Санкт-Петербургский государственный университет

Черников Антон Александрович

Выпускная квалификационная работа

Расширение возможностей профилировщика данных Desbordante по работе с графовыми зависимостями

Уровень образования: магистратура

Направление *02.04.03 «Математическое обеспечение и администрирование информационных систем»*

Основная образовательная программа *ВМ.5665.2023 «Математическое обеспечение и администрирование информационных систем»*

Научный руководитель:
доцент кафедры информационно-аналитических систем, к. ф.-м. н, Е. Г. Михайлова

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

Рецензент:
Java разработчик Каспи Банк, Каспи Магазин М. А. Струтовский

Санкт-Петербург
2025

Saint Petersburg State University

Anton Chernikov

Master's Thesis

Expanding Desbordante data profiler capabilities to handle graph dependencies

Education level: master

Speciality *02.04.03 "Software and Administration of Information Systems"*

Programme *BM.5665.2023 "Software and Administration of Information Systems"*

Scientific supervisor:
C.Sc., Information and Analytical Systems chair docent E.G. Mikhailova

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

Reviewer:
Java developer at Kaspi Shop M.A. Strutovskii

Saint Petersburg
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.

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 чего воз-

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

вращают 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. Эта статья будет основой для создания алгоритма поиска графовых функциональных зависимостей и его последующей интеграции в платформу.

1. Постановка задачи

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

Для достижения этой цели были поставлены следующие задачи:

- Выполнить обзор алгоритма поиска графовых функциональных зависимостей и реализовать его.
- Разработать подсистему, позволяющую запускать реализованный алгоритм из скриптов, написанных на языке программирования Python.
- Создать скрипты-примеры работы реализованного алгоритма на языке программирования Python.
- Обеспечить возможность запускать реализованный алгоритм через консоль путём реализации соответствующей подсистемы.
- Произвести анализ и разработать ускоренный алгоритм поиска графовых функциональных зависимостей.
- Произвести тестирование производительности ускоренного алгоритма поиска графовых функциональных зависимостей.

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* 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* $P[\overline{z}](X \to Y)$*, 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.

Лемма 1. *Пусть* $\phi = P[\overline{z}](X \to Y)$ *— GFD,* G *— граф.*

GFD ϕ выполнена на графе G тогда и только тогда, когда

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

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

Переформулирование выполнимости графовых функциональных зависимостей в терминах данной леммы служит для оптимизации выполнения алгоритма поиска GFD.

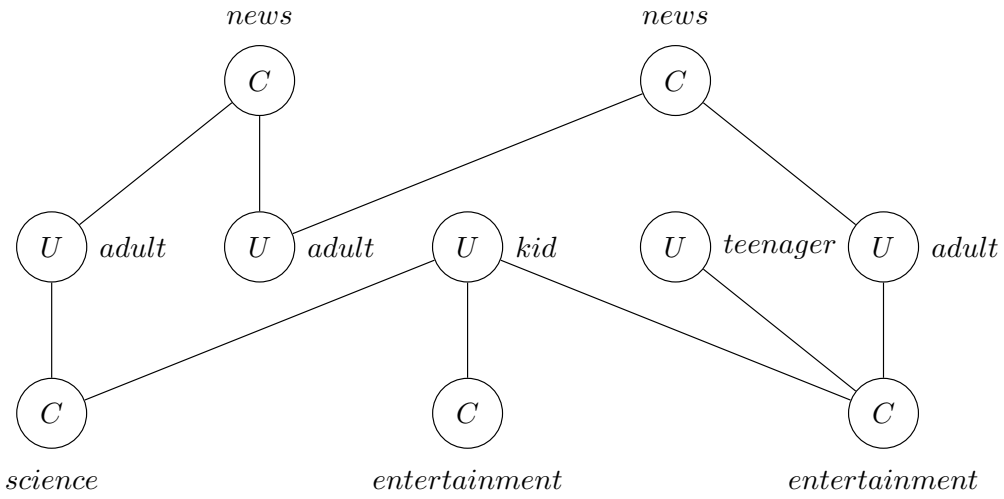
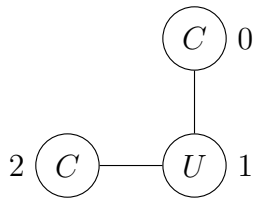


Рис. 1: Связь каналов C (Channel) с пользователями U (User). Атрибуты у вершин с меткой C — $\{topic\}$, с меткой U — $\{age\_group\}$.

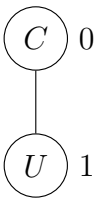
На Рис. 1 представлен пример графа. Вершины графа имеют метки C или U. В зависимости от метки вершина имеет свой собственный набор атрибутов. В данном примере все вершины имеют одноэлементный список атрибутов. У вершин с меткой C он состоит из элемента $topic$, а у вершин с меткой U — age_group. Конкретные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

На Рис. 2 продемонстрированы графовые 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.



$\{0.topic = news\} \rightarrow \{1.age\_group = adult\}$

(a) Графовая зависимость.



$\{0.topic = news\} \rightarrow \{1.age\_group = adult\}$

(b) Минимальная графовая зависимость.

Рис. 2: Пример графовых функциональных зависимостей.

Если не нашлось ни одного вложения паттерна, то такая зависимость считается тривиально выполненной.

Нетрудно заметить, что в данном примере обе зависимости выполняются на приведённом графе.

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

Определение 4 (Минимальная GFD)**.** *Пусть GFD* $\phi = P[X \rightarrow Y]$ *выполнена на графе* G*.*

ϕ *— минимальная, если* $\forall \psi = Q[X \rightarrow Y] \ : \ G \models \psi \ P \subset Q$*.*

Рассмотрим графовые зависимости на Рис. 2. Графовая зависимость, изображённая на Рис. 2b, говорит о том, что в графе все подписчики новостных каналов являются взрослыми. Однако, если мы добавим новую вершину к паттерну, как показано на Рис. 2a, полученная графовая зависимость будет так же выполняться на графе, хотя она несёт в себе избыточную информацию и перегружает понимание сути. Именно по этой причине в данной работе будут находиться только минимальные графовые зависимости.

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.

Данная работа направлена на обзор новой статьи [2], которая предлагает принципиально другой алгоритм — алгоритм поиска графовых функциональных зависимостей, и его реализацию. 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.

4. Алгоритм поиска зависимостей

Алгоритм поиска графовых функциональных зависимостей, который будет рассматриваться в данной работе, описан в статье [2].

4.1. Входные параметры

Дан граф $G = (V, E, L, A)$. Алгоритм состоит из двух этапов: генерация паттернов и генерация правил литералов. После этого проверяется выполнимость сгенерированных зависимостей.

Сложность заключается в том, что количество подграфов-кандидатов с n вершинами растёт экспоненциально с ростом n. Из-за этого долго проверять существование вложений наивно сгенерированных подграфов. Чтобы решить эту проблему, авторы предлагают использовать несколько эвристик, которые существенно сокращают 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 выражает минимальное значение метрики, условно обозначающей количество вложений паттерна в граф, на которых графовая зависимость выполнена. Можно считать, что 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. Он может выставить желаемую частоту, чтобы увидеть только популярные зави-

симости.

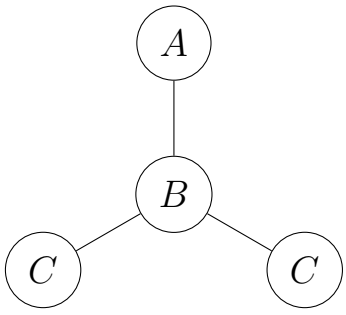
Прежде чем перейти дальше, стоит обсудить вышеупомянутую метрику. Пусть, она будет выражаться функцией $supp(\phi, G)$, где ϕ — графовая зависимость. От этой метрики требуется выполнение двух свойств:

1. Асимптотически она должна выражать частоту встречаемости данной графовой зависимости в графе.

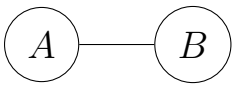
2. Чем больше элементов в паттерне графовой зависимости, тем меньше должно быть значение этой метрики.

Выполнение первого свойства необходимо по очевидным причинам: от пользователя ожидается, что он будет вводить значение этой метрики исходя из ожиданий популярности интересующих зависимостей. Что касается второго свойства, то оно необходимо для того, чтобы сократить количество генерируемых паттернов на текущей итерации, а, следовательно, и время работы алгоритма.

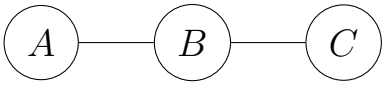
Если взять за функцию $supp(\phi, G)$ количество вложений паттерна графовой зависимости ϕ в граф G, на которых она выполняется, то второе свойство не будет выполняться. Рассмотрим пример на Рис. 3. Легко увидеть, что паттерн P_1 имеет лишь одно вложение в данный граф G. Добавим теперь к нему новую вершину с меткой C и ребро между вершинами с метками B и C. Полученный паттерн P_2 имеет уже два вложения в тот же граф.



(a) Граф G.



(b) Паттерн P_1; вложений в G: 1.



(c) Паттерн P_2; вложений в G: 2.

Рис. 3: Увеличение количества вложений в граф с ростом вершин в паттерне.

Исходя из вышесказанного, видно, что самое естественное определение функции *supp* не подходит для алгоритма. Усовершенствуем эту метрику так, чтобы второе свойство выполнялось. Предлагается посчитать для каждой вершины паттерна количество вершин-образов графа, на вложении в которые при этом выполняется данная зависимость. После этого взять минимальное значение среди посчитанных за значение метрики. Нетрудно заметить, что теперь второе свойство будет выполняться. При добавлении ребра в паттерн количество вложений не увеличится, следовательно, не увеличится и значение метрики. Если же добавляется новая вершина, то количество вложений может увеличиться, остаться прежним или уменьшиться. Последние два случая аналогичны варианту с добавлением ребра. Рассмотрим теперь случай, когда количество вложений увеличилось. Тогда результат метрики не сможет стать больше. Действительно, ведь предыдущее значение содержалось среди старых вершин, а после добавления новой количество их образов не сможет увеличиться, следовательно, как максимум значение метрики для нового паттерна будет принадлежать одной из них, что не больше, чем значение старого паттерна.

Заметим, что естественно определить эту метрику и для паттернов $supp(Q, G)$. Для этого можно пренебречь условием выполнимости зависимости.

Говоря формальным языком, эта метрика может быть определена как:

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

где $I = \{i_1, \ldots, i_m\}$ — множество изоморфизмов паттерна $P[\overline{z}]$ и подграфов графа G, $\overline{z} = \{x_1, \ldots, x_n\}$; $D(x_i) = \{i_1(x_i), \ldots, i_m(x_i)\}$ — множество вершин графа G, которые являются образом вершины x_i посредством изоморфизмов.

Для графовых функциональных зависимостей метрика определяется аналогично за исключением того, что на множество I накладывается дополнительное ограничение: изоморфизм включается только если на нём выполнены все литералы из X, а также все литералы из Y.

Следовательно, частовстречаемость GFD $\phi = P(X \to Y)$ влечёт за собой частовстречаемость паттерна P. Отсюда можно сделать вывод о том, что если паттерн Q не частовстречаемый, то все зависимости вида $\psi = Q(X \to Y)$ также не будут частовстречаемыми. Это понадобится для более эффективной реализации алгоритма.

4.2. Предобработка

Прежде чем перейти к работе алгоритма, понадобится произвести некоторые подготовительные работы. Для этого необходимо обойти начальный граф, запомнить все метки вершин и рёбер, которые в нём присутствуют, все вложения одновершинных паттернов, а так же сгенерировать структуру, упрощающую генерацию литералов для паттернов.

Информация о метках пригодится для генерации паттернов, ведь паттерны с иными метками совершенно точно не будут иметь вложений в данный граф.

По ходу просмотра будут сразу генерироваться паттерны, состоящие из одной вершины, а так же записываться все вложения для каждого из них. Полученные данные будут использоваться как вход на первую итерацию алгоритма.

Структура для генерации литералов представляет собой ассоциативный массив. Его ключами являются метки вершин, а значениями — новые ассоциативные массивы, отвечающие за возможные атрибуты этих вершин. Такое разделение сделано из соображений зависимости атрибутов от меток вершин. Во вложенных массивах в качестве ключей выступают названия атрибутов, а в качестве значений — списки значений этих атрибутов, которые встречаются в графе. Пример такой структуры представлен на Рис. 4.

Как видно из примера, названия атрибутов у разных вершин могут совпадать, однако списки значений у них могут отличаться. Учитывая эту информацию, генерация литералов будет происходить гораздо оптимальнее.

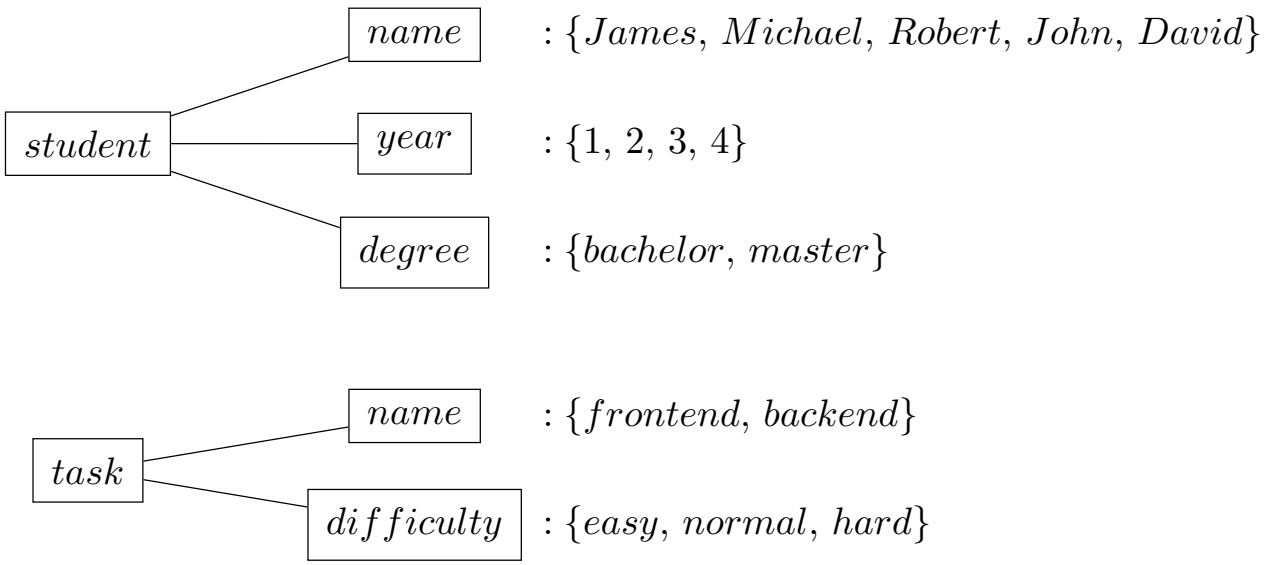


Рис. 4: Структура для генерации литералов.

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

Рассмотрим пример. Пусть, $\Theta = \{a, b, c\}$, $\Psi = \{e\}$. Тогда набор всех сгенерированных паттернов будет таким, как показано на Рис. 5.

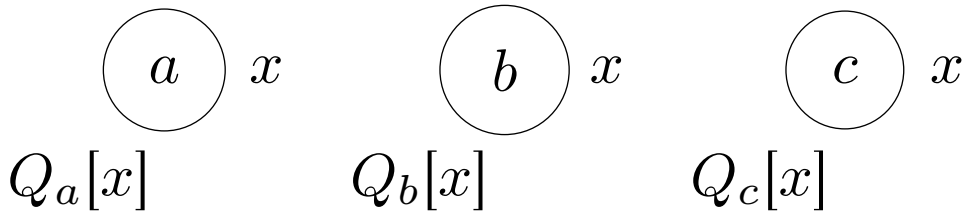


Рис. 5: Пример: начальные паттерны.

Далее вычисляется функция *supp* для каждого сгенерированного паттерна. 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 первой итерации, так как $supp(Q_a, G) > \sigma = 100$, а остальные рассматриваться не будут. Это позволяет существенно сократить время работы алгоритма.

4.3. Основная работа

Алгоритм является итеративным. В целом, каждая итерация состоит из горизонтальной и вертикальной генерации графовых зависимостей. Она обновляет список паттернов, а так же списки вложений и запрещённых правил для каждого паттерна. Для начальных паттернов списки запрещённых литералов являются пустыми.

Горизонтальная генерация получает на вход список паттернов, создаёт всевозможные литералы для каждого из них с помощью структуры литералов, и проверяет, выполняются ли они, при помощи списка вложений. Назовём этот набор литералов буквой Г. Из множества Г выделяется литерал-заключение l, а $\Gamma := \Gamma \setminus \{l\}$. 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. Проверка осуществляется с учётом параметра σ и параллельным подсчётом метрики *supp*. Если для какого-то паттерна нашлась GFD $\phi = P[X \to Y]$ такая, что $supp(\phi, G) > \sigma$ и $G \models \phi$, ϕ добавляется в результат, а правило $X \to Y$ добавляется в список запрещённых правил для этого паттерна. 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 множеств небольшой длины, например, 3-4. Такая эвристика обусловлена тем, что пользователю маловероятно, что понадобится сложные правила с большим количеством литералов, так как они почти наверняка не будут выполнены и весьма не информативны.

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

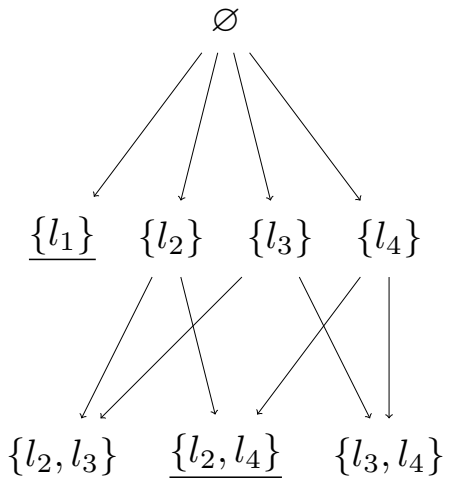


Рис. 6: Пример: дерево литералов.

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.

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.

Далее происходит вертикальная генерация. Она заключается в создании паттернов для следующей итерации. Для каждого паттерна создаётся новый посредством добавления ребра. Ребро может быть добавлено с новой вершиной или без неё. Как только паттерн сгенерирован, происходит проверка на изоморфность с помощью, например, алгоритма ульмана [7] с новыми уже сгенерированными паттернами. Если такой нашёлся, то производится добавление к списку запрещённых правил найденного паттерна запрещённых правил сгенерированного. Иначе список запрещённых правил копируется от того, на основе которого был сгенерирован текущий, а так же создаётся новый список вложений на основе него.

После того, как все новые паттерны сгенерированы, они пропуска-

ются через фильтр метрикой *supp*. Очевидно, что

$$supp(P[X \to Y], G) \leq supp(P, G),$$

Следовательно, если $supp(P, G) < \sigma$, то паттерн P можно дальше не рассматривать.

За счёт того, что когда находится графовая зависимость $\phi = P[X \to Y]$, выполняющаяся на данном графе, все зависимости $\psi = Q[X \to Y]$, такие, что $P \subset Q$, запрещены, так как для них правило $X \to Y$ попадает в список запрещённых, это помогает искать лишь минимальные графовые зависимости.

Алгоритм завершается, как только был сгенерирован пустой список новых паттернов или после k-ой итерации.

5. Реализация

5.1. Алгоритм

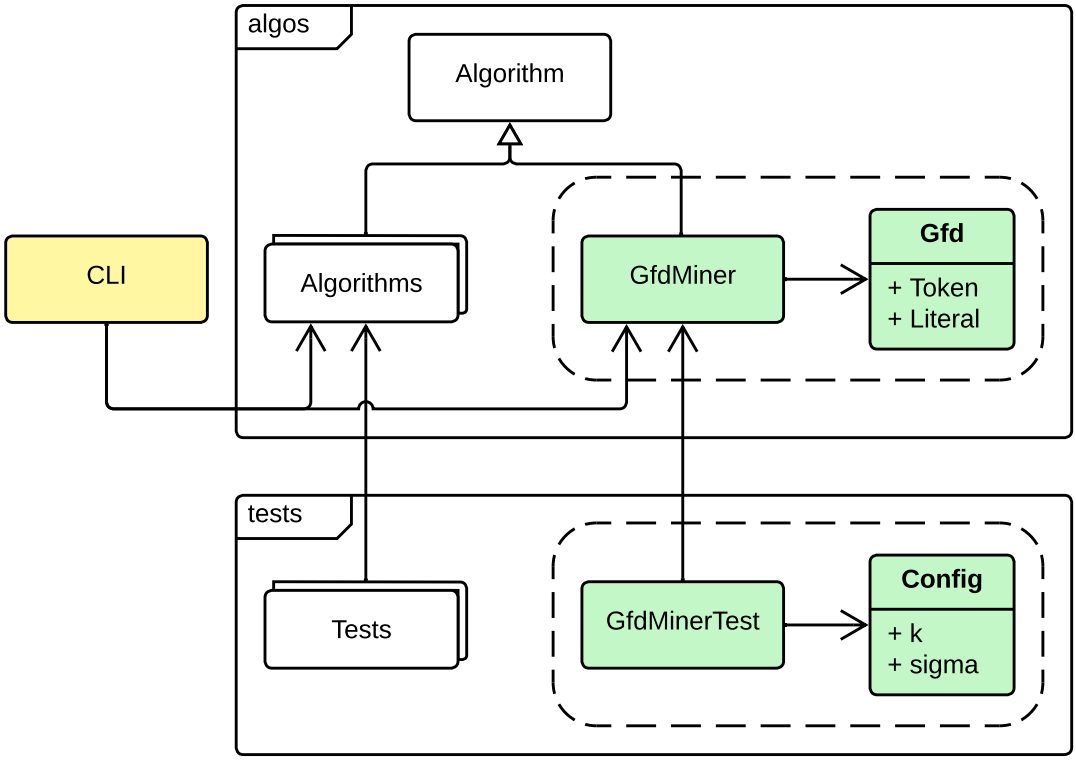


Рис. 7: Диаграмма классов для реализованного алгоритма.

Диаграмма на Рис. 7 показывает, как выглядит описанная реализация в проекте Desbordante. Все алгоритмы лежат в пространстве имён algos и отнаследованы от класса Algorithm, который описывает общий интерфейс для всех алгоритмов, работающих с функциональными зависимостями. Зелёным выделено то, что было реализовано. CLI содержит в себе код для запуска алгоритмов и передачи им параметров через командную строку. Класс, подвергнутый изменениям, выделен на диаграмме жёлтым.

При реализации алгоритма была использована графовая библиотека boost. Она позволяет гибко определить свой собственный тип графа, названный в проекте $graph\_t$, его свойства, а также свойства вершин и рёбер. Так как графовые зависимости требуют от графа быть графом с

атрибутами, все вершины были сконструированы содержать в себе список атрибутов. Помимо этого все элементы графа имеют собственные метки, а графы являются неориентированными.

В качестве литерала было удобно использовать структуру $std :: pair$, так как литерал состоит из двух частей, к тому же такие пары легко сравнивать между собой.

Графовые зависимости представлены в виде класса Gfd, которые содержат в себе граф типа $graph\_t$, а также два множества: посылку и заключение, которые представляют собой списки литералов.

Для алгоритма были реализованы несколько вспомогательных функций:

- $GenerateLiterals()$, которая позволяет по входящему паттерну построить список литералов с помощью вышеописанной стуктуры литералов, которые возможны на данном паттерне.

- $Supp()$, позволяющая считать метрику частоты встречаемости для паттерна или зависимости, реализация которой была подробно описана в разделе “Входные параметры”.

- $AddEdge()$, которая генерирует все возможные паттерны по данному, имеющие хотя бы одно вложение в граф, путём добавления одного ребра.

- $AddVertex()$ аналогична функции $AddEdge()$, за исключением того, что она создаёт новые паттерны через добавление новой вершины к паттерну.

Кроме них была реализована функция $Initialize()$, проводящая работу, описанную в разделе “Предобработка”.

Функция, выполняющая основную работу, состоит из большого цикла, который выполняется, пока множество сгенерированных на прошлой итерации паттернов непусто, или достигнуто максимальное число итераций k. В теле цикла сначала вызывается функция

$HorizontalSpawn()$, производящая правила литералов для каждого паттерна, генерирующая графовые зависимости, а затем проверяющая их выполнимость. После её вызова происходит генерация паттернов для следующей итерации с помощью функций $AddEdge()$ и $AddVertex()$, и выполняется фильтрация тех паттернов, чьё количество вложений меньше входного параметра σ через вызов функции $FilterSupp()$.

5.2. Python bindings

Чтобы можно было выполнять C++ код алгоритма поиска функциональных зависимостей в графах, была использована библиотека pybind11[2]. Это простая библиотека для преобразования типов данных между Python и C++. Она в основном применяется для создания привязок Python к уже существующему коду на C++.

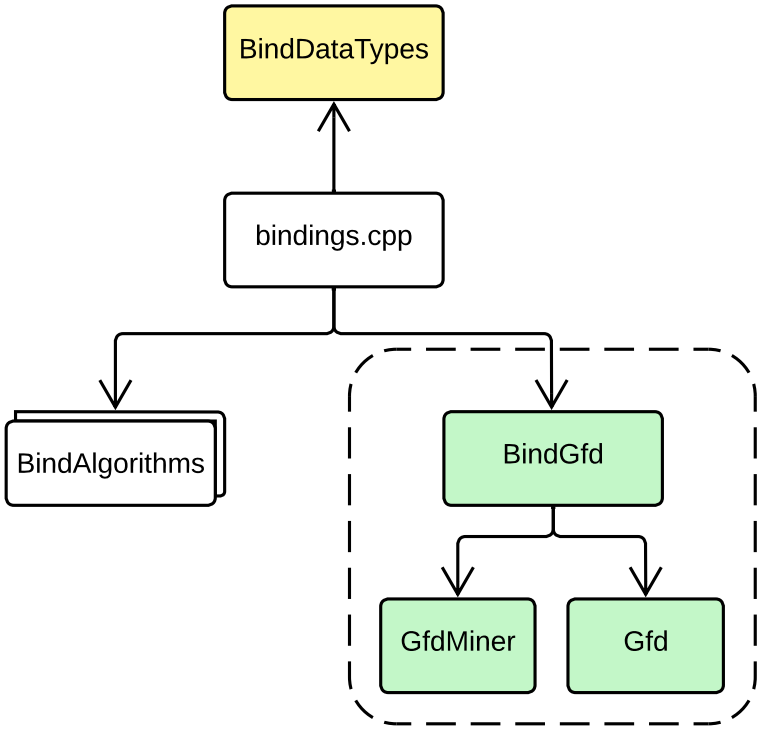


Рис. 8: Схема привязок алгоритмов Desbordante к Python.

На Рис. 8 показана схема привязки C++ кода к Python. Макрос

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

в файле bindings.cpp 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 алгоритм поиска 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 создан новый файл BindGfd, предоставляющий API для работы с алгоритмом поиска графовых зависимостей на Python.

5.3. Примеры

Для наглядной демонстрации пользователям сценариев использования алгоритма поиска графовых функциональных зависимостей дополнительно были написаны следующие примеры.

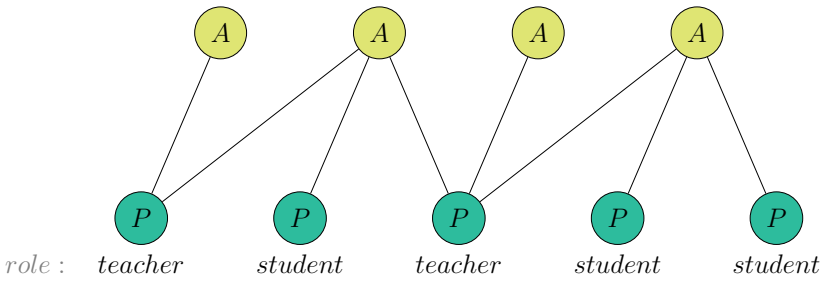
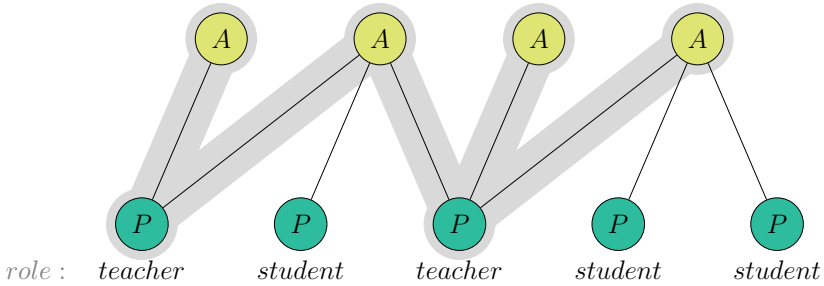
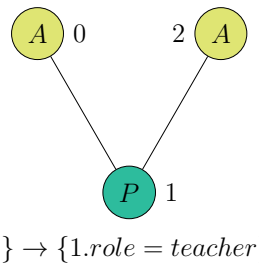


Рис. 9: Пример первого графа.



(a) Вложения паттерна зависимости в граф.



(b) Найденная зависимость.

Рис. 10: Результаты работы первого примера.

Рассмотрим граф, представленный на Рис. 9. Он описывает связи между научными статьями и их авторами. Вершины этого графа име-

ют 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. 10b. Обнаруженная зависимость может быть трактована как следующий факт: если у человека есть две опубликованные статьи, то он является учителем.

Для наглядности, пример также предоставляет все вложения паттерна найденной зависимости (Рис. 10a).

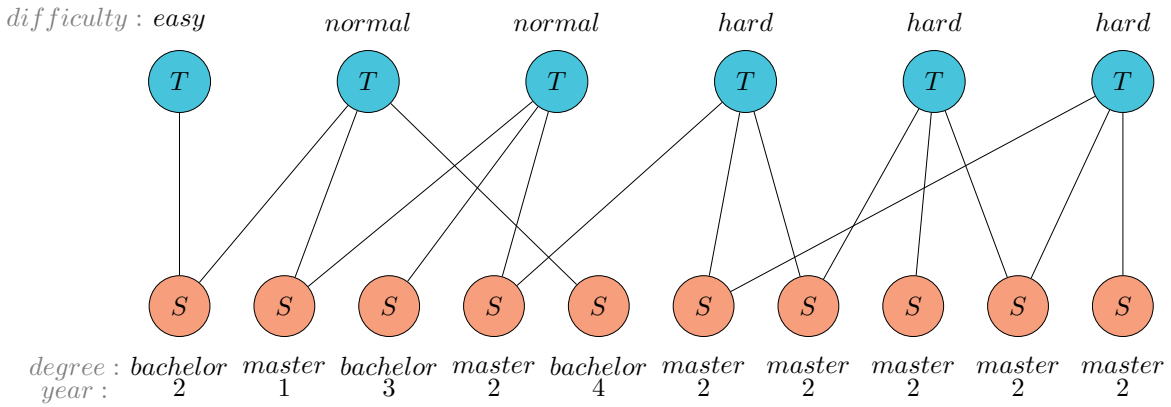
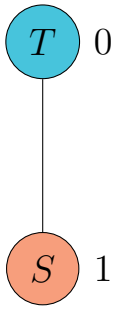


Рис. 11: Пример второго графа.

Рассмотрим теперь второй пример.

На Рис. 11 представлен граф с вершинами двух типов: T (*task*) и S (*student*). У вершин с меткой T есть два атрибута: *name* и *difficulty*. Они показывают название задачи и её сложность в общем понимании. У вершин типа S — три атрибута: *name*, *degree* и *year*. Они обозначают

имя студента, уровень получаемого образования и курс. Значения всех атрибутов подписаны рядом с соответствующими вершинами, кроме атрибута *name*, так как этот атрибут не несёт значимой информации.



$$\{0.difficulty = hard\} \rightarrow$$
$$\{1.degree = master \ \& \ 1.year = 2\}$$

Рис. 12: Результат работы алгоритма на втором графе.

Рассмотрим найденную графовую зависимость, представленную на Рис. 12. Она говорит о том, что над сложной задачей трудятся только магистранты второго курса.

Данные примеры были успешно протестированы с помощью Python, и были получены ожидаемые результаты.

5.4. Python CLI

Для взаимодействия с python-модулем desbordante используется модуль Click[3]. Это инструмент, который позволяет создавать интерфейсы командной строки.

Click выделяется среди аналогичных инструментов тем, что он предоставляет возможность легко и быстро создавать опции командной строки с использованием минимального количества кода. Кроме того, он предлагает инструменты для настройки и обеспечивает лёгкое масштабирование кода.

Командная строка desbordante имеет несколько основных опций, которые необходимы для запуска алгоритмов. Опция help предоставляет информацию о том, как использовать командную строку. Опции task

[3]https://click.palletsprojects.com/en/stable/ (дата обращения 30.05.2025)

и algo отвечают за задачу и конкретный алгоритм, который нужно запустить. Важно отметить, что одну и ту же задачу могут выполнять несколько алгоритмов. Например, для валидации графовых зависимостей доступны наивный, базовый и эффективный алгоритмы проверки графовых зависимостей [8].

Каждый алгоритм имеет свой уникальный набор опций. В случае алгоритма поиска графовых зависимостей этот набор включает путь к dot-файлу, содержащему представление графа, а также целые числа k и σ, которые являются параметрами алгоритма. Набор опций создаётся с помощью декоратора.

Пример вызова алгоритма через Python консоль:

```
$ desbordante --task=gfd_mining --algo=gfd_miner
--graph=examples/graph.dot --gfd_k=3 --gfd_sigma=10
```

5.5. Тестирование

Для апробации алгоритма было добавлено 5 тестов на небольших графах в среднем имеющих 13 вершин и 19 рёбер.

Тесты направлены на:

- Выявление минимальных графовых зависимостей.
- Обработку графов, не имеющих зависимостей.
- Получение графовых зависимостей с заключениями, имеющими более одного литерала.
- Корректную работу на синтетических данных.

6. Улучшение алгоритма поиска

После написания первоначального варианта алгоритма была проведена его проверка и улучшение на основе замечаний и предложений от рецензентов кода.

6.1. 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 соответствующие 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.

6.2. 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.

6.3. Ускорение алгоритма

После замеров производительности алгоритма было принято решение начать работу по ускорению кода, так как на небольших искусственных графах время работы алгоритма было достаточно большим.

Для этого был построен Flame Graph, изображённый на Рис. 13, с помощью инструмента профилирования кода perf.

Отсюда видно, что бóльшую часть времени работают функции, относящиеся к проверке сгенерированных правил. Соответственно, возникает гипотеза о том, что правил генерируется слишком много.

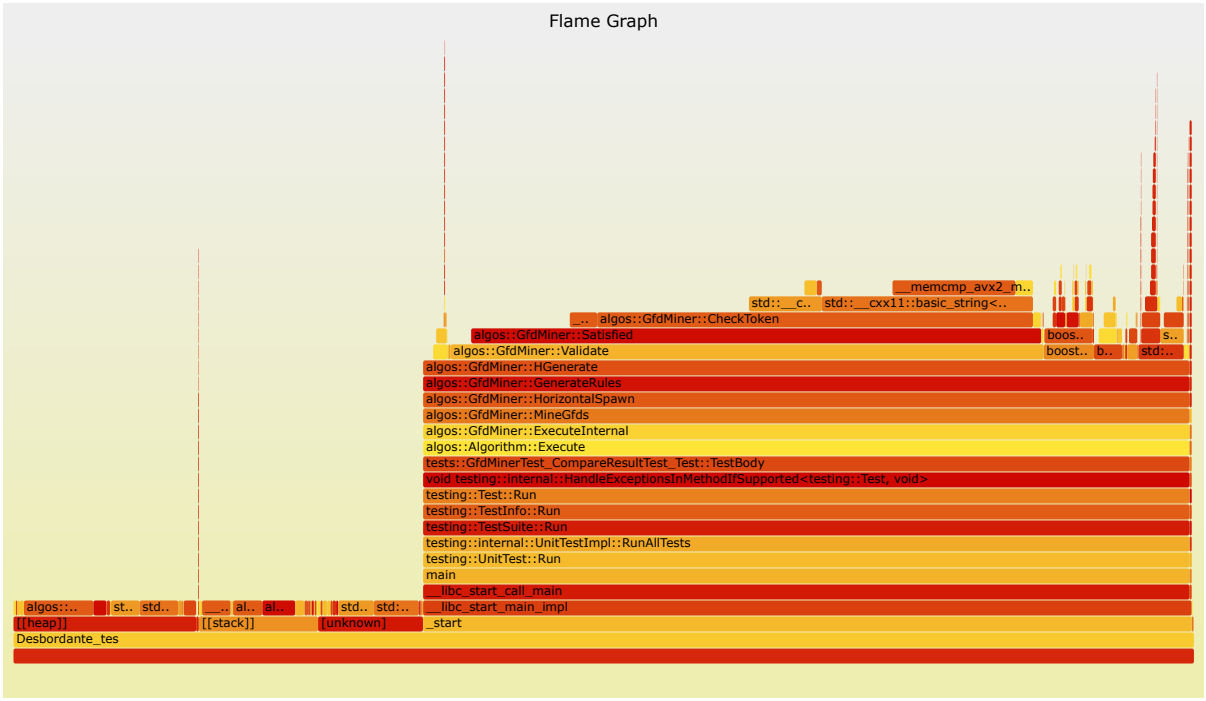


Рис. 13: Flame Graph, полученный в результате профилирования.

Первым делом было предложено пересмотреть подход к генерации литералов. С помощью структуры для генерации литералов, описанной в разделе “Предобработка”, можно упростить генерацию переменных литералов. Пусть у нас есть литерал $i.A = j.B$. Необходимо проверить, имеет ли смысл включать его в результирующую выборку для дальнейшей обработки. Если множество значений атрибута A не пересекается с множеством значений атрибута B, то этот литерал не может быть выполнен ни при каких обстоятельствах даже в теории. Соответственно, с помощью этой дополнительной проверки можно избежать лишних литералов при генерации.

Для анализа существующих подходов к оптимизации генерации правил была изучена статья, описывающая алгоритм поиска графовых дифференциальных зависимостей — структур, являющихся обобщением графовых функциональных зависимостей [1].

В ней рассматривается метод, использующий решётку атрибутов. Это структура данных, представляющая собой дерево с направленными рёбрами. В корне дерева содержится узел, соответствующий пустому множеству (нулевой слой $L0$). Первый слой $L1$ состоит из узлов, содер-

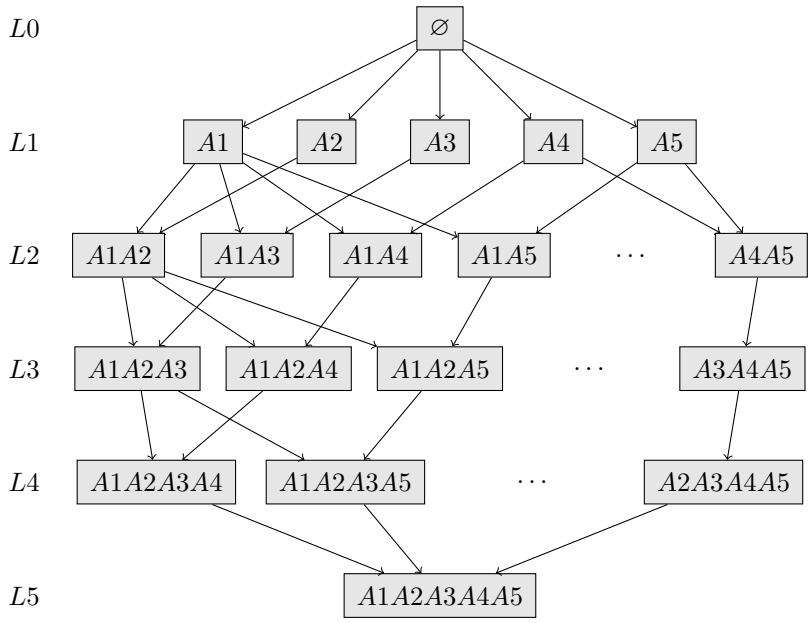


Рис. 14: Пример решётки атрибутов.

жащих 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. 14 изображена решётка атрибутов для пяти атрибутов.

Под алгоритм поиска графовых функциональных зависимостей эта структура может быть адаптирована следующим образом: вместо атрибутов можно рассматривать литералы.

Тогда для генерации правил нужно обойти решётку сверху-вниз слева-направо путём обхода рёбер. Пусть, рассматриваем ребро (A, B), где A — множество литералов узла-предка, а B — множество литералов узла-потомка. Правило генерируется следующим образом: в правой части остаётся литерал из $B \setminus A$ (такое множество будет всегда одноэлементным по построению решётки). Левая часть равна 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 GFD, и записываются в ответ. Чтобы легко считать эту мет-

рику, перед выполнением генерации целесообразно заранее подсчитать её для каждого литерала. Тогда для множеств литералов она будет просчитана тривиальным способом — как мощность пересечения множеств.

Если удалось сгенерировать зависимость, то алгоритм запоминает литерал, который задействовался в правой части, и далее происходит обрезка пространства литералов путём удаления всех таких множеств, которые содержат в себе этот литерал.

В результате работы был разработан Алгоритм 1.

Algorithm 1 RulesGenerator

Input: Паттерн Q, множество литералов $literals$.
Output: Правила, которые выполнены для этого паттерна.

```
1:  result := ∅, Ω := ∅, current := {∅}
2:  while |current| > 0 do
3:      new := ∅
4:      for lhs ∈ current and rhs ∈ literals \ lhs do
5:          if {lhs → rhs} is forbidden or {lhs ∪ X → rhs} ∈ result then
6:              continue
7:          if {lhs → rhs} satisfied then
8:              Ω := Ω ∪ {lhs ∪ {rhs}}
9:              result := result ∪ {lhs → rhs}
10:     for lhs ∈ current and l ∈ literals do
11:         LHS := lhs ∪ {l}
12:         if ∃ ω ∈ Ω : ω ⊆ LHS then
13:             continue
14:         if frequency(LHS) ≥ σ then
15:             new := new ∪ {LHS}
16:         else
17:             Ω := Ω ∪ {LHS}
18:     current := new
19: return result
```

Однако помимо описанных действий были выведены ещё несколько эвристик, которые существенно ускорят выполнение обхода.

Вместо того, чтобы генерировать всё дерево, а потом урезать ненужные узлы, можно генерировать поуровнево новые множества литералов “на ходу” на основе предыдущего уровня (строки 2–3 и 18).

Строки 4–9 описывают генерацию всевозможных правил на текущем уровне и их обход. В строке 5 происходит проверка на запрещённость, так как правило может быть запрещено в том случае, если для данного паттерна уже найдена зависимость, содержащая в себе паттерн-подграф текущего паттерна. Этим обеспечивается минимальность зависимостей. А также проверка на существование более сильного правила, ведь в таком случае очевидно, что текущее правило будет выполнено и дальнейшая проверка бессмысленна. Строки 7–8 содержат проверку правила на выполнимость с помощью Леммы 1, в случае успеха добавляем правило в результат, а также объединение левой и правой части в множество запрещённых узлов Ω. В левой части не может быть литералов, зависящих друг от друга.

Второй цикл (строки 10–17) генерирует узлы для следующей итерации. Для этого в строке 11 добавляется один литерал к уже существующему узлу. Далее проверяется, содержит ли сгенерированный узел в себе запрещённые узлы (строка 12).

Если найдётся узел, для которого метрика частоты будет меньше, чем σ, то очевидно, что всех его потомков дерева можно исключить из рассмотрения (строки 14–17).

Таблица 2: Сравнение производительности двух версий алгоритма.

Датасет	$\|V\| + \|E\|$	Время работы (мс)	
		Базовый	Улучшенный
Blogs	25	15±0	<1
Channels	32	33±0	<1
Movies	19	331±4	6±0
Symbols	63	2599±13	2±0
Shapes	20	1297±8	<1
Всего	**159**	**4276±23**	**9±0**

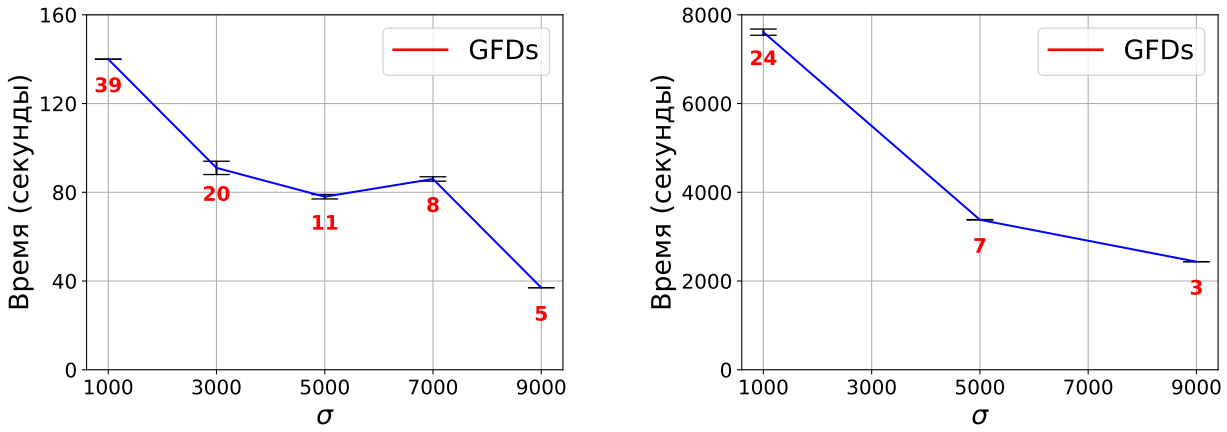
После реализации предложенного алгоритма генерации правил и замене прежнего метода на текущий в первой базовой версии алгоритма поиска графовых функциональных зависимостей получилась новая ускоренная версия алгоритма. Были произведены эксперименты, направленные на сравнение времени работы этих двух версий. Результаты

указаны в Таблице 2.

Как видно улучшенный алгоритм работает в среднем в 420 раз быстрее.

7. Результаты

Программная и аппаратная конфигурация машины, на которой проводились эксперименты: 12th Gen Intel(R) Core(TM) i5-12400F, 16GB RAM, `x86_64`, 22.04.1-Ubuntu.



(a) MOOC. (b) Hyperlink Network.

Рис. 15: Результаты экспериментов.

Для тестирования были использованы датасеты MOOC User Action Dataset[4] и Reddit Hyperlink Network[5]. Данные были модифицированы в соответствии с требованиями к входным данным алгоритма, а именно:

- Переконвертированы в формат DOT.
- Были добавлены новые рёбра и вершины, так как в рассматриваемых примерах атрибутами обладают рёбра, а не вершины.

В результате получились два датасета, общее количество рёбер и вершин которых равно соответственно 1 242 391 и 1 769 856. В качестве параметра k использовалось значение 2, так как зависимости, имеющие более двух вершин в паттерне, составляют менее 1% от числа всех графовых зависимостей [2]. Результаты тестирования приведены на Рис. 15a и Рис. 15b. Красным выделено количество зависимостей, которые нашлись алгоритмом.

[4]https://snap.stanford.edu/data/act-mooc.html (дата обращения 28.05.2025)
[5]https://snap.stanford.edu/data/soc-RedditHyperlinks.html (дата обращения 28.05.2025)

Заключение

Результаты работы:

- Выполнен обзор и реализован алгоритм поиска графовых функциональных зависимостей.
- Разработана подсистема, позволяющая запускать реализованный алгоритм из скриптов, написанных на языке программирования Python.
- Созданы скрипты-примеры работы реализованного алгоритма на языке программирования Python.
- Обеспечена возможность запускать реализованный алгоритм через консоль путём реализации соответствующей подсистемы.
- Произведён анализ и разработан ускоренный алгоритм поиска графовых функциональных зависимостей.
- Произведено тестирование производительности ускоренного алгоритма поиска графовых функциональных зависимостей.

Код работы доступен на GitHub[6][7]. Ссылка на код, обеспечивающий возможность запускать реализованный алгоритм через консоль[8].

[6]https://github.com/Desbordante/desbordante-core/pull/465 (дата обращения 19.04.2025)
[7]https://github.com/Desbordante/desbordante-core/pull/568 (дата обращения 29.05.2025)
[8]https://github.com/Desbordante/desbordante-cli/pull/5 (дата обращения 02.10.2024)

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] 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).

[8] Черников Антон. Реализация эффективного алгоритма проверки графовых функциональных зависимостей в платформе Desbor-

dante. — 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).