Санкт-Петербургский государственный университет

Математическое обеспечение и администрирование информационных систем
Системное программирование

Группа 23.М04-мм

Черников Антон Александрович

Расширение возможностей профилировщика данных Desbordante по работе с графовыми зависимостями

Отчёт по учебной практике

Научный руководитель:
ассистент кафедры ИАС Г. А. Чернышев

Санкт-Петербург
2024

Оглавление

Введение

Профилирование данных является частой задачей у людей, работающих с объёмными массивами данных. Это процесс извлечения дополнительной информации о данных, такой как, например, автор, дата создания или изменения, размер занимаемой памяти. Однако в них может содержаться и другая не столь очевидная метаинформация. О выявлении такого рода информации из данных и будет идти речь в данной работе.

Desbordante[1] — это высокопроизводительный инструмент для профилирования данных с открытым исходным кодом, разрабатываемый группой студентов под руководством Г. А. Чернышева. Производительность содержащихся в Desbordante алгоритмов по обнаружению различных сокрытых закономерностей в данных обеспечивается языком программирования C++, на котором написан весь исходный код.

Графовые функциональные зависимости являются естественным обобщением традиционных функциональных зависимостей на такие структуры данных, как графы [4]. Графовые зависимости помогают устанавливать несоответствия в базах знаний, находить ошибки, определять спам и управлять блогами в социальных сетях.

Данная работа является продолжением работы по расширению функциональности платформы Desbordante. Предыдущая часть была направлена на реализацию и интеграцию эффективного алгоритма валидации графовых функциональных зависимостей в проект. В её результате удалось сконструировать алгоритм [5], использующий эффективный алгоритм поиска подграфа CPI [2]. Следующий шаг — реализация интерфейса, посредством которого пользователи платформы смогут взаимодействовать и запускать созданный валидатор. Именно это является одной из задач данной работы.

Проверка наличия существующей зависимости в графе — очень нужная задача, однако не всегда может быть известно, какой именно структурой обладает интересующая зависимость. Полезным инстру-

[1]https://github.com/Desbordante (дата обращения 1.05.2024)

ментом, расширяющим взаимодействие с графовыми функциональными зависимостями, является автоматическая генерация графовых зависимостей на основе данного графа. Проектирование такого алгоритма поиска (майнинга) зависимостей является нетривиальной задачей. Тем не менее, авторами понятия графовых зависимостей удалось описать такой алгоритм [1]. Эта статья будет основой для создания алгоритма майнинга и его последующей интеграции в платформу. Разбор данной статьи и составляет вторую, теоретическую, задачу настоящей работы.

1. Постановка задачи

Целью данной работы является расширение инструментария Desbordante для работы с графовыми зависимостями.

Для достижения этой цели были поставлены следующие задачи:

- Реализовать возможность запускать валидатор графовых зависимостей из скриптов, написанных на языке программирования Python. Для этого разработать соответствующую подсистему.
- Создать скрипты-примеры работы валидатора графовых зависимостей на языке программирования Python.
- Обеспечить возможность запускать валидатор графовых зависимостей через консоль путём реализации соответствующей подсистемы.
- Выполнить обзор алгоритма поиска графовых функциональных зависимостей и описать его основные свойства.

2. Обзор

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* Σ *— 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: $u.f_i = c_i$.

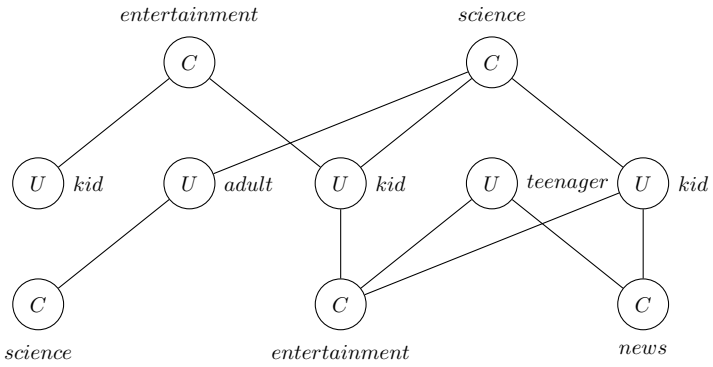
В данной работе графы рассматриваются как неориентированные, то есть, $(u, v), (v, u) \in E$ представляют собой один и тот же объект.

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

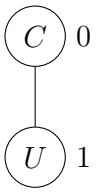
В этом определении под паттерном понимается граф, вершины которого однозначно проиндексированы от 0 до $|V| - 1$ для получения доступа к ним, а под литералом — выражение, имеющее вид $i.f = c$ (константный литерал), где i — индекс вершины паттерна, f — атрибут соответствующей вершины, c — константа (значение),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

Чтобы проверить, выполняется ли GFD, необходимо найти все подграфы графа, изоморфные паттерну, и на каждом вложении проверить



(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.

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.

Допустим, дан граф G и GFD φ. Примем обозначение: $G \models \varphi$ означает, что графовая зависимость φ выполнена на графе G.

3. Валидатор зависимостей

3.1. Python bindings

Для обеспечения возможности запускать C++ код алгоритма валидации графовых функциональных зависимостей на языке программирования Python использовалась библиотека pybind11[2]. Это легковесная заголовочная библиотека, представляющая типы данных на языке C++ в Python, и наоборот. В основном используется для создания привязок Python к существующему коду C++.

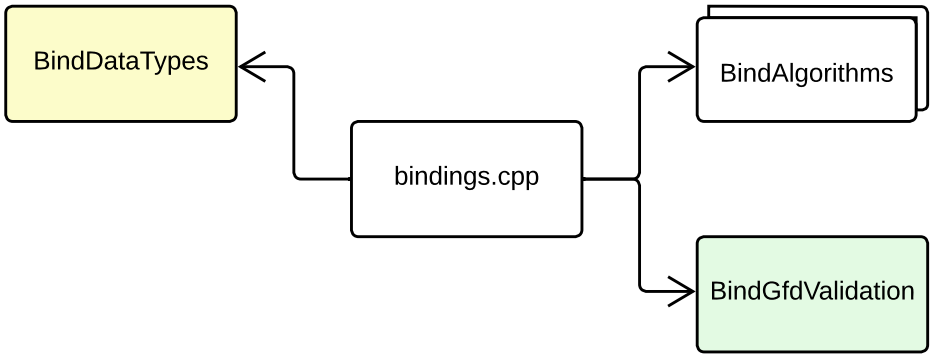


Рис. 2: Схема привязок алгоритмов Desbordante к Python.

Схема привязок C++ кода к Python проиллюстрирована на Рис. 2. Файл bindings.cpp содержит в себе макрос, который позволяет определить Python модуль desbordante. BindDataTypes хранит код для преобразования некоторых составных типов данных, использующихся в проекте, для корректного использования в Python. В отличие от остальных алгоритмов валидатор графовых зависимостей работает напрямую с dot-файлом, содержащим представление графа, вместо таблиц. Поэтому этот модуль был расширен функциональностью, позволяющей преобразовывать типы, предназначенные для хранения путей к файлам.

Также был добавлен новый файл, предоставляющий API для работы с валидатором графовых зависимостей на Python, названный BindGfdValidation.

[2]https://github.com/pybind/pybind11 (дата обращения 9.03.2024)

3.2. Пример

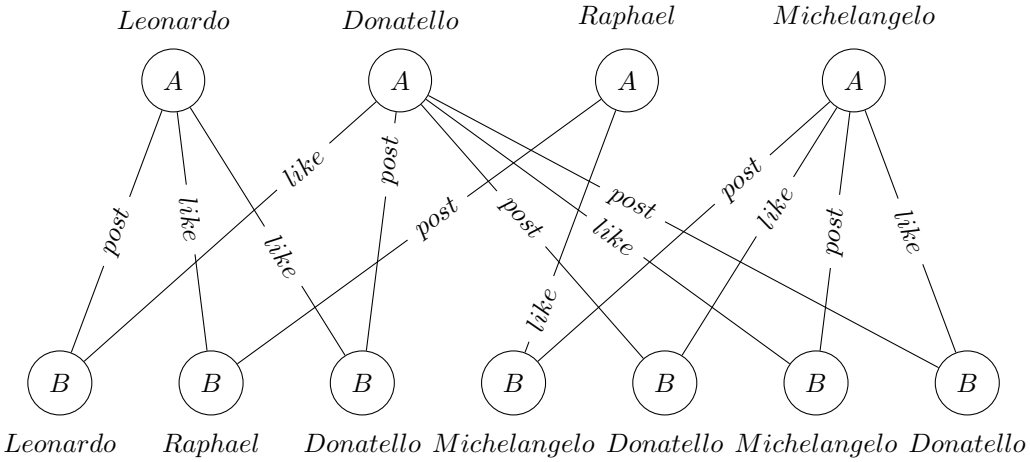
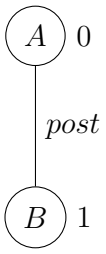
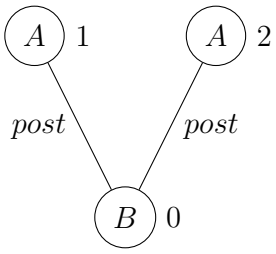


Рис. 3: Пример графа.

Для проверки работы привязок к Python был написан следующий пример. Граф изображён на Рис. 3. Здесь использовались следующие сокращения: A — $account$, B — $blog$. У вершин с меткой A есть атрибут $name$, показывающий никнейм; у вершин с меткой B — $author$, говорящий о том, кто написал данный блог. Рядом с вершинами подписаны значения этих атрибутов. Рёбра так же имеют метки: $post$, означающую то, кем был написан блог, и $like$, означающую одобрение другого человека. На рисунке рёбра с меткой $post$ выделены жирным.



(a) $\{\varnothing \to 0.name = 1.author\}$



(b) $\{\varnothing \to 1.name = 2.name\}$

Рис. 4: Примеры графовых функциональных зависимостей.

Рассмотрим две графовые зависимости, представленные на Рис. 4. С их помощью очень удобно распознавать ошибки в данных.

Если зависимость на Рис. 4a будет не выполнена, значит, была допущена ошибка в данных, так как информация, содержащаяся в метке

ребра противоречит информации об авторе, содержащейся в вершине блога.

Зависимость на Рис. 4b говорит о том, что у одного блога не может быть двух авторов. То есть, выполнение этой зависимости гарантирует, что ошибок, связанных с количеством авторов, в данных нет.

Данный пример был успешно протестирован с помощью Python, и были получены ожидаемые результаты.

3.3. Python CLI

Взаимодействие с python-модулем desbordante производится с помощью модуля Click[3]. Это пакет для создания интерфейсов командной строки. Преимущество этого пакета над аналогами заключается в том, что создание опций командной строки происходит компонуемым способом с минимальным количеством кода, обладает инструментами для настройки, а так же код на нём легко масштабируем.

Командная строка desbordante имеет ряд основных опций, необходимых для запуска алгоритмов. Прежде всего, это опция help, выводящая информацию по пользованию командной строкой. Опции task и algo отвечают за задачу и конкретный алгоритм, который необходимо запустить. Одну и ту же задачу могут выполнять несколько алгоритмов. Например, задачу по валидации графовых зависимостей выполняют наивный, базовый и эффективный алгоритмы проверки графовых зависимостей [7].

Помимо этого каждый алгоритм имеет свой уникальный набор опций. Для валидатора графовых зависимостей он состоит из пути к dot-файлу, содержащим представление графа, и набор путей к dot-файлам, содержащим представления GFD. Создание набора опций производится через декоратор.

Пример вызова валидатора через Python консоль:

```
$ python3 cli/cli.py --task=gfd_verification --algo=gfd_verifier
--graph=examples/graph.dot --gfd=examples/gfd.dot
```

[3]https://click.palletsprojects.com/en/8.1.x (дата обращения 24.03.2024)

4. Алгоритм майнинга

Алгоритм поиска графовых функциональных зависимостей, который будет рассматриваться в данной работе, описан в статье [1].

4.1. Входные параметры

Дан граф $G = (V, E, L, A)$. Основная идея алгоритма заключается в том, чтобы генерировать возможные графовые зависимости в два этапа: генерация паттерна и генерация правил литералов, а затем проверять их на выполнимость. При этом количество подграфов-кандидатов с n вершинами экспоненциально растёт с ростом n. Поэтому проверять существование вложений наивно сгенерированных подграфов может быть очень долго. В связи с данной проблемой авторы предлагают несколько эвристик, существенно сокращающих время работы алгоритма.

Во-первых, помимо графа на вход алгоритму подаётся целое число k, указывающее на то, какое максимальное количество вершин ожидается от паттернов найденных функциональных зависимостей. Если приравнять этот параметр к количеству вершин в графе, то алгоритму придётся перебирать всевозможные подграфы. Однако, зависимости, содержащие паттерны с огромным количеством вершин, в большинстве случаев очень неинформативны, поэтому у пользователя нет необходимости искать зависимости с такими паттернами. Вместо этого он может указать, сколько конкретно вершин ему хотелось бы видеть в итоговом результате.

Во-вторых, алгоритм так же получает пороговое значение σ, которое говорит, сколько вложений в граф минимально должен иметь паттерн найденной зависимости. Иными словами, это минимальная частота встречаемости паттерна в графе. Если зависимость присутствует в графе в единственном экземпляре или по крайней мере всего в нескольких, то скорее всего пользователю не нужна эта информация. С другой стороны алгоритм является итеративным и использует предыдущие результаты для следующей итерации. Сокращение результатов на текущей итерации упростит работу следующих шагов. Пользователь может

выставить желаемую частоту, чтобы увидеть только популярные зависимости.

4.2. Работа алгоритма

Обозначим за Θ множество всех вершинных меток ($L : V \to \Theta$), а за Ψ — множество всех рёберных меток ($L : E \to \Psi$).

Алгоритм работает в несколько итераций. На первой итерации происходит генерация всевозможных паттернов с единственной вершиной, используя все метки из Θ. Пусть, $\Theta = \{a, b, c\}$, $\Psi = \{e\}$. Тогда набор всех сгенерируемых паттернов будет таким, как показано на Рис. 5.

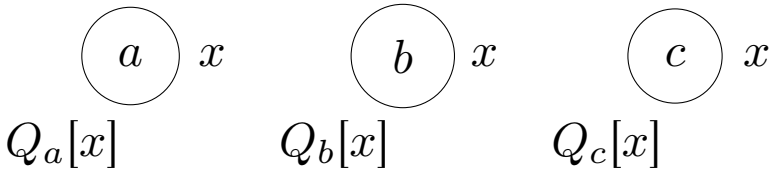


Рис. 5: Пример: паттерны первой итерации.

Далее вычисляется функция *supp* для каждого сгенерированного паттерна. Эта функция принимает на вход паттерн и граф и возвращает количество вложений паттерна в граф. Для её вычисления используется алгоритм поиска подграфа, например [6].

Предположим, пользователь ввёл значение для $\sigma = 100$. А результаты применения функций оказались следующими:

- $supp(Q_a, G) = 216$.
- $supp(Q_b, G) = 97$.
- $supp(Q_c, G) = 54$.

Тогда только паттерн $Q_a[x]$ останется для следующей итерации, так как $supp(Q_a, G) > \sigma = 100$, а остальные рассматриваться не будут.

Следующим этапом происходит генерация всевозможных графовых зависимостей для каждого оставшегося паттерна.

Для начала выписываются все литералы, которые могут встретиться в данной функциональной зависимости. Назовём этот набор буквой Г. Также необходимо выделить отдельный литерал-заключение l.

Процедура генерации использует древовидную струтуру, в узлах которой содержатся множества литералов. На самом верхнем уровне записывается единственное пустое множество. Потом происходит проверка зависимости с данным паттерном и правилом вида $(\varnothing \to l)$. Если оно выполнено, то работа окончена, переходим к следующему паттерну. Иначе достраиваем дерево: на втором, нижнем, уровне записываем все одноэлементные множества из Г, ставим ребро из пустого множества к написанным. Проверяем их по описанному алгоритму. Если нашли такой литерал, на котором GFD выполнилась, дальше вниз от этого узла прекращается построение дерева. В ином случае на третьем уровне пишутся двухэлементные множества, не содержащие выполненный литерал. Рёбра рисуются от узлов к узлам, подмножествами которых они являются. Процедура повторяется, пока нельзя будет сгенерировать следующий слой, или если дошли до конца дерева, то есть, до узла Г.

Как видно, все генерируемые правила будут всегда содержать в правой части один литерал. После того, как все зависимости будут получены, их можно упростить, склеивая две, содержащие в левых частях одно и то же, пока такие не закончатся.

Рассмотрим на примере. Допустим, $\Gamma = \{l_1, l_2, l_3, l_4\}$, и l — заранее выбранный литерал, который будет содержаться в заключении правила.

Тогда дерево будет иметь вид, изображённый на Рис. 6. Где GFD $Q_a[x](l_1 \to l)$ и $Q_a[x](\{l_2, l_4\} \to l)$ удовлетворяют графу G. Эти зависимости записываются в памяти в множество Σ_0.

На следующей итерации алгоритм генерирует все паттерны, содержащие в себе паттерны предыдущей итерации, полученные после фильтрации с помощью параметра σ. Так же эти паттерны содержат ровно одно ребро. Для них проводится такие же шаги, которые уже были описаны. После этого получаем множество GFD Σ_1. Работа алгорит-

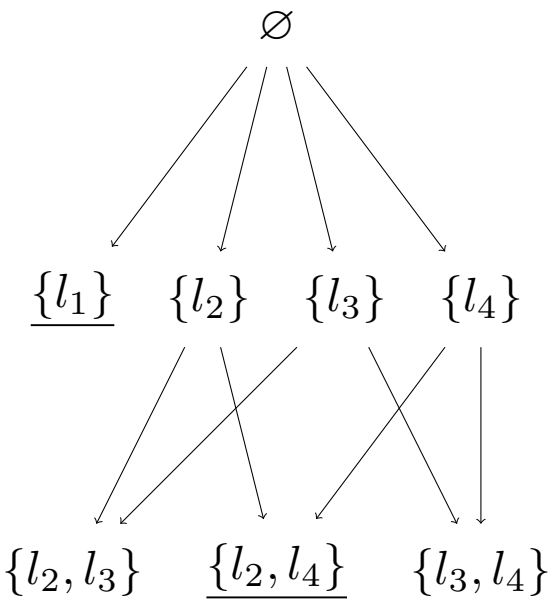


Рис. 6: Пример: дерево литералов.

ма заканчивается, когда все паттерны, количество вершин которых не больше, чем k, будут рассмотрены, или когда $supp(Q, G) < \sigma$ для всех сгенерированных на текущей итерации паттернов Q.

После завершения работы алгоритма необходимо взять объединение всех полученных множеств Σ_i, а затем применить к ним задачу импликации, чтобы избавиться от избыточности. Полученное множество будет ответом.

Заключение

Результаты работы:

- Реализована возможность запускать валидатор графовых зависимостей из скриптов, написанных на языке программирования Python. Разработана соотвутствующая подсистема.
- Созданы скрипты-примеры работы валидатора графовых зависимостей на языке программирования Python.
- Обеспечена возможность запускать валидатор графовых зависимостей через консоль путём реализации соответствующей подсистемы.
- Выполнен обзор алгоритма поиска графовых функциональных зависимостей и описаны его основные свойства.

В дальнейшем планируется:

- Спроектировать быстрый алгоритм поиска на основе рассмотренного алгоритма и реализовать его.
- Произвести тестирование алгоритма поиска.

Исходный код разработанных подсистем был принят, интегрирован и вошёл в релиз Desbordante 2.0.0[4].

[4]https://github.com/Desbordante/desbordante-core/pull/379 (дата обращения 01.05.2024)

Список литературы

[1] Fan Wenfei, Hu Chunming, Liu Xueli, and Lu Ping. Discovering Graph Functional Dependencies. — 2020. — Access mode: https://dl.acm.org/doi/abs/10.1145/3397198 (online; accessed: 2022-10-16).

[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] Chernikov Anton, Litvinov Yurii, Smirnov Kirill, and Chernishev George. FastGFDs: Efficient Validation of Graph Functional Dependencies with Desbordante. — 2023. — Access mode: https://elibrary.ru/item.asp?id=53943942 (online; accessed: 2024-03-09).

[6] 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).

[7] Черников Антон. Реализация эффективного алгоритма проверки графовых функциональных зависимостей в платформе Desbordante. — 2023. — Access mode: https://github.com/Mstrutov/Desbordante/blob/main/docs/papers/FastGFDs%20-%20Anton%20Chernikov%20-%20BA%20thesis.pdf (online; accessed: 2024-03-24).