент	Ускорение в МКОД
Удаление (МКОД и Microblaze)	164,4
Добавление (МКОД и Microblaze)	42,7
Поиск (МКОД и Microblaze)	31,4
Удаление (МКОД и Intel Pentium 4)	22,8
Алгоритм Дейкстры (МКОД и Intel Pentium 4)	19,4
Поиск (МКОД и Intel Pentium 4)	15,3
Алгоритм поиска в глубину (МКОД и ARM11)	12,9
Алгоритм поиска в ширину (МКОД и ARM11)	12,3
Удаление (МКОД и Intel Core i5)	11,8
Алгоритм Прима (МКОД и ARM11)	10,3
Поиск (МКОД и Intel Core i5)	9,8
Алгоритм Дейкстры (МКОД и Intel Core i5)	7,6
Алгоритм Крускала (МКОД и ARM11)	7,8
Добавление (МКОД и Pentium 4)	5,7
Добавление (МКОД и Intel Core i5)	3,2
Алгоритм поиска в глубину (МКОД и Intel Core i5)	3,2
Алгоритм поиска в ширину (МКОД и Intel Core i5)	3,0
Алгоритм Прима (МКОД и Intel Core i5)	2,4
Алгоритм Крускала (МКОД и Intel Core i5)	1,5

Результаты экспериментов показывают, что процессор обработки структур позволяет повысить эффективность вычислительной системы. При этом аппаратная сложность СП примерно в 900 раз меньше, чем у микропроцессора Intel Core i5, а затраченный объем электроэнергии для решения задач в МКОД системе примерно в 5 раз ниже.

1.2 Основные характеристики микроархитектуры СП

Процессор с набором команд дискретной математики (процессор обработки структур, СП) представляет собой вычислительный модуль, работа которого основана на хранении данных вида ключ-значение и их объединении в структуры. Структуры СП представляют собой наборы данных ключ-значение. Процессор способен выполнять действия над структурами, соответствующие основным операциям дискретной математики над множествами. Среди этих операций: добавление, удаление, поиск (в том числе поиск наиболее похожего), а также операции над несколькими структурами (пересечение, сложение, вычитание).

Структуры данных с формальной точки зрения представляет собой совокупность двух сущностей: информационную составляющую о значениях полей данных и структурную составляющую, учитывающую отношения данных. Такая двойственность позволяет разделить процесс обработки структур данных на два потока вычислений: поток обработки структурной составляющей и поток обработки информационной части. В связи с этим в вычислительной системе может быть два параллельно работающих микропроцессора: это специальный микропроцессор, который обрабатывает реляционную (структурную) часть структур данных, в то время как универсальный CPU выполняет вычисления над информационной составляющей структур (Рисунок 6).

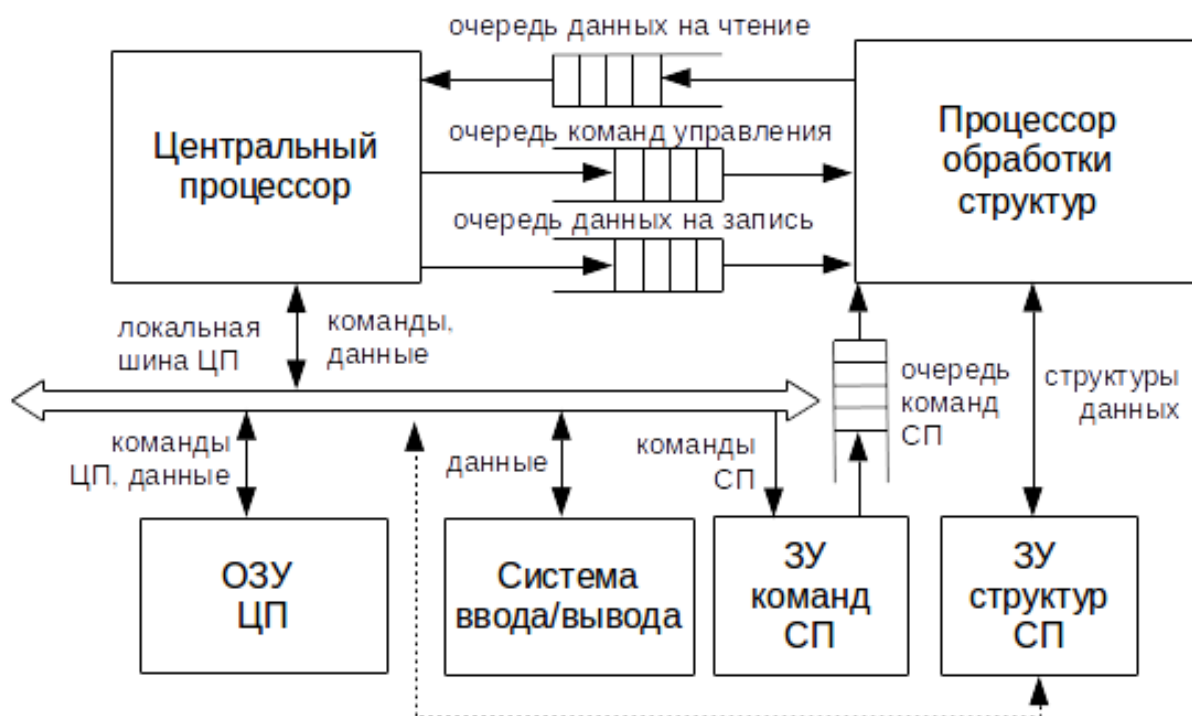


Рисунок 6 – Архитектура вычислительной системы с несколькими потоками команд и одним потоком данных [3]

На рисунке 7 представлена микроархитектура реализации СП Leonhard. Микропроцессор Leonhard x64 представляет собой устройство для обработки структур данных, управляемое специальным набором команд, и предназначено для хранения и обработки больших множеств дискретной информации. Под управлением поступающих команд Leonhard x64 выполняет хранение ключей и значений в многоуровневой подсистеме памяти, выполняет поиск, изменение и выдачу информации другим устройствам системы [2]. Настоящая работа рассматривает эту архитектуру как «чёрный ящик».

Для ускорения поиска и обработки всего набора команд микропроцессор Leonhard x64 использует внутреннее представление множеств в виде В+ дерева, для которого возможна параллельная обработка нескольких вершин дерева как на промежуточных уровнях, используемых для поиска, так и на нижнем уровне, хранящем непосредственно ключи и значения. В связи с этим любая операция над структурой начинается с поиска информации в В+ дереве, а заканчивается обработкой так называемых листьев дерева (вершин нижнего уровня).

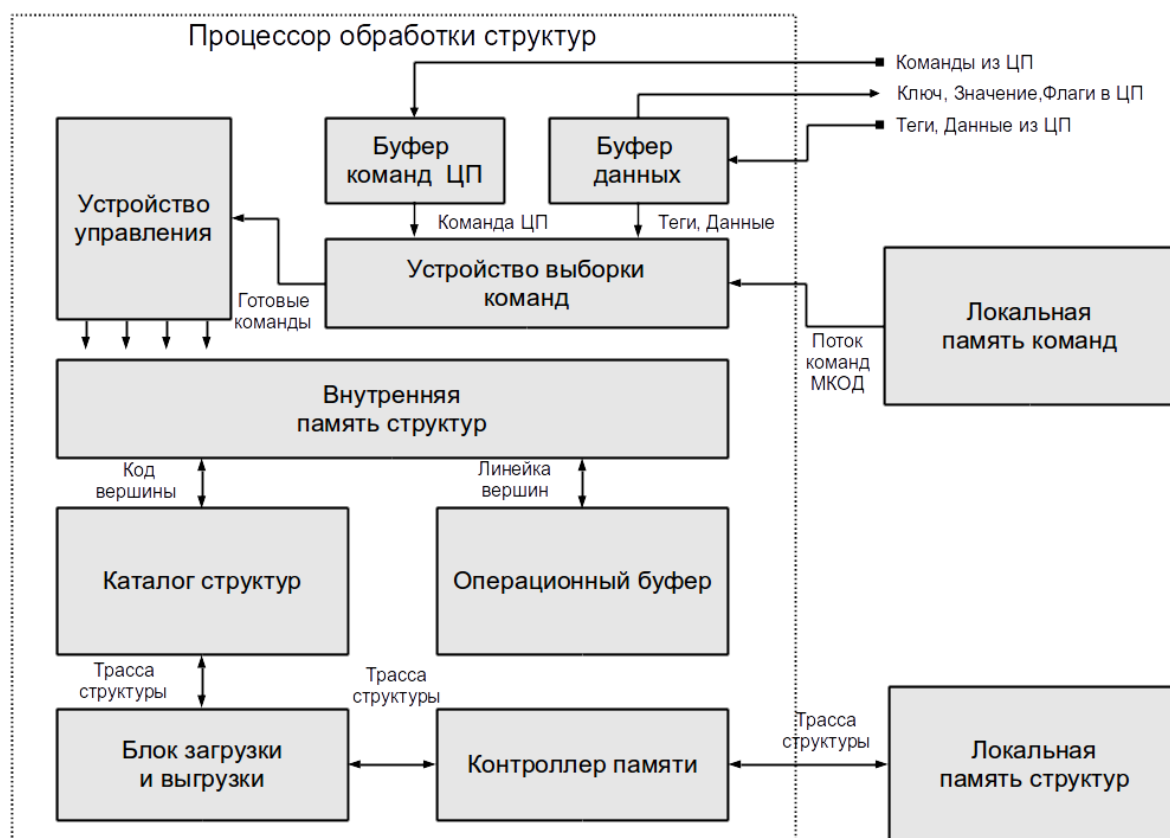


Рисунок 7 – Микроархитектура микропроцессора Leonhard [3]

1.3 Набор команд дискретной математики

Микропроцессор Leonhard x64 хранит информацию о множествах в виде неперекрывающихся В+ деревьев. Последняя версия набора команд Leonhard x64 была расширена двумя новыми инструкциями (NSM и NGR) для обеспечения требований некоторых алгоритмов. Каждая инструкция набора включает до трех операндов (Таблица 3):

Таблица 3 – Формат данных Leonhard x64

Структура	Ключ	Значение
3 бита	64 бита	64 бита

Набор команд состоит из 20 высокоуровневых кодов операций, перечисленных ниже [3].

- команда *search* (*SRCH*) выполняет поиск значения, связанного с ключом;
- команда *insert* (*INS*) вставляет пару ключ-значение в структуру. SPU обновляет значение, если указанный ключ уже находится в структуре;
- операция *delete* (*DEL*) выполняет поиск указанного ключа и удаляет его из структуры данных;
- команда *neighbors* (*NSM*, *NGR*) выполняют поиск соседнего ключа, который меньше (или больше) заданного и возвращает его значение; операции могут быть использованы для эвристических вычислений, где интерполяция данных используется вместо точных вычислений (например, кластеризация или агрегация);
- команда *maximum/minimum* (*MAX*, *MIN*) ищут первый или последний ключи в структуре данных;
- операция *cardinality* (*CNT*) определяет количество ключей, хранящихся в структуре;
- команды *AND*, *OR*, *NOT* выполняют объединения, пересечения и дополнения в двух структурах данных;
- срезы (*LS*, *GR*, *LSEQ*, *GREQ*) извлекают подмножество одной структуры данных в другую;
- переход к следующему или предыдущему (*NEXT*, *PREV*) находят соседний (следующий или предыдущий) ключ в структуре данных относительно переданного ключа; в связи с тем, что исходный ключ должен обязательно присутствовать в структуре данных, операции *NEXT/PREV* отличаются от *NSM/NGR*;
- удаление структуры (*DELS*) очищает все ресурсы, используемые заданной структурой;
- команда *squeeze* (*SQ*) дефрагментирует блоки памяти DSM, используемые структурой;

- команда *jump (JT)* указывает СП код ветвления, который должен быть синхронизирован с CPU (команда доступна только в режиме MISD).

Микропроцессор реализован на базе микросхемы ПЛИС XC6VLX240T-1FFG1156, входящей в состав отладочной платы ML605. В таблице 4 приведены технические характеристики для процессора Leonhard x64.

Таблица 4 – Параметры Leonhard x64

Параметр	Значение
Максимальный размер команды в локальной памяти команд СП (LCM)	144 бит
Максимальный размер команды из ЦП	160 бит
Максимальный размер результата из СП в ЦП	64 бит
Количество разрядов поля ключа	64 бит
Количество разрядов поля значения	64 бит
Расположение байт в памяти	Младший байт по младшему адресу
Размер внешней памяти структур	4 ГБайта
Максимальное количество ключей в структуре	100 663 296
Кратность вершины В ⁺ дерева	8
Количество ключей на нижнем уровне дерева	6
Максимальное количество хранимых структур	7

2 Анализ технологий хранения и обработки графов

2.1 Библиотека Boost Graph Library

2.1.1 Введение в библиотеку Boost Graph Library

Boost — собрание библиотек классов, использующих функциональность языка C++ и предоставляющих удобный кроссплатформенный высокоуровневый интерфейс для решения различных задач программирования (работа с данными, алгоритмами, файлами, потоками и т.п.). Свободно распространяются по лицензии Boost Software License вместе с исходным кодом.

Boost Graph Library предоставляет гибкую и эффективную реализацию концепции графов. В библиотеке присутствуют следующие представления графов:

- список смежности (adjacency_list);
- матрица смежности (adjacency_matrix).

В библиотеке имеется большая база алгоритмов, среди которых:

- поиск в ширину;
- поиск в глубину;
- алгоритм Беллмана-Форда;
- алгоритм Дейкстры;
- алгоритм Прима;
- алгоритм Краскала;
- нахождение компонент связности графа;
- задача о максимальном потоке;
- обратный алгоритм Катхилла-Макки;
- алгоритм топологической сортировки.

2.1.2 Графовые концепции

Библиотека Boost Graph Library (BGL) — это первая библиотека графов C++, применяющая понятия обобщенного программирования при создании алгоритмов на графах. Одна из основных задач обобщенной библиотеки — определить интерфейсы, которые позволят писать алгоритмы, независимые от конкретной структуры данных. Заметим, что под интерфейсом подразумевается не только набор прототипов функций, но и наборы синтаксических условий, таких как имена функций и аргументов, семантических условий (вызываемые функции должны иметь определенные эффекты), гарантии той или иной сложности по времени памяти. [4]

Библиотека Boost Graph Library определяет набор концепций для работы с графами. На рисунке 8 показаны уточняющие отношения между концепциями графа. Причина разделения интерфейса графа на множество концепций состоит в том, чтобы побуждать интерфейсы алгоритмов требовать и использовать только минимальный интерфейс графа, тем самым увеличивая возможность повторного использования алгоритма для других типов графов. [5]

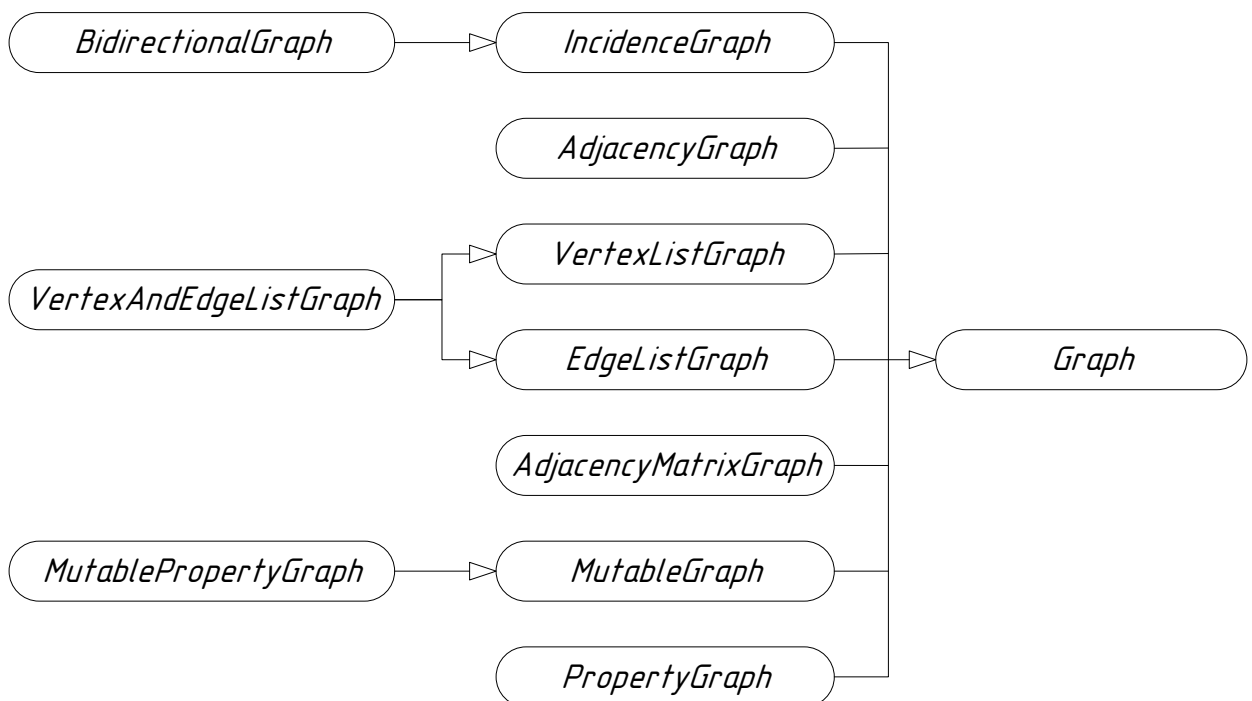


Рисунок 8 – Графовые концепции BGL

Далее в таблицах приводятся сведения о допустимых выражениях и связанных типах для соответствующих концепций графов.

Обозначения:

- G – тип графа;
- g – объект типа G ;
- e – дескриптор ребра, объект типа
boost::graph_traits<G>::edge_descriptor;
- u, v – дескрипторы вершины, объекты типа
boost::graph_traits<G>::vertex_descriptor;
- ep – объект свойств ребра, тип *G::edge_property_type*;
- vp – объект свойств вершины, тип *G::vertex_property_type*;
- *Property* – тег свойства, используемый для указания конкретного свойства вершины или ребра.
- *property* – объект типа *Property*.

Концепция абстрактного графа *Graph* содержит несколько требований (Таблица 5), общих для всех концепций графов.

Таблица 5 – Концепция абстрактного графа (*Graph*)

Выражение	Описание
<i>boost::graph_traits<G>::vertex_descriptor</i>	Тип дескриптора вершины.
<i>boost::graph_traits<G>::edge_descriptor</i>	Тип дескриптора ребра.
<i>boost::graph_traits<G>::directed_category</i>	Указывается является ли граф ориентированным.
<i>boost::graph_traits<G>::edge_parallel_category</i>	Здесь описывается, допускает ли класс графа вставку параллельных ребер (ребер с одинаковым источником и стоком).
<i>boost::graph_traits<G>::traversal_category</i>	Здесь описываются способы посещения вершин и ребер графа.

Концепция графа инцидентности *IncidenceGraph* (Таблица 6) предоставляет интерфейс для эффективного доступа к ребрам каждой вершины графа.

Таблица 6 – Концепция графа инцидентности (*IncidenceGraph*)

Выражение	Описание
<code>boost::graph_traits<G>::out_edge_iterator</code>	Итератор по исходящим ребрам.
<code>boost::graph_traits<G>::degree_size_type</code>	Целочисленный тип степени вершины.
<code>std::pair<out_edge_iterator, out_edge_iterator> out_edges(v, g)</code>	Возвращает диапазон итераторов, обеспечивающий доступ к исходящим ребрам вершины <i>v</i> в графе <i>g</i> .
<code>vertex_descriptor source(e, g)</code>	Возвращает дескриптор источника для ребра <i>e</i> .
<code>vertex_descriptor target(e, g)</code>	Возвращает дескриптор стока для ребра <i>e</i> .

Концепция двунаправленного графа *BidirectionalGraph* (Таблица 7) уточняет концепцию *IncidenceGraph* и добавляет требование для эффективного доступа к входящим ребрам для каждой вершины. Эта концепция отделена от *IncidenceGraph*, потому что для ориентированных графов эффективный доступ к ребрам обычно требует больше места для хранения.

Таблица 7 – Концепция двунаправленного графа (*BidirectionalGraph*)

Выражение	Описание
<code>boost::graph_traits<G>::in_edge_iterator</code>	Итератор по входящим ребрам.
<code>std::pair<in_edge_iterator, in_edge_iterator> in_edges(v, g)</code>	Возвращает диапазон итераторов, обеспечивающий доступ к входящим ребрам для вершины v в графе g .
<code>degree_size_type in_degree(v, g)</code>	Возвращает количество входящих ребер.
<code>degree_size_type degree(e, g)</code>	Возвращает степень вершины.

Концепция графа смежности *AdjacencyGraph* (Таблица 8) обеспечивает интерфейс для эффективного доступа к смежным вершинам для конкретной вершины.

Таблица 8 – Концепция графа смежности (*AdjacencyGraph*)

Выражение	Описание
<code>boost::graph_traits<G>::adjacency_iterator</code>	Итератор по смежным вершинам.
<code>std::pair<adjacency_iterator, adjacency_iterator> adjacent_vertices(v, g)</code>	Возвращает диапазон итераторов, обеспечивающий доступ к смежным вершинам для вершины v в графе g .

Концепция графа списка вершин *VertexListGraph* (Таблица 9) добавляет требование для эффективного обхода всех вершин в графе.

Таблица 9 - Концепция графа списка вершин (*VertexListGraph*)

Выражение	Описание
<code>boost::graph_traits<G>::vertex_iterator</code>	Итератор по всем вершинам.
<code>boost::graph_traits<G>::vertices_size_type</code>	Целочисленный тип для обозначения количества вершин в графе.
<code>std::pair<vertex_iterator, vertex_iterator> vertices(g)</code>	Возвращает диапазон итераторов, обеспечивающий доступ ко всем вершинам в графе <i>g</i> .
<code>vertices_size_type num_vertices(g)</code>	Количество вершин в графе <i>g</i> .

Концепция графа списка вершин *EdgeListGraph* (Таблица 10) добавляет требование для эффективного обхода всех ребер в графе.

Таблица 10 – Концепция графа списка ребер (*EdgeListGraph*)

Выражение	Описание
<code>boost::graph_traits<G>::edge_iterator</code>	Итератор по всем ребрам.
<code>boost::graph_traits<G>::edges_size_type</code>	Целочисленный тип для обозначения количества ребер в графе.
<code>std::pair<edge_iterator, edge_iterator> edges(g)</code>	Возвращает диапазон итераторов, обеспечивающий доступ ко всем ребрам.
<code>edges_size_type num_edges(g)</code>	Количество ребер в графе <i>g</i> .
<code>vertex_descriptor source(e, g)</code>	Возвращает источник для ребра <i>e</i> .
<code>vertex_descriptor target(e, g)</code>	Возвращает сток для ребра <i>e</i> .

Концепция матрицы смежности *AdjacencyMatrix* (Таблица 11) добавляет требование для эффективного доступа к любому ребру графа с учетом исходных и целевых вершин.

Таблица 11 – Концепция матрицы смежности (*AdjacencyMatrix*)

Выражение	Описание
<code>std::pair<edge_descriptor, bool> edge(u, v, g)</code>	Возвращает пару, состоящую из флага, указывающего, существует ли ребро между <i>u</i> и <i>v</i> в графе <i>g</i> , и дескриптора ребра.

Концепция изменяемого графа *MutableGraph* (Таблица 12) добавляет методы для добавления или удаления ребер и вершин.

Таблица 12 - Концепция изменяемого графа (*MutableGraph*)

Выражение	Описание
<code>vertex_descriptor add_vertex(g)</code>	Добавляет вершину в граф <i>g</i> .
<code>void clear_vertex(v, g)</code>	Удаляет все ребра смежные с вершиной.
<code>void remove_vertex(v, g)</code>	Удаляет вершину <i>v</i> из графа <i>g</i> .
<code>std::pair<edge_descriptor, bool> add_edge(u, v, g)</code>	Вставляет ребро в граф <i>g</i> между вершинами <i>u</i> и <i>v</i> . Если граф запрещает параллельные ребра, то флаг устанавливается в значение <i>false</i> .
<code>void remove_edge(u, v, g)</code>	Удаляет все ребра между вершинами.
<code>void remove_edge(e, g)</code>	Удаляет ребро <i>e</i> из графа <i>g</i> .

Концепция изменяемого графа свойств *MutablePropertyGraph* (Таблица 13) уточняет концепцию *MutableGraph* и добавляет возможность при добавлении вершин и ребер устанавливать значения их свойств.

Таблица 13 – Концепция изменяемого графа свойств (*MutablePropertyGraph*)

Выражение	Описание
<code>vertex_descriptor add_vertex(vp, g)</code>	Добавляет вершину со свойствами <i>vp</i> в граф <i>g</i> .
<code>std::pair<edge_descriptor, bool> add_edge(u, v, ep, g)</code>	Добавляет ребро со свойствами <i>ep</i> между вершинами <i>u</i> и <i>v</i> в граф <i>g</i> .

Граф свойств *PropertyGraph* — это граф, у которого есть свойства, связанные с вершинами или ребрами графа. Поскольку у данного графа может быть несколько свойств для определения того, к какому свойству обращаются, используется тег. Граф предоставляет функцию, которая возвращает объект карты свойств.

Таблица 14 – Концепция графа свойств (*PropertyGraph*)

Выражение	Описание
<code>boost::property_map<G, Property>::type</code>	Тип изменяемой карты свойств.
<code>boost::property_map<G, Property>::const_type</code>	Тип неизменяемой карты свойств.
<code>get(property, g)</code>	Возвращает объект карты свойства для графа <i>g</i> .
<code>get(property, g, x)</code>	Получить значение свойства для вершины или ребра <i>x</i> .
<code>put(property, g, x, v)</code>	Установить значение свойства для вершины или ребра <i>x</i> .

2.1.3 Графовые представления

Класс *adjacency_list* реализует интерфейс BGL-графа в виде списка смежности. Представление графа в виде списка смежности содержит последовательность исходящих ребер для каждой вершины. Для разреженных графов это экономит место по сравнению с матрицей смежности, поскольку памяти требуется только порядка $O(|V| + |E|)$, а не $O(|V|^2)$. Кроме того, доступ к исходящим ребрам для каждой вершины более эффективно. Представление ориентированного графа в виде списка смежности показан на рисунке 9.

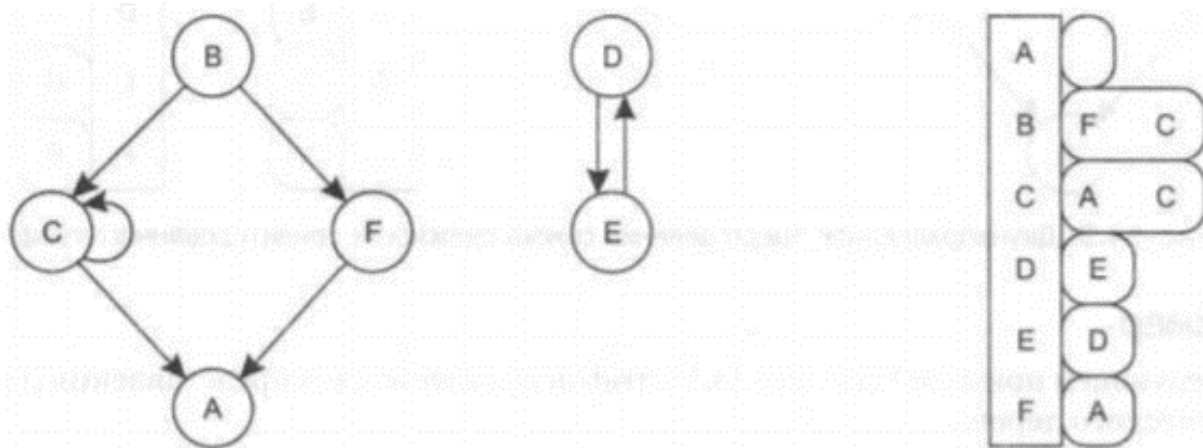


Рисунок 9 – Представление ориентированного графа в виде списка смежности [3]

Класс *adjacency_matrix* реализует интерфейс BGL-графа, используя традиционную графовую структуру матрицу смежности. Для графа с $|V|$ вершинами используется матрица $|V| * |V|$, где каждый элемент является логическим флагом, свидетельствующий о том, имеется ли ребро из вершины i в вершину j . Представление графа в виде матрицы смежности показано на рисунке 10.

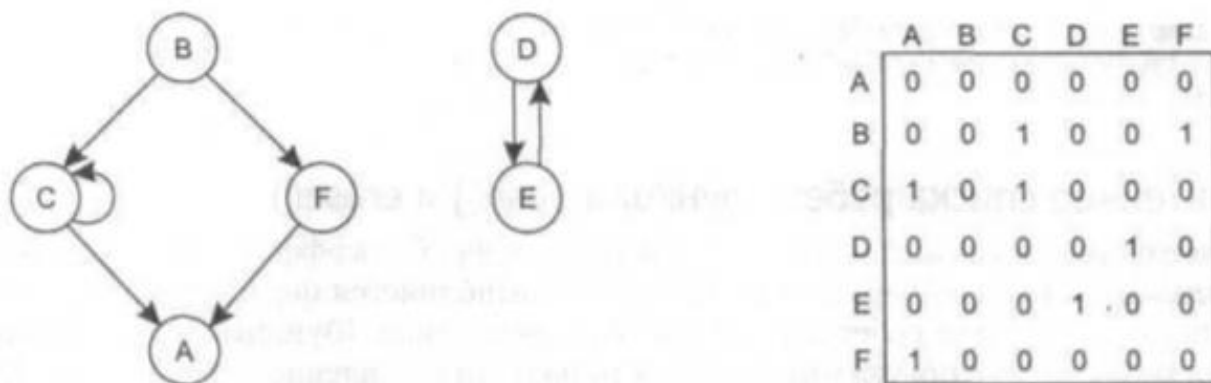


Рисунок 10 – Представление графа в виде матрицы смежности [3]

Преимущество такого матричного формата над списком смежности является то, что ребра добавляются и удаляются за постоянное время. Есть и несколько недостатков. Во-первых, объем используемой памяти имеет порядок $O(|V|^2)$ вместо $O(|V| + |E|)$ (где $|E|$ — число ребер графа). Во-вторых, операции по всем исходящим ребрам каждой вершины (как поиск в ширину) имеют временную сложность $O(|V|^2)$ в отличие от $O(|V| + |E|)$ для списка смежности. Матрицу смежности лучше использовать с плотными

графам (где $|E| \approx |V|^2$), а список смежности — разреженным (где $|E|$ намного меньше $|V|^2$).

2.2 Библиотека LEDA

LEDA — это библиотека классов C++ от AlgoSol для эффективной обработки типов данных и алгоритмов. LEDA предоставляет алгоритмическую базу в области графических и сетевых задач, геометрических вычислений, комбинаторной оптимизации и других. [6]

Также LEDA включает интерфейс для ввода и вывода графов для различных платформ, что позволяет делать визуализацию графов и анимацию работы графовых алгоритмов.

Типа данных `graph` является основным типом данных для представления графов в LEDA. Он может представлять направленные и ненаправленные графы $G = (V, E)$, где V — список узлов, а E — список направленных, соответственно ненаправленных ребер. Узлы и ребра имеют тип `item`. [7]

Основное различие между ориентированными и неориентированными графами заключается в способе хранения ребер, инцидентных с узлом. Ребро инцидентно узлу, если узел является конечной вершиной ребра.

Библиотека LEDA включает огромное количество алгоритмов среди них:

- алгоритмы кратчайшего пути;
- алгоритмы максимального потока;
- минимальные алгоритмы резки;
- алгоритмы потока минимальной стоимости;
- алгоритмы соответствия;
- охватывающие деревья и минимально покрывающие деревья;
- алгоритмы для плоских графов;
- алгоритмы для рисования графов.

2.3 Графовая СУБД Neo4j

Neo4j — графовая система управления базами данных с открытым исходным кодом, реализованная на Java. По состоянию на 2019 год считается самой распространённой графовой СУБД. [8]

Данные хранит в собственном формате, специализированно приспособленном для представления графовой информации, такой подход в сравнении с моделированием графовой базы данных средствами реляционной СУБД позволяет применять дополнительную оптимизацию в случае данных с более сложной структурой. Также утверждается о наличии специальных оптимизаций для SSD-накопителей, при этом для обработки графа не требуется его помещение целиком в оперативную память вычислительного узла, таким образом, возможна обработка достаточно больших графов.

Основные транзакционные возможности — поддержка ACID и соответствие спецификациям JTA, JTS и XA. Интерфейс программирования приложений для СУБД реализован для многих языков программирования, включая Java, Python, Clojure, Ruby, PHP, также реализовано API в стиле REST. Расширить программный интерфейс можно как с помощью серверных плагинов, так и с помощью неуправляемых расширений; плагины могут добавлять новые ресурсы к REST-интерфейсу для конечных пользователей, а расширения позволяют получить полный контроль над программным интерфейсом, и могут содержать произвольный код, поэтому их следует использовать с осторожностью.

В СУБД используется собственный язык запросов — Cypher, но запросы можно делать и другими способами, например, напрямую через Java API. Cypher является не только языком запросов, но и языком манипулирования данными, так как предоставляет функции CRUD для графового хранилища.

Терминология Neo4j и графовых баз данных в целом. [8]

- графовая база данных (graph database) — база данных, построенная на графах — узлах и связях между ними;

- узел (node) — объект в базе данных, узел графа; количество узлов ограничено 2^{35} (примерно 34 биллиона);
- метка узла (node label) — используется как условный «тип узла», например, узлы типа movie могут быть связаны с узлами типа actor (метки узлов — регистрозависимые, причем Cypher не выдает ошибок, если набрать не в том регистре название);
- связь (relation) — связь между двумя узлами, ребро графа. Количество связей ограничено 2^{35} (примерно 34 биллиона);
- тип связи (relation identifier); максимальное количество типов связей 32767;
- свойства узла (properties) — набор данных, которые можно назначить узлу, например, если узел — это товар, то в свойствах узла можно хранить id товара из реляционной СУБД.
- ID узла (node ID) — уникальный идентификатор узла, который по умолчанию отображается при просмотрах результата.

2.4 Заключение по анализу технологий обработки графов

В результате был проведен анализ и исследование таких технологий графовой обработки как: библиотека для работы с графами Boost Graph Library, библиотека для эффективной обработки типов данных и алгоритмов LEDA и графовая система управления базами данных Neo4j.

По результатам проведенного анализа было принято решение, что библиотека элементов программного интерфейса для обработки графов для систем с дискретным набором команд должна реализовывать интерфейс графа, предоставляемого популярной библиотекой Boost Graph Library. Таким образом, реализовав интерфейс Boost графа появится возможность использовать огромный перечень графовых алгоритмов предоставляемой данной библиотекой. А также существующие программные решения, которые

используют графы Boost Graph Library, смогут без труда перейти на граф для систем дискретным набором команд.

3 Исследование принципов обработки графовых моделей для систем с дискретным набором команд

3.1 Анализ требований, предъявляемых к графовым моделям для систем с набором дискретной математики

Главное требование, предъявляемое к моделям – это то, что графы, построенные на базе этой модели, должны соответствовать всем свойствам данного типа графа. Также должен осуществляться эффективный поиск смежных вершин. Если граф взвешенный, то по модели должен осуществляться поиск минимальных и максимальных ребер.

Для осуществления эффективного поиска, важно правильно задавать порядок аргументов в ключе. Наиболее приоритетные аргументы для поиска должны находиться в старших разрядах ключа.

3.2 Моделирование с помощью ультраграфов

Граф называется ультраграфом $HU(X, U, \Gamma_1, \Gamma_2)$, если предикаты $\Gamma_1(X, U)$ и $\Gamma_2(U, X)$ обладают следующим свойством:

$$\exists u_j \in U, (|\Gamma_1 u_j| + |\Gamma_2 u_j|) > 2, \quad (1)$$

т. е. в графе есть хотя бы одно ребро, суммарное количество вершин, которым оно инцидентно и которые инцидентны ему, больше двух. [9]

Ультраграф является универсальной (обобщенной) моделью, так как позволяет в общем случае отобразить всю информацию, необходимую для решения широкого класса задач и другие виды графов (остовные деревья, обыкновенные графы, гиперграфы и др.) являются его частным случаем.

Модель схемы в виде ультраграфа необходима в тех случаях, когда:

- существенной является информация о принадлежности подсистем соединениям с указанием: является ли подсистема источником сигнала для данной цепи или приемником из нее;
- количество подсистем проектируемого объекта, являющихся источниками/приемниками информации, более одного, т.е. в схеме есть цепи, соединяющие более двух подсистем.

К числу таких задач можно отнести, например, задачи идентификации и покрытия, временного анализа топологической реализации схемы и др. Ниже на рисунке 1 показан пример моделирования схмотехнической цепи при помощи ультраграфа.

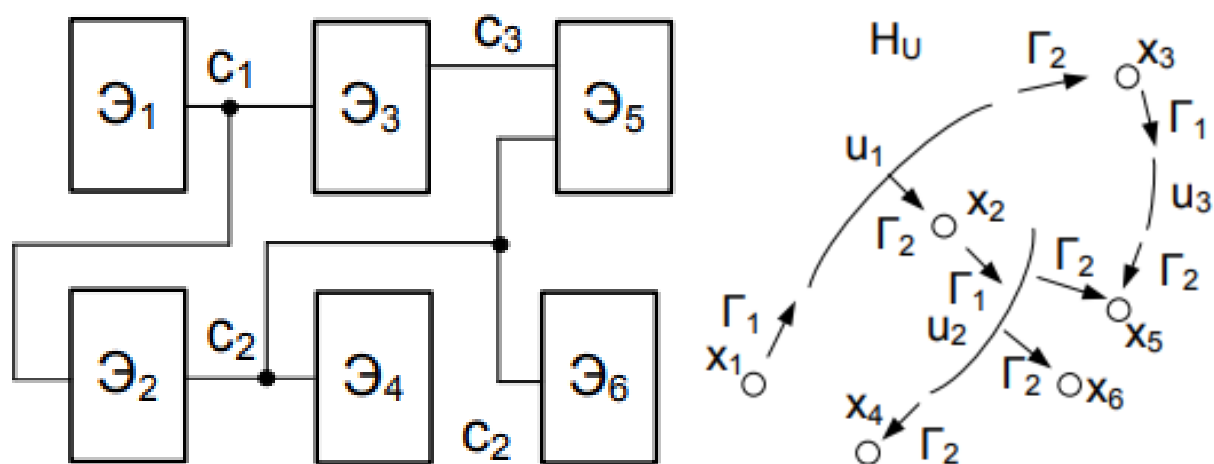


Рисунок 11 – Пример моделирования схмотехнической цепи с помощью ультраграфа

В силу универсальности ультраграфов было принято решение в библиотеке элементов программного интерфейсов для обработки графов реализовать интерфейс для работы с ультраграфами, т.к. на базе данного вида графа, можно строить другие типы графов, являющиеся его «предшественниками».

3.3 Графовые модели для системы с набором дискретной математики

3.3.1 Модель для ориентированного остовного графа

Текущая модель подходит для невзвешенных ориентированных остовых графов и обычных ориентированных графов, у которых нельзя двумя и более ребрами соединить две одинаковых вершины. Модель использует всего одну структуру СП (Таблица 15):

Таблица 15 – Структура хранения ориентированного остовного графа

Ключ		Значение
id вершины	0..0	Данные вершины
id вершины	id смежной вершины	

Структура хранит смежные вершины. Старшим аргументом ключа является идентификатор вершины, а младшим - идентификатор смежной вершины. Данные вершины хранятся в значении для записи с ключом, у которого аргумент *id вершины* соответствует этой вершине и аргумент *id смежной вершины* равен 0.

Для неориентированного остовного графа необходимо каждое ребро представлять дважды относительно входящих в него вершин.

Ограничения:

- идентификаторы вершин не должны быть равны 0;
- нельзя двумя и более ребрами соединить две одинаковых вершины.

3.3.2 Модель для взвешенного ориентированного остовного графа

Эта модель подходит для взвешенных ориентированных остовых графов. Модель использует всего одну структуру СП (Таблица 16).

Таблица 16 – Структура хранения взвешенного ориентированного остовного графа

Ключ			Значение
id вершины	0..0		Данные вершины
id вершины	0..0	id смежной вершины	Вес ребра
id вершины	Вес	id смежной вершины	

Структура хранит смежные вершины. Старшим аргументом ключа является идентификатор вершины, вторым – вес ребра, а младшим – идентификатор смежной вершины. Данные вершины хранятся в значении для записи с ключом, у которого аргумент *id вершины* соответствует этой вершине и аргумент *id смежной вершины* равен 0.

Для нахождения веса ребра между вершинами необходимо добавлять запись, в которой аргументы *id вершины* и *id смежной вершины* соответствуют смежным вершинам, аргумент вес равен 0, а значение равно весу ребра.

Для неориентированного остовного графа необходимо каждое ребро представлять дважды относительно входящих в него вершин.

Отличие данной модели от предыдущей – в SPU структуру был добавлен аргумент *Вес ребра*, благодаря которому будет осуществляться быстрый поиск смежной вершины, соединенной через ребро с минимальным или максимальным весом.

Ограничения:

- идентификаторы вершин не должны быть равны 0;
- вес ребра представлен натуральным числом.

3.3.3 Модель для ориентированного мультиграфа

Модель подходит для ориентированных мультиграфов и использует одну структуру СП (Таблица 17):

Таблица 17 - Структура хранения ориентированного мультиграфа

Ключ			Значение
0..0	id ребра	id смежной вершины 1	id смежной вершины 2
id вершины	0..0		Данные вершины
id вершины	id ребра	id смежной вершины	

Структура хранит смежные вершины. Старшим аргументом ключа является идентификатор вершины, вторым – идентификатор ребра, а младшим – идентификатор смежной вершины. Данные вершины хранятся в значении для записи с ключом, у которого аргумент *id вершины* соответствует этой вершине, а остальные аргументы равны 0.

Для поиска по идентификатору вершины необходимо добавлять запись, в которой аргумент *id вершины* равен 0, *id ребра* соответствует ребру, а младший аргумент и значение соответствует идентификаторам смежных вершин. Поиск такой записи будет происходить с помощью команды *ngr(0,id,0)*.

Для неориентированного графа необходимо каждое ребро представлять дважды относительно входящих в него вершин.

Ограничения: идентификаторы вершин не должны быть равны 0.

3.3.4 Модель для взвешенного ориентированного мультиграфа

Модель подходит для взвешенных ориентированных мультиграфов и использует одну структуру СП (Таблица 18). Структура хранит смежные вершины. Старший аргумент ключа – это идентификатор вершины, второй – вес ребра, третий - идентификатор ребра, а младший – идентификатор смежной вершины. Данные вершины хранятся в значении для записи с ключом, у которого аргумент *id вершины* соответствует этой вершине, а остальные аргументы равны 0.

Таблица 18 – Структура взвешенного ориентированного мультиграфа

Ключ				Значение
0..0	id ребра	id смежной вершины 1	id смежной вершины 2	
id вершин	0..0			Данные вершины
id вершины	Вес	id ребра	id смежной вершины	

Отличие данной модели от предыдущей – в структуру СП был добавлен аргумент «*Вес ребра*», благодаря которому, будет осуществляться быстрый поиск смежной вершины, соединенной ребром с минимальным / максимальным весом.

Для поиска по идентификатору вершины необходимо добавлять запись, в которой аргумент *id вершины* и *вес* равны 0, *id ребра* соответствует ребру, а младший аргумент и значение соответствует идентификаторам смежных вершин. Поиск такой записи будет происходить с помощью команды *ngr(0,0,id, 0)*.

Ограничения:

- идентификаторы вершин не должны быть равны 0;
- вес ребра представлен целым положительным числом.

3.3.5 Модель для гиперграфа

Эта модель подходит для гиперграфов и использует 2 структуры СП: для связи вершина – ребро (Таблица 19) и обратной связи ребро – вершина (Таблица 20).

Таблица 19 – Структура связи вершина – ребро для гиперграфа

Ключ		Значение
id вершины	0..0	Данные вершины
id вершины	id ребра	

Структура хранит вершины и исходящие из них ребра. По этой структуре происходит поиск смежных ребер для указанной вершины. Старший аргумент ключа – это идентификатор вершины, а младший – идентификатор смежного ребра. Данные вершины хранятся в значении для записи с ключом, у которого аргумент *id вершины* соответствует этой вершине и аргумент *id ребра* равен 0.

Таблица 20 – Структура связи ребро – вершина для гиперграфа

Ключ		Значение
id ребра	0..0	Вес ребра
id ребра	id вершины	

Структура хранит ребра и вершины, в которые входят эти ребра. По этой структуре происходит поиск смежных вершин для указанного ребра. Старший аргумент ключа является идентификатор ребра, а младший - идентификатор смежной вершины.

Для взвешенных графов в значении для записи с ключом, у которого аргумент *id ребра* соответствует указанному ребру, а аргумент *id вершины* равен 0, значение будет соответствовать весу ребра. Таким образом, модель будет обеспечивать быстрое изменение веса ребра, однако, чтобы получить ребро с минимальным весом, придется вытащить все ребра и найти среди них ребро с минимальным весом.

Может показаться, что если в первой структуре размещать исходящие из вершины ребра, а во второй - входящие в вершину ребра, то модель будет подходить для ультраграфов. Однако, в таком случае нельзя по вершине будет найти все входящие в неё ребра.

Ограничения:

Идентификаторы вершины и ребра не должны быть равны 0.

3.3.6 Модель для ультраграфа

Данная модель подходит для ультраграфов и использует 2 структуры СП: связь вершина ребра (Таблица 21) и обратная связь ребро-вершина (Таблица 22).

Таблица 21 – Структура связи вершина – ребро для ультраграфа

Ключ			Значение
0	id вершины	0..0	Данные вершины
Бит инцидентности	id вершины	id ребра	

Структура хранит вершины и исходящие из них ребра. По этой структуре происходит поиск смежных ребер для указанной вершины. Старший бит ключа (*Бит инцидентности*) показывает является ли данное ребро входящим или исходящим (0 - входящее, 1 - исходящее), второй аргумент ключа является идентификатор вершины, а младший аргумент - идентификатор смежного ребра. Данные вершины хранятся в значении для записи с ключом, у которого аргумент *id вершины* соответствует этой вершине и аргумент *id ребра* равен 0.

Таблица 22 – Структура связи вершина – ребро для ультраграфа

Ключ			Значение
0	id ребра	0..0	Вес ребра
Бит инцидентности	id ребра	id вершины	

Структура хранит ребра и вершины, в которые входят эти ребра. По этой структуре происходит поиск смежных вершин для указанного ребра. Старший бит ключа (*Бит инцидентности*) показывает является ли это ребро входящим или исходящим (0 - входящее, 1 - исходящее), второй аргумент ключа является идентификатор ребра, а младший аргумент - идентификатор смежной вершины.

Для взвешенных графов в значении для записи с ключом, у которого аргумент *id ребра* соответствует указанному ребру, а аргумент *id вершины* равен 0, значение будет соответствовать весу ребра. Таким образом, модель будет обеспечивать быстрое изменение веса ребра, однако, чтобы получить ребро с минимальным весом, придется вытащить все ребра и найти среди них ребро с минимальным весом.

У данной модели есть недостатки, например, нельзя эффективно за одно обращение к СП узнать смежны ли две вершины. Для решения этой проблемы можно добавить третью структуру СП для связи *вершина – вершина – ребро* (Таблица 23). Но это увеличит затратность на операции изменения графа и значительно уменьшит допустимое количество вершин и ребер в графе.

Таблица 23 – Структура связи вершина – вершина – ребро для ультраграфа

Ключ				Значение
Бит инцидентности	id вершины	id вершины	id ребра	

Ограничения:

Идентификаторы вершины и ребра не должны быть равны 0.

3.3.7 Модель для взвешенного ультраграфа

Данная модель подходит для взвешенных ультраграфов и использует 2 структуры СП: связь вершина ребра (Таблица 24) и обратная связь ребро-вершина (Таблица 25).

Структура связи вершина – ребро хранит вершины и исходящие из них ребра. По этой структуре происходит поиск смежных ребер для указанной вершины. Старший бит ключа (*Бит инцидентности*) показывает является ли данное ребро входящим или исходящим (0 - входящее, 1 - исходящее), второй аргумент ключа является идентификатор вершины, третий - вес ребра, а младший аргумент - идентификатор смежного ребра. Данные вершины

хранятся в значении для записи с ключом, у которого аргумент *id вершины* соответствует этой вершине и аргумент *id ребра* равен 0.

Таблица 24 – Структура связи вершина – ребро для взвешенного ультраграфа

Ключ			Значение
0	0..0		Общее кол-во вершин
0	id вершины	0..0	Данные вершины
0	id вершины	1..1	Кол-во входящих ребер
Бит инцидентности	id вершины	Вес ребра	id ребра
1	id вершины	1..1	Кол-во исходящих ребер

Чтобы не пересчитывать количество смежных ребер предлагается для записи с битом инцидентности равным 0 и старшими аргументами (*Вес ребра* и *id ребра*), у которых биты равны 1, хранить количество входящих ребер, а для записи с битом инцидентности равным 1 хранить количество исходящих ребер. По адресу 0 хранится общее количество вершин.

Структура связи ребро - вершина (Таблица 25) хранит ребра и вершины, в которые входят эти ребра. По этой структуре происходит поиск смежных вершин для указанного ребра. Старший бит ключа (*Бит инцидентности*) показывает является ли данное ребро входящим или исходящим (0 - входящее, 1 - исходящее), второй аргумент ключа – идентификатор ребра, а младший аргумент – идентификатор смежной вершины.

Для взвешенных графов в значении для записи с ключом, у которого аргумент *id ребра* соответствует указанному ребру, а аргумент *id вершины* равен 0, значение будет соответствовать весу ребра.

Чтобы не пересчитывать количество смежных вершин предлагается для записи с битом инцидентности равным 0 и старшими аргументами (*Вес ребра* и *id ребра*), у которых биты равны 1, хранить количество вершин, из которых выходит ребро, а для записи с битом инцидентности равным 1

хранить количество вершин, в которое входит ребро. По адресу 0 хранится общее количество ребер.

Таблица 25 – Структура связи ребро – вершина для взвешенного ультраграфа

Ключ			Значение
0	0..0		Общее кол-во ребер
0	id ребра	0..0	Вес ребра
0	id ребра	1..1	Кол-во вершин, в кот. входит ребро
Бит инцидентности	id ребра	id вершины	
1	id ребра	1..1	Кол-во вершин, из кот. выходит ребро

Отличие данной модели от предыдущей - в первую SPU структуру был добавлен аргумент *Вес ребра*, благодаря которому будет осуществляется быстрый поиск ребер вершины с минимальным / максимальным весом. Однако для изменения веса у ребра придется пробежаться по всем связям *вершина – ребро* и у каждой изменить вес.

Ограничения:

- идентификаторы вершины и ребра не должны быть равны 0, а также все биты идентификаторов не должны быть равными 1;
- вес ребра должен быть целым положительным числом.

Эта графовая модель для системы с набором дискретной математики наиболее универсальна, т.к. другие виды графов являются частным случаем ультраграфа. Поэтому модель будет выбрана в качестве реализации в библиотеке элементов программного интерфейса для обработки графов.

3.3.8 Модель ультраграфа с атрибутами для ребер

Данная модель предназначена для ультраграфов, у которых ребра могут иметь несколько атрибутов, по которым должен происходить быстрый доступ

для смежных вершин. У такой модели будет достаточно долгое изменение атрибутов ребер, однако, появляется выигрыш при поиске смежных ребер с нужным атрибутом. Ниже показана структуры СП такой модели:

Таблица 26 – Структура связи вершина – ребро для ультраграфа с атрибутами для ребер

Ключ					Значение
0	0..0				Общее кол-во вершин
0	id вершины	0..0			Данные вершины
0	id вершины	1..1			Кол-во входящих ребер
Бит инцидентности	id вершины	id атрибута	Значение атрибута	id ребра	
1	id вершины	1..1			Кол-во исходящих ребер

Таблица 27 – Структура связи ребро – вершина для взвешенного ультраграфа с атрибутами для ребер

Ключ				Значение
0	0..0			Общее кол-во ребер
0	id ребра	0..0	id атрибута	Значение атрибута
0	id ребра	1..1		Кол-во вершин, в кот. входит ребро
Бит инцидентности	id ребра	id вершины	0..0	
1	id ребра	1..1		Кол-во вершин, из кот. выходит ребро

Ограничения:

- идентификаторы вершины и ребра не должны быть равны 0, а также все биты идентификаторов не должны быть равными 1;
- идентификатор атрибут должен быть натуральным числом;
- значение атрибута должно быть целым положительным числом.

3.3.9 Модель для хранения множества графов

Для реализации нескольких графов на одной системе с набором дискретной математики предлагается в графовые модели добавлять старший аргумент - идентификатор графа. Так, например, будет выглядеть модель (таблицы 28, 29) для реализации нескольких графов на основе модели ультраграфа:

Таблица 28 – Структура связи вершина – ребро для множества ультраграфов

Ключ				Значение
id графа	0	0..0		Общее кол-во вершин
id графа	0	id вершины	0..0	Данные вершины
id графа	0	id вершины	1..1	Кол-во входящих ребер
id графа	Бит инцидентности	id вершины	id ребра	
id графа	1	id вершины	1..1	Кол-во исходящих ребер

Таблица 29 - Структура связи ребро – вершина для множества ультраграфов

Ключ				Значение
id графа	0	0..0		Общее кол-во ребер
id графа	0	id ребра	0..0	Вес ребра
id графа	0	id ребра	1..1	Кол-во вершин, в кот. входит ребро
id графа	Бит инцидентности	id ребра	id вершины	
id графа	1	id ребра	1..1	Кол-во вершин, из кот. выходит ребро

4 Проектирование и разработка библиотеки

4.1 Анализ требований

Согласно техническому заданию библиотека элементов программного интерфейса для обработки графов будет использоваться для решения графовых задач для систем с дискретным набором команд. В библиотеке реализуется интерфейс для создания ультраграфов для таких систем.

Ультраграф – это ориентированный граф, в котором имеются ребра смежные с более чем двумя вершинами. Это наиболее универсальный тип графов, поскольку на базе него можно реализовывать другие типы графов: остовные деревья, обычные графы, мультиграфы, гиперграфы и т.д.

Библиотека должна реализовывать интерфейс графа предоставляемой библиотекой Boost Graph Library. Таким образом, для программ, использующих графы предоставляемый библиотекой boost, можно легко перейти на данный граф, просто подменив их. Также реализовав интерфейс графа boost, появляется возможность использовать множество его графовых алгоритмов.

На рисунке 12 показано структурно функциональная схема разрабатываемой системы. Библиотека для обработки графов для систем с дискретным набором команд должна реализовывать следующий функционал для обработки ультраграфов:

- добавление вершин и ребер;
- добавление и изменение свойств у вершин и ребер;
- добавление ребра между двумя вершинами;
- соединение вершины и ребра;
- удаление вершин и ребер;
- удаление ребер между двумя вершинами;
- отсоединение ребра от вершины;
- получение свойств вершины и ребра;

- проверка существования вершины или ребра в графе;
- проверка смежности двух вершин;
- получение смежных вершин и ребер.

Библиотека для обработки графов взаимодействует с системой с дискретным набором команд через программный интерфейс взаимодействия с процессором для обработки структур. У данного интерфейса имеются проблемы в обработке структур данных: ошибки при выполнении операций длинной арифметики (арифметические, логические и битовые операции с типом *data_t*). Необходимо доработать, отладить и исправить баги в программном интерфейсе взаимодействия с процессором для обработки структур. Также необходимо добавить возможность класть в структуры СП любые типы данных.

Для отладки на локальной машине необходимо реализовать симулятор структурного процессора, который должен со схожими асимптотическими сложностями выполнять большинство команд структурного процессора, а именно:

- создание структуры;
- удаление структуры;
- вставка ключ-значение (INS);
- удаление по ключу (DEL);
- поиск по ключу (SRCH);
- поиск соседнего ключа (NSM, NGR);
- переход к следующему и предыдущему ключу (NEXT, PREV);
- получение первого и последнего ключа в структуре (MIN, MAX).

Для написания исходного кода к библиотеке для обработки графов должен использоваться язык C++14. Для сборки библиотеки должна использоваться система автоматизированной сборки CMake.

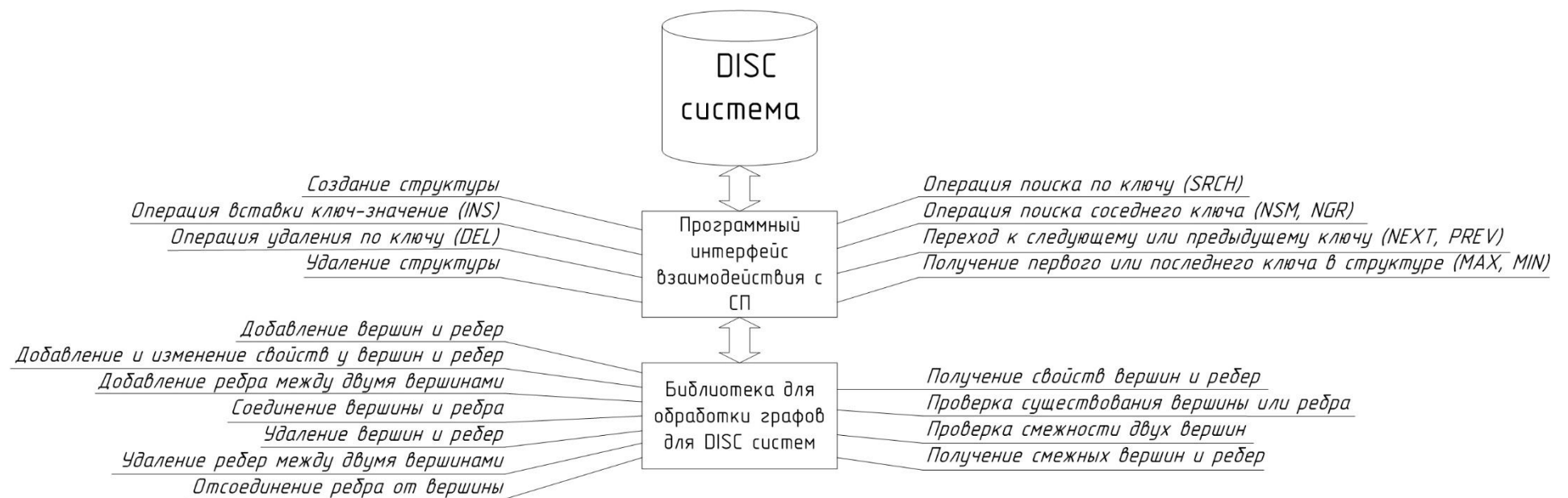


Рисунок 12 – Структурно-функциональная схема

4.2 Модель жизненного цикла разработки

Библиотеки элементов программного интерфейсов для обработки графов для систем с дискретным набором команд разрабатывалась согласно V-модели жизненного цикла разработки (Рисунок 13). Согласно этой модели, на каждом этапе происходит контроль текущего процесса, для того чтобы убедиться в возможности перехода на следующий уровень. В этой модели тестирование начинается еще со стадии написания требований, причем для каждого последующего этапа предусмотрен свой уровень тестового покрытия.

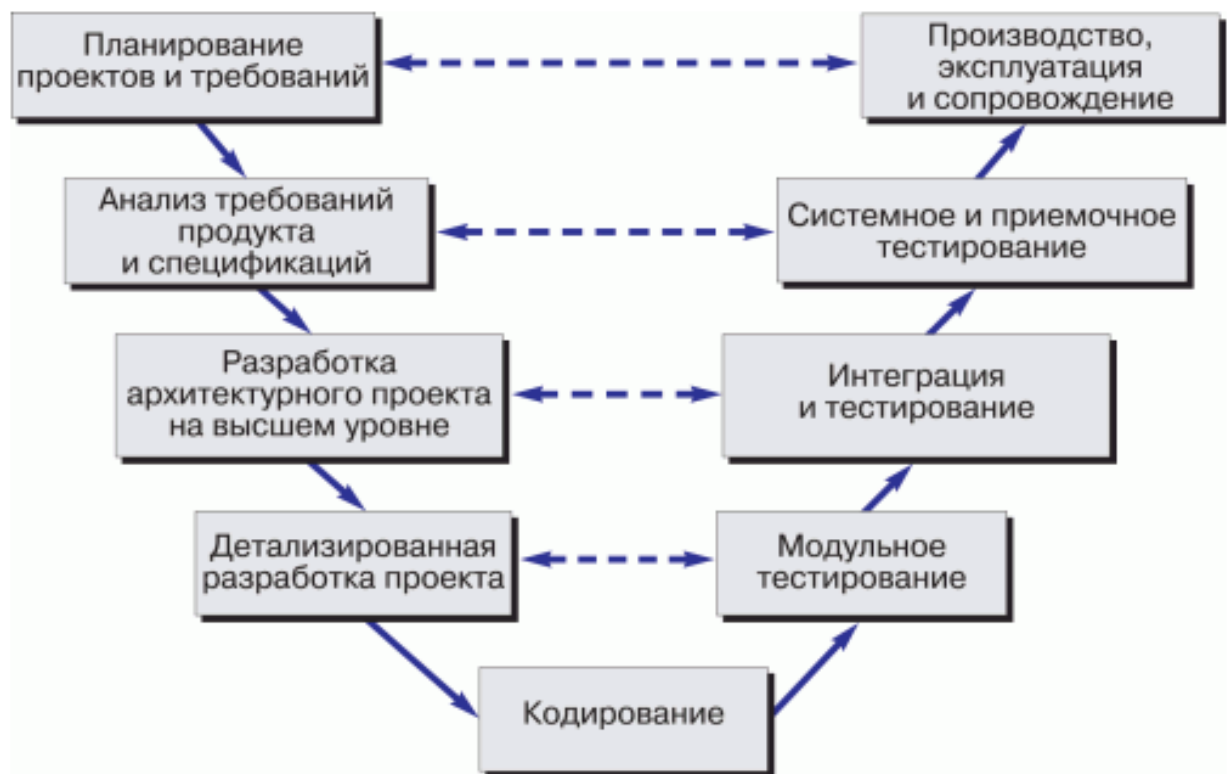


Рисунок 13 – V-модель жизненного цикла разработки [10]

На первом этапе были сформированы требования к интерфейсу библиотеки обработки графов. Был произведен анализ существующих технологий обработки графов, по результату которого было принято решение, что библиотека должна реализовывать интерфейс графа предоставляемой библиотекой Boost Graph Library. Также были сформированы требования к симулятору СП.

После чего был произведен анализ структурного процессора и анализ существующего программного интерфейса СП. В программном интерфейсе

были обнаружены проблемы в обработке структур данных. В частности, были обнаружены баги в «длинной арифметике». Было принято решение доработать и отладить существующий программный интерфейс.

Далее началась разработка программного интерфейса взаимодействия со структурным процессором. Для отладки на локальной машине был создан симулятор структурного процессора. При добавлении нового функционала были написаны соответствующие покрывающие тесты.

После разработки программного интерфейса СП началась разработка библиотеки элементов программного интерфейса для обработки графов для систем с дискретным набором команд. При реализации каждой концепции графов BGL производилось соответствующее тестирование.

4.3 Выбор методологии проектирования

Графовые абстракции — это мощный инструмент решения задач, используемых для описания отношений между дискретными объектами. Практические задачи могут быть смоделированы в виде графов для различных областей, например таких, как маршрутизации пакетов в Интернете, проектировании телефонной сети, систем сборки программного обеспечения, поисковых машин, молекулярная биология, систем автоматизированного планирования дорожного маршрута, научных вычислений и т. п. Достоинство графовой абстракции является тот факт, что найденное решение проблем теории графов может быть использовано для решения проблем в широком диапазоне областей. Например, задачи нахождения выхода из лабиринта и задачи нахождения групп взаимно достижимых веб-страниц могут быть решены с помощью поиска в глубину — важнейшего положения из теории графов.

Обобщенное программирование хорошо зарекомендовало себя при решении проблем повторного использования кода для библиотек алгоритмов на графах. В рамках обобщенного программирования алгоритмы на графах

могут быть сделаны более гибкими и легко используемыми для большого набора задач. Каждый графовый алгоритм пишется не в терминах специфической структуры данных, а для графовой абстракции, которая может быть реализована многим различным структурам данных. Написание обобщенных графовых алгоритмов имеет дополнительно преимущество, являясь более естественным. [4]

Как выше было сказано, библиотека элементов программного интерфейса для обработки графов должна реализовывать интерфейс графа предоставляемой библиотекой Boost Graph Library. Библиотека Boost Graph Library применяет понятия обобщенного программирования. Следовательно, при разработке библиотеки для обработки графов было решено использовать методологию обобщенного программирования для реализаций большинства концепций графов BGL.

При этом для реализации внутренних компонентов библиотеки применялась объектно-ориентированная методология разработки, т.к. объектно-ориентированный подход значительно повышает унификацию и читаемость кода, что увеличивает скорость разработки.

4.4 Программный интерфейс взаимодействия с СП

4.4.1 Принципы построения библиотеки

При разработке программного интерфейса взаимодействия с процессором обработки структур использовался объектно-ориентированный подход. Т.к. объектный подход существенно повышает уровень унификации разработки и пригодности для повторного использования кода, что увеличивает скорость разработки.

Ниже перечислены основные принципы построения библиотеки программного интерфейса взаимодействия с процессором обработки структур.

- все описания выполнены в пространстве имён *SPU*;
- структуры в памяти СП представляются как объекты (и управляются объектами);
- каждому объекту-структуре присваивается уникальный идентификатор – *GSID*;
- использование "длинной арифметики" для поддержки любой разрядности регистров СП. Создан специальный тип *data_t*;
- тип данных *data_t* репрезентует данных в регистрах СП (порядок следования байт – Little-endian);
- поддержка разметки *data_t* на поля с естественным порядком следования, произвольной длиной и любым типом данных;
- поддержка хранения в ОЗУ данных, больших разрядной сетки СП.

Типы данных *data_t* и *gsid_t* представляют собой структуры, в которых заключён массив 32-разрядных беззнаковых целых. Над типами данных определены операции в файле *containres_operations.hpp*. *GSID* является чисто суррогатным ключом структуры в памяти и имеет ограниченную поддержку "длинной арифметики".

Тип данных *data_t* репрезентует данных в регистрах СП и обязан соответствовать им. Тип строго Little-endian, при этом кратность элементов 32 разрядам соответствует кратности регистра СП. Компиляция заголовочного файла *spu.h* с поддержкой C++ позволяет использовать шаблонный конструктор типа *data_t* от любого типа.

4.4.2 Описание структур СП программного интерфейса

Класс *BaseStructure* (Рисунок 14) является базовым классом структуры и непосредственно взаимодействует с СП. Методы *insert*, *del*, *search*, *min*, *max*, *next*, *prev*, *nsm*, *ngr*, *get_power* реализуют соответствующие команды процессора с набором команд дискретной математики. Также имеются *protected* методы *adds* и *dels*, которые соответственно вызываются для

создания и удаления структуры СП. Все вышеперечисленные методы являются виртуальными для легкой подмены при использовании симулятора процессора. Поле *gsid* содержит глобальный идентификатор структуры СП, к которому относится текущий экземпляр *BaseStructure*.

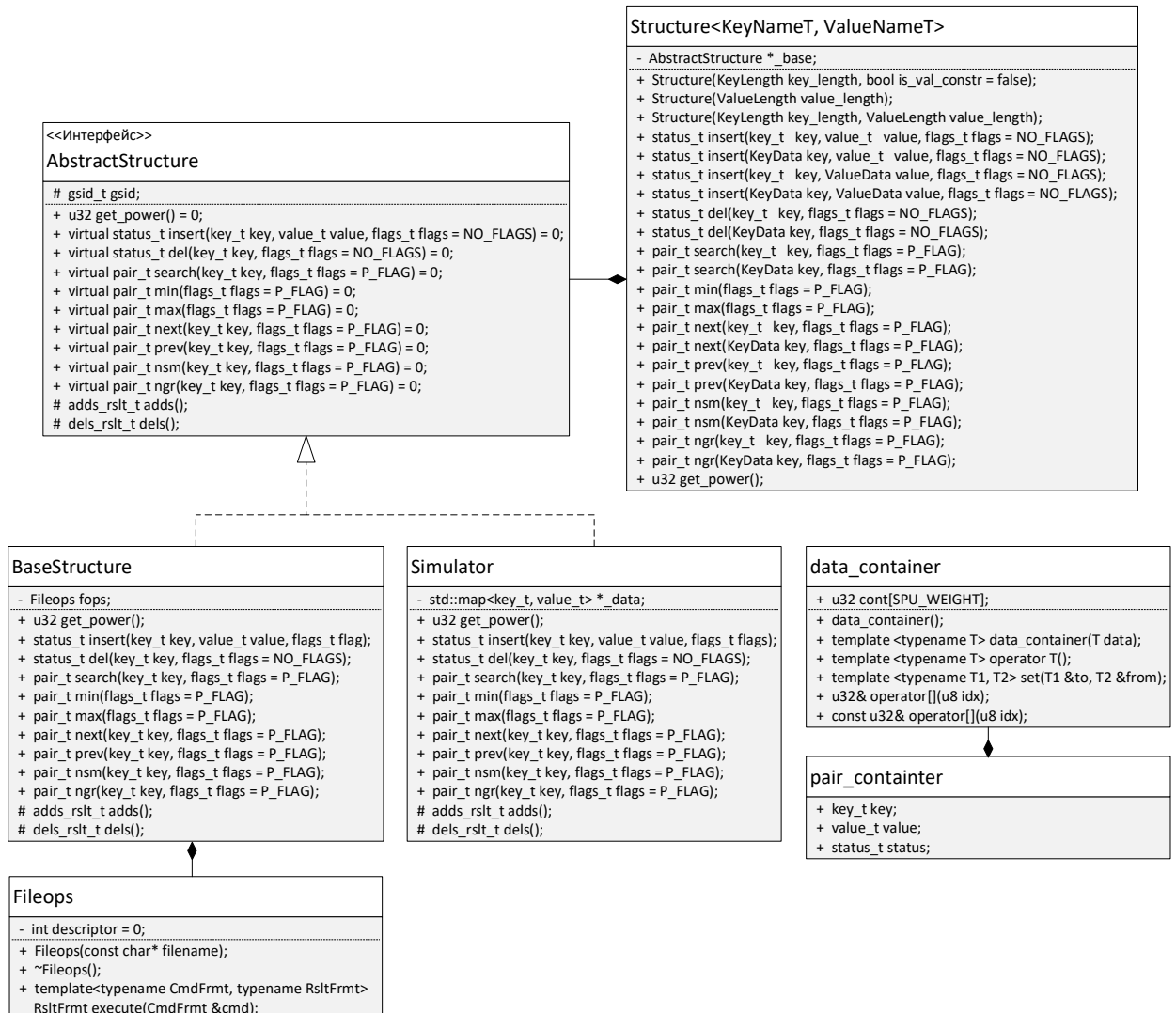


Рисунок 14 – Диаграмма классов структур СП

Класс структура *Structure* является шаблоном и реализует интерфейс взаимодействия со структурой СП. Описание структуры несколько разнится в зависимости от заданного типа названий в разбиении полей ключа структуры. Переданный в шаблон тип определяет тип идентификаторов аргументов ключа. Этот класс имеет конструкторы для создания новой структуры.

Структура *data_container* является основным типом данных, соответствующих данным, хранимым в процессоре обработки структур. У данного типа имеются сокращенные наименования: *data_t*, *key_t*, *value_t*. Для

этого типа был определен шаблонный оператор преобразования типа и шаблонный конструктор, таким образом есть возможность для конвертации любого типа данных, не превышающего 8 байт, к типу *data_container* и обратного преобразования. При конвертации к *data_container* байты базового типа записываются в содержимое этой структуры и аналогично происходит при обратной конвертации. Также были реализованы все основные арифметические, логические и битовые операторы для работы с типами *data_container*.

Ниже в листингах 1, 2 и 3 показаны примеры работы со структурами СП.

Листинг 1 – Пример создания структуры, вставки и поиска пары ключ-значение

```
#include "libspu/structure.hpp"

// Создание структуры struct1 с аргументами ключа: "one", "two",
// "three" и их размерами в битах
Structure<string> struct1({
    { "one",    5 },
    { "two",    7 },
    { "three", 10 }
});

// Вставка пары ключ-значение, где
// ключ: {"one": 3, "two": 3, "three": 3}
// значение: 1.123
struct1.insert({
    { "one",    3 },
    { "two",    3 },
    { "three",  3 }
}, 1.123);

// Получить значение для ключа {"one": 3, "two": 3, "three": 3}
pair_t pair = struct1.search({
    { "one",    3 },
    { "two",    3 },
    { "three",  3 }
});
if (pair.status == OK) {
    // value является парой двух 32-рядных чисел
    unsigned int val0 = pair.value[0];
    unsigned int val1 = pair.value[1];
    cout << "Found: " << val0 << " " << val1 << endl;
    cout << "Full pair is " << to_string(pair) << endl;
}
```

```

    // Также можно обратно получить исходное значение
    double dval = (BitFlow&) pair.value;
    cout << "Found double: " << dval << endl;
}

```

Листинг 2 – Пример демонстрации работы со структурой с шаблонным типом *void*

```

struct Point2D {float x; float y};
Point2D point = {3.14, 2.7};

// Создание структуры struct2 без определения аргументов ключа
Structure<> struct2;

// Можно вставлять любые структуры данных, размер которых меньше
8 байт
struct2.insert(123, 120);
struct2.insert("abc", 10.123);
struct2.insert(5.321, 50);
struct2.insert(8, point);

// И соответственно даस्ताвать
pair = struct2.search(123);
int val123 = pair.value;
pair = struct2.search("abc");
double valabc = pair.value;
pair = struct2.search(5.321);
int val5321 = pair.value;
pair = struct2.search(8);
Point2D valpoint = pair.value;

```

Листинг 3 - Пример демонстрации работы с методами min, max, next, prev, nsm, ngr

```

Structure<> struct3;
struct3.insert(2, 120);
struct3.insert(1, 10);
struct3.insert(5, 50);
struct3.insert(4, 40);

pair = struct3.search(2);
long long val2 = (BitFlow&) pair.value;
cout << "Value for struct 2: " << val2 << endl;

/// Получить пару ключ-значение с минимальным ключом в структуре
pair_t min = struct3.min();    //=> 1: 10
/// Получить пару ключ-значение с максимальным ключом в
структуре
pair_t max = struct3.max();    //=> 5: 50

/// Получить следующую пару ключ-значение
pair_t next = struct3.next(1); //=> 2: 120

```

```

/// Получить предыдущую пару ключ-значение
pair_t prev = struct3.prev(5); //=> 4: 40

/// Получить ближайшую пару ключ-значение у которой ключ меньше
заданного
pair_t nsm = struct3.nsm(3); //=> 2: 120
/// Получить ближайшую пару ключ-значение у которой ключ больше
заданного
pair_t ngr = struct3.ngr(3); //=> 4: 40

```

4.4.3 Разметка данных на поля

Класс *template<NameT> class Fields<NameT>* (Рисунок 15) отвечает за разметку полей данных. Конструктор класса принимает дескриптор длин полей *FieldsLength<NameT>*, в котором описывается имя поля с заданным типом NameT и произвольным значением длины (количество бит для этого поля).

Конкретные данные могут быть заданы как дескриптором *FieldsData<NameT>*. Данные по полям доступны с использованием оператора. Тип преобразуется к *data_t* автоматически при необходимости.

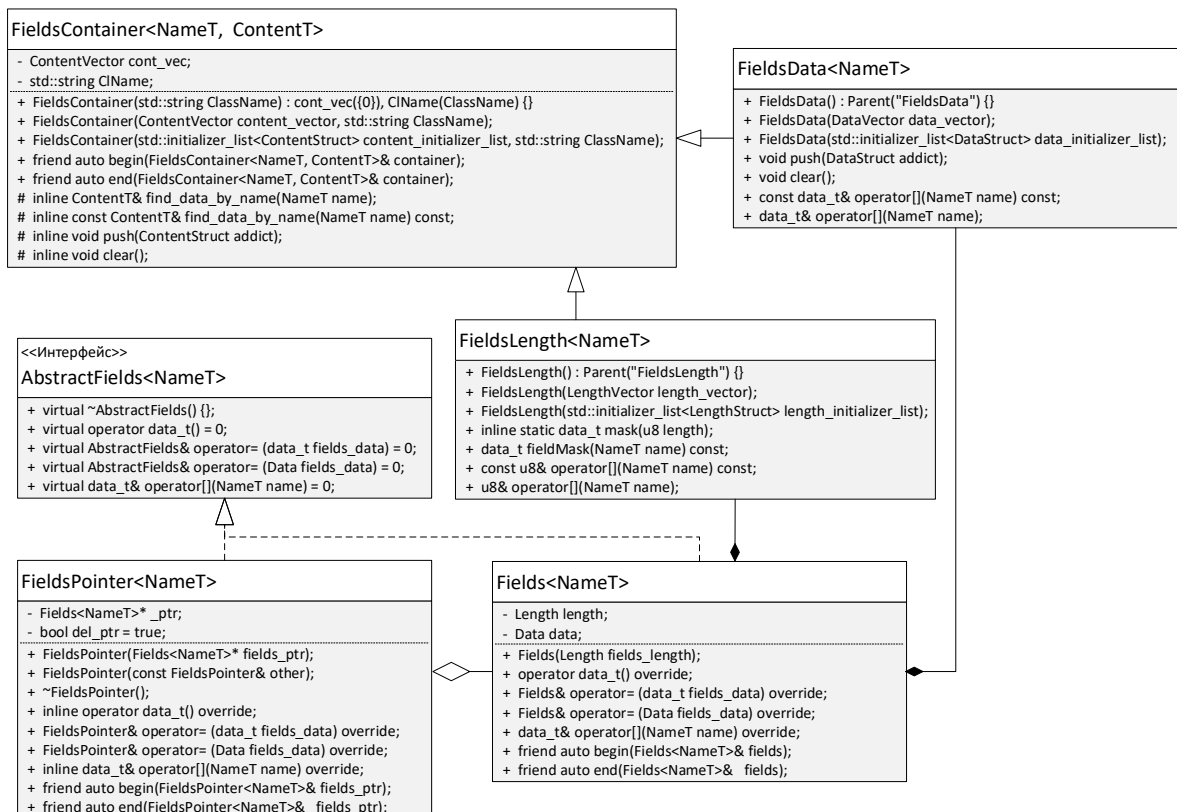


Рисунок 15 – Диаграмма классов полей

Для класса *Fields* существует "умный" указатель - класс *template<NameT> class FieldsPointer<NameT>*. Класс призван эффективно управлять указателем на разметку полей и не допускать повторного удаления. Этот класс необходим для хранения множества указателей на класс *Fields* с теми же методами, что и у класса *Fields* и удалением только одного.

Листинг 4 – Пример разметки данных по полям

```
Fields<string> F({
    { "a", 8 },
    { "b", 8 },
    { "c", 8 },
    { "d", 8 },
});

F = 0x1234;

cout << to_string(F["a"]) << endl; // 4
cout << to_string(F["b"]) << endl; // 3
cout << to_string(F["c"]) << endl; // 2
cout << to_string(F["d"]) << endl; // 1

F = {
    { "a", 255 },
    { "b", 15 },
    { "c", 13 },
    { "d", 0 },
};

cout << to_string(F) << endl; // 0x00000000-0x000D0FFF
```

4.4.4 Хранение крупных структур данных

Т.к. размер поля значения в структуре СП равен 64 битам, нельзя хранить в СП данные, размер которых превышает 8 байт. Как вариант можно в структуре СП хранить указатель на эти данные, либо воспользоваться реализованным интерфейсом *BaseExternValue* (Рисунок 16).

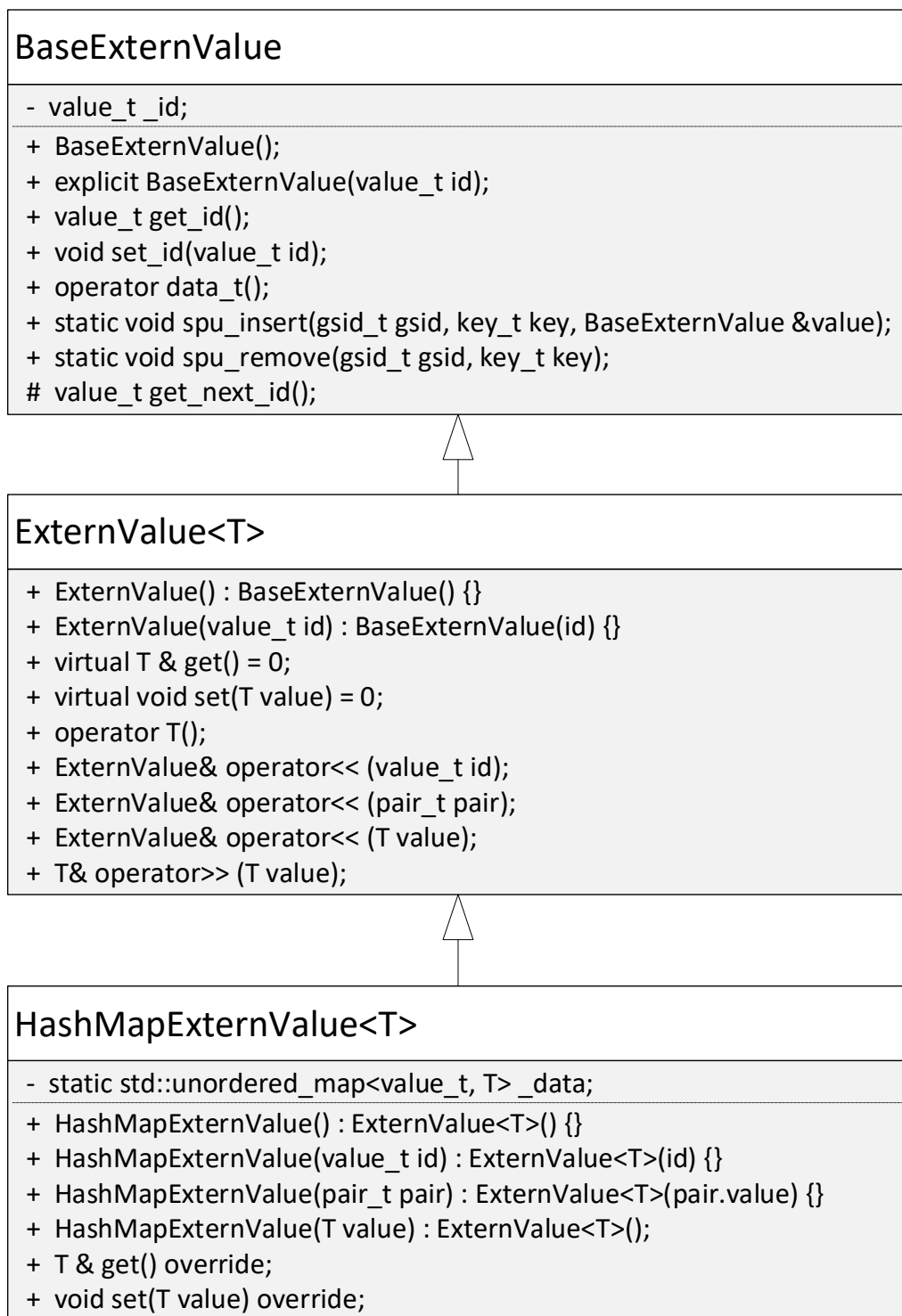


Рисунок 16 – Диаграмма классов хранилища крупных структур

Для хранения крупных структур, размер которых больше 8 байт, был определен интерфейс *BaseExternValue* (Рисунок 16), а также дочерний шаблонный интерфейс *ExternValue<class T>*, определяющий тип хранимого значения. У *BaseExternValue* имеется поле *_id*, которое является идентификатором для текущей структуры. Данный идентификатор будет записан как значение в структуру СП. Когда потребуется достать данные по

этому идентификатору будет найдена, соответствующая ему, крупная структура данных.

Для определения класса внешнего хранилища значений необходимо унаследоваться от интерфейса *ExternValue<class T>* и реализовать методы *T & get()*, *void set(T value)*, для соответствующего получения и записи данных. Ниже в листинге 5 приводится определение класса *HashMapExternValue<class T>*, который реализует хранение крупных структур внутри хеш-таблицы в оперативной памяти.

Листинг 5 – Определение класса *HashMapExternValue<class T>*

```
template <class T>
class HashMapExternValue : public ExternValue<T> {
    static std::unordered_map<value_t, T> _data;

    public:
    HashMapExternValue() : ExternValue<T>() {}
    /// Данный конструктор инициализирует id. Используется при
    получении данных из SPU.
    HashMapExternValue(value_t id) : ExternValue<T>(id) {}
    /// Данный конструктор инициализирует id. Используется при
    получении данных из SPU.
    HashMapExternValue(pair_t pair) :
    ExternValue<T>(pair.value) {}
    /// Данный конструктор должен записать данные.
    HashMapExternValue(T value) : ExternValue<T>() { set(value);
}

    T & get() override { return
_data[BaseExternValue::get_id()]; }
    void set(T value) override {
_data[BaseExternValue::get_id()] = value; }
};

template <class T>
std::unordered_map<value_t, T> HashMapExternValue<T>::_data =
std::map<value_t, T>();
```

Листинг 6 – Пример работы с крупными структурами данных

```
/// С помощью HashMapExternValue можно сохранять структуры
любого размера.
string string1 = "This string stored at hash map. In SPU stored
id for a string";
BaseExternValue extern_val =
HashMapExternValue<string>(string1);
struct2.insert(1, extern_val);
```

```

pair = struct2.search(1);
string res_str = (HashMapExternValue<string>) pair.value;
cout << res_str << endl;

struct Point {double x; double y; double z};
Point p = {1.5, 2.3, 3.7};
HashMapExternValue<Point> point_ext;
/// Операторы << и >> делают тоже, что и методы set и get
point_ext << p;
struct2.insert(2, point_ext);

pair = struct2.search(2);
if (pair.status == OK) {
    point_ext << pair;
    Point p_res;
    point_ext >> p;
    cout << "Point struct X=" << p.x << " Y=" << p.y << " Z=" <<
p.z << endl;
}

```

4.4.5 Разработка симулятора СП

Был реализован симулятор СП, для обеспечения ускорения при разработке программного обеспечения. Благодаря симулятору можно тестировать программный продукт при отсутствии процессора с набором команд дискретной математики.

Для включения симулятора СП достаточно в любом месте кода сделать объявление *SPU_SIMULATOR*.

Листинг 7 – Пример объявления *SPU_SIMULATOR*

```

/// Объявление SPU_SIMULATOR определяет использовать ли
симулятор SPU по умолчанию
#define SPU_SIMULATOR

/// Теперь все структуры по умолчанию используют в качестве
базовой структуры Simulator
Structure<> struct3;
struct3.insert(2, 120);
pair = struct3.search(2);

```


Также в конструктор структуры можно передать симулятор:

Листинг 8 – Пример инициализации структуры на базе симулятора

```
BaseStructure *baseStructure = new Simulator;  
Structure<string> struct1({  
    { "one",    5 },  
    { "two",    7 },  
    { "three", 10 }  
}, baseStructure);
```

Симулятор был реализован на базе отсортированного ассоциативного контейнера *std::map*, который имеет схожие асимптотики: добавление, удаление, обращение к элементам происходит за $O(\log n)$, где n — размер контейнера. Симулятор реализовывает интерфейс структуры, предоставляемый абстрактным классом *AbstractStructure*, и реализовывает следующие операции структурного процессора:

- операция создания и удаления структуры;
- вставка ключ-значение;
- удаление ключа из структуры;
- поиск по ключу;
- операции поиска соседнего ключа;
- команды перехода к следующему и предыдущему ключу;
- операции получения первого и последнего ключа в структуре.

4.5 Модель хранения графов

Для хранения ультраграфа используются следующие 2 структуры СП: связь вершина – ребро (Таблица 30) и обратная связь ребро – вершина (Таблица 31).

Таблица 30 – Структура связь вершина - ребро

Ключ				Значение
id графа	Бит инцидентности	id вершины	id ребра	
id графа	0	0 ... 0		Общее кол-во вершин
id графа	0	id вершины	0 ... 0	Данные вершины
id графа	0	id вершины	1 ... 1	Кол-во исходящих ребер
id графа	1	id вершины	1 ... 1	Кол-во входящих ребер

Структура связи вершина - ребро хранит вершины и исходящие из них ребра. По этой структуре происходит поиск смежных ребер для указанной вершины. Старший аргумент - идентификатор графа. Аргумент «Бит инцидентности» показывает является ли данное ребро входящим или исходящим (0 - исходящее, 1 - входящее) для вершины, третий аргумент ключа является идентификатор вершины, а младший аргумент - идентификатор смежного ребра. Данные вершины хранятся в значении для записи с ключом, у которого аргумент бит инцидентности равен 0, аргумент id вершины соответствует этой вершине и аргумент id ребра равен 0.

Таблица 31 – Структура связь ребро - вершина

Ключ				Значение
id графа	Бит инцидентности	id ребра	id вершины	
id графа	0	0 ... 0		Общее кол-во ребер
id графа	0	id ребра	0 ... 0	Данные ребра
id графа	0	id ребра	1 ... 1	Кол-во вершин, из кот. выходит ребро
id графа	1	id ребра	1 ... 1	Кол-во вершин, в кот. входит ребро

Структура связи ребро - вершина хранит ребра и вершины, в которые входят эти ребра. По этой структуре происходит поиск смежных вершин для указанного ребра. Аргумент «Бит инцидентности» показывает является ли данное ребро входящим или исходящим (0 - исходящее, 1 - входящее) для вершины, третий аргумент ключа является идентификатор ребра, а младший аргумент - идентификатор смежной вершины. К ребру можно прикрепить информацию о ребре. Эти данные хранятся в значении для записи с ключом, у которого аргумент бит инцидентности равен 0, аргумент id ребра соответствует этому ребру, и аргумент id вершины равен 0.

При поиске приоритет имеет та вершина или ребро, что имеет меньший id. Таким образом, для взвешенных графов следует формировать id ребра так, чтобы в старших битах идентификатора ребра находился вес этого ребра (для формирования такого идентификатора имеется вспомогательный метод `edge_descriptor get_edge_descriptor(id_t edge_id, weight_t weight);`).

4.6 Концепции ультраграфов

Библиотека элементов программного интерфейса для обработки графов реализует все концепции графов, предоставляемые библиотекой Boost Graph Library (п.п. 2.1.2 Графовые концепции). Однако, библиотека Boost Graph Library предоставляет концепции лишь для обыкновенных графов. Для решения этой проблемы были добавлены три концепции реализующий интерфейс для работы с ультраграфами. Ниже на рисунке 17 показана диаграмма концепций, реализованных в библиотеке элементов программного интерфейса для обработки графов.

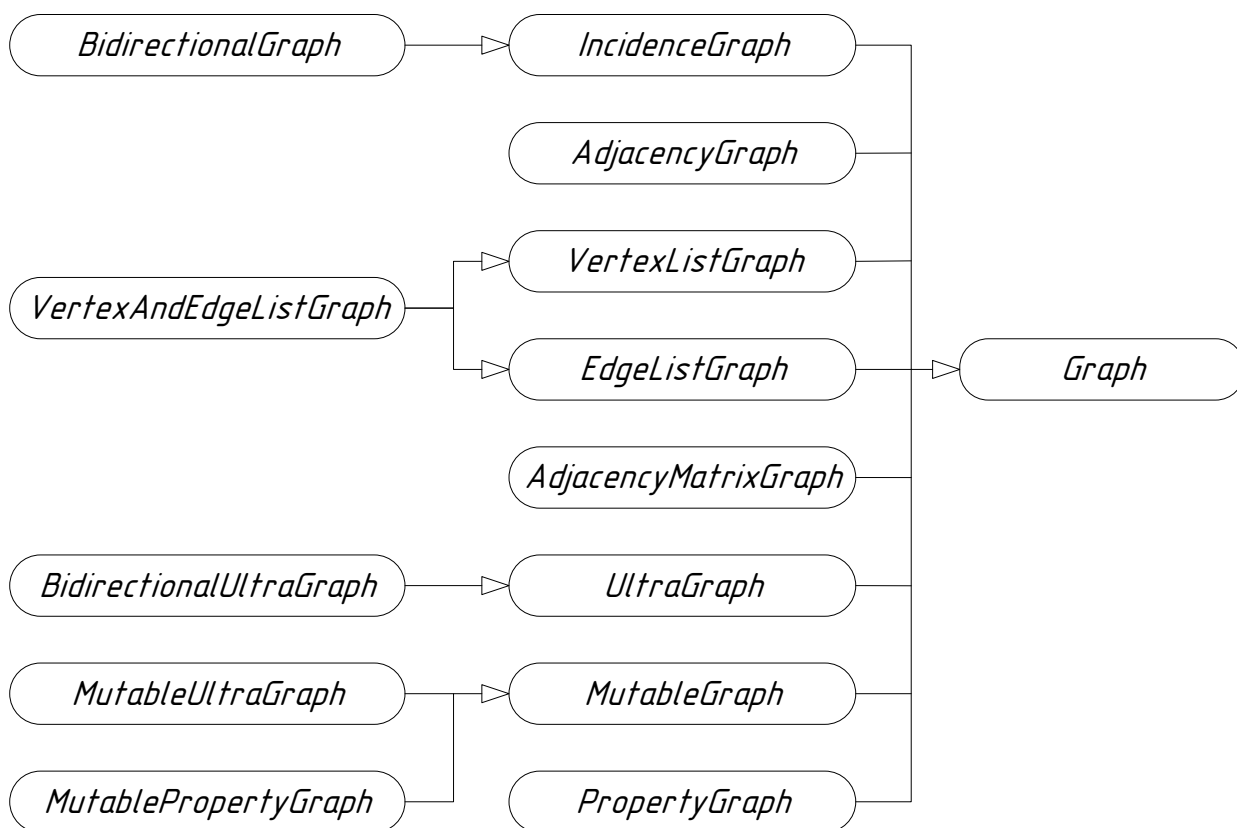


Рисунок 17 – Диаграмма концепций

Концепция ультраграфа *UltraGraph* (Таблица 32) добавляет итераторы для обхода по вершинам «стокам» для выбранного ребра.

Таблица 32 – Концепция ультраграфа (*UltraGraph*)

Выражение	Описание
<code>graph_traits<G>::target_iterator</code>	Итератор по вершинам «стокам».
<code>std::pair<target_iterator, target_iterator> targets(e, g)</code>	Возвращает диапазон итераторов, обеспечивающий доступ к стокам для ребра <i>e</i> в графе <i>g</i> .
<code>degree_size_type targets_cnt(e, g)</code>	Возвращает количество вершин «стоков» для ребра <i>e</i> .

Концепция двунаправленного ультраграфа *BidirectionalUltraGraph* (Таблица 33) уточняет концепцию *UltraGraph* и добавляет итераторы для обхода по вершинам «источника» для выбранного ребра. Эта концепция отделена от *UltraGraph*, потому что для ультраграфов эффективный доступ к вершинам «источников» обычно требует больше места для хранения.

Таблица 33 – Концепция двунаправленного ультраграфа (*BidirectionalUltraGraph*)

Выражение	Описание
<code>graph_traits<G>::source_iterator</code>	Итератор по вершинам «источников».
<code>std::pair<source_iterator, source_iterator> sources(e, g)</code>	Возвращает диапазон итераторов, обеспечивающий доступ к источникам для ребра <i>e</i> в графе <i>g</i> .
<code>degree_size_type sources_cnt(e, g)</code>	Возвращает количество вершин «источников» для ребра <i>e</i> .

Концепция изменяемого ультраграфа *MutableUltraGraph* (Таблица 34) уточняет концепцию *MutableGraph* и добавляет методы для соединения ребер с вершинами.

Таблица 34 – Концепция изменяемого ультраграфа (*MutableUltraGraph*)

Выражение	Описание
<code>void connect_source(e, v, g)</code>	Присоединяет как «источник» вершину <i>v</i> к ребру <i>e</i> .
<code>void connect_target(e, v, g)</code>	Присоединяет как «сток» вершину <i>v</i> к ребру <i>e</i> .
<code>void disconnect_source(e, v, g)</code>	Отсоединяет вершину «источник» <i>v</i> от ребра <i>e</i> .
<code>void disconnect_target(e, v, g)</code>	Отсоединяет вершину «сток» <i>v</i> от ребра <i>e</i> .

4.7 Основные классы обработки графов

На рисунке 18 показана диаграмма основных классов библиотеки элементов программного интерфейса обработки графов для систем с дискретным набором команд.

Класс *SpuUltraGraph* является основным классом для работы с ультраграфами. Также класс *SpuUltraGraph* позволяет создавать графы с параллельными ребрами, т.е. добавлять несколько ребер между двумя вершинами.

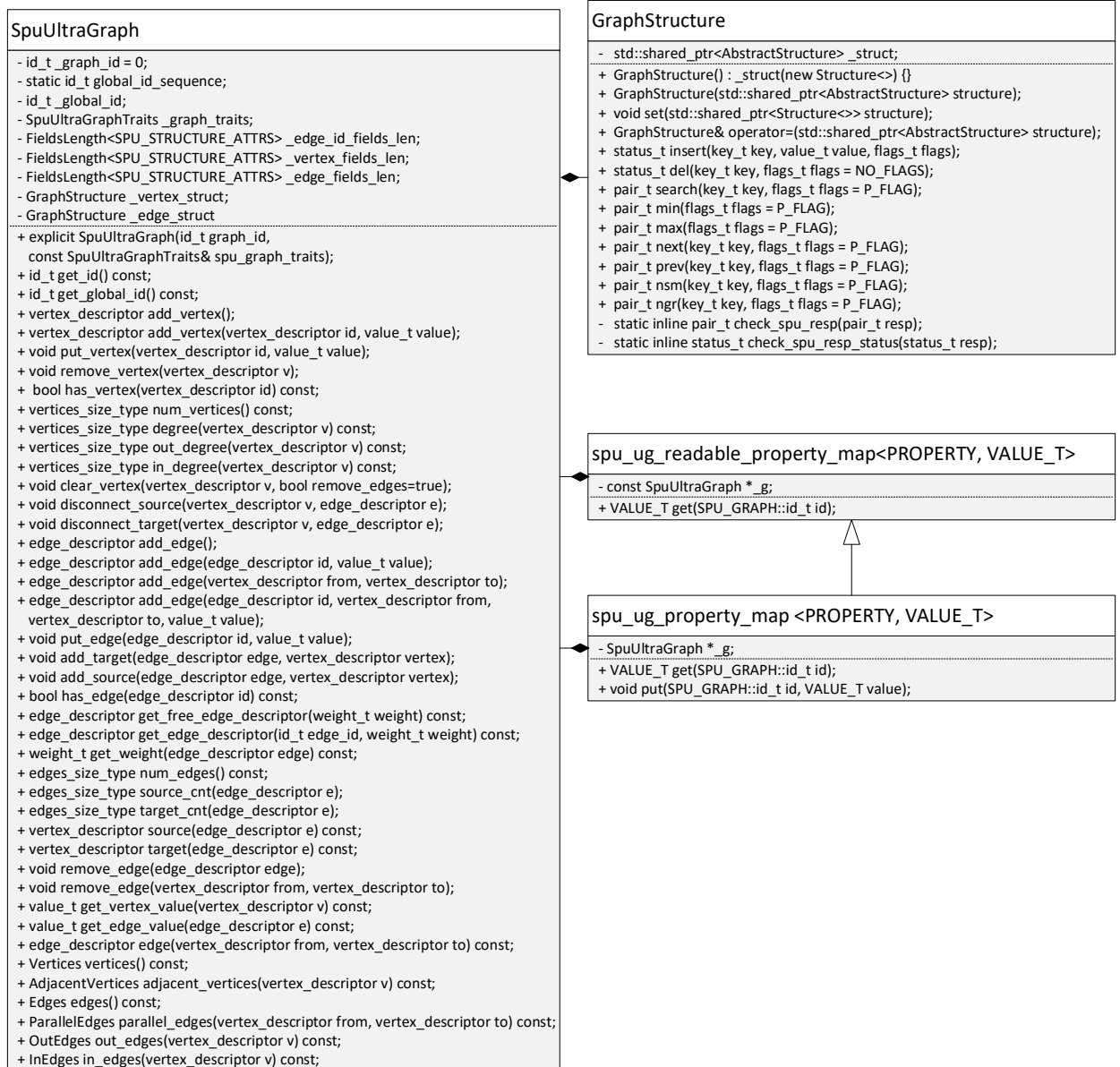


Рисунок 18 – Диаграмма основных классов библиотеки обработки графов

Класс *SpuUltraGraph* реализовывает все концепции boost графов (п.п. 2.1.2 Графовые концепции):

- концепция изменяемых графов (*MutableGraphConcept*);
- концепция списка вершин (*VertexListGraphConcept*);
- концепция списка ребер (*EdgeListGraphConcept*);
- концепция графа инцидентности (*IncidenceGraphConcept*);

- концепция двунаправленного графа (*BidirectionalGraphConcept*);
- концепция графа смежности (*AdjacencyGraphConcept*);
- концепция матрицы смежности (*AdjacencyMatrixConcept*);
- концепция свойств графа (*PropertyGraphConcept*).

А также он реализовывает все концепции ультраграфов (п.п. 4.6 Концепции ультраграфов):

- концепция ультраграфа (*UltraGraph*);
- концепция двунаправленного ультраграфа (*BidirectionUltraGraph*);
- концепция изменяемого ультраграфа (*MutableUltraGraph*).

Дескрипторы вершины и ребра (*vertex_descriptor*, *edge_descriptor*) имеют тип *unsigned long* и являются идентификаторами на соответствующую вершину или ребро в графе. Эти дескрипторы служат для предоставления доступа к вершинам и ребрам графа. Нужно отметить, что при поиске вершины или ребра приоритет имеет та вершина или ребро у кого меньше идентификатор (*id*). Таким образом, для взвешенных графов следует формировать идентификатор ребра так, чтобы в старших битах идентификатора ребра находился вес этого ребра. Для формирования такого идентификатора имеется вспомогательный метод *edge_descriptor get_edge_descriptor(id_t edge_index, weight_t weight) const*. Также имеется метод для поиска свободного дескриптора ребра для определенного веса *edge_descriptor get_free_edge_descriptor(weight_t weight) const*.

Класс *GraphStructure* является декоратором над классом *Structure* от программного интерфейса СП и реализовывает все его методы. Класс предназначен для выявления ошибок при работе со структурным процессором и отправки соответствующих исключений.

Шаблонные классы *sru_ug_readable_property_map* и *sru_ug_readable_property_map* предназначены для реализации концепции свойств графа. Они предоставляют интерфейс для получения и изменения соответствующего свойства у вершины или ребра.

Для реализации всех концепций графов были разработаны соответствующие контейнеры и итераторы (Рисунок 19), которые позволяют проводить все возможные виды обходов вершин и ребер графа.

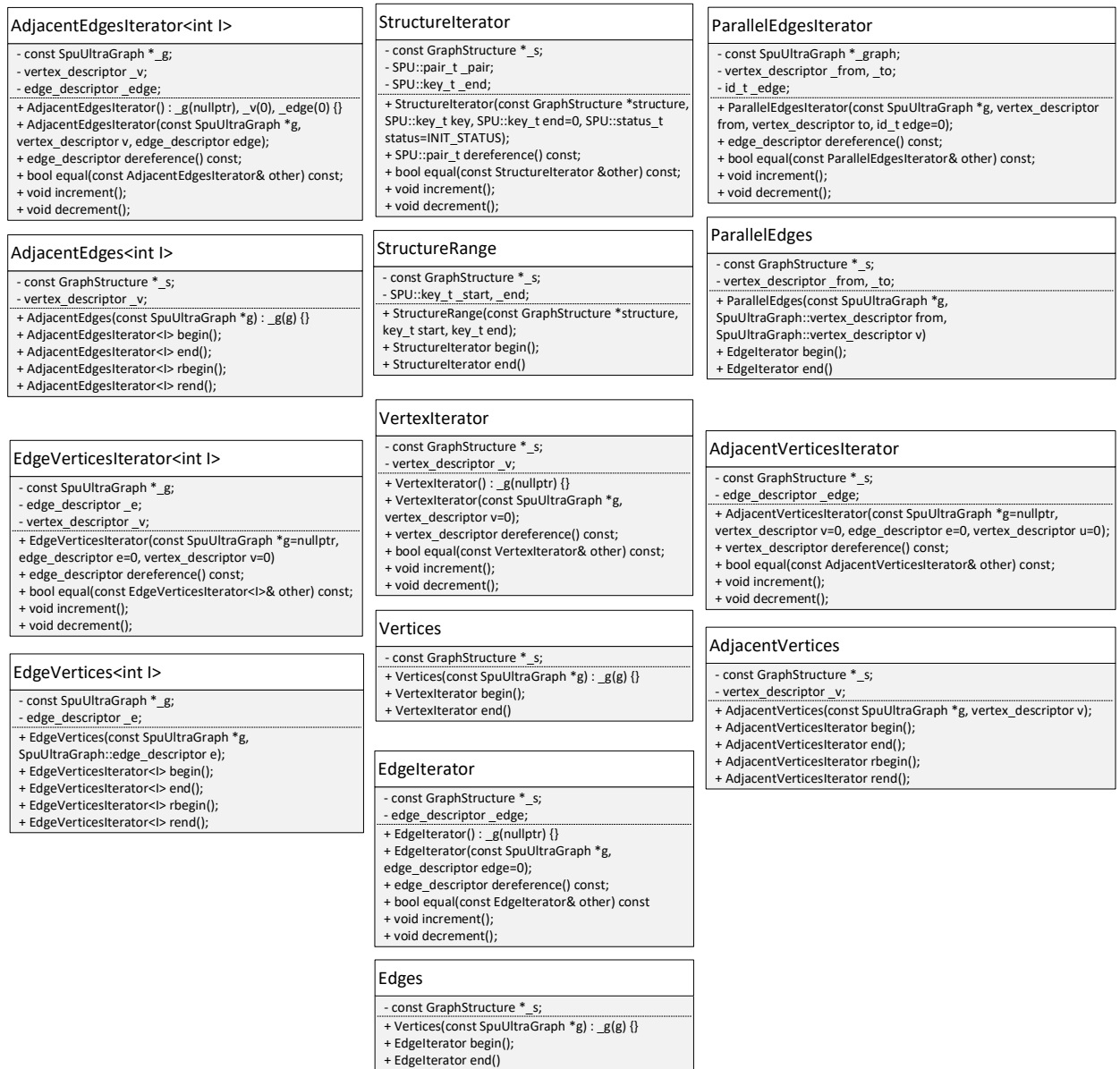


Рисунок 19 – Диаграмма классов контейнеров и итераторов

4.8 Алгоритм поиска свободного домена

В структурах СП хранение любых данных организуется следующим образом (Таблица 35): в старших битах структуры СП находится префикс, который объединяет в себе множество структур данных. У каждой структуры данных имеется свой собственный идентификатор, значение которого

соответствует значению домена этой структуры данных. И вся информация об этой структуре данных кладется внутрь этого домена под соответствующие адреса. Т.е. домен — это группа ключей, у которых одинаковая старшая часть. Например, для ребер графа (Таблица 31 – Структура связь ребро - вершина) префиксом будут являться поля с идентификатором графа и битом инцидентности, доменом – идентификатор ребра, а внутри адреса будет храниться информация о количестве смежных вершин и информация о самих смежных вершинах.

Таблица 35 – Организация хранения данных в СП

Ключ (64 бит)			Значение (64 бит)
Префикс	Домен	Адрес	

Была разработана функция для поиска свободного домена в структуре СП, рекурсивный алгоритм которого представлен на рисунке 20. Идея алгоритма заключается в рекурсивном делении пополам области доменов в структуре СП и поиска внутри незанятого домена. Ниже показана асимптотическая сложность данного алгоритма:

$$O(\log_2(2^{D_{domain}}) * \log_8(n)) = O(D_{domain} * \log_8(n)), \quad (2)$$

где D_{domain} – битность домена,

n – количество записей в структуре СП,

$\log_8(n)$ – асимптотическая сложность команд поиска в структуре СП.

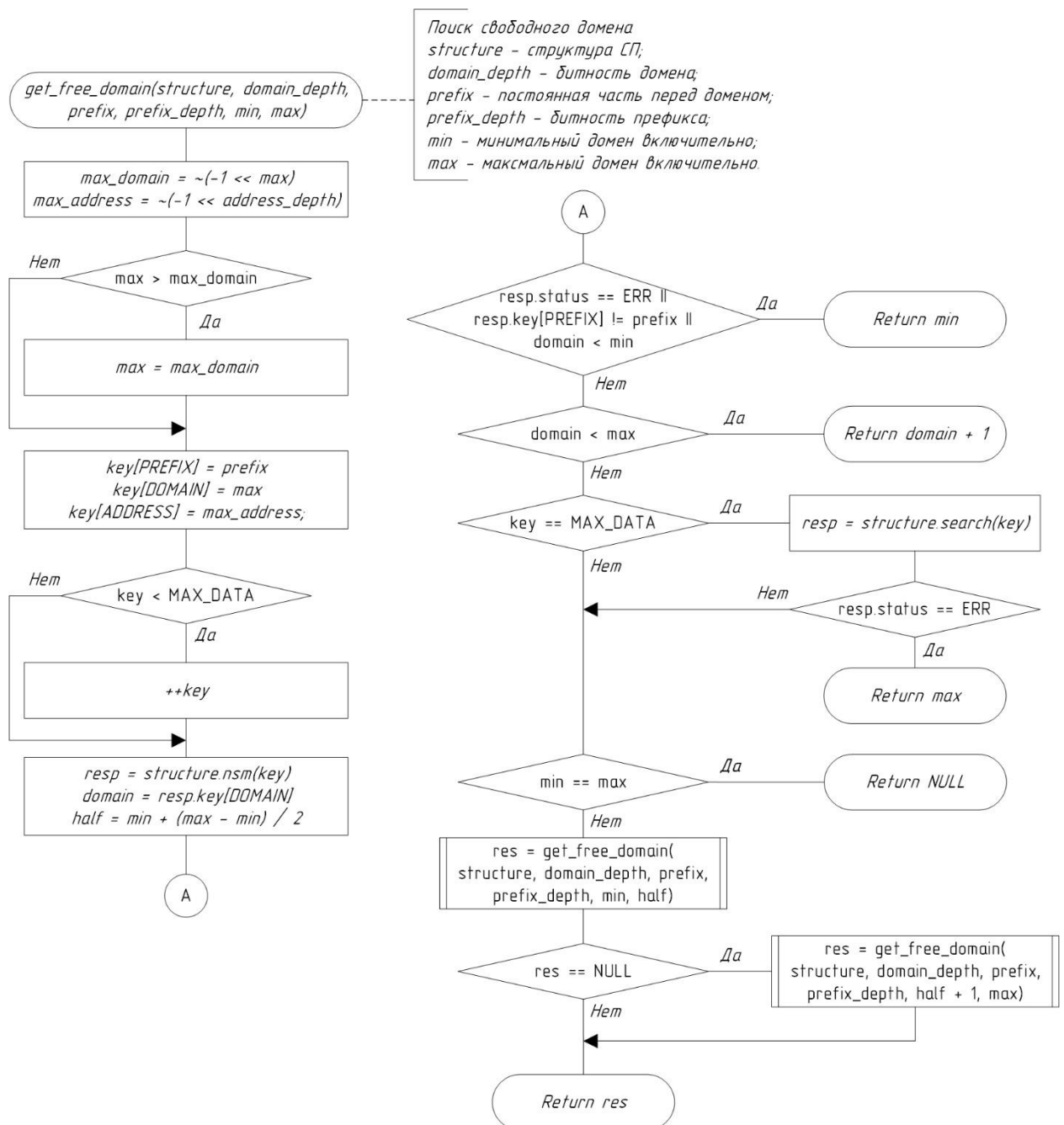


Рисунок 20 – Схема алгоритма поиска свободного домена

Данный алгоритм используется при поиске свободного идентификатора для вершин и ребер в графе при добавлении. Как правило, идентификаторы вершин и ребер располагаются последовательно по порядку, поэтому можно применить оптимизацию: сохранять последний добавленный идентификатор вершины или ребра и перед выполнением алгоритма поиска домена проверить не свободен ли следующий идентификатор. Схема озвученного алгоритма для поиска свободного id ребра представлена рисунке 21.

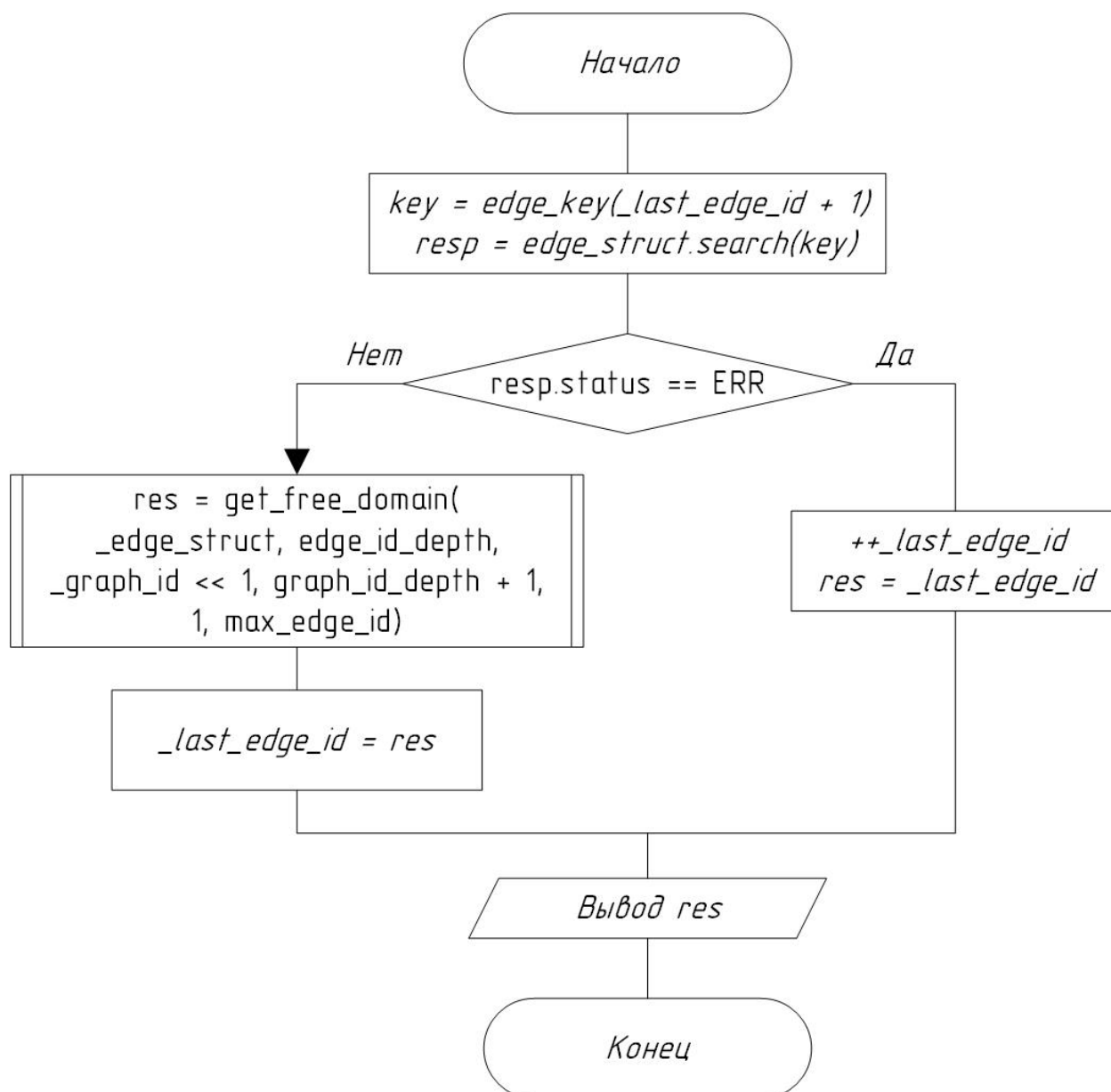


Рисунок 21 – Схема алгоритма поиска свободного id ребра

4.9 Руководство по установке библиотеки

Данное руководство показывает процесс установки, разработанной библиотеки для Linux систем. Перед установкой библиотеки программного интерфейса для обработки графов понадобится установить систему управления версиями Git, библиотеку Boost версии 1.72, а также компилятор GCC и систему автоматизации сборки CMake. После чего переходим к установке:

1. Вначале необходимо в папку с исходным кодом проекта скачать репозиторий с библиотекой элементов программного интерфейса для обработки графов. Для этого выполняем команду:

```
git clone https://github.com/kiryanenko/graph-api.git
```

2. После необходимо инициализировать подмодуль с программным интерфейсом взаимодействия с СП. Для этого переходим в папку библиотеки и выполняем следующие команды:

```
cd ./graph-api
```

```
git submodule init
```

```
git submodule update
```

3. Далее в файле проекта *CMakeLists.txt* необходимо подключить библиотеку в листинге 9 показан пример файла *CMakeLists.txt*.

Листинг 9 – Пример файла CMakeLists.txt

```
cmake_minimum_required(VERSION 3.14)
project(my_project)

set(CMAKE_CXX_STANDARD 14)

set(SPU_ARCH 64)

# Включаю симулятор СП
set(SPU_SIMULATOR ON)
if(${SPU_SIMULATOR})
    set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -DSPU_SIMULATOR")
endif(${SPU_SIMULATOR})

set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS}
${GCC_COVERAGE_COMPILE_FLAGS} -Wall -ggdb -DSPU${SPU_ARCH}")

add_executable(main main.cpp)

# Подключаю библиотеку
include_directories(graph-api)
target_link_libraries(main graph-api)
```

4. Теперь можно пользоваться библиотекой элементов программного интерфейса для обработки графов. В папке *graph-api/examples* лежат примеры работы с библиотекой элементов программного интерфейса для обработки графов. Также в папке

graph-api/doxygen/html находится HTML документация по данной библиотеке.

Подробнее об установке и использовании библиотеки элементов программного интерфейса для обработки графов описано в приложении Б. Далее в следующей главе показаны примеры работы библиотекой для обработки графов.

4.10 Примеры работы с библиотекой

В листинге 10 приводятся примеры инициализации графов. Показано как можно настроить количество выделяемых бит под идентификаторы графов, вершин и ребер.

Листинг 10 – Пример инициализации графа

```
// Инициализация графа по умолчанию
// В этом случае id графа будет равно 0
SpuUltraGraph graph;

// Объект черт графа
SpuUltraGraphTraits traits;
// Задаю битность. По умолчанию она такая:
traits.graph_id_depth = 3;          // Кол-во бит под id графа
traits.vertex_id_depth = 28;        // Кол-во бит под id вершины
traits.edge_id_depth = 28;          // Кол-во бит под индекс ребра
// (Индекс ребра НЕ id ребра. id ребра состоит из веса и индекса)
traits.weight_depth = 4;            // Кол-во бит под вес ребра
// Указываю структуры, кот будет использовать наш граф
GraphStructure vertex_struct, edge_struct;
traits.vertex_struct = vertex_struct;
traits.edge_struct = edge_struct;

// Инициализация графа с id = 5 и указанием черт
SpuUltraGraph custom_graph = SpuUltraGraph(5, traits);
```

В листинге 11 приводится пример работы с библиотекой. Показано как можно добавлять и удалять вершины и ребра, а также показаны примеры выполнения других базовых операций с графом.

Листинг 11 – Пример выполнения базовых операций с графом

```
// Добавление вершины
SpuUltraGraph::vertex_descriptor v1 = graph.add_vertex();
// Добавление вершины с указанием id
auto v2 = graph.add_vertex(2);
// Добавление вершины с id = 3 и value = 123.123
auto v3 = graph.add_vertex(3, 123.123);

// Добавление ребра
SpuUltraGraph::edge_descriptor e1 = graph.add_edge();
// Добавление ребра с указанием id
auto e2 = graph.add_edge(2);
// К ребру можно прикрепить данные
auto e3 = graph.add_edge(3, data_t("abc"));
// Добавление ребра от v1 к v2
auto e12 = graph.add_edge(v1, v2);

// Чтобы добавить ребро с весом,
// прежде необходимо сформировать edge_descriptor.
// Здесь формируется edge_descriptor,
// у которого вес равен 10 и индекс ребра равен 5.
// Индекс ребра это не id ребра.
// Как раз id ребра состоит из веса и индекса.
SpuUltraGraph::edge_descriptor e4 =
    graph.get_edge_descriptor(5, 3);
// Добавление ребра e4 от вершины v2 к v3
graph.add_edge(e4, v2, v3);

// Получение веса ребра из edge_descriptor
cout << "Вес ребра e4 = " << graph.get_weight(e4) << endl;
// Получение данных вершины.
// Если вершина не найдена будет исключение
cout << "Данные вершины v3 = "
    << (double) graph.get_vertex_value(v3) << endl;
// Получение данных вершины.
// Если вершина не найдена будет исключение
cout << "Данные ребра e3 = "
    << (const char *) graph.get_edge_value(e3) << endl;

// Проверка наличия вершины
cout << "Наличие вершины v1 = " << graph.has_vertex(v1) << endl;
// Проверка наличия ребра
cout << "Наличие ребра e1 = " << graph.has_edge(e1) << endl;

cout << "Кол-во вершин = " << graph.num_vertices() << endl;
cout << "Кол-во ребер = " << graph.num_edges() << endl;
cout << "Кол-во исходящих ребер у вершины v1 = "
    << graph.out_degree(v1) << endl;
cout << "Кол-во входящих ребер у вершины v1 = "
    << graph.in_degree(v1) << endl;
cout << "Кол-во вершин 'источников' у ребра e12 = "
    << graph.num_sources(e12) << endl;
```

```

cout << "Кол-во вершин 'стоков' у ребра e12 = "
      << graph.num_targets(e12) << endl;

// Удаление вершины
graph.remove_vertex(v1);
// Удаление всех ребер соединенных с вершиной v2
graph.clear_vertex(v2);

// Удаление ребра
graph.remove_edge(e1);
// Удаление всех ребер от v2 к v3
graph.remove_edge(v2, v3);

```

В листинге 12 приводятся примеры обходов вершин и ребер в графе.

Листинг 12 – Примеры обхода графа

```

cout << "Итерация по всем вершинам: ";
for (auto v : graph.vertices())
    cout << v << ' ';
cout << endl;

cout << "Итерация по всем ребрам: ";
for (auto e : graph.edges())
    cout << e << ' ';
cout << endl;

cout << "Итерация по исходящим ребрам у вершины v1: ";
for (auto e : graph.out_edges(v1))
    cout << e << ' ';
cout << endl;

cout << "Итерация по входящим ребрам у вершины v1: ";
for (auto e : graph.in_edges(v1))
    cout << e << ' ';
cout << endl;

cout << "Итерация по параллельным ребрам от вершины v1 к v2: ";
for (auto e : graph.parallel_edges(v1, v2))
    cout << e << ' ';
cout << endl;

cout << "Итерация по смежным вершинам для v1: ";
for (auto v : graph.adjacent_vertices(v1))
    cout << v << ' ';
cout << endl;

```

В следующем примере (Листинг 13) при помощи утилиты автоматической визуализации графов GraphViz происходит создание SVG изображения заранее заданного графа.

Листинг 13 – Пример визуализации графа

```
// Класс для печати свойств вершины для graphviz
class vertex_property_writer {
    const SpuUltraGraph &_g;
public:
    vertex_property_writer(SpuUltraGraph &g) : _g(g) {}
    void operator()(std::ostream& out, const
SpuUltraGraph::vertex_descriptor &v) const {
        cout << "[label=\"" << v << "\"]";
    }
};

// Класс для печати свойств ребра для graphviz
class edge_property_writer {
    const SpuUltraGraph &_g;
public:
    edge_property_writer(SpuUltraGraph &g) : _g(g) {}
    void operator()(std::ostream& out, const
SpuUltraGraph::edge_descriptor &e) const {
        auto weight = get(edge_weight, _g, e);
        cout << "[label=\"" << weight << "\",weight=\""
            << weight << "\"]";
    }
};

// ...

// Распечатать в консоль граф в формате graphviz
vertex_property_writer vpw(graph);
edge_property_writer epw(graph);
write_graphviz(cout, graph, vpw, epw);

// С помощью GraphViz создадим svg изображение графа
std::ofstream f("graph.dot");
boost::write_graphviz(f, graph, vpw, epw);
f.close();
system("dot graph.dot -Kcirco -Tsvg -o graph.svg");
```

В результате выполнения данного кода генерируется следующее SVG изображение графа:

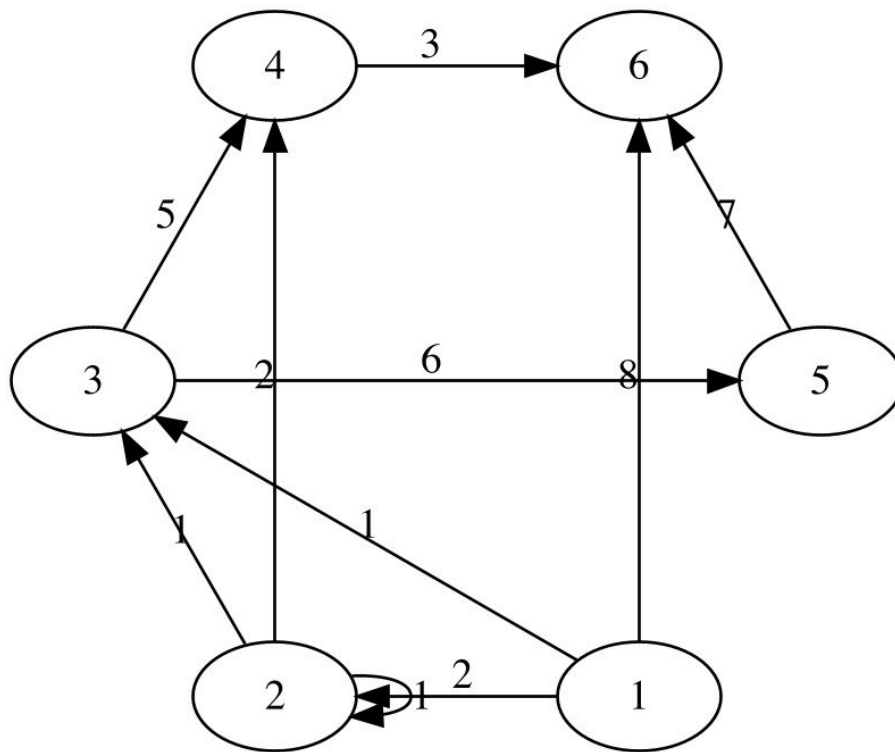


Рисунок 22 – Сгенерированное изображение графа.

В последнем примере (Листинг 14) показано применение алгоритма Дейкстры для подсчета расстояний от вершины №1 до остальных. Перед применением алгоритма Дейкстры необходимо для вершин создать два внешних свойства: предшественник и расстояние.

Листинг 14 – Пример применения алгоритма Дейкстры

```

// Создаю свойство предшественник для вершин
map<SpuUltraGraph::vertex_descriptor,
SpuUltraGraph::vertex_descriptor> vertex_to_predecessor;
associative_property_map<
    map<SpuUltraGraph::vertex_descriptor,
        SpuUltraGraph::vertex_descriptor>
> predecessor_property_map(vertex_to_predecessor);
// Создаю свойство расстояние для вершин
map<SpuUltraGraph::vertex_descriptor, size_t>
vertex_to_distance;
associative_property_map<
    map<SpuUltraGraph::vertex_descriptor, size_t>
> distance_property_map(vertex_to_distance);

// Выполняю алгоритм дейкстра для подсчета расстояний
// от вершины #1 до остальных
dijkstra_shortest_paths_no_color_map(graph, 1,
predecessor_map(predecessor_property_map).distance_map(distance_
property_map));

```

```
std::cout << "Distances and parents:" << std::endl;
for (auto v: graph.vertices()) {
    cout << "Vertex " << v
        << " Distance = " << vertex_to_distance[v]
        << ", parent = " << vertex_to_predecessor[v] + 1
        << endl;
}
```

5 Обеспечение качества библиотеки

5.1 Особенности работы с ультраграфами

Для реализации интерфейса для работы с ультраграфами в библиотеку были добавлены методы для добавления к ребру и удаления из ребра вершин «источников» и «стоков». А также были добавлены соответствующие итераторы для обхода вершин «источников» и «стоков» для выбранного ребра.

Нужно понимать, что для реализации ультраграфа требуется две структуры СП (п.п. 4.5). Поэтому, увеличиваются в 2 раза затраты по производительности и ресурсам для операций изменения графа.

В текущей версии библиотеки нельзя за одно обращение к СП узнать смежны ли две вершины в графе. На данный момент это решается путем «умного» перебора смежных ребер для этих двух вершин.

Для решения этой проблемы можно добавить третью структуру СП для связи *вершина – вершина – ребро* (Таблица 23 – Структура связи вершина – вершина – ребро для ультраграфа). Но это увеличит затратность на операции изменения графа и значительно уменьшит допустимое количество вершин и ребер в графе.

5.2 Тестирование программного интерфейса взаимодействия СП

Для тестирования на локальной машине был разработан симулятор СП (п.п. 4.4.5 Разработка симулятора СП). Симулятор был реализован на базе отсортированный ассоциативный контейнера *std::map*, который имеет схожие асимптотики: добавление, удаление, обращение к элементам происходит за $O(\log n)$, где n — размер контейнера.

Была проведена отладка программного интерфейса, были исправлены проблемы с «длинной арифметикой» (арифметические, логические, битовые операции с типом *data_t*).

Для обеспечения качества и надежности были написаны 14 юнит-тестов, которые тестируют корректность выполнения команд процессора обработки структур, а также тестируют корректность выполнения операций «длинной арифметики».

▼ ✓	testDataContainerOperators	0 ms
✓	test_compare_data_containers	0 ms
✓	test_addition_data_containers	0 ms
✓	test_overflow_addition_data_containers	0 ms
✓	test_subtract_data_containers	0 ms
✓	test_overflow_subtract_data_containers	0 ms
✓	test_increment_data_containers	0 ms
✓	test_overflow_increment_data_containers	0 ms
✓	test_decrement_data_containers	0 ms
✓	test_overflow_decrement_data_containers	0 ms
✓	test_shift_right	0 ms
✓	test_shift_left	0 ms
▼ ✓	testSpuApi	0 ms
✓	test_mask	0 ms
✓	test_nsm_command	0 ms
✓	test_prev_command	0 ms

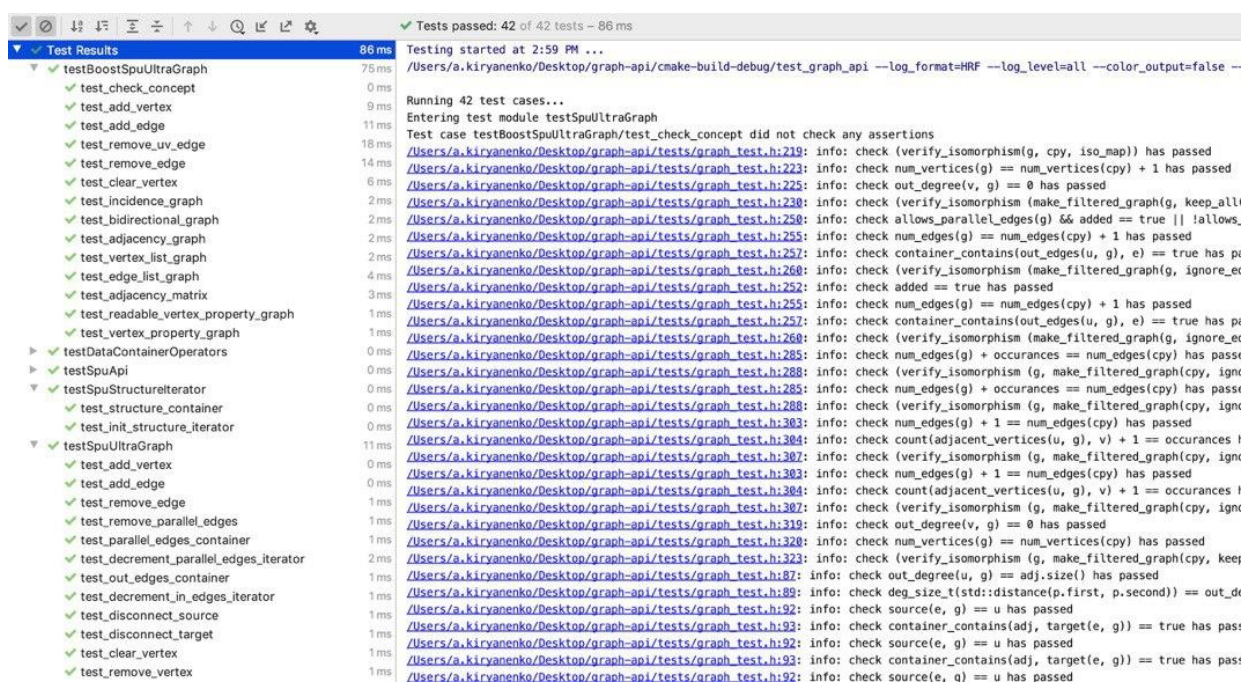
Рисунок 23 – Результаты проведения тестирования программного интерфейса взаимодействия СП

Работа программного интерфейса, помимо тестирования на локальной машине, была протестирована на процессоре Leonhard. В результате проведенного тестирования (Рисунок 23) было выяснено, что программный интерфейс взаимодействия с процессором обработки структур соответствует всем функциональным требованиям.

5.3 Тестирование библиотеки элементов программного интерфейса для обработки графов

5.3.1 Функциональное тестирование

Для обеспечения качества и надежности были написаны 28 юнит-тестов, покрывающие весь функционал библиотеки. Для написания юнит-тестов использовалась библиотека Boost Test Library. Также для тестирования концепций графов использовались тесты, предоставляемые библиотекой Boost Graph Library. В общей сложности вместе с тестами программного интерфейса взаимодействия СП было написано 42 юнит-теста (Рисунок 24).



```
Tests passed: 42 of 42 tests - 86 ms
Testing started at 2:59 PM ...
/Users/a.kiryanenko/Desktop/graph-api/cmake-build-debug/test_graph_api --log-format=HRF --log_level=all --color_output=false --

Running 42 test cases...
Entering test module testSpuUltraGraph
Test case testBoostSpuUltraGraph/test_check_concept did not check any assertions
/Users/a.kiryanenko/Desktop/graph-api/tests/graph_test.h:219: info: check (verify_isomorphism(g, cpy, iso_map)) has passed
/Users/a.kiryanenko/Desktop/graph-api/tests/graph_test.h:223: info: check num_vertices(g) == num_vertices(cpy) + 1 has passed
/Users/a.kiryanenko/Desktop/graph-api/tests/graph_test.h:225: info: check out_degree(v, g) == 0 has passed
/Users/a.kiryanenko/Desktop/graph-api/tests/graph_test.h:230: info: check (verify_isomorphism(make_filtered_graph(g, keep_all), cpy, iso_map)) has passed
/Users/a.kiryanenko/Desktop/graph-api/tests/graph_test.h:250: info: check allows_parallel_edges(g) && added == true || !allows_parallel_edges(g) has passed
/Users/a.kiryanenko/Desktop/graph-api/tests/graph_test.h:255: info: check num_edges(g) == num_edges(cpy) + 1 has passed
/Users/a.kiryanenko/Desktop/graph-api/tests/graph_test.h:257: info: check container_contains(out_edges(u, g), e) == true has passed
/Users/a.kiryanenko/Desktop/graph-api/tests/graph_test.h:260: info: check (verify_isomorphism(make_filtered_graph(g, ignore_edges(e)), cpy, iso_map)) has passed
/Users/a.kiryanenko/Desktop/graph-api/tests/graph_test.h:252: info: check added == true has passed
/Users/a.kiryanenko/Desktop/graph-api/tests/graph_test.h:255: info: check num_edges(g) == num_edges(cpy) + 1 has passed
/Users/a.kiryanenko/Desktop/graph-api/tests/graph_test.h:257: info: check container_contains(out_edges(u, g), e) == true has passed
/Users/a.kiryanenko/Desktop/graph-api/tests/graph_test.h:260: info: check (verify_isomorphism(make_filtered_graph(g, ignore_edges(e)), cpy, iso_map)) has passed
/Users/a.kiryanenko/Desktop/graph-api/tests/graph_test.h:285: info: check num_edges(g) + occurrences == num_edges(cpy) has passed
/Users/a.kiryanenko/Desktop/graph-api/tests/graph_test.h:288: info: check (verify_isomorphism(g, make_filtered_graph(cpy, ignore_edges(e)), iso_map)) has passed
/Users/a.kiryanenko/Desktop/graph-api/tests/graph_test.h:288: info: check (verify_isomorphism(g, make_filtered_graph(cpy, ignore_edges(e)), iso_map)) has passed
/Users/a.kiryanenko/Desktop/graph-api/tests/graph_test.h:303: info: check num_edges(g) + 1 == num_edges(cpy) has passed
/Users/a.kiryanenko/Desktop/graph-api/tests/graph_test.h:304: info: check count(adjacent_vertices(u, g), v) + 1 == occurrences(v) has passed
/Users/a.kiryanenko/Desktop/graph-api/tests/graph_test.h:307: info: check (verify_isomorphism(g, make_filtered_graph(cpy, ignore_edges(e)), iso_map)) has passed
/Users/a.kiryanenko/Desktop/graph-api/tests/graph_test.h:303: info: check num_edges(g) + 1 == num_edges(cpy) has passed
/Users/a.kiryanenko/Desktop/graph-api/tests/graph_test.h:307: info: check count(adjacent_vertices(u, g), v) + 1 == occurrences(v) has passed
/Users/a.kiryanenko/Desktop/graph-api/tests/graph_test.h:307: info: check (verify_isomorphism(g, make_filtered_graph(cpy, ignore_edges(e)), iso_map)) has passed
/Users/a.kiryanenko/Desktop/graph-api/tests/graph_test.h:319: info: check out_degree(v, g) == 0 has passed
/Users/a.kiryanenko/Desktop/graph-api/tests/graph_test.h:320: info: check num_vertices(g) == num_vertices(cpy) has passed
/Users/a.kiryanenko/Desktop/graph-api/tests/graph_test.h:323: info: check (verify_isomorphism(g, make_filtered_graph(cpy, keep_all), cpy, iso_map)) has passed
/Users/a.kiryanenko/Desktop/graph-api/tests/graph_test.h:87: info: check out_degree(u, g) == adj.size() has passed
/Users/a.kiryanenko/Desktop/graph-api/tests/graph_test.h:89: info: check deg_size_t(std::distance(p.first, p.second)) == out_degree(u, g) has passed
/Users/a.kiryanenko/Desktop/graph-api/tests/graph_test.h:92: info: check source(e, g) == u has passed
/Users/a.kiryanenko/Desktop/graph-api/tests/graph_test.h:93: info: check container_contains(adj, target(e, g)) == true has passed
/Users/a.kiryanenko/Desktop/graph-api/tests/graph_test.h:92: info: check source(e, g) == u has passed
/Users/a.kiryanenko/Desktop/graph-api/tests/graph_test.h:93: info: check container_contains(adj, target(e, g)) == true has passed
/Users/a.kiryanenko/Desktop/graph-api/tests/graph_test.h:92: info: check source(e, q) == u has passed
```

Рисунок 24 – Результаты проведения юнит-тестирования

Была проведена оценка покрытия тестами (Рисунок 25), в ходе которой было выяснено, что тесты покрывают 82% исходного кода библиотеки.

Coverage: All in test_graph_api ×	
88% files, 82% lines covered	
Element	Statistics, %
cmake-build-debug	
cmake-build-release	
docs	
doxygen	
examples	
performance_tests	
spu-api	81% files, 68% lines covered
tests	100% files, 98% lines covered
Doxyfile	
exceptions.h	0% lines covered
GraphStructure.h	80% lines covered
LICENSE	
SpuUltraGraph.cpp	76% lines covered
SpuUltraGraph.h	86% lines covered
SpuUltraGraphAdapter.h	100% lines covered
SpuUltraGraphProperty.h	79% lines covered
StructureIterator.h	100% lines covered
UltraGraphConcepts.h	
utils.cpp	45% lines covered
utils.h	

Рисунок 25 – Результаты оценки покрытия тестами

В результате проведенного тестирования было выяснено, что библиотека элементов программного интерфейса для обработки графов соответствует всем функциональным требованиям.

5.3.2 Тестирование производительности

Для проведения тестирования производительности библиотеки элементов программного интерфейса для обработки графов был разработан специальный шаблонный класс *GraphPerformanceTest* (Рисунок 26). Класс *GraphPerformanceTest* отвечает за проведение теста производительности в зависимости от количества ребер и вершин в графе. Он автоматически заполняет граф, измеряет время выполнения переданного теста и сохраняет

результаты в CSV файл. Для более точного результата на каждой итерации проводится 10 повторных тестов и находится среднее время выполнения.

GraphPerformanceTest<G>

```
+ void (*test_func)(G&) = nullptr;
+ string results_file = "results.csv";
+ size_t avg_iterations_cnt = 10;
+ bool should_fill = true;
+ bool is_mutable_test = true;
+ pair<edge_t, bool> (*add_edge_func)(vertex_t, vertex_t, G&) = nullptr;
+ size_t start_vertices_cnt = 500;
+ size_t inc_vertices_value = 500;
+ size_t end_vertices_cnt = 100000;
+ size_t edges_per_vertex = 3;
+ GraphPerformanceTest(void (*test_func)(G&), string results_file);
+ void start();
```

Рисунок 26 – Класс *GraphPerformanceTest*

При проведении тестирования производительности использовался двумерный граф решетки, показанный на рисунке 27. Вначале, для проведения тестирования строился случайный граф. Однако, такой граф неоднороден, и это может в какой-то момент повлиять на результаты тестирования. Поэтому было принято решение использовать граф решетки, потому что он имеет четкую структуру и обладает свойством однородности.

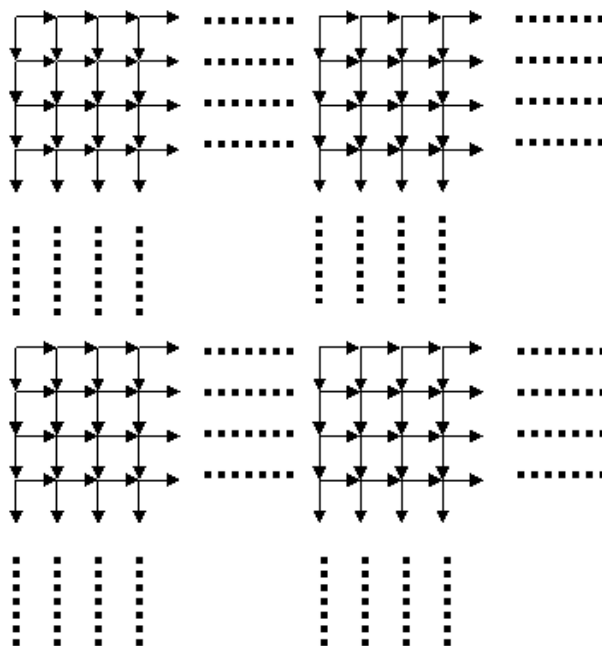


Рисунок 27 – Граф решетки

Далее на рисунках 28 – 31 показаны графики производительности в зависимости от количества ребер и вершин в графе для соответствующих графовых алгоритмов. На графиках сравнивается производительность разработанного графа *SpuUltraGraph*, BGL графа списка смежности *boost::adjacency_list* и BGL графа матрицы смежности *boost::adjacency_matrix*. Тестирование происходило на локальной машине, характеристики которой приведены ниже в таблице 36.

Таблица 36 – Технические характеристики ЭВМ

Параметр	Значение
Процессор	Intel Core i5-6600
Тактовая частота процессора	3,3 ГГц
Объем ОЗУ	8 Гб
Операционная система	Ubuntu 18.04

Класс *GraphPerformanceTest* позволяет гибко настраивать параметры проводимого тестирования производительности. В таблице 37 приведены параметры, которые были выставлены при проведении тестов производительности. Для BGL графа матрицы смежности *boost::adjacency_matrix* из-за нехватки объема оперативной памяти параметр «Конечное количество вершин» был выставлен в значение 20000 и соответственно параметр «Конечное количество ребер» – в значение 40000.

Таблица 37 – Параметры тестирования производительности

Параметр	Значение
Вид графа	Граф решетки
Начальное количество вершин	1000
Инкремент количества вершин	1000
Конечное количество вершин	100000
Начальное количество ребер	2000
Инкремент количества ребер	2000
Конечное количество ребер	200000
Количество тестов производительности	99
Количество повторных тестов для подсчета среднего	10
Общее количество тестов	990

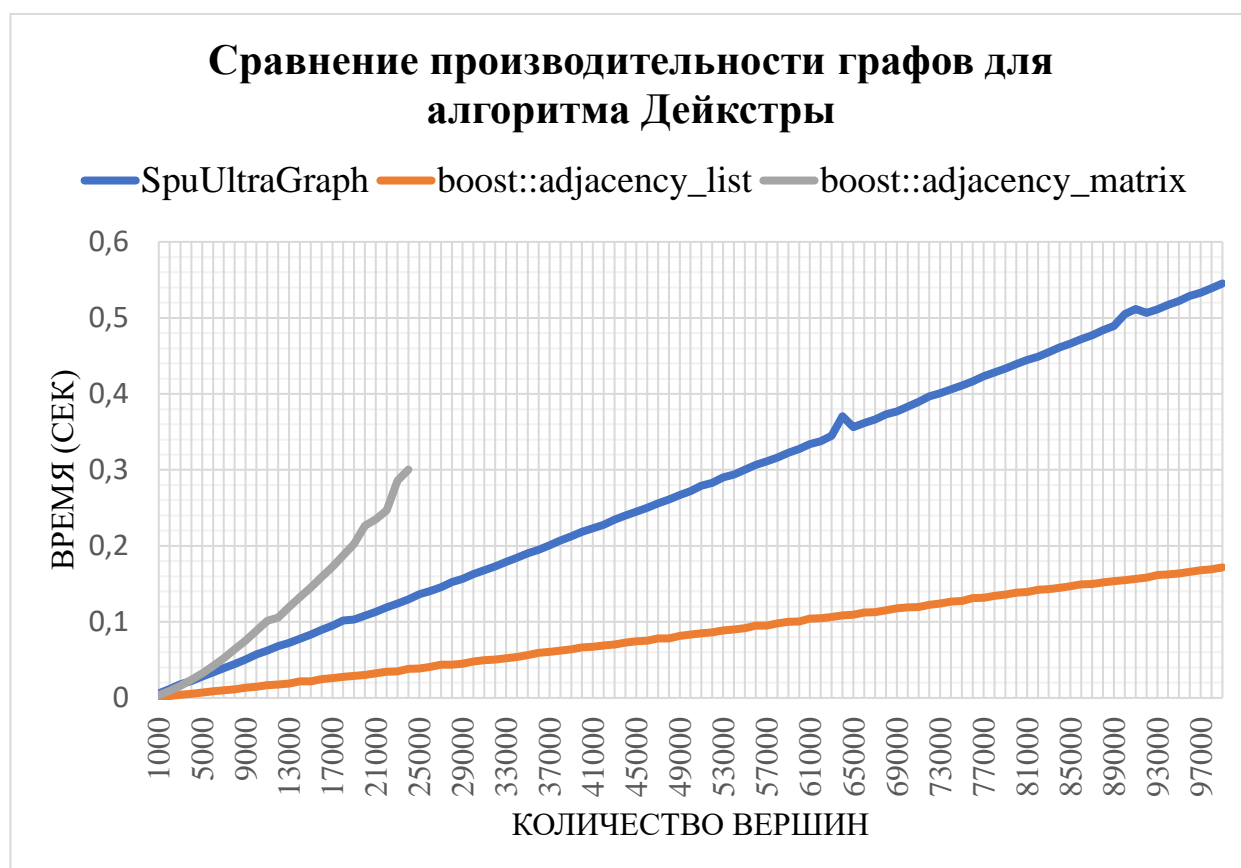


Рисунок 28 – График сравнения производительности графов для алгоритма Дейкстры

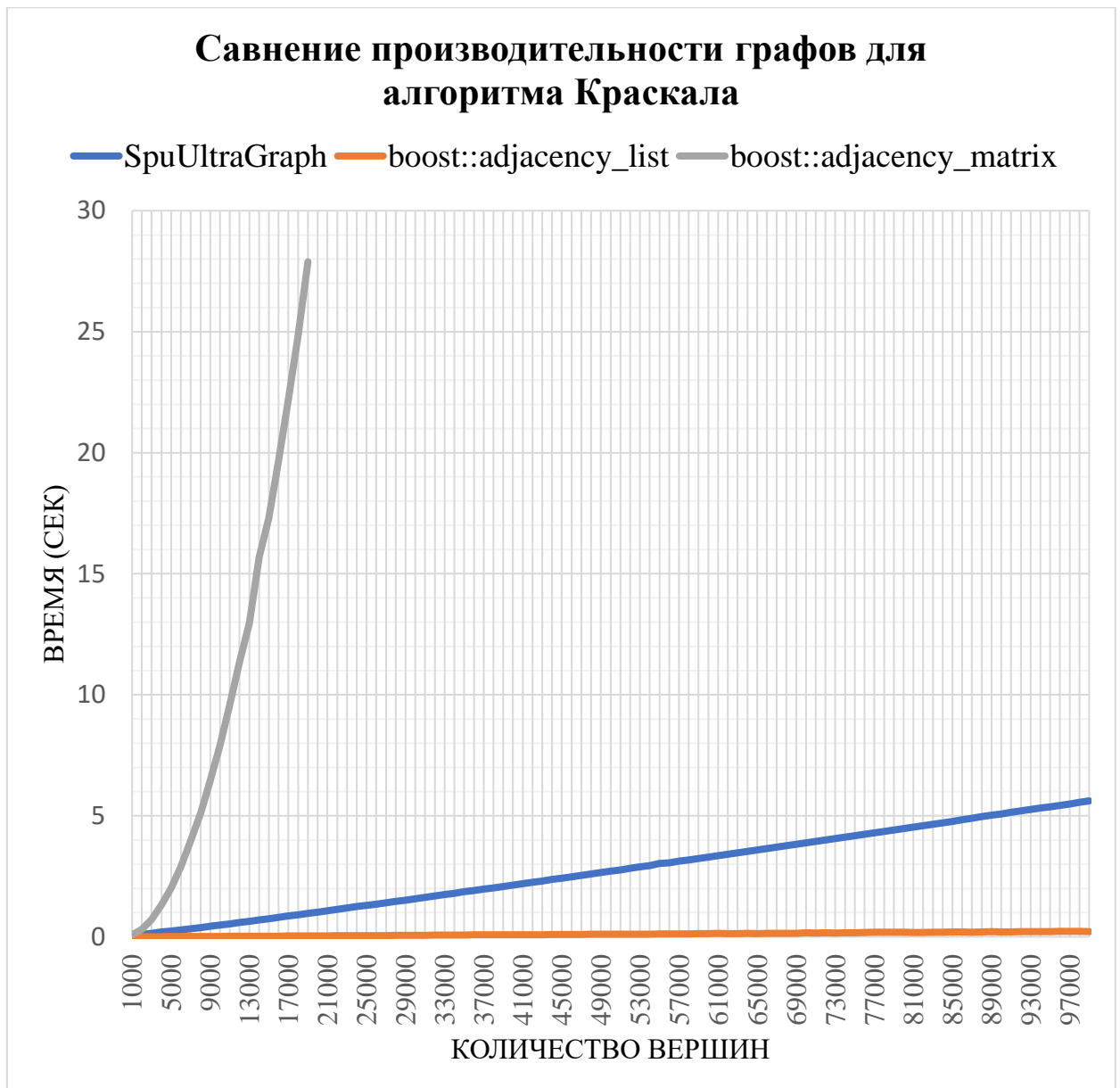


Рисунок 29 – График сравнения производительности графов для алгоритма Краскала

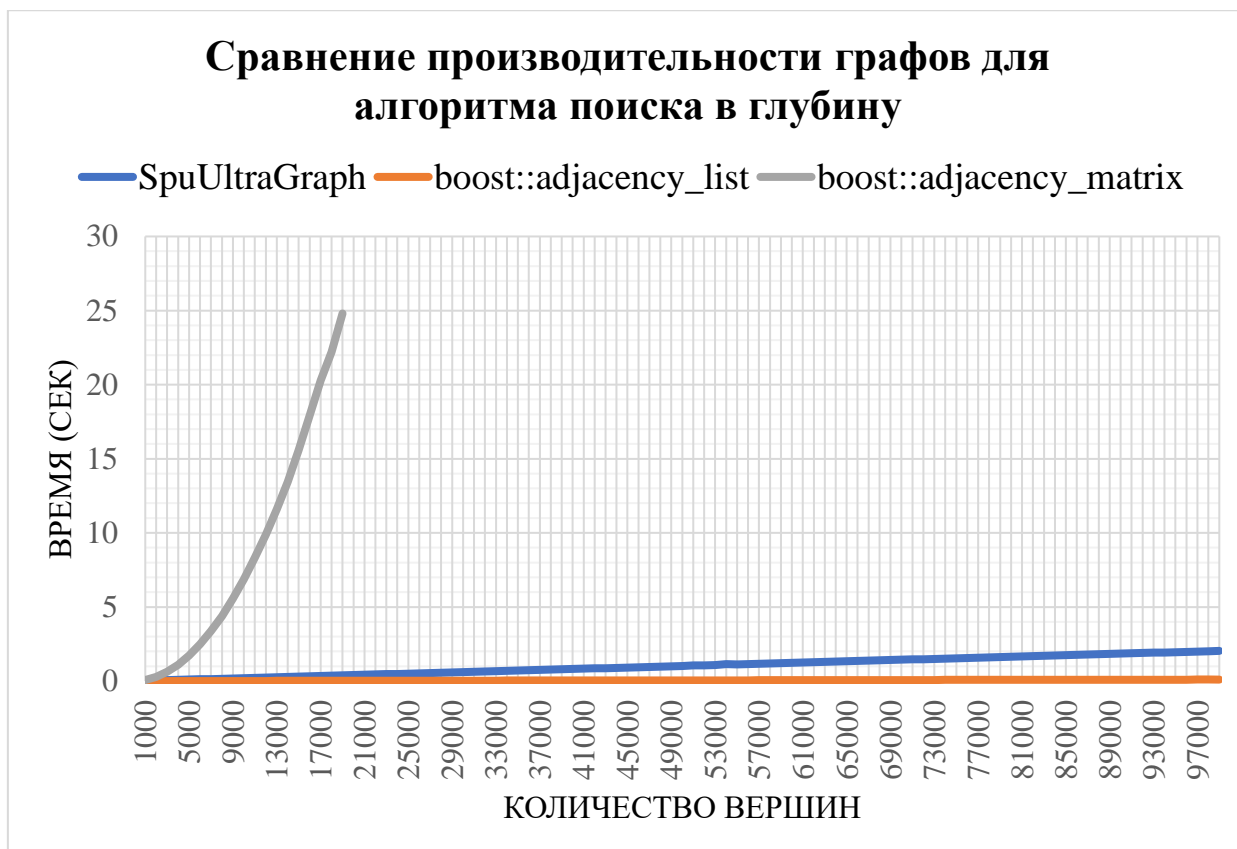


Рисунок 30 – График сравнения производительности графов для алгоритма поиска в глубину

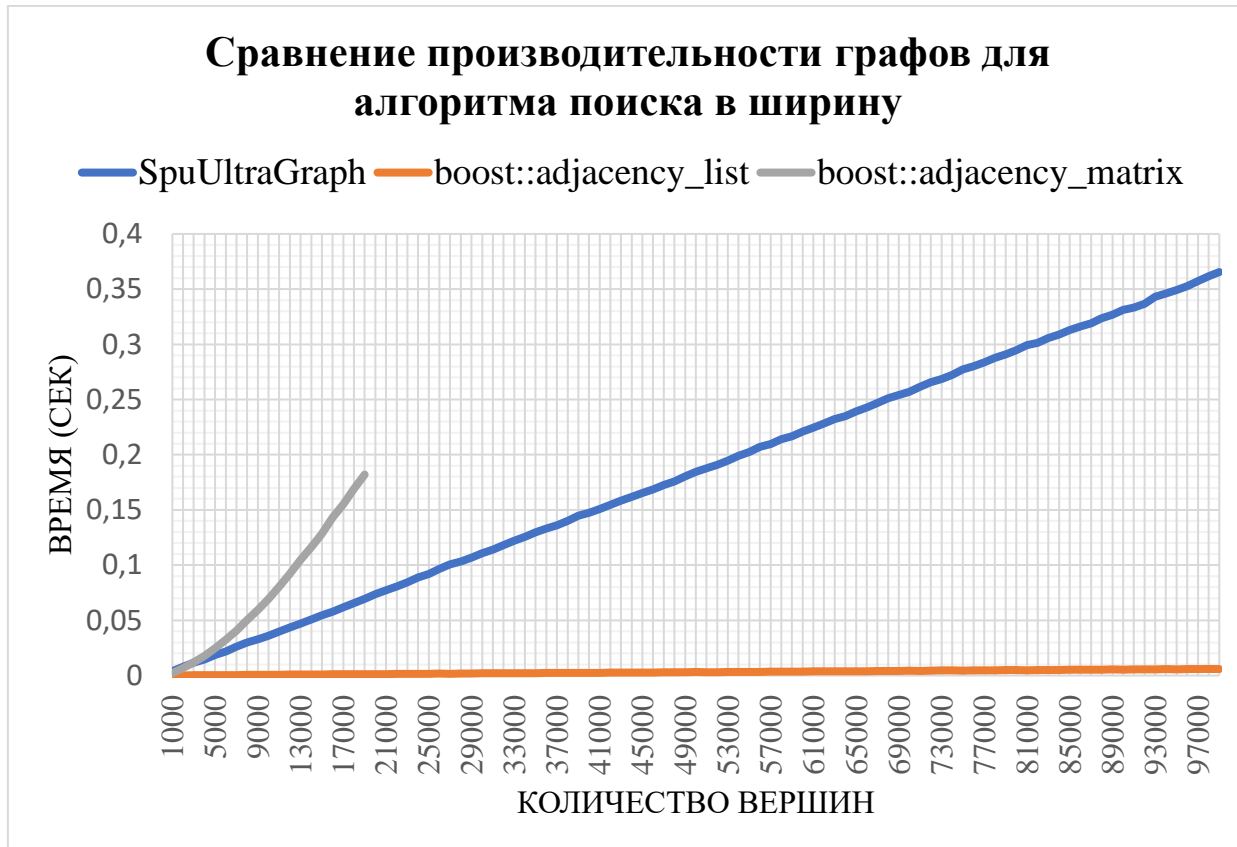


Рисунок 31 – График сравнения производительности графов для алгоритма поиска в ширину

Как можно видеть на графиках выше производительность разработанного графа *SpuUltraGraph* значительно превышает стандартного BGL графа матрицы смежности *boost::adjacency_matrix*. Однако все же граф *SpuUltraGraph* уступает по производительности BGL графу списка смежности *boost::adjacency_list*. Например, для алгоритма Дейкстры для нахождения кратчайших расстояний до вершин производительность графа *SpuUltraGraph* уступает примерно в 3 раза производительности стандартному BGL графу списка смежности *boost::adjacency_list*. Нужно понимать, что граф *SpuUltraGraph* предназначен для работы с ультраграфами и в этом случае он взаимодействует с двумя структурами СП. Из этого следует, что графу *SpuUltraGraph* приходится делать в 2 раза больше обращений к СП. Тогда как BGL граф список смежности *boost::adjacency_list* предназначен для работы с мультиграфами. Стоит отметить, что при работе с процессором обработки структур производительность графа *SpuUltraGraph* должна значительно увеличиться.

Очевидно, что при работе с любыми типами графов для операций добавления, удаления и поиска вершин и ребер графики производительности графа *SpuUltraGraph* будет иметь вид схожий с видом графиков производительности команд СП (Рисунки 3 – 5), т.к. происходят те же самые команды добавления, удаления и поиска записи в структуре СП.

Таким образом, в ходе тестирования производительности библиотеки элементов программного интерфейса для обработки графов было выяснено, что в дальнейшем необходимо произвести ряд оптимизаций. Программный интерфейс взаимодействия СП нуждается в оптимизации обработки данных. В библиотеке обработки графов необходимо произвести оптимизации в операциях взаимодействия с графами, необходимо уменьшить количество обращений к процессору обработки структур.

В целом библиотека элементов программного интерфейса для обработки графов показала достаточно хороший результат в тестах производительности.

5.4 Документирование исходного кода библиотеки

Для автоматизации документирования исходного кода библиотеки элементов программного интерфейса для обработки графов использовался система Doxygen.

Система Doxygen позволяет генерировать на основе исходного кода, содержащего комментарии специального вида, красивую и удобную документацию, содержащую в себе ссылки, диаграммы классов, вызовов и т.п. в различных форматах: HTML, LaTeX, и другие. [11]

Таким образом, все классы и методы были задокументированы и была сформирована HTML документация библиотеки (Рисунок 32).

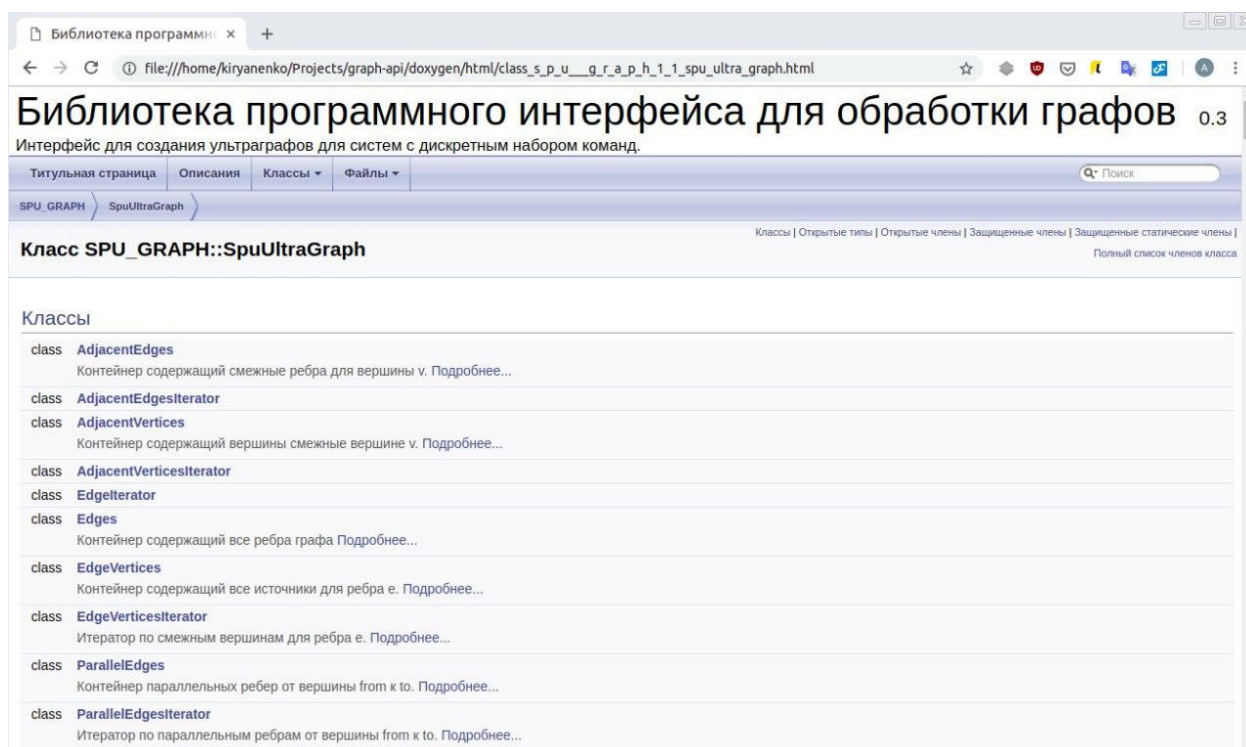


Рисунок 32 – Страница с документацией по библиотеке

ЗАКЛЮЧЕНИЕ

В результате выполнения настоящей работы был проведен анализ и исследование программной модели процессора с набором команд дискретной математики. А также был проведен анализ технологий хранения и обработки графов. Были разработаны графовые модели для систем с дискретным набором команд.

По итогам работы разработан весь функционал библиотеки элементов программного интерфейса для обработки графов, который предназначен для решения графовых задач для систем с дискретным набором команд. В библиотеке реализован интерфейс для создания ультраграфов для таких систем. Библиотека реализует интерфейс графа, предоставляемый библиотекой Boost Graph Library, который предоставляет набор алгоритмов для решения графовых задач. Также разработаны концепции для ультраграфов, которые были реализованы в библиотеке.

Для получения доступа к вычислительным ресурсам СП используется программный интерфейс СП, используемый для включения в разрабатываемое программное обеспечение. Был доработан и отлажен существующий программный интерфейс для работы с библиотекой для обработки графов.

Для проведения тестирования на локальной машине разработан симулятор для программного интерфейса процессора обработки структур. Для обеспечения качества и надежности написаны юнит-тесты. Также было проведено тестирование производительности библиотеки. По результатам тестирования выяснено, что библиотека элементов программного интерфейса для обработки графов соответствует всем предъявляемым требованиям.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Попов А.Ю. Исследование производительности процессора обработки структур в системе с многими потоками команд и одним потоком данных / Попов А.Ю. // Инженерный журнал: наука и инновации. – 2013. – № 11. – URL: <http://engjournal.ru/catalog/it/hidden/1048.html> (дата обращения 01.02.2020).
2. Принципы организации гетерогенной вычислительной системы с набором команд дискретной математики. / Попов А.Ю. – Москва: Издательство МГТУ им. Н.Э. Баумана, 2020. – 28 с.
3. Попов А.Ю. Руководство системного программиста. Микропроцессор Leonhard x64. / Попов А.Ю. – Москва: Издательство МГТУ им. Н.Э. Баумана, 2019. – 29 с.
4. Дж. Сик, Л. Ли C++ Boost Graph Library. Библиотека программиста / Дж. Сик, Л. Ли, Э. Ламсдэйн; пер. с англ. Сузи Р. – СПб.: Питер, 2006. – 304 с.
5. The Boost Graph Library [Электронный ресурс] – URL: https://www.boost.org/doc/libs/1_72_0/libs/graph/doc/index.html (дата обращения 14.05.2020)
6. LEDA for C++ [Электронный ресурс] – URL: <https://www.algorithmic-solutions.com/index.php/products/leda-for-c> (дата обращения 14.05.2020)
7. Gabriel Valiente. Algorithms on Trees and Graphs / Gabriel Valiente. – Люксембург: Springer, 2002 – 489 с.
8. Neo4j [Электронный ресурс] – URL: <https://evilinside.ru/neo4j/> (дата обращения 14.05.2020)
9. Овчинников В.А. Графы в задачах анализа и синтеза структур сложных систем. / Овчинников В.А. – Москва: Издательство МГТУ им. Н.Э. Баумана, 2014. – 423 с.

10. V-модель [Электронный ресурс]. – URL: <https://qalight.com.ua/baza-znaniy/v-model-v-model/> (дата обращения 14.05.2020).
11. Документируем код эффективно при помощи Doxygen [Электронный ресурс] – URL: <https://habr.com/ru/post/252101/> (дата обращения 14.05.2020)