Санкт-Петербургский государственный университет

Кафедра информационно-аналитических систем

Группа 22.Б08-мм

Улучшение алгоритма верификации запрещающих ограничений в проекте “Desbordante”

Аносов Павел Игоревич

Отчёт по учебной практике
в форме «Решение»

Научный руководитель:
асс. кафедры ИАС Г. А. Чернышев

Санкт-Петербург
2025

Оглавление

Введение

Desbordante [3] — это высокоэффективный инструмент для наукоёмкого профилирования данных. Такой подход направлен на выявление сложных паттернов и зависимостей, которые невозможно обнаружить стандартными методами профилирования.

Одним из ключевых типов таких зависимостей являются запрещающие ограничения (Denial Constraints, DC) [1]. Эти ограничения обобщают функциональные и условные функциональные зависимости, зависимости порядка и другие, которые не всегда достаточно гибки для описания сложных взаимосвязей в данных. Запрещающие ограничения (ЗО) определяют комбинации значений атрибутов, которые не должны встречаться в наборе данных. Это позволяет выявлять различные ошибки и несоответствия в данных, такие как пропущенные значения, дубликаты и прочие аномалии.

Существуют алгоритмы для решения следующих задач, связанных с ЗО:

1. Верификация ЗО — проверка выполнения заданного ограничения на заданных данных.

2. Поиск ЗО — выявление конкретных ограничений, которые верны на наборе данных.

Данная работа является продолжением работы [8], в которой была добавлена проверка выполнимости переменных запрещающих ограничений (variable DC), а также различные оптимизации алгоритма для частных случаев. В данной работе мы продолжим развивать тему запрещающих ограничений, а именно: реализуем верификацию ЗО, содержащих предикаты с константными значениями — константных ЗО (constant DC), а также обеспечим поиск кортежей, нарушающих зависимость.

1 Постановка задачи

Данная работа является продолжением работы [8], в которой отсутствовала поддержка константных ЗО.

Целью работы является реализация алгоритма для эффективной верификации запрещающих ограничений, включающих в себя предикаты с константными значениями, а также нахождение кортежей, нарушающих зависимости. Для достижения этой цели были поставлены следующие задачи:

1. Добавить проверку запрещающих ограничений, которые включают в себя предикаты с константными значениями;
2. Сделать возможным выявление кортежей, нарушающих зависимость;
3. Реализовать алгоритм минимизации набора кортежей, нарушающих зависимость, без которых ЗО будет выполняться;
4. Выполнить нагрузочное тестирование алгоритмов верификации и минимизации нарушений, а также провести анализ результатов.

2 Обзор

Данный раздел повторяет соответствующий раздел моего предыдущего отчета [8].

Несмотря на то, что ЗО являются довольно молодой научной областью, тем не менее, в ней накопилось немало работ, описывающих алгоритмы верификации и поиска ЗО. Алгоритмы поиска в данной работе затронуты не будут, более подробную информацию и сравнительную характеристику о них можно найти в работе Ивана Морозко[1]. В данной разделе мы рассмотрим лишь алгоритмы верификации, они представляют большую ценность, так как позволяют проверять различные гипотезы о данных.

Перед началом работы необходимо было провести анализ предметной области для выявления алгоритма верификации ЗО который был бы наиболее эффективным и не требующим большого объема инфраструктуры, которую необходимо реализовать для его поддержки в программном коде.

В ходе изучения предметной области были найдены четыре работы, описывающие алгоритмы верификации ЗО, но у каждого из них были свои преимущества и недостатки.

1. Rapidash Verification [7] — поддержка только переменных гетерогенных и гомогенных ЗО (variable heterogeneous and homogeneous DC).

2. One-Pass Inconsistency Detection Algorithms [5] — поддержка переменных и константных ЗО (variable and constant DC), но только гомогенных (homogeneous).

3. FAst Constraint-based Error DeTector (FACET) [6] — поддержка только переменных гетерогенных и гомогенных ЗО (variable heterogeneous and homogeneous DC).

[1] https://github.com/Desbordante/desbordante-core/blob/main/docs/papers/FastADC%20-%20Ivan%20Morozko%20-%202024%20autumn.pdf

4. VioFinder [4] — поддержка только переменных гетерогенных и гомогенных ЗО (variable heterogeneous and homogeneous DC).

Определения константных и переменных ЗО будут приведены в следующем разделе.

Также были изучены работы, описывающие алгоритмы поиска, так как зачастую такие алгоритмы тоже применяют верификацию. К сожалению, все алгоритмы либо не применяли верификацию в явном виде, либо предполагали построение объемной инфраструктуры, что нам не подходит.

После изучения доступных вариантов алгоритмов верификации основой работы стала статья [7], так как, во-первых, гетерогенные ЗО представляют наибольшую ценность (такими ЗО, например, являются функциональные зависимости и зависимости порядка). А во-вторых, по заявлению авторов, алгоритм быстрее алгоритма FACET, представленного в статье [6], который в свою очередь быстрее алгоритма VioFinder, представленного в статье [4].

3 Предварительные сведения

Данные определения повторяют соответствующий раздел моего предыдущего отчета [8]. Перед тем, как начать реализацию алгоритма проверки константных ЗО, необходимо ввести основные определения и понятия. Подавляющее большинство определений взяты из статьи [7], исключения будут указаны явно.

Таблица 1: Основные понятия.

Понятие	**Значение**
R	таблица с исходными данными
$vars(R)$	конечное множество атрибутов (колонок) таблицы
$\|R\|$	количество строк в R
$t,\ s$	кортежи в R
$t.A$	значение атрибута A в кортеже t
предикат p	выражение вида: $s.A\ op\ t.B$, где $s, t \in R,\ A,\ B \in vars(R)$ и $op \in \{=\ ,\ \neq\ ,\ \geq\ ,\ >\ ,\ \leq\ ,\ <\}$
ЗО φ	конъюнкция предикатов: $\forall t, s \in R,\ t \neq s : \neg(p_1 \wedge \cdots \wedge p_m)$
$vars_{op}(\varphi)$	множество атрибутов гомогенного ЗО, для которых есть предикат с оператором op в ЗО

Определение 1. *Пара кортежей (s, t) будет считаться нарушением (violation), если все предикаты в φ верны.*

Определение 2. *φ удерживается (holds) на R, если нет нарушений.*

Определение 3. *Предикат (predicate) называется гомогенным (homogeneous), если он представим в виде $s.A$ op $t.A$ или $s.A$ op $a.B$*

Определение 4. *Предикат называется гетерогенным (heterogeneous), если его можно представить в виде: $s.A$ op $t.B$.*

Определение 5. *Гомогенный предикат вида $s.A$ op $t.A$ называется строчным гомогенным (row-level homogeneous) предикатом.*

Определение 6. *Согласно [2] ЗО называется константной (constant DC), если она содержит предикат вида: $s.A$ op c, где c — константа.*

Определение 7. *Согласно [2] ЗО называется переменной (variable DC), если $\forall p_i$ представим в виде $s.A$ op $t.B$.*

Определение 8. *Гомогенный предикат вида $s.A$ op $s.B$ называется колоночным гомогенным (column-level homogeneous) предикатом.*

Так как большинство ЗО (зависимости порядка, функциональные зависимости и пр.) содержат только строчные гомогенные предикаты, то для удобства будем называть гомогенным ЗО такое ЗО, которая содержит только строчные гомогенные предикаты.

Определение 9. *Если ЗО Содержит только строчные гомогенные предикаты, то такое ЗО — гомогенное (homogeneous DC).*

Определение 10. *Если ЗО содержит и строчные, и колоночные гомогенные предикаты, то такое ЗО — смешанная гомогенная (mixed homogeneous DC).*

Определение 11. *Гетерогенное ЗО (heterogeneous DC) — ЗО, содержащая любые типы предикатов.*

Не умаляя общности, будем считать, что каждая колонка в R находится в максимум одном предикате гомогенного ЗО.

Пример. В таблице 2 приведен набор данных, в котором содержится информация о налоговой ставке для жителей разных штатов США. Вот некоторые правила, которые выполнятся на данном наборе данных: (1) SSN — потенциальный ключ, (2) $Zip \rightarrow State$ — функциональная зависимость и (3) для всех жителей одного штата у человека с большей заработной платой больше налоговая ставка.

Таблица 2: Налоговые ставки для жителей разных штатов США.

	SSN	Zip	Salary	FedTaxRate	State
t1	100	10108	3000	20%	NewYork
t2	101	53703	5000	15%	Wisconsin
t3	102	53703	6000	20%	Wisconsin
t4	103	53703	4000	10%	Wisconsin

Все вышеперечисленные правила можно записать в терминах запрещающих ограничений следующим образом:

1. $\varphi_1 : \forall t, s \in R,\ t \neq s : \neg(s.SSN = t.SSN)$

2. $\varphi_2 : \forall t, s \in R,\ t \neq s : \neg(s.Zip = t.Zip \wedge s.State \neq t.State)$

3. $\varphi_3 : \forall t, s \in R,\ t \neq s : \neg(s.State = t.State \wedge s.Salary < t.Salary \wedge s.FedTaxRate > t.FedTaxRate)$

Все вышеперечисленные ЗО являются гомогенными, т.е. содержат строчные гомогенные предикаты. Также приведем пример гетерогенного ЗО: $\varphi_4 : \forall t, s \in R,\ t \neq s : \neg(s.Salary < t.FedTaxRate)$. Отметим, что $vars_{=}(\varphi_3) = \{State\}$, $vars_{<}(\varphi_3) = \{Salary\}$ и $vars_{>}(\varphi_3) = \{FedTaxRate\}$.

4 Алгоритм

Как уже было упомянуто, реализация основного алгоритма верификации запрещающих ограничений была представлена в предыдущей работе [8]. В данном разделе же будут описаны алгоритм проверки ЗО, содержащих предикаты с константными значениями, а также алгоритмы выявления нарушающих кортежей — исключений, и их минимизация.

Не умаляя общности, отметим, что алгоритмы минимизации и верификации константных ЗО работают с запрещающими ограничениями, не содержащими предикаты с оператором не равно и гетерогенные предикаты с равенством. Эти случаи сводятся к проверке ЗО, не содержащих таких предикатов следующим способом.

Любой предикат $s.A \neq t.B$ может быть записан в виде: $(s.A < t.B) \vee (s.A > t.B)$. Заметим, что $\neg(\varphi \wedge s.A \neq t.B) \iff \neg(\varphi \wedge s.A < t.B) \wedge \neg(\varphi \wedge s.A > t.B)$. Таким образом, ЗО, содержащее ℓ предикатов, может быть записано в равносильной форме как конъюнкция 2^ℓ ЗО, не содержащих операторов не равно.

Гетерогенный предикат с равенством $s.C = t.D$ может быть записан в виде: $s.C \leq t.D \wedge s.C \geq t.D$. Подробное доказательство корректности и алгоритм поиска ортогональных диапазонов представлены в оригинальной статье.

4.1 Константные ЗО

Как уже было упомянуто, константное ЗО — это ЗО, которое содержит хотя бы один предикат с константным значением вида: $s.A\ op\ c$ $(t.A\ op\ c)$, где $op \in \{=, \neq, \geq, >, \leq, <\}$, а c — константа.

Для того, чтобы решить проблему верификации константных ЗО, напомним алгоритм верификации смешанных гомогенных ЗО (mixed homogeneous DC), которые могут содержать предикаты следующего вида: $t.A\ op\ t.B$, $s.A\ op\ s.B$ и $s.A\ op\ t.A$.

Пусть $\forall s, t : \neg\varphi(s, t)$ — ЗО. Сначала переписывается φ в следующем виде: $\varphi_S(s) \wedge \varphi_T(t) \wedge \varphi_{ST}(s, t)$, где φ_S содержит все предикаты, которые

содержат только s (и не t), φ_T содержит все предикаты, которые содержат только t (и не s), и φ_{ST}, которое содержит и s, и t. Тогда ЗО $\forall s, t : \neg\varphi$ может быть записано в равносильной форме:

$$\begin{aligned}\forall s, t : \neg\varphi &\Leftrightarrow \forall s, t : \neg\left(\varphi_S(s) \wedge \varphi_T(t)\right) \vee \neg\varphi_{ST}(s, t) \\ &\Leftrightarrow \forall s, t : \left(\varphi_S(s) \wedge \varphi_T(t)\right) \Rightarrow \neg\varphi_{ST}(s, t) \\ &\Leftrightarrow \forall s \in \mathbf{S} : \forall t \in \mathbf{T} : \neg\varphi_{ST}(s, t),\end{aligned}$$

где $\mathbf{S}$ — множество всех кортежей в $\mathbf{R}$, таких что φ_S верно, и $\mathbf{T}$ — множество всех кортежей в $\mathbf{R}$, таких что φ_T верно. Отметим, что $\mathbf{S}$ и $\mathbf{T}$ могут пересекаться.

Вводятся две структуры данных для ортогонального поиска (такие же, как и структура H в алгоритме 1) $H_{\mathbf{S}}$ и $H_{\mathbf{T}}$ для точек в $\mathbf{S}$ и $\mathbf{T}$ соответственно. Для каждого кортежа (то есть точки) $q \in \mathbf{R}$, сначала проверяется, принадлежит ли она $\mathbf{S}$ и $\mathbf{T}$ (то есть просто проверяется выполнимость набора предикатов на кортеже).

1. Если $q \in \mathbf{S}$, то выполняется ортогональный поиск по $H_{\mathbf{T}}$ чтобы найти какую-нибудь точку r такую, что $\varphi_{ST}(q, r)$ верно. Если находится такая точка, то зависимость не выполняется и алгоритм возвращает $false$. Если же ортогональный поиск вернул $false$ (не нашлось ни одной точки в ортогональном диапазоне, образующей нарушение с текущей точкой), то выполняется вставка q в $H_{\mathbf{S}}$.

2. Аналогично, если $q \in \mathbf{T}$, выполняется ортогональный поиск по H_{S}, чтобы найти точку r, такую, что $\varphi_{ST}(r, q)$ верно. Если такой точки нет, выполняется вставка q в H_{T}. Иначе алгоритм вернет $false$.

Алгоритм проверки константных ЗО не отличается от алгоритма проверки смешанных гомогенных ЗО, необходимо лишь отметить, что любой константный предикат с s или t входит соответственно в $\varphi_S(s)$ или $\varphi_T(t)$, так как в нём участвуют только значение из одного кортежа и константа.

Так как любая константная ЗО — это общий случай смешанной гомогенной ЗО, то данного алгоритма достаточно для верификации любых

константных ЗО.

4.2 Алгоритм нахождения исключений

В данной части будет описана модификация алгоритма, представленного в статье [7].

Исключения — важный аспект в работе алгоритма, который позволяет получить больше информации о данных: не только факт того, выполняется или не выполняется зависимость, но и те записи, на которых она не выполняется.

В случае когда в ЗО встречаются кортежи s и t, то под исключением подразумеваются такие пары записей таблицы, которые нарушают зависимость. Если же ЗО содержит единственный кортеж: только s, либо t, то исключением будет являться единственное число — номер записи из таблицы.

Например: пары чисел (1, 4), (1, 3) соответствуют тому, что записи с номерами 1 и 4, 1 и 3 нарушают зависимость, соответственно, достаточно убрать первую, для того чтобы зависимость выполнялась.

Алгоритм 1 описывает детали поиска нарушений ЗО ϕ на заданной таблице R. Красным цветом выделены изменённые строки исходного алгоритма. На вход алгоритма подается ЗО, не содержащее предикаты с оператором не равно и гетерогенные предикаты с равенством.

Описанный в предыдущей работе [8] алгоритм не поддерживал возможность выявления исключений во время обработки входных данных: с первым найденным нарушением алгоритм завершал работу. Для того чтобы добавить такую возможность, необходимо модифицировать представленный алгоритм.

Идея модификации следующая: при нахождении точек в дереве при ортогональном поиске будем не завершать работу, а помечать найденные точки как нарушения, а затем добавлять обрабатываемую запись в дерево. Таким образом для каждого кортежа мы найдём все записи, с которыми он образует нарушение.

При обработке смешанной гомогенной ЗО воспользуемся тем же при-

ёмом, что и в основном алгоритме: будем добавлять точки в соответствующее дерево даже после того как нашли нарушения.

Также отметим, что не получится применить оптимизацию описанную в оригинальной статье в случае ЗО, содержащего только строковые гомогенные предикаты с равенством и единственный предикат с неравенством.

В данном случае оптимизация не предполагает хранение отсмотренных записей, а только обновление минимума и максимума по колонке с неравенством. Таким образом, информация об уже просмотренных записях не хранится. Поэтому в случае нахождения нарушения не получится формировать пары с предшествующими записями. Соответственно, для того, чтобы можно собирать нарушения в случае такого ЗО, необходимо просто отключить данную оптимизацию (для этого, в частности, и предполагается добавление поля do_collect_violations_).

Algorithm 1: Поиск нарушений ЗО

Input: Таблица R, ЗО ϕ

Output: Список пар записей таблицы R

k $\leftarrow$| vars(ϕ)\ vars $_=$(ϕ) |

$H \leftarrow$ пустая хэш-таблица

$V \leftarrow$ пустой список

foreach $t \in R$ **do**

  $v \leftarrow \pi_{\text{vars } =(\phi)}(t)$

  if $v \notin H$ **then**

    $H[v] \leftarrow$ new OrthogonalRangeSearch()

  L, U, $\mathbf{L'}$, $\mathbf{U'} \leftarrow$ CreateBothSearchRanges (t, ϕ)

  SearchRes $\leftarrow H[v]$.RangeSearch $(\mathbf{L}, \mathbf{U})$

  InvSearchRes $\leftarrow H[v]$.RangeSearch $(\mathbf{L'}, \mathbf{U'})$

  $H[v]$.insert $\left(\pi_{\text{vars } (\phi)\setminus \text{vars}=(\phi)}(t)\right)$

  foreach $p \in SearchRes \cup InvSearchRes$ **do**

    $V \leftarrow V \cup \{p, t\}$

return V

Procedure `CreateBothSearchRanges`(r, ϕ)

  foreach *предикат с неравенством* $s.C$ *op* $t.D \in \phi$ **do**

    if *op is* $\leq$ *or* $\leq$ **then**

      $U.C \leftarrow min\{U.C, r.D\}$

      $L'.D \leftarrow max\{L'.D, r.C\}$

    if *op is* $>$ *or* $\geq$ **then**

      $L.C \leftarrow max\{L.C, r.D\}$

      $U'.D \leftarrow min\{U'.D, r.C\}$

    return $\mathbf{L}, \mathbf{U}, \mathbf{L'}, \mathbf{U'}$

Procedure `InvertRange(L, U)`

  U$' \leftarrow$ L, L$' \leftarrow$ U

  изменить $-\infty$ на ∞ и ∞ на $-\infty$ в $\mathbf{U'}$ и $\mathbf{L'}$ соответственно

  return L$'$, U$'$

4.3 Алгоритм минимизации исключений

Верификация ЗО позволяет проверять различные гипотезы на данных, зачастую содержащих различные ошибки. В предыдущем разделе нами уже был предложен алгоритм нахождения исключений.

Одним из возможных применений таких ограничений является очистка данных. В процессе очистки можно не только выявлять, но и модифицировать, удалять ошибочные записи, добавлять новые записи, так, чтобы привести данные в консистентное состояние. В данной работе мы предложим алгоритм, который только удаляет ошибочные записи.

Важной особенностью подобного алгоритма является его эффективность, под которой мы будем понимать минимизацию потери полезной информации.

Реализовать оптимальный алгоритм минимизации, удаляющий минимально возможное количество записей, не является целью данной работы. Вместо этого предлагается наивное решение, которое можно быстро разработать и которое является наглядным. Мотивацию такого решения хорошо отражает русская народная пословица: лучше синица в руках, чем журавль в небе. Хотя предложенное решение не является оптимальным, оно позволяет достичь приемлемых результатов и проиллюстрировать основные идеи.

Начнем с того, что мы хотим предложить такой алгоритм для результатов проверки ЗО, задействующей два кортежа s, t. Для ЗО, задействующей только один кортеж (только s или только t), работа подобного алгоритма тривиальна: все не удовлетворяющие ЗО записи просто удаляются.

Ниже представлен алгоритм минимизации. На вход алгоритма подается список пар чисел — результаты проверки ЗО.

1. Представим данные исключения в виде графа: каждая вершина — запись, каждая пара из списка исключений — ребро между вершинами;

2. Отсортируем вершины по количеству инцидентных им рёбер;

3. Удаляем вершину с наибольшим количеством инцидентных ей рёбер;

4. Продолжаем работу до тех пор, пока в графе не останется рёбер;

5. Удалённые вершины будут являться набором записей, которые нужно удалить, чтобы зависимость выполнялась.

На рисунке 2 приведён пример графа, для которого наивный алгоритм выдаст не наилучший результат. Результат работы алгоритма — список из удаленных вершин.

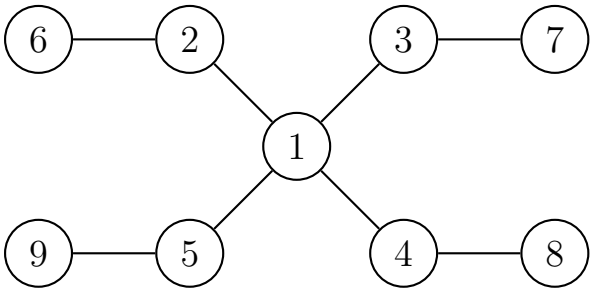
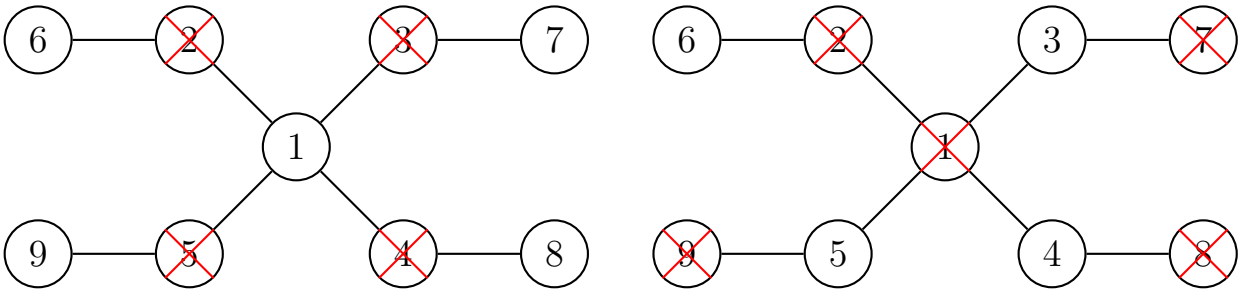


Рис. 1: Полный граф исключений



(a) Оптимальный алгоритм (b) Наивный алгоритм

Рис. 2: Результаты работы алгоритмов

Как видно из рисунка 2, наивный алгоритм удаляет большее количество вершин, что соответствует большему количеству удаленных записей в исходной таблице. Стоит отметить, что в наивном алгоритме возможны различные варианты удаления вершин. В нашем примере первой всегда удаляется вершина с номером 1, после чего алгоритм с равной вероятностью может выбрать любую из оставшихся вершин, поскольку все они имеют степень один. Однако, независимо от выбора,

после завершения работы алгоритма в графе останется четыре вершины.

Подробная работа алгоритма представлена в разделе 5.3.

5 Реализация

5.1 Поддержка константных ЗО

Алгоритм верификации смешанных гомогенных ЗО, представленный в статье [7], позволяет верифицировать и константные ЗО. Однако такой возможности в предыдущей работе [8] не было предоставлено. И для того чтобы добавить такую возможность в Desbordante, необходимо было реализовать соответствующую инфраструктуру, а именно, решить две задачи:

1. Добавить поддержку синтаксического разбора строк, задающих константные ЗО;

2. Приспособить уже существующий класс ColumnOperand для хранения константных значений.

Для выполнения первой задачи был создан отдельный класс, который отвечает за преобразование строки в список предикатов. При создании экземпляра класса DCParser ему передается строка, задающая ЗО, и объект типа ColumnLayoutRalationData, который представляет собой отношение и через который происходит доступ к именам колонок. Также для получения типа соответствующей колонки в конструктор класса передается объект типа std::vector<TypedColumnData>.

Чтобы пользователю было удобно, и не было необходимости искать специальные символы, обозначающие конъюнкцию ($\wedge$), класс DCParser предполагает разбиение строки по задаваемому разделителю — в данном случае это “ *and* ”.

Для выполнения второй задачи был модифицирован класс ColumnOperand: удалено поле is_first_tuple_ и добавлено опциональное поле std::optional<dc::Tuple> tuple_, которое задает кортеж s либо t. Это сделано для того, чтобы различать константные и переменные операнды: если предикат представляет собой константу, то и кортеж получить не получится. В ином случае поле содержит один из кортежей s, t. Поле type_ отвечает за тип операнда, val_ — за значение операнда. Если

операнд представляет собой переменную, то val_ хранит *nullptr*. Поле column_ содержит в себе указатель на заданную колонку. Если операнд константный, то и нет колонки, на которую он мог бы указывать, соответственно column_ содержит *nullptr*.

5.2 Поиск нарушений

Для того, чтобы добавить возможность выявлять нарушения, был модифицирован алгоритм верификации ЗО DCVerifier. Были добавлены новые поля: *violations_* и *do_collect_violations_*. Первое отвечает за сбор всех нарушений во время верификации ЗО, а второе поле — это задаваемая пользователем опция, которая позволяет включать или выключать сбор нарушений во время верификации ЗО. Это позволяет не тратить лишнее время, если нет необходимости в сборе нарушений.

Также были исправлены все методы, отвечающие за непосредственную проверку ЗО (ЗО сначала классифицируется, а затем проверяется подходящим методом). Для каждого метода (VerifyOneTuple, VerifyTwoTuples, VerifyMixed, VerifyOneInequality, VerifyAllEquality) была добавлена возможность выявлять нарушения.

Из класса DCVerifier также были удалены все методы, отвечающие за классификацию ЗО. Теперь это ответственность класса DC, в котором определен метод GetType, предоставляющий тип ЗО, для которого он вызывается.

Ниже для класса синтаксического разбора DCParser, алгоритма верификации DCVerifier, а также других вспомогательных классов приведена диаграмма 3.

Рис. 3: Диаграмма класса алгоритма и вспомогательных классов.

- Удаленный метод
- Добавленный класс/метод/поле;
- Отредактированный метод.

На диаграмме выделены изменения, внесённые в процессе работы: добавленные, удалённые и отредактированные элементы. Классы/методы/поля, не выделенные отдельным цветом, уже существовали на момент начала работы и не были изменены.

5.3 Алгоритм минимизации

В данном разделе приведена реализация алгоритма минимизации и пример его использования.

Пример использования. В строках 32–34 задаются исходные таблица и запрещающее ограничение. Далее в строках 37–41 загружаются данные из исходной таблицы и запускается алгоритм верификации. В строках 49–55 в качестве результата работы алгоритма формируется список пар чисел — исключения, а также запускается алгоритм минимизации. В качестве результата работы алгоритма минимизации в строке 58 формируется приблизительное решение задачи минимизации — список записей, которые нужно удалить, чтобы ЗО выполнялась.

Работа алгоритма минимизации. В строках 2–11 создается граф на основе переданных исключений. В строках 26–28 исполняется основной цикл алгоритма: удаляются вершины с наибольшим числом инцидентных ребер, пока в графе есть ребра. В строках 13–19 выбирается такая вершина и удаляется из графа.

example.py:

```
1  class DataCleaner:
2      def __init__(self, violations: List[Tuple]) -> None:
3          self.graph: Dict[int, List[int]] = defaultdict(list)
4          for v1, v2 in violations:
5              if v1 != v2:
6                  self.graph[v1].append(v2)
7                  self.graph[v2].append(v1)
8              else:
9                  self.graph[v1].append(v1)
10         self.nodes: List[int] = list(self.graph.keys())
11         self.removed_nodes: List[int] = []
12
13     def __remove_highest_degree_node(self) -> None:
14         max_key = max(self.graph.items(), key=lambda x: len(x[1]))[0]
15         for neighbour in self.graph[max_key]:
16             self.graph[neighbour].remove(max_key)
17         del self.graph[max_key]
18         self.nodes.remove(max_key)
19         self.removed_nodes.append(max_key)
```

```
20
21      # Check if the graph contains any edges
22      def __has_edges(self) -> bool:
23          return any(self.graph[node] for node in self.graph)
24
25      # Remove highest degree node while graph has edges
26      def clean(self) -> None:
27          while self.__has_edges():
28              self.__remove_highest_degree_node()
29
30
31
32  table = 'examples/datasets/taxes_2.csv'
33  dc = "!(t.State == s.State and t.Salary > s.Salary
34  and t.FedTaxRate < s.FedTaxRate)"
35
36  # Creating a verificator and loading data in algortihm
37  verificator = db.dc_verification.algorithms.Default()
38  verificator.load_data(table=(table, ';', False))
39
40  # Algorithm execution
41  verificator.execute(denial_constraint=dc, do_collect_violations=True)
42
43  result = verificator.dc_holds()
44
45  print("DC " + dc + " holds: " + str(result))
46  print("Average algo execution time: " + str(algo_time))
47
48  # Get list of all tuple pairs that violate the constraint
49  violations = verificator.get_violations()
50
51  # Creating data cleaner for minimization
52  cleaner = DataCleaner(violations)
53
54  # Execute minimization algorithm
55  cleaner.clean()
56
57  # Get minimum number of records that should be removed in order to make constraint hold
58  nodes = sorted(cleaner.removed_nodes)
59  print(f"Removed records: {", ".join(map(str, nodes))}")
```

6 Эксперимент

В данном разделе описаны результаты проведенного нагрузочного тестирования алгоритма верификации константных запрещающих ограничений и алгоритма минимизации.

В качестве набора данных для тестирования был выбран adult.csv (32 тысячи строк, 15 столбцов), так как в нём присутствуют данные и числовых типов, и строчных.

В экспериментах будут использованы две группы запрещающих ограничений: A и B. Группа A начинается с исходного ЗО A1, каждое следующее ЗО в группе формируется добавлением одного предиката к предыдущему. Группа B построена аналогично. Такой подход позволяет анализировать влияние количества предикатов на результаты тестирования.

Группа A:
ЗО A1: $\neg(s.0 = 30)$
ЗО A2: $\neg(s.0 = 30 \wedge s.1 = Private)$
ЗО A3: $\neg(s.0 = 30 \wedge s.1 = Private \wedge s.2 < 141297)$
ЗО A4: $\neg(s.0 = 30 \wedge s.1 = Private \wedge s.2 < 141297 \wedge s.3 = 11th)$
ЗО A5: $\neg(s.0 = 30 \wedge s.1 = Private \wedge s.2 < 141297 \wedge s.3 = 11th \ \wedge s.4 >= 10)$

Группа B:
ЗО B1: $\neg(s.0 <= 30 \wedge t.0 <= 17)$
ЗО B2: $\neg(s.0 <= 30 \wedge t.0 <= 17 \wedge s.1 = Private)$
ЗО B3: $\neg(s.0 <= 30 \wedge t.0 <= 17 \wedge s.1 = Private \wedge t.2 >= 287927)$
ЗО B4: $\neg(s.0 <= 30 \wedge t.0 <= 17 \wedge s.1 = Private \wedge t.2 >= 287927 \wedge s.3 = 10th)$
ЗО B5: $\neg(s.0 <= 30 \wedge t.0 <= 17 \wedge s.1 = Private \wedge t.2 >= 287927 \wedge s.3 = 10th \wedge t.4 = 11)$

В ЗО, приведенных выше, используется следующее обозначение: $s.i$, i — номер колонки.

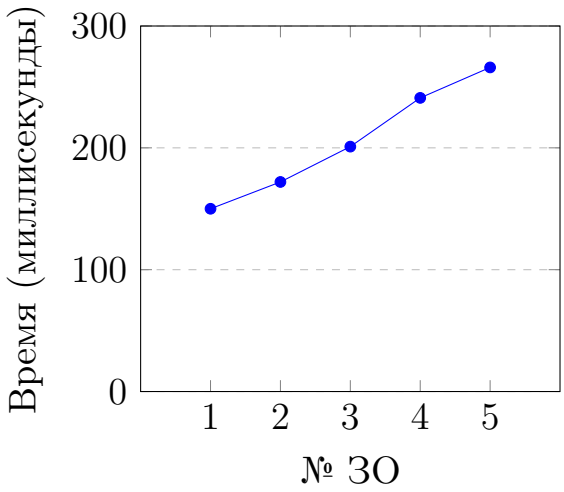
Данные группы имеют следующие различия. Первая группа включает в себя ЗО, содержащие только один кортеж s: эти ЗО затрагивают лишь одну запись в таблице. В то же время ЗО из второй группы содержат в себе оба кортежа s и t: данные ЗО включают в себя две записи в

таблице.

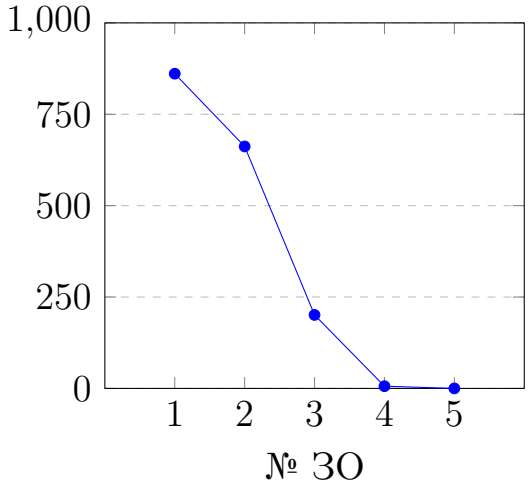
Наконец, отметим что итоговое время проверки включает только проверку выполнения алгоритма, без учёта времени загрузки данных.

6.1 Алгоритм верификации

На графиках 4 и 5 показаны результаты нагрузочного тестирования алгоритма верификации константных ЗО.

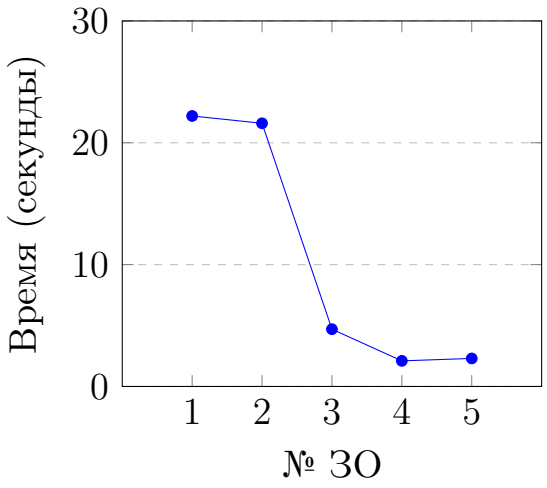


(a) Время работы алгоритма

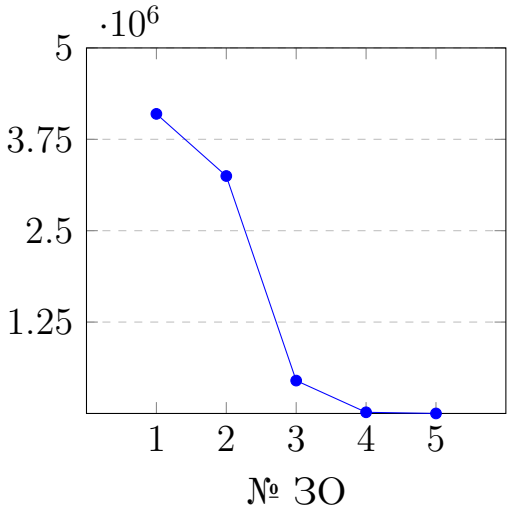


(b) Количество нарушений

Рис. 4: Результаты тестирования группы A



(a) Время работы алгоритма



(b) Количество нарушений

Рис. 5: Результаты тестирования группы B

Проверка запрещающих ограничений A1–A5 выполняется за линейное время, поскольку для проверки всех предикатов в ЗО требуется доступ только к одной записи из таблицы (каждая ЗО включает лишь кортеж s). При этом время выполнения проверки увеличивается пропорционально количеству предикатов в ЗО: чем больше предикатов, тем больше времени занимает проверка.

Для ЗО B1–B5 проверка за линейное время уже невозможна, так как они включают в себя два различных кортежа s и t. В таком случае в алгоритме 1 используется k-d дерево. ЗО B1 затрагивает только колонку с индексом ноль, поэтому в k-d дереве выполняется одномерный поиск по значениям из этой колонки. Однако поскольку дерево в процессе работы алгоритма не балансируется, поиск становится неэффективным и занимает время, линейно зависящее от количества элементов.

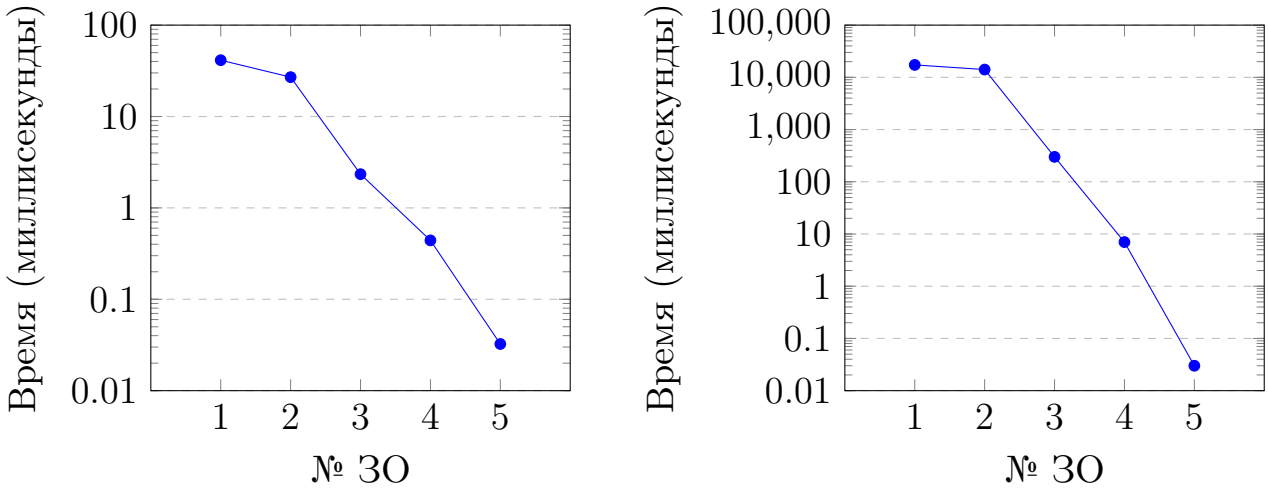
Одномерное k-d дерево аналогично сбалансированному двоичному дереву поиска, за исключением того, что последнее балансируется после каждой вставки элемента. Использование сбалансированного двоичного дерева поиска вместо k-d дерева открывает возможности для дальнейшей оптимизации.

С увеличением количества колонок, участвующих в ЗО, время работы алгоритма уменьшается. Это связано с тем, что высота k-d дерева значительно снижается за счет увеличения размерности хранимых точек, что, в свою очередь, сокращает время поиска.

Кроме того, с ростом количества предикатов в ЗО количество нарушений уменьшается. Это объясняется тем, что каждый новый предикат накладывает дополнительное ограничение, сужая круг записей, которые могут считаться нарушениями. Поскольку ЗО A1–A5 и B1–B6 формируются путем последовательного добавления предикатов к предыдущим ЗО, наибольшее количество нарушений наблюдается у ЗО с наименьшим числом предикатов (A1 и B1), а наименьшее — у ЗО с наибольшим числом предикатов (A5 и B5).

6.2 Алгоритм минимизации

На графиках 6 представлены результаты нагрузочного тестирования алгоритма минимизации.

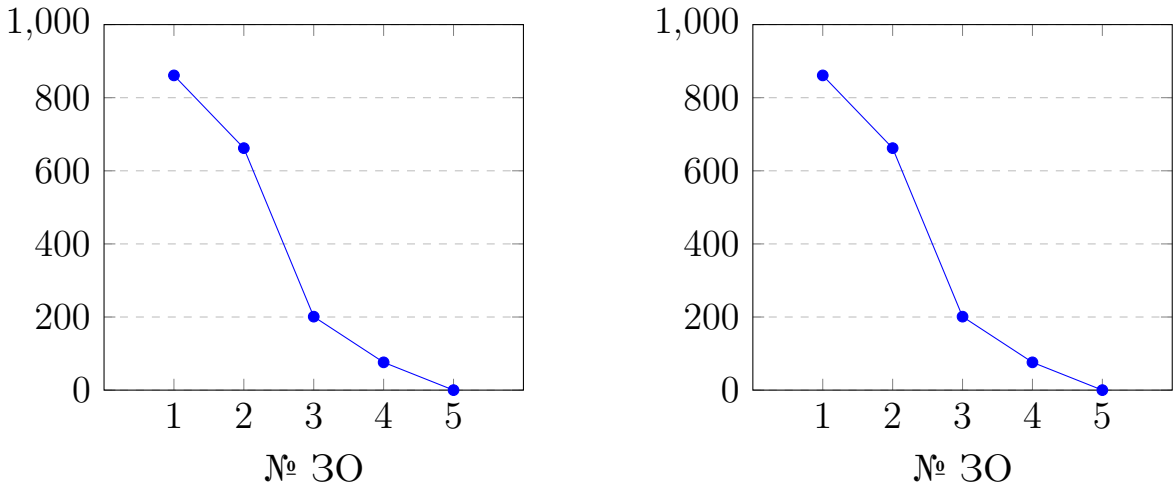


(a) Группа A (b) Группа B

Рис. 6: Время работы алгоритма минимизации

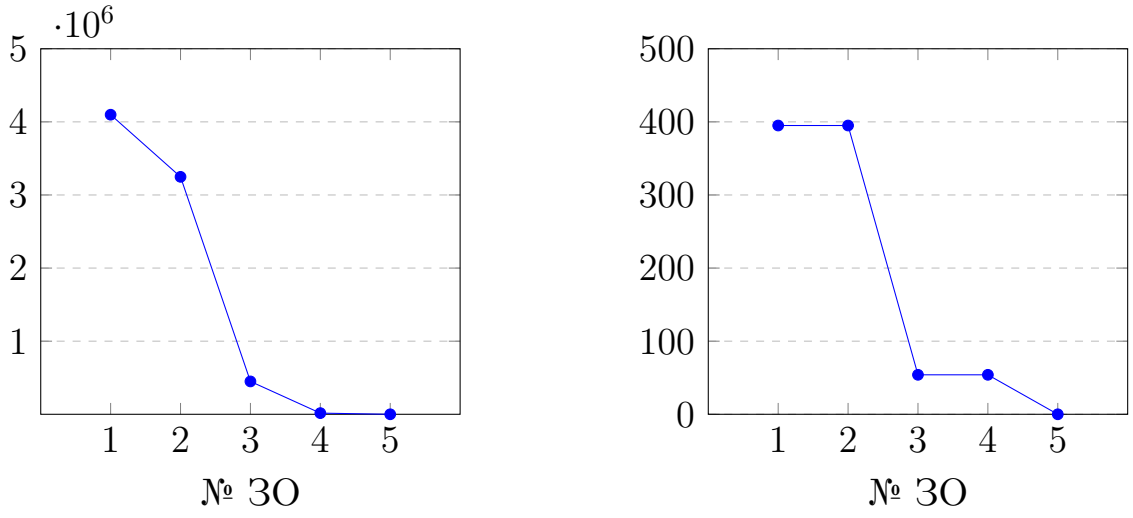
На графиках 7 и 8 представлена сравнительная характеристика числа нарушений и количества удаленных записей алгоритмом минимизации для каждой группы ЗО.

Для наглядности добавлен график с количеством нарушений для каждой группы запрещающих ограничений.



(a) Количество нарушений (b) Количество удаленных записей

Рис. 7: Результаты тестирования ЗО группы A



(a) Количество нарушений (b) Количество удаленных записей

Рис. 8: Результаты тестирования ЗО группы B

Проверка нарушений ЗО B1, B2 выполняется на порядки дольше всех остальных ЗО, так эти ограничения имеют наибольшее количество нарушений, соответственно алгоритм тратит больше времени на удаление и перебор вершин в графе.

Также отметим, что для ЗО A1–A5 удаленные записи полностью совпадают с выявленными нарушениями. Так как ЗО из этой группы содержат в себе лишь кортеж s, это означает, что выявленные нарушения будут являться минимальным набором записей, который необходимо удалить, чтобы ЗО выполнялась.

Алгоритм минимизации работает за приемлемое время для большого количества нарушений: для порядка миллиона пар нарушений алгоритм отрабатывает в пределах 17 секунд. С уменьшением количества пар нарушений время работы алгоритма минимизации также уменьшается, так как построение графа и поиск вершины с наименьшим числом инцидентных рёбер занимают меньшее количество времени.

Также необходимо отметить, что алгоритм минимизации удаляет небольшое, относительно числа нарушений, количество записей. При размере набора нарушений порядка миллиона удаляется 400 записей, что также немного относительно общего числа записей в таблице (32 тысячи строк).

Заключение

По итогам учебной практики стало возможным использование алгоритма для верификации ЗО с константными предикатами. По результатам выполнения работы:

1. Добавлена проверка запрещающих ограничений, которые включают в себя предикаты с константными значениями;
2. Стало возможным выявление кортежей, нарушающих зависимость;
3. Реализован алгоритм минимизации набора кортежей, нарушающих зависимость, без которых ЗО будет выполняться;
4. Выполнено нагрузочное тестирование алгоритмов верификации и минимизации нарушений, а также проведен анализ результатов.

Исходный код[2] доступен на GitHub, PR 510. Изменения на стадии рассмотрения.

[2]https://github.com/Desbordante/desbordante-core/pull/510

Список литературы

[1] Chu Xu, Ilyas Ihab F., Papotti Paolo. Discovering denial constraints // Proc. VLDB Endow. — 2013. — Aug.. — Vol. 6, no. 13. — P. 1498–1509. — URL: https://doi.org/10.14778/2536258.2536262.

[2] Chu Xu, Ilyas Ihab F., Papotti Paolo. Discovering denial constraints // Proc. VLDB Endow. — 2013. — aug. — Vol. 6, no. 13. — P. 1498–1509. — URL: https://doi.org/10.14778/2536258.2536262.

[3] Desbordante: a Framework for Exploring Limits of Dependency Discovery Algorithms / Maxim Strutovskiy, Nikita Bobrov, Kirill Smirnov, George Chernishev // 2021 29th Conference of Open Innovations Association (FRUCT). — 2021. — P. 344–354.

[4] Efficient Detection of Data Dependency Violations / Eduardo H. M. Pena, Edson R. Lucas Filho, Eduardo C. de Almeida, Felix Naumann // Proceedings of the 29th ACM International Conference on Information & Knowledge Management. — CIKM '20. — New York, NY, USA : Association for Computing Machinery, 2020. — P. 1235–1244. — URL: https://doi.org/10.1145/3340531.3412062.

[5] One-Pass Inconsistency Detection Algorithms for Big Data / Meifan Zhang, Hongzhi Wang, Jianzhong Li, Hong Gao // IEEE Access. — 2019. — Vol. 7. — P. 22377–22394.

[6] Pena Eduardo H. M., de Almeida Eduardo C., Naumann Felix. Fast detection of denial constraint violations // Proc. VLDB Endow. — 2021. — dec. — Vol. 15, no. 4. — P. 859–871. — URL: https://doi.org/10.14778/3503585.3503595.

[7] Rapidash: Efficient Detection of Constraint Violations / Zifan Liu, Shaleen Deep, Anna Fariha et al. // Proc. VLDB Endow. — 2024. — may. — Vol. 17, no. 8. — P. 2009–2021. — URL: https://doi.org/10.14778/3659437.3659454.

[8] Павел Аносов. Верификация DC в Desbordante. — URL: https://github.com/Desbordante/desbordante-core/blob/main/docs/papers/DC%20Verification%20-%20Anosov%20Pavel%20-%202024%20spring.pdf (дата обращения: 2025-01-05).