Санкт-Петербургский государственный университет

***Черников Антон Александрович***

Выпускная квалификационная работа

# Реализация эффективного алгоритма проверки графовых функциональных зависимостей в платформе Desbordante

Уровень образования: бакалавриат

Направление *02.03.03 «Математическое обеспечение и администрирование информационных систем»*

Основная образовательная программа *СВ.5006.2019 «Математическое обеспечение и администрирование информационных систем»*

Научный руководитель:
доцент кафедры системного программирования, к.т.н, Ю. В. Литвинов

Консультант:
ассистент кафедры информационно-аналитических систем, Г. А. Чернышев

Рецензент:
младший разработчик ООО “УК ЮД-КАПИТАЛ” М. А. Струтовский

Санкт-Петербург
2023

Saint Petersburg State University

***Anton Chernikov***

Bachelor's Thesis

# Implementation of efficient graph functional dependency validation algorithm in the Desbordante platform

Education level: bachelor

Speciality *02.03.03 "Software and Administration of Information Systems"*

Programme *CB.5006.2019 "Software and Administration of Information Systems"*

Scientific supervisor:
C.Sc., System Programming chair docent Y.V. Litvinov

Consultant:
Information and Analytical Systems chair assistant G.A. Chernishev

Reviewer:
junior developer at Unidata M.A. Strutovskii

Saint Petersburg
2023

# Оглавление

# Введение

Профилирование данных — это процесс извлечения дополнительной информации о данных. В первом приближении под дополнительной информацией может пониматься такая метаинформация как автор, дата создания, размер занимаемой памяти. Однако помимо этого данные могут содержать неочевидные зависимости и закономерности, сокрытые в них. О выявлении такого рода информации из данных и будет идти речь в данной работе.

Профилирование является частой задачей у людей, работающих с массивами данных. Выявленные зависимости могут представлять ценность для довольно широкого круга людей. Некоторые примеры приведены ниже:

- Специалисты по машинному обучению.

  Перед обучением алгоритма, решающего некую задачу, данные для обучения необходимо подготовить. Одно из важных действий — определение параметров, которые являются производными от других параметров. Такие параметры стоит исключать из обучающей выборки, ведь они не несут новой полезной информации об объекте. Их наличие только увеличит количество потребляемой памяти, время обучения, а также повысит шансы переобучения.

- Люди, работающие с финансовыми данными.

  Проверка или получение гипотезы из финансовых данных могут быть очень полезны с точки зрения бизнеса. Например, найденная зависимость “если покупатель приобретает наушники, то он приобретает и микрофон” может указать на то, что целесообразно размещать указанные товары физически ближе друг к другу в магазинах техники, или же продавать наушники со встроенным микрофоном.

- Учёные, работающие с экспериментальными данными.

> Автоматическое нахождение зависимости в данных может натолкнуть на формулирование гипотезы, открытие нового закона или по крайней мере указать направление для дальнейшего исследования.

Функциональные зависимости хорошо изучены для таблиц. Их описание можно найти в любом учебнике по базам данных. Необходимость в функциональных зависимостях также очевидна в графах, довольно распространённом способе хранения данных. Они помогают устанавливать несоответствия в базах знаний, находить ошибки, определять спам и управлять блогами в социальных сетях.

Desbordante[1] — это инструмент для профилирования данных, разрабатываемый группой студентов под руководством Г. А. Чернышева. Проект содержит множество алгоритмов, которые способны обнаруживать различные закономерности в данных. Весь проект имеет открытый исходный код и высокую производительность, так как реализован на C++. Однако все имеющиеся инструменты позволяют работать только с данными, представленными в виде таблиц. Данная работа направлена на то, чтобы расширить функциональность Desbordante — добавить алгоритм выявления функциональных зависимостей в данных, представленных в виде графов.

Обобщение обычных функциональных зависимостей на графы приведено в статье [4]. Кроме этого авторы статьи описывают и оценивают алгоритм проверки выполнения набора графовых зависимостей на больших реальных графах, представленный в двух вариантах: для входящего монолитного графа и распределённого графа, разбитого на части, хранящиеся на нескольких носителях. Далее вторая версия алгоритма рассматриваться не будет, а первая будет принята как базовая версия алгоритма для данной работы.

[1] https://github.com/Mstrutov/Desbordante (дата обращения 18.04.2023)

# 1. Постановка задачи

Целью данной работы является создание эффективного алгоритма валидации графовых функциональных зависимостей.

Для достижения этой цели были поставлены следующие задачи:

- Выявить наиболее ресурсозатратную часть наивного алгоритма валидации.
- Выполнить обзор существующих подходов к оптимизации выявленной части и выбрать наиболее релевантный.
- Спроектировать быстрый алгоритм валидации на основе выбранного подхода и реализовать его.
- Произвести сравнение базового и быстрого алгоритмов валидации.

# 2. Обзор

## 2.1. Графовые зависимости

**Определение 1** (Функциональная зависимость)**.** *Отношение $R$ удовлетворяет функциональной зависимости $X \to Y$ (где $X, Y \subset R$) тогда и только тогда, когда для любых кортежей $t_1,\ t_2 \in R$ выполняется: если $t_1[X] = t_2[X]$, то $t_1[Y] = t_2[Y]$.*

Таблица 1: Характеристики мобильных устройств

| name | OS | memory | bluetooth_codec |
|---|---|---|---|
| Honor 20 | Android | 128 GB | SBC |
| iPhone 14 | iOS | 128 GB | AAC |
| Redmi Note 8t | Android | 64 GB | SBC |
| Realme 8 | Android | 128 GB | SBC |

Пусть, отношение представлено в виде Таблицы 1. Здесь видно, что фукциональная зависимость $OS \to bluetooth\_codec$ выполняется, так как все строчки, имеющие значение $Android$ в столбце $OS$ содержат одинаковое значение в столбце $bluetooth\_codec$ ($SBC$). В это же время зависимость $OS \to memory$ не выполняется, потому что третья строчка в столбце $memory$ имеет значение, отличное от остальных.

Функциональные зависимости могут быть обобщены на графы. Одно из таких обобщений предлагают авторы статьи [4], на котором и основана данная работа. В этой статье определяются и исследуются графовые зависимости, формулируется задача проверки (validation) выполнения зависимостей на графе, а также задачи выполнимости (satisfiability) и импликации (implication) набора зависимостей.

Задачи выполнимости и импликации были более подробно изучены в статье [3], в которой предложены эффективные алгоритмы работы под каждую из них.

Прежде чем рассматривать графовые зависимости, нужно формально определить данные, на которых они определены — графы.

**Определение 2** (Граф)**.** *Граф — это структура данных, состоящая из четвёрки* $(V, E, L, A)$*, где* $V$ *— множество вершин;* $E \subseteq V \times V$ *— множество рёбер;* $L : V \cup E \to \Sigma$ *— сюръекция, где* $\Sigma$ *— множество меток (алфавит),* $A$ *— функция, которая сопоставляет каждой вершине список её атрибутов.*

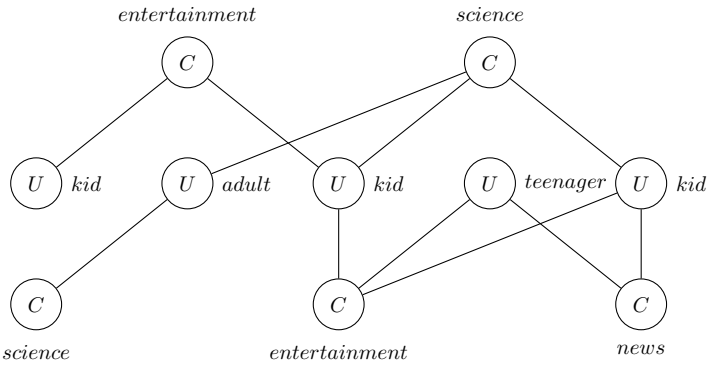
Список атрибутов содержит названия атрибутов и соответствующие этим атрибутам значения. Пусть, $A(u) = (f_1 = c_1, f_2 = c_2, \ldots, f_m = c_m),\ u \in V$, здесь вершина $u$ имеет атрибуты $f_i\ i = 1, 2, \ldots, m$, а число $m$ зависит от конкретной вершины, то есть, у каждой вершины может быть свой набор атрибутов (обычно набор атрибутов зависит от метки вершины). $c_i$ — значение, которое принимает атрибут $f_i$, обозначение: $u.f_i = c_i$.

В данной работе графы рассматриваются как неориентированные, то есть, $(u, v), (v, u) \in E$ представляют собой один и тот же объект.

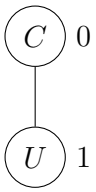
**Определение 3** (Графическая функциональная зависимость)**.** *GFD (Graph Functional Dependency) — это конструкция* $P[X \to Y]$*, где* $P$ *— паттерн, а* $X$ *и* $Y$ *— множества литералов.*

В этом определении под паттерном понимается граф, вершины которого однозначно проиндексированы от 0 до $|V| - 1$ для получения доступа к ним, а под литералом — выражение, имеющее вид $i.f = c$ (константный литерал), где $i$ — индекс вершины паттерна, $f$ — атрибут соответствующей вершины, $c$ — константа (значение), или $i.f_i = j.f_j$ (переменный литерал), где $i, j$ — индексы вершин паттерна, $f_i, f_j$ — атрибуты соответствующих вершин.

На Рис. 1 представлен пример графовой зависимости и графа. Вершины графа имеют метки $C$ или $U$. В зависимости от метки вершина имеет свой собственный набор атрибутов. В данном примере все вершины имеют одноэлементный список атрибутов. У вершин с меткой $C$ он состоит из элемента *topic*, а у вершин с меткой $U$ — *age_group*. На Рис. 1а конкретные значения этих атрибутов указаны рядом с вершинами.




(a) Связь каналов $C$ (Channel) с пользователями $U$ (User). Атрибуты у вершин с меткой $C$ — $\{topic\}$, с меткой $U$ — $\{age\_group\}$.




(b) Графовая зависимость $\{0.topic = entertainment \rightarrow 1.age\_group = kid\}$.

Рис. 1: Пример графовой функциональной зависимости.

Чтобы проверить, выполняется ли GFD, необходимо найти все подграфы графа, изоморфные паттерну, и на каждом вложении проверить выполнимость зависимости литералов, то есть, выполнено ли: если все литералы в левой части выполняются, то все литералы в правой части так же выполняются. Если есть хотя бы одно вложение, на котором зависимость не выполняется, то графовая зависимость не выполняется на всём графе. Если не нашлось ни одного вложения паттерна, то такая зависимость считается тривиально выполненной.

## 2.2. Наивный алгоритм

Наивный алгоритм проверки графовой зависимости можно разделить на две стадии:

1. Поиск всех вложений паттерна в основной граф.

2. Проверка зависимостей для каждого из найденных вложений.

Поиск вложения — это ничто иное, как поиск изоморфного подграфа, широко распространённая NP-полная задача. Если изоморфный подграф уже найден, остаётся лишь проверить необходимые зависимости за линейное время от количества литералов. Очевидно, что поиск подграфа гораздо ресурсозатратнее, чем проверка выполнимости литералов. Для практического подтверждения этих рассуждений были про-

изведены замеры работы наивного алгоритма по перечисленным стадиям. В результате оказалось, что около 99% всего времени занимала первая часть. В связи с этим для улучшения производительности в первую очередь необходимо оптимизировать именно поиск подграфа.

При обсуждении задачи поиска подграфа невозможно не упомянуть алгоритм Ульмана [5], который является классическим алгоритмом поиска точных изоморфизмов. Однако этот алгоритм в настоящее время считается устаревшим, вместо него используют другие более эффективные алгоритмы. Одним из таких является VF2. Он производительнее, чем алгоритм Ульмана [8], и реализован в качестве стандартного алгоритма поиска подграфа в библиотеке boost. Именно этот алгоритм был использован в данной работе в качестве наивного.

## 2.3. Базовый алгоритм

В рассматриваемой статье предлагается подход, основанный на распараллеливании поиска подграфов. Граф, представляющий базу данных, делится на локальные участки, на которых необходимо проверить ту или иную графовую зависимость, далее генерируются сообщения, включающие в себя функциональную зависимость и часть графа, на которой предположительно может оказаться соответствие паттерна зависимости. Центральный процессор, так называемый координатор, распределяет эти сообщения между процессорами. Каждое сообщение имеет свой вес, равный размеру локального участка графа (сумме количества вершин и количества рёбер), содержавшегося в нём. В результате распределения, каждый процессор получает множество сообщений, сумма весов которых примерно одинакова для каждого процессора.

Под процессором можно подразумевать как один поток на одной машине, так и отдельную машину в кластере высокопроизводительных устройств. Описываемый алгоритм ориентирован именно на второй случай. Однако арендовать распределённую систему для своих нужд может быть дорого. Идея этой работы — открыть проверку графовых функциональных зависимостей для более широкой аудитории,

предоставив возможность запускать алгоритм проверки на персональном компьютере среднестатистического пользователя. Первоначальные эксперименты показали, что существующий алгоритм может быть не оптимален для таких целей. Поэтому возникла идея разработать свой алгоритм проверки графовых функциональных зависимостей, подходящий для запуска на одном устройстве.

Ввиду вышеперечисленных причин была реализована многопоточная версия базового алгоритма. Процесс балансировки сообщений основан на статье [1] и тоже был реализован вместе с базовым алгоритмом по причине отсутствия открытого кода.

Стоит обратить внимание, что этот алгоритм довольно ресурсозатратен по памяти, так как балансировка сообщений возможна только в случае хранения всех сообщений в один момент времени, в то время как самих сообщений может быть достаточно много.

## 2.4. Обзор алгоритмов поиска подграфа

Для анализа существующих подходов к оптимизации поиска подграфа были отобраны и изучены несколько статей, описывающие алгоритм, который использует эвристики для повышения производительности при поиске изоморфных подграфов. Статьи отбирались в соответствии со следующими критериями:

- Статья должна описывать алгоритм, решающий задачу поиска изоморфного подграфа.
- Предлагаемый алгоритм должен не только давать ответ на вопрос, существует ли вложение графа-запроса в основный граф, но и возвращать все точные изоморфизмы.
- Алгоритм должен принимать на вход только один граф-запрос и один основной граф.

Первый критерий необходим ввиду очевидных причин. Точные изоморфизмы, описанные во втором критерии, нужны для дальнейшей ра-

боты, чтобы проверить на каждом вложении выполнимость зависимости. Третий критерий позволяет не рассматривать алгоритмы, принимающие несколько графов-запросов или несколько основных графов, так как такие алгоритмы направлены не непосредственно на поиск подграфа, а скорее на уменьшение количества графов-запросов для проверки, которые точно не содержатся в основном графе, или основных графов, в которых гарантированно нет изоморфизмов графа-запроса. В худшем случае после выполнения этих алгоритмов данные не поменяются, следовательно, алгоритмы, удовлетворяющие предложенным критериям, будут давать выигрыш.

Таблица 2: Характеристики изученных алгоритмов.

| Название | Дата | Авторы | Конференция | Результат |
|---|---|---|---|---|
| SPath | 2010 | Peixiang Zhao<br>Jiawei Han | VLDB | кандидаты на изоморфизм |
| BR-Index | 2011 | Jiong Yang<br>Wei Jin | SSDBM | кандидаты на изоморфизм |
| CPI | 2016 | Fei Bi<br>Lijun Chang<br>Xuemin Lin<br>Lu Qin<br>Wenjie Zhang | SIGMOD | точные изоморфизмы |

В результате были отобраны и изучены три статьи, информация о которых содержится в Таблице 2. Рассмотрим алгоритмы более подробно.

Начнём с алгоритма SPath [7]. Основная идея заключается в использовании двух эвристик:

- Минимизировать количество попарных сравнений (количество вершин в запросе).
- Минимизировать количество кандидатов для каждой вершины.

Для достижения этих целей используется индексация основного графа с помощью путей. Первый пункт достигается следующим образом: граф-запрос представляется в виде совокупности небольших путей, после чего в в основном графе находятся комбинации этих путей. Тем

самым алгоритм достаточно эффективно убирает гарантированно ложноположительные вложения. Второй пункт достигается более строгим отбором кандидатов, то есть, наложением дополнительных ограничений помимо равенства меток и сравнения степеней вершин.

Следующий алгоритм — BR-Index (Bounded Region Index) [6]. Алгоритм требует провести некоторую предобработку основного графа, которая не зависит от графа-запроса. Граф разбивается на перекрывающие друг друга области. После чего случайным образом генерируются маленькие графы с небольшим количеством вершин и радиусом 1, так называемые индексы. Для каждой области создаётся её соответствие со списком индексов. Для этого процесса используется наивный алгоритм для поиска подграфа, однако это происходит очень быстро, так как индексы представляют собой небольшие графы, которые в свою очередь ищутся в небольших областях основного графа. Следующим этапом происходит поиск индексов в графе-запросе. В основном графе, в зависимости от их взаимного расположения в графе-запросе, находятся области, содержащие эти индексы, с этим же самым расположением друг относительно друга, и происходит поиск вложения в найденном маленьком участке графа.

Оставшийся алгоритм носит название CPI [2]. Алгоритм генерирует специальную одноимённую структуру данных, помогающую перебирать все частичные вложения графа-запроса. Работа разделена на несколько этапов.

Сначала производится декомпозиция графа-запроса на ядро и лес. Ядро — это минимальный связный подграф, содержащий все вершины степени больше единицы. Оставшиеся вершины, не принадлежащие ядру, составляют множество деревьев, называемое лесом.

На следующем этапе происходит выделение из графа-запроса дерева путём обхода в ширину из корневой вершины. Корневая вершина обязательно должна принадлежать ядру, а её выбор обусловлен наименьшим количеством кандидатов на изоморфизм среди всех вершин ядра.

Далее используются два подалгоритма для построения вспомога-

тельной структуры CPI (Compact Path-Index). Эта структура имеет такую же топологию, как выделенное дерево графа-запроса. Однако узлы этого графа представляют собой списки кандидатов на изоморфизм для соответствующей вершины в графе-запросе. Рёбра между узлами носят условный характер, так как представляют собой набор рёбер, которые существуют в основном графе, между кандидатами. Первый подалоритм построения CPI можно назвать нисходящим, а второй — восходящим. Сначала строится поверхностная версия CPI, обходя дерево запроса в ширину, потом производится обход в обратном порядке, чтобы удалить тупиковые вложения.

Теперь можно использовать построенную структуру для перебора всех изоморфизмов дерева запроса, а потом убирать из них те, которые не содержат рёбер, принадлежащим графу-запросу, но не принадлежащим дереву запроса. Однако чтобы ускорить этот процесс была проведена оптимизация: по вышеописанному алгоритму находятся лишь вложения ядра, а только потом добавляется лес, так как он гарантированно совпадёт. Поэтому нет необходимости перебирать полные изоморфизмы, будет достаточно только ядра.

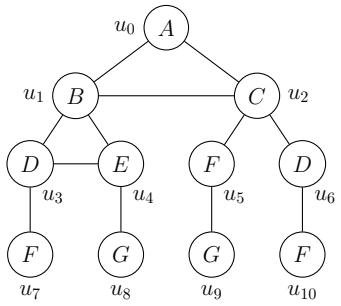



Рис. 2: Граф-запрос.

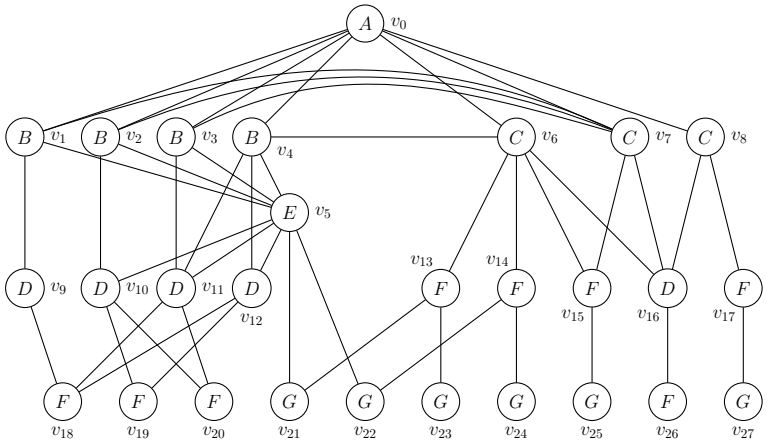



Рис. 3: Основной граф.

Рассмотрим пример, состоящий из графа-запроса, показанного на Рис. 2 и основного графа, изображённого на Рис. 3. Этот пример взят из статьи [2]. Декомпозиция графа-запроса показана на Рис. 4. Здесь серым выделены вершины, принадлежащие ядру. Также жирными показаны рёбра, которые принадлежат дереву запроса, которое получилось

путём обхода графа в ширину, начиная из корневой вершины, принадлежащей ядру. В данном случае эта вершина — $u_0$. Чтобы получить полное дерево запроса, нужно взять жирные рёбра из ядра и все рёбра из каждого дерева, принадлежащего лесу.

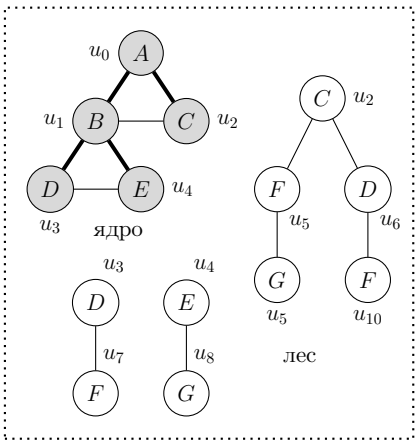



Рис. 4: Декомпозиция запроса.

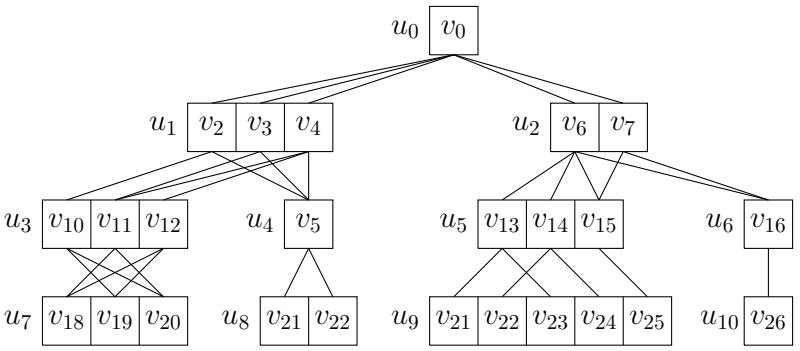



Рис. 5: CPI.

На Рис. 5 показана структура CPI, которую строит рассматриваемый алгоритм. Как можно заметить, эта структура содержит все изоморфизмы полного дерева запроса. Однако механизм возвращения следующего изоморфизма всего запроса учитывает рёбра запроса, не принадлежащие дереву. Сначала происходит поиск всех вложений ядра, например, при изоморфизме $\{u_0 \to v_0, u_1 \to v_2, u_2 \to v_6, u_3 \to v_{10}, u_4 \to v_5\}$ дальнейшее вложение не будет рассматриваться, так как ребра $(v_2, v_6)$ не существует. В таком случае алгоритм переходит к следующему кандидату на вложение ядра.

# 3. Предлагаемое решение

В результате, среди этих алгоритмов был выбран CPI по нескольким причинам. Во-первых, статья с этим алгоритмом является новейшей. Во-вторых, она была опубликована в сильной конференции. В-третьих, в качестве результата она возвращает все точные изоморфизмы подграфа, а не убирает ложноположительные.

На основании выбранного алгоритма был спроектирован и реализован алгоритм проверки функциональной зависимости на данном графе. Наряду с ним были также реализованы наивный и базовый алгоритмы. Первый был необходим для сравнения производительности. Открытый исходный код не был предоставлен в статье [4], поэтому, чтобы произвести анализ предлагаемого алгоритма, и был реализован базовый алгоритм.

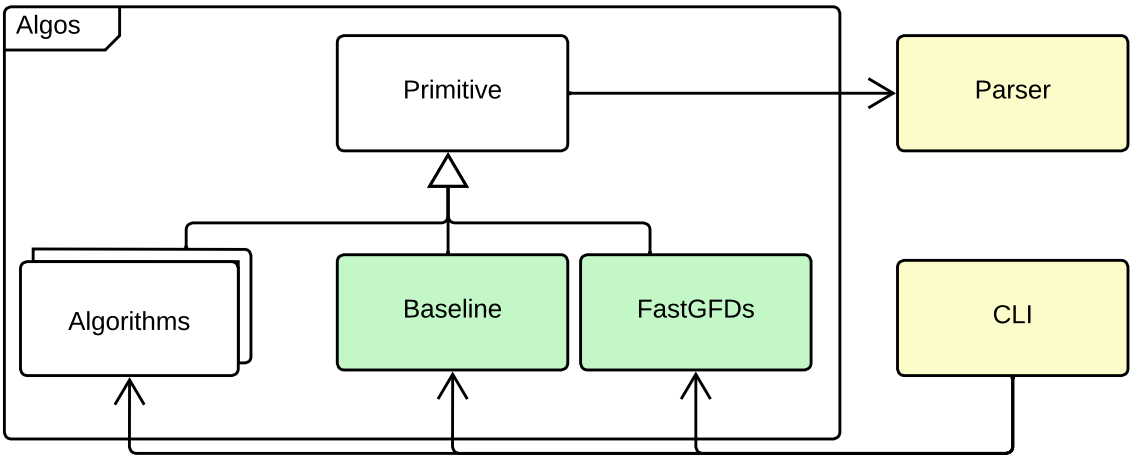


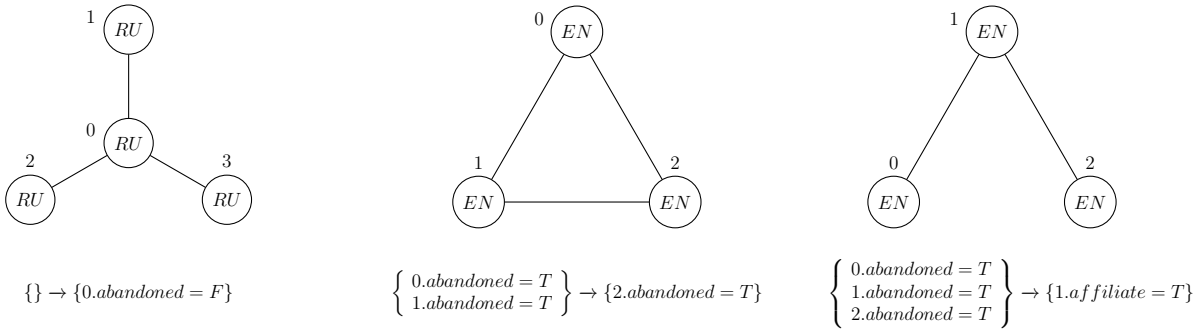

Рис. 6: Диаграмма классов для реализованных алгоритмов.

Диаграмма на Рис. 6 показывает, как выглядит описанная реализация в проекте Desbordante. Все алгоритмы лежат в пространстве имён algos и отнаследованы от класса Primitive, который описывает общий интерфейс для всех алгоритмов, работающих с функциональными зависимостями. Зелёным выделено то, что было реализовано (Baseline — базовый алгоритм, FastGFDs — предложенный). Parser представляет собой класс, описывающий работу с чтением данных для обработки. CLI

содержит в себе код для запуска алгоритмов и передачи им параметров через командную строку. Все классы, подвергнутые изменениям, выделены на диаграмме жёлтым.

# 4. Результаты

Программная и аппаратная конфигурация машины, на которой проводились эксперименты: SAMSUNG NP350E5C-S07RU, Intel(R) Core(TM) i5-3210M, 6GB RAM, `x86_64`, 20.04.1-Ubuntu, `g++ 9.4.0`, `boost 1.72.0`.




(a) Звезда. (b) Треугольник. (c) Путь.

Рис. 7: Графовые зависимости для тестов.

Для тестирования был использован датасет Twitch Gamers Social Network[2]. В связи с тем, что датасет был слишком объёмным для используемого оборудования, были использованы только первые 20 тысяч вершин и все индуцированные ими рёбра (92798). А также три графовые зависимости, представленные на Рис. 7.

Таблица 3: Время работы реализованных алгоритмов (мс).

| Алгоритм | Запрос | | |
|---|---|---|---|
| | Звезда | Треугольник | Путь |
| Наивный | 81±6 | 1638±64 | 5245±92 |
| Базовый | 74±5 | 677±8 | 4016±46 |
| Предложенный | 25±3 | 474±19 | 1390±45 |

Таблица 4: Затрачиваемая память реализованных алгоритмов (МБ).

| Алгоритм | Запрос | | |
|---|---|---|---|
| | Звезда | Треугольник | Путь |
| Наивный | 36 | 36 | 36 |
| Базовый | 173 | 173 | 173 |
| Предложенный | 36 | 36 | 36 |

Наивный, базовый и предложенный алгоритмы были протестированы на описанных данных. Результаты тестирования приведены в Таблицах 3 и 4. Таблица 3 отражает затрачиваемое время, а Таблица 4 — максимальное количество занимаемой памяти.

Как видно из таблиц, наивный алгоритм работает дольше всех, что было ожидаемо и лишь подтвердилось экспериментально. Предложенный алгоритм в свою очередь использует меньше памяти и времени, чем

[2]http://snap.stanford.edu/data/twitch_gamers.html (дата обращения 30.03.2023)

базовый. По времени улучшение варьировалось в промежутке от 1,4 до 3,3 раз, в среднем составило 2,6 раз. Потребление памяти снизилось в 4,8 раз.

# Заключение

Результаты работы:

- Выявлена наиболее ресурсозатратная часть наивного алгоритма валидации.
- Выполнен обзор существующих подходов к оптимизации выявленной части и выбран наиболее релевантный.
- Спроектирован и реализован быстрый алгоритм валидации на основе выбранного подхода.
- Произведено сравнение базового и быстрого алгоритмов валидации.

Код этой работы доступен на GitHub[3].

По теме данной работы была написана статья “FastGFDs: Efficient Validation of Graph Functional Dependencies with Desbordante”, которая была доложена на конференции FRUCT’23.

[3]https://github.com/Mstrutov/Desbordante/pull/154

# Список литературы


[1] Aggarwal Gagan, Motwani Rajeev, and Zhu An. The Load Rebalancing Problem. — 2003. — Access mode: https://dl.acm.org/doi/abs/10.1145/777412.777460 (online; accessed: 2022-11-24).

[2] Bi Fei, Chang Lijun, Lin Xuemin, Qin Lu, and Zhang Wenjie. Efficient Subgraph Matching by Postponing Cartesian Products. — 2016. — Access mode: https://dl.acm.org/doi/abs/10.1145/2882903.2915236 (online; accessed: 2023-02-23).

[3] Fan Wenfei, Liu Xueli, and Cao Yingjie. Parallel Reasoning of Graph Functional Dependencies. — 2018. — Access mode: https://ieeexplore.ieee.org/abstract/document/8509281 (online; accessed: 2022-10-17).

[4] Fan Wenfei, Wu Yinghui, and Xu Jingbo. Functional Dependencies for Graphs. — 2016. — Access mode: https://dl.acm.org/doi/abs/10.1145/2882903.2915232 (online; accessed: 2022-09-14).

[5] Ullmann J. R. An Algorithm for Subgraph Isomorphism. — 1976. — Access mode: https://dl.acm.org/doi/abs/10.1145/321921.321925 (online; accessed: 2022-11-11).

[6] Yang Jiong and Jin Wei. BR-Index: An Indexing Structure for Subgraph Matching in Very Large Dynamic Graphs. — 2011. — Access mode: https://link.springer.com/chapter/10.1007/978-3-642-22351-8_20 (online; accessed: 2023-02-14).

[7] Zhao Peixiang and Han Jiawei. On Graph Query Optimization in Large Networks. — 2010. — Access mode: https://dl.acm.org/doi/abs/10.14778/1920841.1920887 (online; accessed: 2023-02-17).

[8] Lee Jinsoo, Han Wook-Shin, Kasperovics Romans, and Lee Jeong-Hoon. An in-depth comparison of subgraph isomorphism algorithms in graph databases. — 2013. — Access mode: https://dl.acm.org/doi/abs/10.14778/2535568.2448946 (online; accessed: 2023-05-09).