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
2025

# Оглавление

# Введение

В современном мире объёмы информации растут с невероятной скоростью. Специалисты, работающие с данными, сталкиваются с необходимостью их анализа и обработки. Одной из ключевых задач в этой области является профилирование данных — процесс, который позволяет получить дополнительную информацию о данных.

Профилирование данных включает в себя извлечение дополнительной информации о данных, такой как авторство, дата создания или изменения, а также размер занимаемой памяти. Однако данные могут содержать множество неочевидных зависимостей и закономерностей, которые могут быть не сразу заметны человеческому глазу, но могут иметь важное значение для понимания структуры и содержания данных. В данной работе рассматривается вопрос выявления такого рода информации из данных.

Desbordante[1] — это высокопроизводительный инструмент для профилирования данных с открытым исходным кодом, разработанный группой студентов под руководством Г. А. Чернышева. Проект содержит множество алгоритмов, которые способны обнаруживать различные закономерности в данных, а также предоставляет соответствующие пользовательские интерфейсы для них. В качестве основного языка используется C++, что в целом повышает производительность.

Графовые функциональные зависимости (Graph Functional Dependencies) представляют собой естественное обобщение традиционных функциональных зависимостей на структуры данных, такие как графы [5]. Они помогают выявлять несоответствия в базах знаний, находить ошибки, определять спам и управлять блогами в социальных сетях.

Desbordante уже содержит алгоритмы, работающие с графовыми зависимостями, такие как алгоритмы проверки выполнимости существующих графовых зависимостей. Эти алгоритмы получают на вход граф и множество графовых функциональных зависимостей, после че-

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

го возвращают только те из них, которые выполнены на данном графе. Полезным алгоритмом, расширяющим взаимодействие с графовыми зависимостями, является автоматическая генерация графовых зависимостей на основе входного графа. Авторы понятия графовых зависимостей разработали такой алгоритм и описали его в статье [2], однако ими не был предоставлен код алгоритма. Desbordante содержит и этот алгоритм. Но код данного алгоритма в проекте нуждается в улучшении. Именно это и является практической составляющей этой работы. Анализ кода и выявление замечаний будет производиться с помощью рецензентов кода.

Следующий шаг — разработка нового алгоритма, который по входящему графу ищет структуры, являющиеся обобщением графовых функциональных зависимостей. Эти структуры называются графовыми дифференциальными зависимостями (Graph Differential Dependencies), они введены авторами статьи [1] и будут подробно описаны в данной работе. Теоретическая часть работы состоит из изучения и обзора алгоритма поиска графовых дифференциальных зависимостей. Этот алгоритм использует как один из своих этапов алгоритм генерации подграфов [7], занимающий существенную часть работы, и нуждающийся в дополнительном глубоком изучении.

# 1. Постановка задачи

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

Для достижения этой цели были поставлены следующие задачи:

- Произвести анализ и улучшение алгоритма поиска функциональных зависимостей в соответствии с предложениями рецензентов кода.
- Создать скрипт-пример работы алгоритма поиска графовых зависимостей на языке программирования Python.
- Выполнить обзор алгоритма поиска графовых дифференциальных зависимостей и описать его основные свойства.

# 2. Предварительные сведения

**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: Данные о студентах и их оценках

| ID | Name | Course | Grade |
|---|---|---|---|
| 1 | Alice | Math | A |
| 2 | Bob | Math | B |
| 3 | Charlie | Science | A |
| 1 | Alice | Science | B |
| 2 | Bob | Science | A |

Пусть, отношение представлено в виде Таблицы 1. Заметим, что фукциональная зависимость $ID \to Name$ выполнена, так как в этом случае каждый идентификатор студента (ID) уникально определяет его имя (Name). Например, для ID = 1 всегда будет Name = “Alice”. В это же время зависимость $Name \to Course$ не выполнена. Здесь мы видим, что одно и то же имя может соответствовать нескольким курсам. Например, “Alice” изучает как “Math”, так и “Science”. Это означает, что имя не может однозначно определить курс, что делает эту зависимость невыполненной.

Функциональные зависимости могут быть обобщены на графы. Одно из таких обобщений предлагают авторы статьи [5], 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 [4], 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: $u.f_i = c_i$. У каждой вершины есть атрибут *eid* (entity identificator) — идентификатор сущности. Он широко используется в графах со свойствами. Несколько вершин могут представлять собой один и тот же объект. Чтобы это обозначить используется данный атрибут.

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 дифференциальная зависимость)** *GDD (Graph Differential Dependency) — это конструкция* $Q[\overline{z}](X \to Y)$*, где* $Q$ *— паттерн, а* $X$ *и* $Y$ *— множества ограничений расстояния.*

В этом определении под паттерном понимается граф, вершины которого однозначно проиндексированы от 0 до $|V| - 1$ для получения доступа к ним, а под ограничением расстояния — выражение, представляющее собой одну из следующих конструкций:

$$\delta_A(x.A, c) \leq t_A;\ \delta_{A_1A_2}(x.A_1, x'.A_2) \leq t_{A_1A_2};$$

$$\delta_{eid}(x.eid, c_e) = 0;\ \delta_{eid}(x.eid, x'.eid) = 0;$$

$$\delta_{\equiv}(x.rela, c_r) = 0;\ \delta_{\equiv}(x.rela, x'.rela) = 0;$$

Где $x, x'$ — индексы паттерна; $A, A_1, A_2$ — атрибуты; $c$ — значение атрибута; $\delta_{A_1A_2}(x.A_1, x'.A_2)$ (или $\delta_{A_1}(x, x')$, если $A_1 = A_2$) — заданная

пользователем функция расстояния для значений $A_1, A_2$; $t_{A_1A_2}$ — порог для $\delta_{A_1A_2}(\cdot,\cdot)$; $\delta_{eid}(\cdot,\cdot)$ (соответственно $\delta_{\equiv}(\cdot,\cdot)$) — функция для $eid$ (соответственно $\equiv$), возвращающая 0 или 1. $\delta_{eid}(x.eid, c_e) = 0$, если значение $eid$ у $x$ равно $c_e$; $\delta_{eid}(x.eid, x'.eid) = 0$, если у $x$ и $x'$ одинаковое значение $eid$; $\delta_{\equiv}(x.rela, c_r) = 0$, если у $x$ есть отношение с именем $rela$, заканчивающееся узлом $c_r$; $\delta_{\equiv}(x.rela, x'.rela) = 0$, если у $x$ и $x'$ есть отношение с именем $rela$, заканчивающееся одним и тем же узлом.

Заданная пользователем функция расстояния $\delta_{A_1A_2}(x.A_1, x'.A_2)$ зависит от типов $A_1$ и $A_2$. Это может быть арифметическая операция с интервальными значениями, расстояние редактирования для строковых значений или расстояние между двумя категориальными значениями в таксономии.

В предыдущей работе рассматривались так называемые графовые функциональные зависимости. Графовые дифференциальные зависимости в сущности являются их обобщением. Соответственно, любой алгоритм поиска графовых функциональных зависимостей находит только подкласс зависимостей, когда алгоритм поиска GDD находит всеобъемлющее исчерпывающее множество.

**Лемма 1** *Пусть $\sigma = Q[\overline{z}](X \to Y)$ — GDD, $G$ — граф.*

*GDD $\sigma$ выполнена на графе $G$ тогда и только тогда, когда*

$$sat(X) \subseteq sat(Y)$$

Здесь $sat(A)$ — множество всех вложений паттерна $Q$ в граф $G$ таких, что все ограничения расстояния из $A$ выполнены.

Переформулирование выполнимости графовых дифференциальных зависимостей в терминах данной леммы служит для удобства выполнения алгоритма поиска GDD.

**Определение 4 (Частовстречаемая GDD (Frequent GDD))** *Пусть $I = \{i_1, \ldots, i_m\}$ — множество изоморфизмов паттерна $Q[\overline{z}]$ и подграфов графа $G$, где $\overline{z} = \{x_1, \ldots, x_n\}$; и пусть*

*$D(x_i) = \{i_1(x_i), \ldots, i_m(x_i)\}$ — множество вершин графа $G$, которые являются образом вершины $x_i$ посредством изоморфизмов. Частота встречаемости определяется следующим образом:*

$$mni(Q[\overline{z}], G) = min\{|D(x_i)|,\ x_i \in \overline{z}\}$$

*Таким образом, при наличии минимальной поддержки $MNI = \tau$ паттерн $Q[\overline{z}]$ является частовстречаемым в $G$ тогда и только тогда, когда $mni(Q[\overline{z}], G) \geq \tau$.*

*GDD $\sigma = Q[\overline{z}](X \rightarrow Y)$ является частовстречаемой, когда паттерн $Q[\overline{z}]$ частовстречаемый.*

Нетрудно заметить, что при добавлении в паттерн ребра или вершины его частота встречаемости не будет увеличиваться. Это свойство монотонности обеспечит более эффективную генерацию паттернов.

**Определение 5 (Неизбыточная GDD (Non-redundant GDD))** *GDD $\sigma = Q[\overline{z}](X \rightarrow Y)$ является неизбыточной, тогда и только тогда, когда паттерн $Q[\overline{z}]$:*

a. *Является частовстречаемым, то есть $mni(Q[\overline{z}], G) \geq \tau$.*

b. *Не существует паттерна $P[\overline{w}]$: $Q$ — подграф $P$ и $mni(Q[\overline{z}], G) = mni(P[\overline{w}], G)$.*

**Определение 6 (Неприводимая GDD (Irreducible GDD))** *GDD $\sigma = Q[\overline{z}](X \rightarrow Y)$ является неприводимой, тогда и только тогда, когда:*

a. *Любой атрибут $A \in Y$ не встречается в $X$*

b. *Не существует GDD $\psi = P[\overline{w}](X' \rightarrow Y')$:*

   (*i*) *$Q$ — подграф $P$,*

   (*ii*) *$X' \succeq X$,*

   (*iii*) *$Y \succeq Y'$.*

Запись $X \succeq X'$ (множество ограничений расстояния $X$ подчиняет себе множество ограничений расстояния $X'$) обозначает, что:

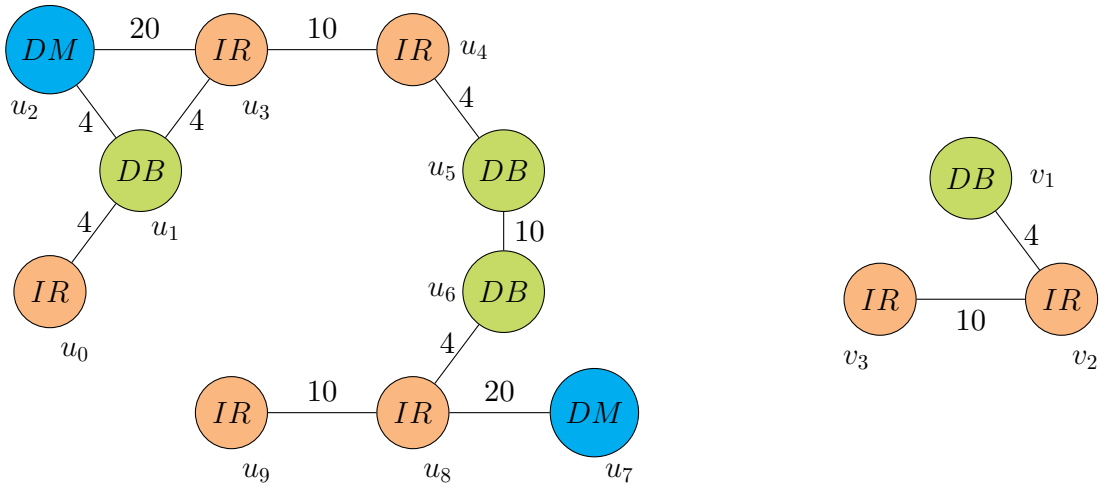
(*i*) Для каждого ограничения расстояния вида $\delta_A(x.A, c) \leq t_A^{(1)}$ или $\delta_{A_1A_2}(x.A_1, x'.A_2) \leq t_{A_1A_2}^{(1)}$ из $X$ существует ограничение расстояния вида $\delta_A(x.A, c) \leq t_A^{(2)}$ или $\delta_{A_1A_2}(x.A_1, x'.A_2) \leq t_{A_1A_2}^{(2)}$ из $X'$ такое, что $t_A^{(1)} \geq t_A^{(2)}$ или $t_{A_1A_2}^{(1)} \geq t_{A_1A_2}^{(2)}$ соответственно,

(*ii*) Для каждого ограничения расстояния вида $\delta_{\equiv/eid}(\cdot, \cdot) = 0$ из $X$ существует ограничение расстояния вида $\delta_{\equiv/eid}(\cdot, \cdot) = 0$ из $X'$.

**Определение 7 (Constraint Satisfaction Problem (CSP))** *Модель CSP представляет собой тройку* $(X, D, C)$*, где*

- $X$ — *упорядоченный список переменных*
- $D$ — *список доменов, соответствующих переменным* $X$
- $C$ — *набор ограничений между переменными* $X$

Для лучшего понимания природы CSP предлагается ознакомиться с примером на Рис. 4. Граф и запрос из примера взяты из статьи [7].




(a) Граф $G$ (b) Запрос $S$

Рис. 1: Пример графа и запроса.

Задача поиска вхождений запроса $S$ в граф $G$ может быть сформулирована в терминах модели CSP:

$$CSP = \left( \begin{array}{c} (v_1, v_2, v_3), \{\{u_1, u_5, u_6\}, \{u_3, u_4, u_8\}, \{u_4, u_3, u_9\}\}, \\ \{v_1 \neq v_2 \neq v_3, L(v_1) = DB, L(v_2) = L(v_3) = IR, \\ L(v_1, v_2) = 4, L(v_2, v_3) = 10\} \end{array} \right)$$

Первое множество $(v_1, v_2, v_3)$ представляет собой список переменных, которые обозначают вершины графа-запроса. Второе множество — это набор доменов для вершин $v_1$, $v_2$ и $v_3$ соответственно. Каждый домен — ничто иное, как список вершин графа $G$, которые являются образами при отображении запроса $S$ в данный граф. Запрос имеет три вхождения в граф, при каждом вхождении вершина $v_1$ может быть отображена в вершину $u_1$,$u_5$ или $u_6$. Этот список и есть домен вершины $v_1$. Для остальных вершин домены задаются аналогично.

Перейдём к третьему множеству. Оно содержит в себе так называемые правила или ограничения, которые должны соблюдаться на каждом вхождении запроса $S$ в граф $G$. В примере перечислен минимум: вершины должны быть различными, а также все вершины и рёбра должны иметь указанные метки. При необходимости этот список ограничений можно пополнить, например, указав минимальные степени вершин.

Алгоритм поиска графовых дифференциальных зависимостей будет использовать формулировку задачи поиска вхождений при генерации частовстречаемых паттернов.

# 3. Обзор

Графовые зависимости были предложены авторами статьи [5], в которой также описывают и оценивают алгоритм проверки выполнения набора графовых зависимостей на больших реальных графах.

В результате одной из предыдущих работ [8] был реализован и интегрирован алгоритм проверки графовых функциональных зависимостей в проект Desbordante. Для него были реализованы три версии. Одна из них наивная, необходимая для сравнения с остальными алгоритмами. Вторая является реализацией алгоритма из рассмотренной статьи, а третья — сконструированная улучшенная его версия [6], использующая эффективный алгоритм поиска подграфа CPI [3]. Кроме этого, проект Desbordante был расширен возможностью запускать интегрированный алгоритм проверки графовых зависимостей на Python и через консоль, а также снабжён скриптами-примерами работы этого алгоритма.

Другая предыдущая работа [9] была направлена на обзор новой статьи [2], которая предлагает принципиально другой алгоритм — алгоритм поиска графовых функциональных зависимостей. Их различие с алгоритмом проверки графовых зависимостей в том, что в первом случае перед работой алгоритма пользователю известна вся информация об интересующей зависимости, и задача алгоритма — дать ответ на вопрос выполняется ли данная зависимость на графе. А алгоритм поиска зависимостей позволяет генерировать такие зависимости автоматически. Исходные данные ограничиваются лишь графом, на котором необходимо произвести поиск выполненных зависимостей. Этот алгоритм был реализован в проекте Desbordante.

Данная работа направлена на улучшение реализованного алгоритма поиска функциональных зависимостей и обзор новой статьи, описывающей алгоритм поиска графовых дифференциальных зависимостей.

# 4. Улучшение алгоритма поиска.

В рамках прошлой работы была поставлена задача реализовать алгоритм поиска графовых функциональных зависимостей. После написания первоначального варианта алгоритма была проведена его проверка и улучшение на основе замечаний и предложений от рецензентов кода.

## 4.1. Исправление замечаний по коду.

Рецензенты обратили внимание на следующие аспекты:

- Была произведена полная переработка пространств имён. Вспомогательные структуры и псевдонимы лежали в пространстве имён $algos$, после исправлений это пространство содержало только алгоритмы, а вся сопутствующая периферия была перемещена в отдельные пространства имён, такие как $gfd$ и $model$.

- Для алгоритмов, отвечающих за валидацию и поиск графовых зависимостей были созданы соответсвующие директории для более удобного хранения кода.

- Было предложено использовать метод $try\_emplace$ вместо двойной проверки наличия элемента в контейнере.

- Была подчёркнута важность комментариев и описаний. Были добавлены комментарии с описанием алгоритмов и их параметров.

- Использование ссылок в качестве полей в классе $CmpCallback$ было признано неоптимальным решением. Было предложено использовать лямбда-выражения для упрощения кода и повышения его читаемости.

- Было предложено использовать алгоритмы вместо циклов для упрощения кода и повышения его эффективности. Например, вместо цикла $for$ было предложено использовать $std :: ranges :: min$.

- Было отмечено, что некоторые методы могут быть упрощены путём использования алгоритмов, таких как $std :: ranges :: all\_of$.

- Было предложено рассмотреть возможность использования $std :: unordered\_map$ вместо $std :: map$ для оптимизации работы с данными.

- Было отмечено, что некоторые объекты создаются и затем перемещаются, что может быть неэффективным. Было предложено использовать $emplace\_back$ для непосредственного создания объектов в контейнере.

- Был извлечён общий функционал из методов $Validate$ и $Support$ в отдельные функции, что позволило избежать дублирования кода.

- Вместо передачи параметров алгоритма в некоторые функции было предложено использовать поля класса ($graph\_$, $k\_$ и $sigma\_$). Это упростило код и сделало его более читаемым.

## 4.2. Обновление тестов.

Кроме того, рецензенты кода предложили обновить тесты для проверки корректности работы алгоритма. Было предложено создать отдельный класс для конфигурации тестов и использовать его для инициализации параметров тестов. Это позволило упростить код тестов и сделать его более структурированным.

Пример теста до переработки:

```
TEST_F(GfdMiningTest, CompareResultTest) {
    std::vector<Gfd> gfds = {MakeGfd(kGfdTestGfd)};
    std::filesystem::path const graph_path = kGfdTestGraph;
    std::unique_ptr<GfdMiner> algorithm =
            CreateGfdMiningInstance(graph_path, 2, 3);
    algorithm->Execute();
    auto const gfd_list = algorithm->GfdList();
    ASSERT_EQ(expected_gfds.size(), gfd_list.size());
```

```
    ASSERT_THAT(gfd_list,
        ::testing::ElementsAreArray(expected_gfds));
}
```

Пример обновлённого теста:

```
TEST_P(GdfMiningTest, CompareResultTest) {
    auto algorithm =
        algos::CreateAndLoadAlgorithm<GfdMiner>(Params());
    algorithm->Execute();
    ASSERT_THAT(algorithm->GfdList(),
        ::testing::ElementsAreArray(GetExpectedGfds()));
}
```

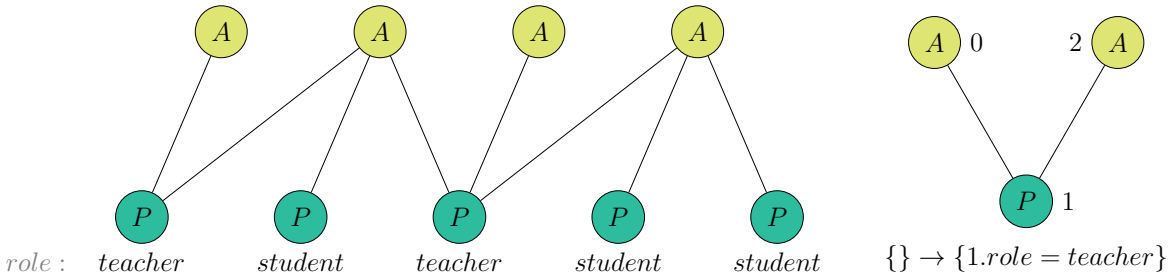
Как видно, новый вариант содержит более читаемый и понятный код. Были удалены так называемые магические константы при создании алгоритма. Блоки кода, ответственные за загрузку параметров (графа и графовой зависимости), выделены в отдельные функции.

Для более эффективной загрузки датасетов было предложено создать отдельный файл $all\_gfd\_paths.h$, содержащий переменные, хранящие информацию о путях к входным данным. Это помогло структурировать код тестов.

После выполнения всех исправлений обновлённый пулл-реквест был одобрен рецензентами и успешно принят. В общей сложности исправлено 403 комментария (включая минорные замечания), было осуществлено порядка двадцати проходов рецензирования.

# 5. Пример

Для наглядной демонстрации пользователям сценариев использования алгоритма поиска графовых функциональных зависимостей дополнительно был написан следующий пример.




(a) Граф. (b) Найденная зависимость.

Рис. 2: Пример.

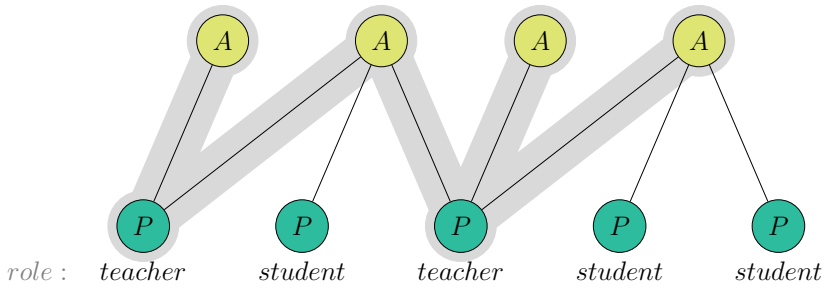



Рис. 3: Вложения паттерна зависимости в граф.

Рассмотрим граф, представленный на Рис. 2a. Он описывает связи между научными статьями и их авторами. Вершины этого графа имеют две метки: $A$ (*Article*, Статья) и $P$ (*Person*, Человек). У каждой вершины есть свой набор атрибутов в зависимости от метки.

- *Article*:
  - *title* — обозначает название статьи.
- *Person*:
  - *name* обозначает имя человека.
  - *role* может принимать одно из двух значений: “*teacher*” (преподаватель) или “*student*” (студент).

Для поиска используем следующие параметры: $k = 3, \sigma = 2$.

В результате работы алгоритма получаем на выходе одну зависимость, изображённую на Рис. 2b. Обнаруженная зависимость может быть трактована следующим фактом: если у человека есть две опубликованные статьи, то он является учителем.

Для наглядности, пример также предоставляет все вложения патерна найденной зависимости (Рис. 3).

# 6. Алгоритм поиска дифференциальных зависимостей

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

## 6.1. Входные параметры

Дан граф $G = (V, E, L, A)$. Алгоритм находит частовстречаемые графовые дифференциальные зависимости. На это есть несколько причин. Во-первых, предполагается, что зависимости, которые встречаются в графе в очень малом количестве, неинтересны пользователю, соответственно, их рассмотрение нецелесообразно. Во-вторых, такая эвристика поможет значительно повысить производительность, существенно сократив время выполнения алгоритма. За пороговое значение, задающее минимальную частоту встречаемости зависимостей, отвечает входной параметр $\tau$.

Также алгоритм принимает на вход множество ограничений расстояния $\Delta$. Оно представляет собой конкретные функции $\delta$, определённые на атрибутах, задаваемых пользователем, а также пороговые значения для каждой из таких функций. Однако этот параметр является опциональным. Если пользователь не предоставил функции для ограничений расстояния, то в качестве них берутся алгоритмом функции по умолчанию с равнораспределёнными пороговыми значениями.

Работу алгоритма можно разделить на три глобальных этапа, которые будут описаны далее более подробно:

1. Генерация частовстречаемых паттернов.
2. Генерация выполненных зависимостей на основе сгенерированных паттернов.
3. Удаление избыточных зависимостей.

Псевдокод приведён в Алгоритме 1.

**Algorithm 1** GDDMiner

**Input:** Граф $G$, частотный порог $\tau$, ограничения расстояний $\Delta$ (опционально)

**Output:** Минимальное покрытие GDD $\Sigma_c$

1: $\Sigma := \emptyset, \Sigma_c := \emptyset$
/* получение частых неизбыточных паттернов */
2: $Q \leftarrow rGrami(G, \tau)$
/* поиск GDD на основе полученных паттернов */
3: **for** $Q_i[\overline{z_i}] \in Q$ **do**
4: $H(Q_i[\overline{z_i}], G) \leftarrow hMatches(Q_i[\overline{z_i}], G)$
5: $\Sigma_i \leftarrow redGDDs(H, \Delta)$
6: $\Sigma \leftarrow \Sigma \cup \Sigma_i$
/* отбрасывание избыточных GDD */
7: $\Sigma_c \leftarrow cover(\Sigma)$
8: **return** $\Sigma_c$

Более подробный разбор каждого этапа описывается в следующих разделах.

## 6.2. Генерация частовстречаемых паттернов

Первый этап использует алгоритм GraMi, описанный в статье [7].

**Algorithm 2** FrequentSubgraphMining

**Input:** Граф $G$, частотный порог $\tau$

**Output:** Все подграфы $S$ графа $G$ такие, что $s_G(S) \geq \tau$

1: $result \leftarrow \emptyset$
2: **Let** $fEdges$ be the set of all frequent edges in $G$
3: **for** $e \in fEdges$ **do**
4: $result \leftarrow result \cup SubgraphExtension(e, G, \tau, fEdges)$
5: Remove $e$ from $G$ and $fEdges$
6: **return** $result$

Рассмотрим псевдокод общего подхода к поиску частовстречаемых паттернов. Он представлен в Алгоритме 2. Для начала определяется множество $fEdges$, обозначающее все рёбра графа, которые имеют частоту встречаемости не меньше, чем $\tau$ (строчка 2). Имеет смысл рассматривать только такие рёбра, ведь по свойствам монотонности редкие

рёбра не могут быть включены в частовстречаемые подграфы.

Далее происходит последовательный перебор всех найденных рёбер из множества $fEdges$, и для каждого такого ребра вызывается функция SubgraphExtension, возвращающая все частовстречаемые подграфы, в которых содержится рассматриваемое ребро (строчки 3–5). После чего все локальные результаты для каждого ребра объединяются и возвращается общий результат. Более детальное описание функции предъявлено в Алгоритме 3.

**Algorithm 3** SubgraphExtension

**Input:** Подграф $S$ графа $G$, частотный порог $\tau$ и множество частовстречающихся рёбер $fEdges$ в графе $G$

**Output:** Все частовстречающиеся подграфы графа $G$, которые расширяют $S$

```
1:  result ← S, candidateSet ← ∅
2:  for e ∈ fEdges and node u of S do
3:      if e can be used to extend u then
4:          Let ext be the extension of S with e
5:          if ext is not already generated then
6:              candidateSet ← candidateSet ∪ ext
7:  for c ∈ candidateSet do
8:      if s_G(c) ≥ τ then
9:          result ← result ∪ SubgraphExtension(c, G, τ, fEdges)
10: return result
```

Функция SubgraphExtension вызывается рекурсивно. Она принимает на вход построенный подграф $S$, о котором уже известно, что он является частовстречаемым. Далее происходит попытка расширить подграф $S$ новым ребром и проверить новый граф на уникальность во избежание ненужных дополнительных проверок (строчки 2–6). Генерируется множество $candidateSet$, в котором содержатся графы, включающие в себя подграф $S$. После чего производится проверка графов-кандидатов на частоту встречаемости (строчки 7–9). Для тех из них, которые прошли проверку, рекурсивно вызывается функция SubgraphExtension. Результат содержит в себе сам подграф $S$, а также всё, что получилось в итоге рекурсивного вызова SubgraphExtension

для всех новых графов.

Теперь рассмотрим более подробно как происходит проверка на частоту встречаемости (строчка 8). Поведение функции описано в Алгоритме 4.

**Algorithm 4** IsFrequentCsp

**Input:** Графы $S$ и $G$, частотный порог $\tau$

**Output:** $true$, если $S$ — частовстречающийся подграф графа $G$, иначе $false$

1: Consider the subgraph $S$ to graph $G$ CSP
2: Apply node and arc consistency
3: **if** *the size of any domain is less than* $\tau$ **then**
4: **return** $false$
5: **for** *solution Sol of the S to graph G CSP* **do**
6: Mark all nodes of *Sol* in the corresponding domains
7: **if** *all domains have at least* $\tau$ *marked nodes* **then**
8: **return** $true$
9: **return** $false$ ▷ Domain is exhausted

Алгоритм IsFrequentCsp использует определённую в разделе "Предварительные сведения" модель CSP. Он строит такую модель для вложений подграфа $S$ в граф $G$. Для начала множество доменов инициализируется значениями $V$ для каждой вершины подграфа $S$ (где $V$ — это множество вершин графа $G$; строчка 1). Далее проверяются две согласованности (строчка 2):

- Согласованность вершин.

  Для каждой вершины $u$ подграфа $S$ происходит обход всех вершин из соответствующего домена модели CSP и проверка всех ограничений модели $L(u) = C$. Если вершина из домена не удовлетворяет хотя бы одному условию, она вычёркивается из домена. Все остальные вершины там остаются.

- Согласованность рёбер.

  Для каждой пары вершин $u, v$ из подграфа $S$ проверяется наличие такой пары вершин из соответствующих доменов этих вершин, что все условия ограничений $L(u, v) = C$ выполнены. Если

не находится ни одной такой пары, это значит, что нет ни одного вложения подграфа $S$ в граф $G$. Следовательно, в таком случае можно с уверенностью сказать, что подграф не является частовстречаемым.

После проверки согласованностей алгоритм считает, сколько вершин осталось в доменах после удаления. По определению, частота подграфа определяется как минимальная мощность среди всех доменов. Соответственно, если найдётся хотя бы один домен, количество элементов которого меньше, чем $\tau$, то частота такого паттерна будет также меньше, чем $\tau$, отсюда вывод, что этот паттерн не частовстречаемый (строчки 3–4).

Следующим шагом в алгоритме является последовательный перебор всех вложений подграфа $S$ в граф $G$ (под словом $Sol$ подразумевается вхождение). Найденное вхождение означает, что для вершин подграфа в модели CSP совершенно точно должны присутствовать в соответствующих доменах вершины графа, в которые они отобразились. Алгоритм помечает такие вершины. После каждого вхождения производится подсчёт помеченных вершин в каждом домене. Если все домены набрали по крайней мере по $\tau$ помеченных вершин каждый, то по определению частовстречаемости паттерна дальше можно не проверять, так как он уже частовстречаемый. Если же все вхождения были рассмотрены, и остался хотя бы один домен с количеством помеченных вершин меньше $\tau$, то функция возвращает $false$ (строчки 5–9).

Для этой функции также может быть применён ряд оптимизаций.

- Первая оптимизация заключается в пересмотре способа обхода вершин. Вместо того, чтобы рассматривать все вхождения по порядку, можно рассматривать вершины подграфа и соответствующие им домены. Для каждой вершины $v$ из домена $D$, который соответствует вершине подграфа $u$ рассматривается возможность вложения подграфа при условии отображения вершины $u$ в вершину $v$. Если такое находится, то мы помечаем вершину $v$, а также все остальные вершины из соответствующих доменов. Иначе

вычёркиваем вершину $v$ из домена $D$. Как только в домене набирается $\tau$ помеченных вершин, алгоритм переходит к следующей вершине. Процесс повторяется, пока не будут просмотрены все вершины или когда найдётся домен, в котором останется помеченных вершин меньше, чем $\tau$.

- Сокращение сверху вниз. Идея заключается в том, что для построения модели CSP для рассматриваемого графа можно пользоваться знанием об уже построенных моделях CSP для подграфов данного графа. Соответственно, если из домена для произвольной вершины произвольного графа была вычеркнута вершина-кандидат, то эта же вершина должна быть вычеркнута из соответствующих доменов всех графов, включающих в себя данный граф как подграф.

- Уникальные метки. Если выполнено условие: “Среди меток подграфа нет повторяющихся и структура подграфа является деревом”, то алгоритм можно завершить после второй строчки (то есть, будет достаточно проверки согласованности вершин и рёбер). Доказательство валидности такой оптимизации представлено в статье [7].

- Автоморфизмы. Если подграф может быть нетривиально отображён сам в себя, то можно собрать несколько кластеров вершин, внутри каждого кластера будут лежать вершины, которые могут получиться в результате автоморфизма. При рассмотрении вершины, можно лишь пройтись по кластерам и производить одни и те же манипуляции со всеми вершинами рассматриваемого кластера вместо того, чтобы идти по всем вершинам.

- Ленивый поиск. Применяется следующая эвристика: во время рассмотрения вершины можно считать, сколько ещё вершин осталось проверить. Тогда если их количество будет слишком маленьким, например, даже в самом лучшем случае (все нерассмотренные вершины станут помеченными) количество всех помеченных вершин

не превысит $\tau$, то можно сказать, что подграф не является частовстречаемым.

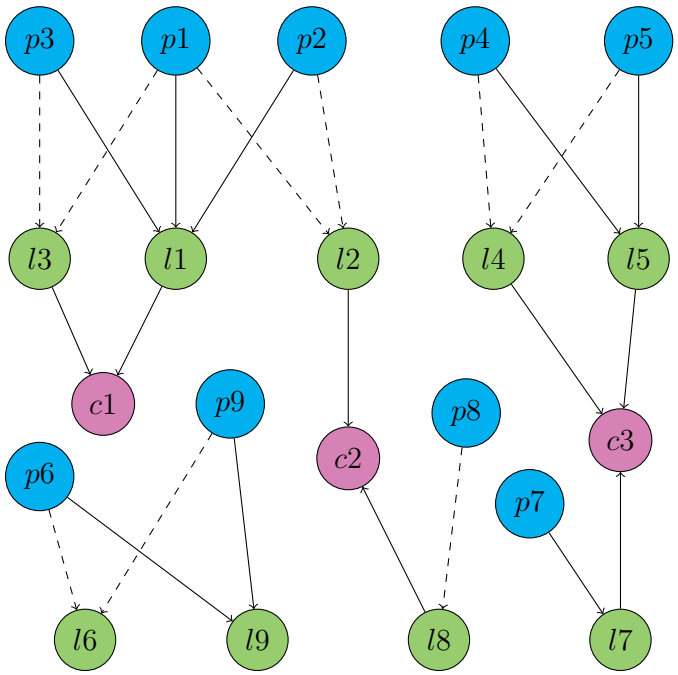
- Декомпозиционное сокращение. Каждый граф был получен из какого-то конкретного ребра $e$ по построению. С помощью сокращения сверху вниз все лишние вершины были удалены из доменов, но были рассмотрены только графы, содержащие ребро $e$. Оптимизация предлагает воспользоваться другими моделями CSP, но на этот раз использовать графы, которые не содержат ребро $e$.

## 6.3. Поиск графовых дифференциальных зависимостей

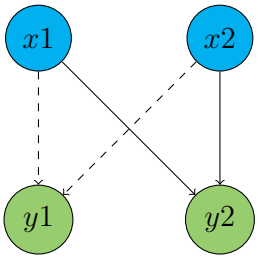
Следующим этапом является создание дифференциальных зависимостей на основе сгенерированных паттернов. Эти паттерны заведомо являются частовстречаемыми по построению. Алгоритм использует поэлементную обработку множества всех сгенерированных паттернов в цикле. Для каждого паттерна находятся все его вхождения в граф. За это отвечает функция $hMatches$ в Алгоритме 1.

Рассмотрим на примере, взятом из статьи [1]. Пусть дан граф $G$, изображённый на Рис. 4a, и паттерн на Рис. 4b. Рис. 4c иллюстрирует обозначения для меток вершин и рёбер соответственно. В таком случае функция $hMatches$ сгенерирует всевозможные вхождения рассматриваемого паттерна в граф $G$. Таких вложений будет четыре. Эти вложения можно представить в виде Таблицы 2, в которой каждая строка отвечает за вложение паттерна, а столбцы — это атрибуты для каждой вершины.

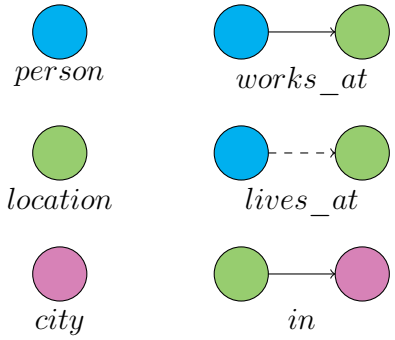
Далее происходит генерация правил для графовых дифференциальных зависимостей. Из $\Delta$ берутся ограничения расстояния, подходящие для рассматриваемых атрибутов. Они представляют собой неравенство: функцию из набора функций, определённых на некоторых атрибутах; и число из ряд пороговых значений для каждой такой функции. Если же пользователь предпочёл опустить информацию об ограничени-




(a) Граф $G$.




(b) Паттерн $Q$




(c) Обозначения для меток.

Рис. 4: Пример графа и паттерна.

Таблица 2: Пример таблицы вложений $Q$ в $G$.

| | $x_1$ : *person* | | | $y_1$ : *location* | | | | $x_2$ : *person* | | | $y_2$ : *location* | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | id | Name (A1) | DOB (A2) | id | Type (A3) | Street name (A4) | State (A5) | id | Name (A1) | DOB (A2) | id | Type (A3) | Street name (A4) | State (A5) |
| h1 | p1 | William Johnson | 1970.7.31 | l3 | Res | 46 Adrian Ave | SA | p3 | Bill Johnson | 7/31/70 | l1 | Comm | 17 Tea Tree Gully | SA |
| h2 | p1 | William Johnson | 1970.7.31 | l2 | Res | 28 Main Rd | NSW | p2 | Mary Smith | Jun. 1990 | l1 | Comm | 17 Tea Tree Gully | SA |
| h3 | p4 | Paul H. Colbert | | l4 | Com | 93 High St | NSW | p5 | Colber P. H. | 25/12/89 | l5 | Com | 93 High Street | NSW |
| h4 | p6 | Tom Engel | 1 Jun. 1956 | l6 | | 99 Mawson blvd | WA | p9 | Engel, Thomas | 06/01/65 | l9 | Com | 199 Main Road | WA |

ях, то алгоритм берёт стандартные заранее определённые функции и равномерно удалённые пороговые значения. Для каждой функции с пороговым значением на основе постренной таблицы вычисляется множество вложений, на которых это ограничение выполняется. В результате получается набор ограничений расстояния, каждое из которых имеет множество удовлетворяющих этому ограничению вложений.

Рис. 5 иллюстрирует пример ограничений расстояний, введённых пользователем, а также множества вложений, на которых ограничения выполнены. Эти ограничения могут быть произвольными.

В заключение, строится так называемая решётка атрибутов. Это структура данных, представляющая собой дерево с направленными

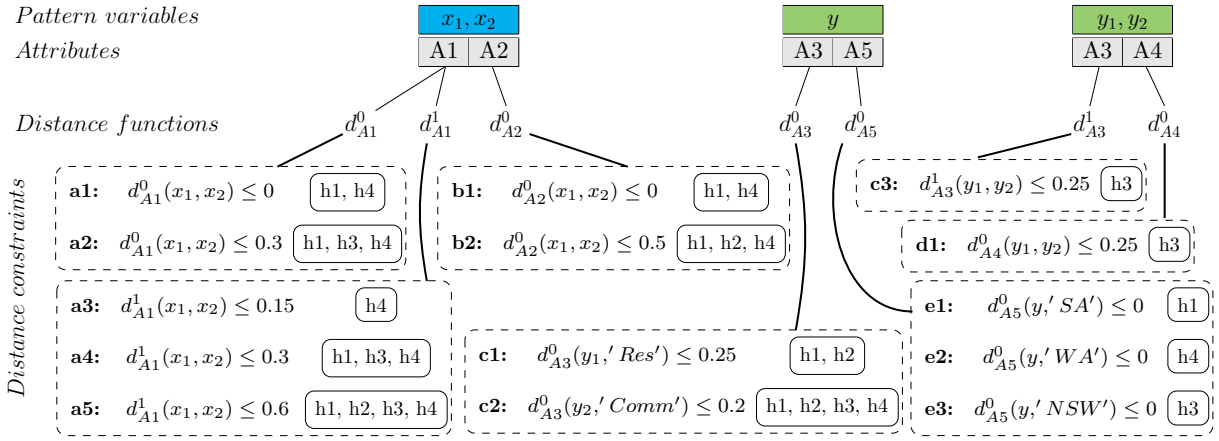



Рис. 5: Примеры ограничений расстояния для вхождений $Q$.

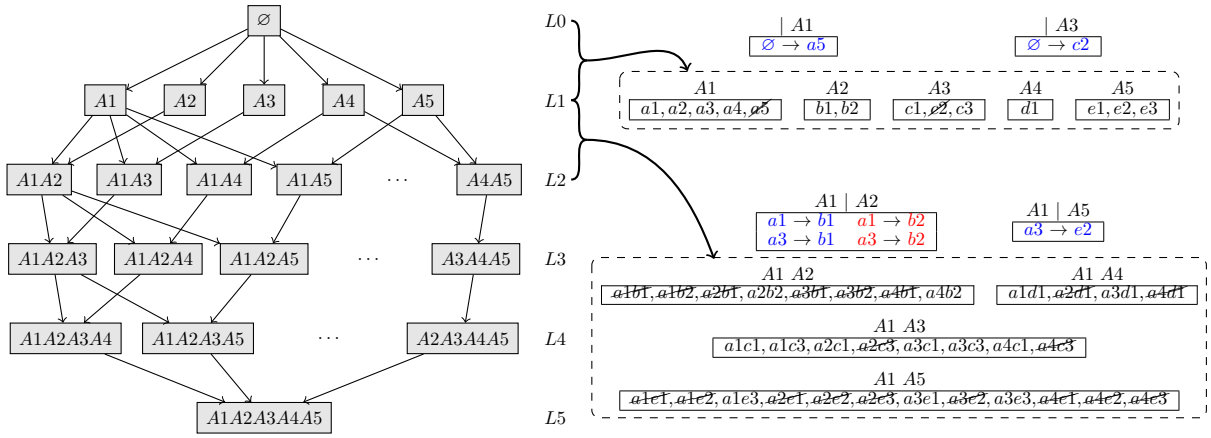



Рис. 6: Пример решётки атрибутов (слева); и пример обхода и обрезки пространства поиска кандидатов GDD (справа).

рёбрами. В корне дерева содержится узел, соответствующий пустому множеству (нулевой слой $L0$). Первый слой $L1$ состоит из узлов, содержащих одноэлементные множества — элементы являются атрибутами. Каждый последующий слой $Li$ содержит в себе всевозможные множества атрибутов мощностью $i$. Рёбра решётки атрибутов соединяют узлы только соседних слоёв. Между двумя узлами есть ребро, если множество в узле-предке является подмножеством множества в узле-потомке. На Рис. 6 изображена решётка атрибутов для рассматриваемого примера (слева).

После построения решётки атрибутов алгоритм приступает к генерации правил. Решётка обходится сверху-вниз слева-направо путём обхода рёбер. Пусть, рассматриваем ребро $(A, B)$, где $A$ — множество атрибутов узла-предка, а $B$ — множество атрибутов узла-потомка. Пра-

вило генерируется следующим образом: в правой части остаётся ограничение расстояния из блока ограничений, принадлежащих атрибуту $B \setminus A$ (такое множество будет всегда одноэлементным по построению решётки атрибутов). Левая часть получает любое ограничение из блока ограничений для каждого атрибута из $A$.

В результате описанных действий сгенерируется множество правил для данного паттерна. Теперь по Лемме 1 легко проверить выполнимость каждого правила, так как задача сводится к сравнению множеств. Все правила, которые оказались выполнены, служат для генерации на их основе GDD, и записываются в ответ.

Тонкость работы алгоритма на этом этапе заключается в том, что могут быть сгенерированы два правила, одно из которых является избыточным. Если правило выполняется для некоторого порога в ограничении расстояния в правой части, то оно же будет выполняться для всех таких же ограничений, но с бóльшим пороговым значением. В таком случае для генерации зависимости используется минимальный порог, а все пороги, которые больше сгенерированного, игнорируются несмотря на их выполнимость.

Если удалось сгенерировать зависимость, то алгоритм запоминает ограничение, которое задействовалось в правой части, и далее происходит обрезка пространства ограничений путём удаления всех таких множеств, которые содержат в себе это ограничение. Для более детального понимания рекомендуется обратиться к правой части Рис. 6. Здесь синим выделены правила, которые оказались выполнены, а красным — избыточные правила. Так же множества ограничений в левых частях подверглись обрезке (помечено зачёркиванием). Рисунок предоставляет обход только первых двух слоёв.

## 6.4. Удаление избыточных зависимостей

Завершительный этап алгоритма — выделение минимального покрытия из множества получившихся графовых дифференциальных зависимостей $\Sigma$. $\Sigma$ можно представить как мультимножество, так как

каждый паттерн генерирует множество своих зависимостей. По построению, внутри каждого кластера, порождаемого паттерном, гарантируется, что не существует избыточных зависимостей. Соответственно, необходимо лишь проверить наличие избыточных зависимостей попарно для кластеров.

Для любой пары $\Sigma_i, \Sigma_j \in \Sigma$ такой, что $Q_i[\overline{z_i}] \supseteq Q_j[\overline{z_j}]$: любое GDD $\sigma' \in \Sigma_j$ исключается из $\Sigma_j$, если существует $\sigma \in \Sigma_i$ такое, что $LHS(\sigma) \succeq LHS(\sigma')$ и $RHS(\sigma') \succeq RHS(\sigma)$. Эта процедура повторяется для всех пар $\Sigma_i, \Sigma_j$ до тех пор, пока не перестанут происходить изменения, после чего они добавляются в $\Sigma_c$.

# Заключение

Результаты работы:

- Произведены анализ и улучшение алгоритма поиска функциональных зависимостей в соответствии с предложениями рецензентов кода.
- Создан скрипт-пример работы алгоритма поиска графовых зависимостей на языке программирования Python.
- Выполнен обзор алгоритма поиска графовых дифференциальных зависимостей и описаны его основные свойства.

Код работы доступен на GitHub[2]. Исправленный пулл-реквест был одобрен и принят.

[2]https://github.com/Desbordante/desbordante-core/pull/465 (дата обращения 19.04.2025)

# Список литературы


[1] Zhang Yidi, Kwashie Selasi, Bewong Michael, Hu Junwei, Mahboubi Arash, Guo Xi, and Feng Zaiwen. Discovering Graph Differential Dependencies. — 2023. — Access mode: https://link.springer.com/chapter/10.1007/978-3-031-47843-7_18 (online; accessed: 2024-04-20).

[2] 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).

[3] 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).

[4] 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).

[5] 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).

[6] 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).

[7] Elseidy Mohammed, Abdelhamid Ehab, Skiadopoulos Spiros, and Kalnis Panos. GRAMI: Frequent subgraph and pattern mining in a single large graph. — 2014. — Access mode: https://repository.kaust.edu.sa/items/7e0f406b-4243-4c63-86d5-940ab866b4fd (online; accessed: 2024-03-21).

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

[9] Черников Антон. Расширение возможностей профилировщика данных Desbordante по работе с графовыми зависимостями. — 2024. — Access mode: https://github.com/Desbordante/desbordante-core/blob/main/docs/papers/Bindings%2C%20examples%2C%20CLI%20and%20mining%20review%20for%20GFD%20-%20Anton%20Chernikov%20-%202023%20autumn.pdf (online; accessed: 2024-10-03).