Санкт-Петербургский государственный университет

Кафедра информационно-аналитических систем

Группа 20.Б11-мм

Коробицын Игорь Андреевич

Обзор алгоритма FASTDC

Отчёт по учебной практике
в форме «Эксперимент»

Научный руководитель:
Ассистент кафедры информационно-аналитических систем Г.А. Чернышев

Консультант:
М.А. Струтовский

Санкт-Петербург
2024

Оглавление

Введение

Integrity constraints (IC) являются полезным инструментом для поддержки выполнения определённых правил на массиве данных. Различные языки IC формализуют требования к данным, которые должны быть в массиве данных, что упрощает задачу поддержки качества данных. Однако, существующие языки IC не способны выразить некоторые требования, которые встречаются довольно часто.

С другой стороны, мы не можем использовать слишком общие языки, вроде логики первого порядка: для выявление требований необходима либо экспертиза, которая требует больших затрат, либо автоматизированный поиск, который почти невозможно оптимизировать если язык слишком универсален. Это связано с тем, что количество допустимых комбинаций символов, которые могут считаться уникальными IC, оказывается астрономическим, что делает автоматическую проверку их всех или значительной их части почти невозможной задачей. Также, из-за недостаточно ограниченного синтаксиса, сложно вывести аксиомы, позволяющие как можно раньше отсеивать бесперспективные IC.

Xu Chu и др. [5] предложили алгоритм для поиска DC — нового языка IC, который позволяет как выразить требования уже существующих языков, так и те, которые через них выразить невозможно, а именно язык Denial constraints (DC).

Целью данной работы является обзор DC и алгоритма их поиска, с целью дальнейшей реализации данного алгоритма.

1. Постановка задачи

Целью данной работы является изучение алгоритма поиска DC, с целью дальнейшей его реализации. Для достижения данной цели были поставлены следующие задачи:

1. Рассмотреть язык DC и аксиомы, которые к нему применимы;
2. Провести обзор алгоритма FASTDC и его принципа работы;
3. Разобрать оптимизации алгоритма, предложенные авторами статьи;
4. Привести и проанализировать результаты экспериментов авторов статьи;
5. Сделать выводы об эффективности предложенных оптимизаций алгоритма.

ϕ	$=$	$\neq$	$>$	$<$	$\geq$	$\leq$
$\bar{\phi}$	$\neq$	$=$	$\leq$	$\geq$	$<$	$>$
$Imp(\phi)$	$=, \geq, \leq$	$\neq$	$>, \geq, \neq$	$<, \leq, \neq$	$\geq$	$\leq$

Рис. 1: Операторы $\bar{\phi}$ и $Imp(\phi)$ для каждого оператора [5]

2. Обзор

2.1. Основные понятия

Перед тем, как перейти непосредственно к алгоритму, необходимо упомянуть основные понятия. Пусть у нас есть база данных R. Тогда:

1. *Предикат*: Выражение P вида $v_1 \phi v_2$ или $v_1 \phi c$, где v_1, v_2 имеют вид $t_x.A$, где $t_x \in R, A$ — столбец, c — это константа, а оператор $\phi \in \{=, >, <, \geq, \leq, \neq\}$.

   Т.е. v_1 и v_2 — это ячейки в некоторой строке t_x и некотором столбце A, и *предикат* — это сравнение либо двух ячеек, либо ячейки и константы;

2. *Denial constraint (DC)*: Выражение φ вида $\forall t_\alpha, t_\beta, t_\gamma ... \in R$, $\neg(P_1 \wedge ... \wedge P_m)$;

   Т.е. для любого множества строк (авторы статьи сошлись на предикатах с не более чем двумя строками) хотя бы один из *предикатов* $P_1, ...P_m$ не выполняется. Чаще всего записывается как $\neg(P_1 \wedge ... \wedge P_m)$.

   Если в некотором DC все предикаты имеют вид $v_1 \phi v_2$, то такой DC называется variable (VDC). Иначе — константным (CDC).

   Примеры DC можно найти в следующем разделе;

3. *Инверсия*: *Инверсия предиката* P $v_1 \phi_1 v_2$ — это предикат $\bar{P}$ $v_1 \phi_2 v_2$, где $\phi_2 = \bar{\phi_1}$ (см. Рис. 1). Если P выполняется, то $\bar{P}$ не выполняется;

4. *Подразумеваемый (implied) предикат*: *предикат* Q $v_1 \phi_2 v_2$ *подразумевается предикатом* P $v_1 \phi_1 v_2$, если $\phi_2 \in Imp(\phi_1)$ (см. Рис. 1);

5. *Множество подразумеваемых (implied) предикатов предиката P* $Imp(P)$, соответственно, это множество $\{Q : Q = v_1\phi_2v_2\}$, где $\phi_2 \in Imp(\phi_1)$. $\forall Q \in Imp(P)$ если P выполняется, то Q выполняется тоже.

Утверждение "DC φ выполняется на массиве данных I" будем записывать так: $I \models \varphi$. Аналогично, для множества DC Σ запись $I \models \Sigma$ означает, что $\forall\varphi \in \Sigma$ $I \models \varphi$.

Теперь можно определить *тривиальные*, *симметричные* и *минимальные DC*, ровно как и *замыкание* множества *предикатов* т.к. эти понятия необходимы для алгоритма:

1. *Тривиальные DC*: DC φ называется *тривиальным*, если оно всегда выполняется: $\forall I$, $I \models \varphi$. Далее мы будем рассматривать только нетривиальные DC;

2. *Симметричные DC*: DC φ_1 и φ_2 называются *симметричными*, если φ_2 получается из φ_1 заменой t_α на t_β и наоборот. Если φ_1 и φ_2 *симметричны*, то $\varphi_1 \models \varphi_2$ и $\varphi_2 \models \varphi_1$ (здесь это означает, что $I \models \varphi_1 \Rightarrow I \models \varphi_2$ и наоборот);

3. *Минимальные DC*: DC φ_1 называется *минимальным*, если не существует φ_2 такого, что $I \models \varphi_1$ и $I \models \varphi_2$, и при этом $\varphi_2.Pres \subset \varphi_1.Pres$, где $\varphi.Pres$ — это множество предикатов в DC φ;

4. *Замыкание (Closure)*: *Замыкание* $Clo_\Sigma(\mathbf{W})$ множества *предикатов* $\mathbf{W}$ под множеством DC Σ — это множество *предикатов* P таких, что $\forall P \in Clo_\Sigma(\mathbf{W})$, $\Sigma \models \neg(\mathbf{W} \wedge \bar{P})$, Т.е. $\neg(\mathbf{W} \wedge \bar{P})$ подразумевается множеством Σ (подробнее в разделе 3.1. "Правила вывода DC").

2.2. Примеры

Пусть у нас есть следующая таблица (см. Рис. 2). Допустим, у нас есть 5 ограничений:

TID	FN	LN	GD	AC	PH	CT	ST	ZIP	MS	CH	SAL	TR	STX	MTX	CTX
t_1	Mark	Ballin	M	304	232-7667	Anthony	WV	25813	S	Y	5000	3	2000	0	2000
t_2	Chunho	Black	M	719	154-4816	Denver	CO	80290	M	N	60000	4.63	0	0	0
t_3	Annja	Rebizant	F	636	604-2692	Cyrene	MO	64739	M	N	40000	6	0	4200	0
t_4	Annie	Puerta	F	501	378-7304	West Crossett	AR	72045	M	N	85000	7.22	0	40	0
t_5	Anthony	Landram	M	319	150-3642	Gifford	IA	52404	S	Y	15000	2.48	40	0	40
t_6	Mark	Murro	M	970	190-3324	Denver	CO	80251	S	Y	60000	4.63	0	0	0
t_7	Ruby	Billinghurst	F	501	154-4816	Kremlin	AR	72045	M	Y	70000	7	0	35	1000
t_8	Marcelino	Nuth	F	304	540-4707	Kyle	WV	25813	M	N	10000	4	0	0	0

Рис. 2: Данные о налогообложении [5]

1. *area code* (AC) и *номер телефона* (PH) однозначно определают человека;

2. два человека с одним и тем же *ZIP-кодом* (американская система почтовых индексов) будут жить в одном и том же *штате* (ST);

3. если кто-то живёт в *городе* (CT) Денвер, то он живёт в *штате* Колорадо;

4. если два человека живут в одном и том же *штате*, то у того, у кого *зарплата* (SAL) меньше, и *налоговая ставка* (TR) будет меньше;

5. *налоговая льгота для неженатых и бездетных* (STX) не может быть больше, чем зарплата.

Первые три ограничения могут быть выражены уже существующими языками IC: *Key*, *FD*, и *CFD* соответственно:

1. $Key\{AC, PH\}$

2. $ZIP \rightarrow ST$

3. $[CT = \text{`}Denver\text{'}] \rightarrow [ST = \text{`}CO\text{'}]$

Однако с четвёртым и пятым ограничениями возникают проблемы. Впрочем, их можно выразить с помощью языка *DC*. Например, так будет выглядит запись четвёртого ограничения:

$$\forall t_\alpha, t_\beta \in R, \neg(t_\alpha.ST = t_\beta.ST \wedge t_\alpha.SAL < t_\beta.SAL \wedge t_\alpha.TR > t_\beta.TR)$$

Этот DC означает, что какие бы строчки t_α и t_β мы ни взяли, *предикаты* $t_\alpha.ST = t_\beta.ST$, $t_\alpha.SAL < t_\beta.SAL$ и $t_\alpha.TR > t_\beta.TR$ не будут выполняться для них одновременно.

А так выглядит DC для пятого ограничения:

$$\forall t_\alpha \in R, \neg(t_\alpha.SAL < t_\alpha.STX)$$

Также, DC могут использоваться для того, чтобы выразить другие языки Integrity constraints, например, functional dependencies (FD). Так будет выглядеть второе ограничение, записанное языком DC:

$$\forall t_\alpha, t_\beta \in R, \neg(t_\alpha.ZIP = t_\beta.ZIP \wedge t_\alpha.ST \neq t_\beta.ST)$$

2.3. Проблемы с DC

Необходимо решить ряд проблем с языком DC:

1. **Аксиомы**: работа с FD возможна благодаря наличию правил вывода, известных как аксиомы Армстронга [4]. Было бы удобно иметь что-то похожее для DC. Аналог аксиом Армстронга для DC будет приведён в начале следующей главы;

2. **Количество возможных DC**: каждый DC может включать или не включать любой из допустимых *предикатов*. Таким образом, если P — множество всех *предикатов*, то общее количество допустимых DC будет $2^{|P|}$. Мощность же множества всех *предикатов*, если допустить, что каждый DC работает с двумя строками, будет $6 * 2m * (2m - 1)$, где m — это количество столбцов (каждый *предикат* — это одна из 6 операций $\{=, \neq, <, >, \leq, \geq\}$, и две уникальные ячейки). Из-за этого требуется создать алгоритм, который будет как можно раньше отсекать те DC, которые рассматривать не обязательно;

3. **Верификация**: алгоритмы поиска IC часто страдают от проблемы overfitting'a [3], т.е. те закономерности, которые работают на

одном массиве данных, не обязательно выполняются в общем случае. Эта проблема может быть решена консультациями с экспертами, но это требует времени и денег. Авторы статьи предлагают параметр *interestingness*, чтобы отранжировать обнаруженные DC, и таким образом отсеять те из них, которые слишком громоздкие или слабо подтверждаются конкретными строками.

3. Алгоритм

3.1. Правила вывода *DC*

Как понятно из раздела 2 (“Обзор”), язык DC достаточно выразителен (он не только позволяет выражать другие языки IC, но и добавляет новую семантику), но при этом имеет формализованный и простой синтакс. Это позволяет составить систему правил вывода DC, которые напоминают таковую для FD [1]:

1. **Тривиальность**: $\forall P_i, P_j : \bar{P}_i \in Imp(P_j)$, $\neg(P_i \wedge P_j)$ является *тривиальным DC*.

   Т.е если в DC есть два *предиката*, которые не могут быть верны одновременно ($\bar{P}_i \in Imp(P_j)$), то DC *тривиален*, т.е. всегда выполняется;

2. **Аугментация**: Если $\neg(P_1 \wedge ... \wedge P_n)$ выполняется, то $\forall Q$, $\neg(P_1 \wedge ... \wedge P_m \wedge Q)$ выполняется тоже.

   Другими словами, если DC уже выполняется, то, добавив к нему любые другие *предикаты*, мы получим DC, который тоже выполняется;

3. **Транзитивность**: Если $\neg(P_1 \wedge ... \wedge P_n \wedge Q_1)$ и $\neg(R_1 \wedge ... \wedge R_m \wedge Q_2)$ выполняются, и при этом $Q_2 \in Imp(\bar{Q}_1)$, то $\neg(P_1 \wedge ... \wedge P_n \wedge R_1 \wedge ... \wedge R_m)$ тоже выполняется.

   Или, если есть два DC которые выполняются, и два *предиката*, которые не могут быть ложными одновременно ($Q_2 \in Imp(\bar{Q}_1)$) по одному в каждом DC, то, если объединить эти два DC и убрать эти два *предиката*, мы получим DC, который выполняется.

Обозначим эту систему правил вывода как $\mathcal{I}$. $\mathcal{I}$ позволяет определить, подразумевает ли множество DC Σ некий DC φ, т.е. $\Sigma \models \varphi$. Задача определения для неких Σ и φ того, верно ли что $\Sigma \models \varphi$, называется **проблемой выводимости** (**Implication Problem**), и установле-

но, что она принадлежит классу сложности coNP-Complete [2]. По этой причине, авторы статьи остановились на неполном алгоритме.

3.2. Проблема выводимости

Алгоритм 1 [5] проверяет, верно ли, что $\Sigma \models \varphi$. В нём упоминается функция $getClosure()$, которая берёт на вход множество *предикатов* $\mathbf{W}$ и множество DC Σ, и возвращает *замыкание* $Clo_\Sigma(\mathbf{W})$. Мы вернёмся к ней в следующем разделе.

Algorithm 1 Проверка на выводимость

Ввод: Множество DC Σ и DC φ.

Вывод: Булевское значение: DC Σ подразумевает φ или нет.

1: **if** φ — *тривиальный* DC **then**
2: **return** true
3: **end if**
4: $\Gamma \leftarrow \Sigma$
5: **for** $\phi \in \Sigma$ **do**
6: $\Gamma \leftarrow \Sigma$ + DC, который *симметричен* ϕ
7: **end for**
8: $Clo_\Gamma(\varphi.Pres) \leftarrow getClosure(\varphi.Pres, \Gamma)$
9: **if** $\exists \phi \in \Gamma : \phi.Pres \subseteq Clo_\Gamma(\varphi.Pres)$ **then**
10: **return** true
11: **end if**
12: **return** false

3.3. *getClosure()*

Алгоритм 2 [5] вычисляет частичное *замыкание* $Clo_\Sigma(\mathbf{W})$, получив на вход множество *предикатов* $\mathbf{W}$ и множество DC Σ.

Algorithm 2 getClosure()

Ввод: множество *предикатов* $\mathbf{W}$ и множество DC Σ.

Вывод: $Clo_\Sigma(\mathbf{W})$.

1: 1. Инициализируем $Clo_\Sigma(\mathbf{W})$:

2: **for** $P \in \mathbf{W}$ **do**
3: $Clo_\Sigma(\mathbf{W}) \leftarrow Clo_\Sigma(\mathbf{W}) + Imp(P)$
4: $Clo_\Sigma(\mathbf{W}) \leftarrow Clo_\Sigma(\mathbf{W}) + Imp(Clo_\Sigma(\mathbf{W}))$
5: **end for**
6: 2. Для каждого *предиката* P составим список L_P из DC, содержащих P. Также, для каждого DC φ составим список L_φ из его *предикатов*, которых ещё нет в $Clo_\Sigma(\mathbf{W})$. Также, создадим очередь J из DC, все *предикаты* которых содержатся в *замыкании*, кроме одного:
7: **for** $\varphi \in \Sigma$ **do**
8: **for** $P \in \varphi.Pres$ **do**
9: $L_P \leftarrow L_P + \varphi$
10: **end for**
11: **end for**
12: **for all** $P \notin Clo_\Sigma(\mathbf{W})$ **do**
13: **for** $\varphi \in L_P$ **do**
14: $L_\varphi \leftarrow L_\varphi + P$
15: **end for**
16: **end for**
17: **for** $\varphi \in \Sigma$ **do**
18: **if** $|L_\varphi| = 1$ **then**
19: $J \leftarrow J + \varphi$
20: **end if**
21: **end for**
22: 3. Теперь, используя очередь J, дополним $Clo_\Sigma(\mathbf{W})$:
23: **while** $|J| > 1$ **do**
24: $\varphi \leftarrow J.pop()$
25: $P \leftarrow L_\varphi.pop()$
26: **for** $Q \in Imp(\bar{P})$ **do**
27: **for** $\varphi \in L_Q$ **do**
28: $L_\varphi \leftarrow L_\varphi - Q$
29: **if** $|L_\varphi| = 1$ **then**
30: $J \leftarrow J + \varphi$

31: **end if**
32: **end for**
33: **end for**
34: $Clo_\Sigma(\mathbf{W}) \leftarrow Clo_\Sigma(\mathbf{W}) + Imp(\bar{P})$
35: $Clo_\Sigma(\mathbf{W}) \leftarrow Clo_\Sigma(\mathbf{W}) + Imp(Clo_\Sigma(\mathbf{W}))$
36: **end while**
return $Clo_\Sigma(\mathbf{W})$

Пройдёмся по стадиям алгоритма:

1. $\forall P \in \mathbf{W}$, $Imp(P) \subset Clo_\Sigma(\mathbf{W})$ согласно *тривиальности*. То же самое верно и для $P \in Clo_\Sigma(\mathbf{W})$ ввиду транзитивности $Imp(P)$;

2. Эти списки понадобятся нам на следующей стадии;

3. Для очередного $\varphi \in J$ берём тот его *предикат*, которого нет в *замыкании*. Добавив к $\mathbf{W}$ тот $P \in \varphi.Pres$, которого ещё нет в *замыкании*, мы получим DC, содержащий все предикаты из φ, а т.к. $\Sigma \models \varphi$, то, по *аугментации*, $\bar{P} \in Clo_\Sigma(\mathbf{W})$.

   Теперь можно проверить, не появились ли у нас новые DC φ, у которых все предикаты кроме одного в *замыкании*. Для этого пройдёмся по DC, содержащим *предикаты*, которые $\in Imp(\bar{P})$. Если какой-то DC нас устраивает, мы добавляем его в J.

   Наконец, т.к. $\bar{P} \in Clo_\Sigma(\mathbf{W})$, повторяем для него действия из стадии 1.

3.4. Алгоритм FASTDC

Алгоритм 3 [5] занимается непосредственно поиском минимальных DC.

Algorithm 3 FASTDC

Ввод: Таблица I и схема этой таблицы R.

Вывод: Множество всех минимальных DC Σ.

1: $\mathbf{P} \leftarrow buildPredicateSpace(I, R)$

2: $Evi_I \leftarrow buildEvidenceSet(I, \mathbf{P})$
3: $\mathbf{MC} \leftarrow searchMinimalCovers(Evi_I, Evi_I, \emptyset, >_{init}, \emptyset)$
4: **for** $\mathbf{X} \in \mathbf{MC}$ **do**
5: $\quad \Sigma \leftarrow \Sigma + \neg(\bar{\mathbf{X}})$
6: **end for**
7: **for** $\varphi \in \Sigma$ **do**
8: $\quad$ **if** $(\Sigma - \varphi) \models \varphi$ **then**
9: $\quad\quad \Sigma \leftarrow \Sigma - \varphi$
10: $\quad$ **end if**
11: **end for**
12: **return** Σ

Функция $buildPredicateSpace()$, принцип работы которой подробнее описан в подразделе 3.4.1), возвращает множество возможных предикатов, $buildEvidenceSet()$ — множество, обозначаемое как Evi_I, состоящее из множеств верных предикатов для каждой пары строк, которые обозначаются как $SAT(\langle t_x, t_y \rangle)$ (подробнее в подразделе 3.4.1), а $searchMinimalCovers()$ находит *минимальные покрытия для* Evi_I: такие множества $\mathbf{X} \subseteq \mathbf{P}$, что $\forall E \in Evi_I, \mathbf{X} \cap E \neq \varnothing$ и $\nexists \mathbf{Y} \subset \mathbf{X}$ такого, что $\forall E \in Evi_I, \mathbf{Y} \cap E \neq \varnothing$ (подробнее в подразделе 3.4.3).

Теорема: $\neg(\overline{X_1} \wedge ... \wedge \overline{X_n})$ — валидный *минимальный DC* тогда и только тогда когда $\mathbf{X} = \{X_1, ..., X_n\}$ — *минимальное покрытие для* Evi_I [5].

Таким образом, получив *минимальные покрытия*, мы также получаем *минимальные DC*.

3.4.1. *buildPredicateSpace()*

Для каждого столбца между двумя строками на нём создаются предикаты с операциями $\{=, \neq\}$, и, если тип данных в столбце числовой, то ещё и с операциями $\{>, <, \geq, \leq\}$.

Столбцы считаются *объединимыми*, если значения в них одного типа, а также некоторые из значений одного столбца также присутствуют в другом (на основании данных экспериментов [5], приведённых в раз-

деле 4.5, авторы статьи пришли к выводу, что необходимо требовать как минимум тридцатипроцентного совпадения). Для них создаются предикаты с операциями $\{=, \neq\}$.

Сравнимыми столбцы считаются, если они оба числового типа, и их среднее арифметическое отличается не более чем на порядок. Для них создаются предикаты с операциями $\{>, <, \geq, \leq\}$.

Для поиска *объединимых* и *сравнимых* столбцов могут быть использованы алгоритмы профайлинга [6].

3.4.2. $buildEvidenceSet()$

Evi_I вычисляется ровно так, как написано в определении: для каждой возможной пары строк перебираются предикаты, чтобы найти те, которые на ней выполняются. Сложость, таким образом, $O(|\mathbf{P}| \times |I|^2)$, однако для каждой пары строк проверку можно проводить независимо, что позволяет распараллелить алгоритм.

Авторы статьи предлагают разделить массив данных на B блоков, а процесс поиска Evi_I разбить на подзадачи, каждая из которых заключается в сравнении строк двух из этих блоков. Общее число подзадач, таким образом, $\frac{B^2}{2}$, и если имеется M машин, то необходимо, чтобы выполнялось уравнение $\frac{B^2}{2} = w \times M$, где w — число задач для каждой из машин. Таким образом, $B = \sqrt{2wM}$. Помимо этого, т.к. в любой момент времени в памяти должно находиться два блока, необходимо убедиться, что два блока помещаются в оперативную память каждой машины.

3.4.3. $searchMinimalCovers()$

Алгоритм 4 [5] проводит поиск *минимальных покрытий* поиском в глубину. Псевдокод приведён ниже. Алгоритм дополнительно оптимизирован отсеиванием веток, которые не ведут ни к каким DC, которые бы не следовали напрямую из других DC, а также динамическим порядком перебора предикатов, к которому мы вернёмся в той части алгоритма, которая отвечает непосредственно за поиск в глубину.

Отсеивание бесперспективных веток поиска:

1. Если перебирать эту ветку дальше, то получатся DC вида $\neg(\overline{\mathbf{X} - P} \wedge \overline{P} \wedge \mathbf{W})$, и, если $\exists Q \in \overline{\mathbf{X} - P}$, такой, что предикат P им подразумевается, то все эти DC *тривиальные*;

2. Если $\mathbf{Y} \in \mathbf{MC}$, то $\neg(\overline{\mathbf{Y}})$ — валидный DC. Любой DC, который мы получим при дальнейшем переборе, будет иметь вид $\neg(\overline{\mathbf{X}} \wedge \mathbf{W})$, и, если $\mathbf{X} \subseteq \mathbf{Y}$, такой DC подразумевается DC $\neg(\overline{\mathbf{Y}})$ согласно *аугментации*;

3. Если $\mathbf{X} \supseteq ((\mathbf{Y} - Y_i) \cup \overline{Q})$, то любой DC, который мы получим, будет иметь вид $\neg(\overline{\mathbf{Y} - Y_i} \wedge Q \wedge \mathbf{W})$. Также, из того, что $\mathbf{Y} \in \mathbf{MC}$ следует, что $\neg(\overline{\mathbf{Y} - Y_i} \wedge \overline{Y_i})$ — валидный DC. По *акисиоме транзитивности*, $\neg(\overline{\mathbf{Y} - Y_i} \wedge \mathbf{W})$ — тоже валидный DC, и по *аксиоме аугментации* он подразумевает $\neg(\overline{\mathbf{Y} - Y_i} \wedge Q \wedge \mathbf{W})$;

К остальным двум методам отсеивания мы вернёмся чуть позже. Если вкратце, то четвёртый if (строки 12-14) работает засчёт разделения пространства DC с целью минимизировать повторную проверку уже учтённых DC, а пятый (строки 15-17) — за счёт свойств метрики $Interestingness$, предложенной авторами статьи (позже будет показано, что для того, чтобы проверить это условие, не необходимо перебирать все DC вида $\neg(\overline{\mathbf{X}} \wedge \mathbf{W})$)

Algorithm 4 $searchMinimalCovers()$

Ввод:

1. Evi_I

2. Evi_{curr} — пока не покрытые множества из Evi_I

3. текущий путь дерева поиска $\mathbf{X} \subseteq \mathbf{P}$

4. текущий порядок предикатов $>_{curr}$

5. Σ — множество DC, обнаруженных на данный момент

Вывод: Множество *минимальных покрытий* для Evi_I, обозначаемое как **MC**

1: $P \leftarrow \mathbf{X}.last$ // последний добавленный предикат
2: Отсеивание бесперспективных веток поиска:
3: **if** $\exists Q \in \overline{\mathbf{X} - P}$ такой, что $P \in Imp(Q)$ **then**
4: **return**
5: **end if**
6: **if** $\exists \mathbf{Y} \in \mathbf{MC}$ такой, что $\mathbf{X} \supseteq \mathbf{Y}$ **then**
7: **return**
8: **end if**
9: **if** $\exists \mathbf{Y} = \{Y_1, ...Y_n\} \in \mathbf{MC}$ и $\exists i \in [1, n]$ и $\exists Q \in Imp(Y_i)$ такой, что $\mathbf{X} \supseteq ((\mathbf{Y} - Y_i) \cup \overline{Q})$ **then**
10: **return**
11: **end if**
12: **if** $\exists \varphi \in \Sigma$ такой, что $\overline{\mathbf{X}} \supseteq \varphi.Pres$ **then**
13: **return**
14: **end if**
15: **if** $Inter(\varphi) < t, \forall \varphi$ вида $\neg(\overline{\mathbf{X}} \wedge \mathbf{W})$ **then**
16: **return**
17: **end if**
18: Завершение поиска с позитивным или негативным результатом:
19: **if** $>_{curr} = \emptyset$ и $Evi_{curr} \neq \emptyset$ **then**
20: **return**
21: **end if**
22: **if** $Evi_{curr} = \emptyset$ **then**
23: **if** нет подмножества размера $|\mathbf{X}| - 1$, которое бы покрывало Evi_{curr} **then**
24: $\mathbf{MC} \leftarrow \mathbf{MC} + \mathbf{X}$
25: **end if**
26: **return**
27: **end if**
28: Рекурсивный поиск в глубину:
29: **for all** $P \in >_{curr}$ **do**

30: $\mathbf{X} \leftarrow \mathbf{X} + P$

31: $Evi_{next} \leftarrow$ множества в Evi_{curr}, ещё не покрытые P

32: $>_{next} \leftarrow$ новый порядок предикатов P' таких, что $\{P' | P >_{curr} P'\}$ с учётом Evi_{next}

33: $searchMinimalCovers(Evi_I, Evi_{next}, \mathbf{X}, >_{next}, \Sigma)$

34: $\mathbf{X} \leftarrow \mathbf{X} - P$

35: **end for**

Завершение поиска с негативным либо позитивным результатом:

1. Если ещё есть необработанные предикаты, но при этом ещё есть непокрытые множества в Evi_{curr}, значит, данная ветка поиска в глубину ни к чему не привела;

2. Если все множества в Evi_{curr} покрыты, то если $\mathbf{X}$ — минимальное покрытие, то мы добавляем его в $\mathbf{MC}$.

Для того, чтобы произвести рекурсивный поиск в глубину, для каждого *предиката* P в текущем динамическом порядке:

1. Добавляем в $\mathbf{X}$ *предикат* P;

2. Составляем Evi_{next} из множеств из Evi_{curr}, которые ещё не покрыты *предикатом* P;

3. Определяем порядок новый порядок предикатов с учётом Evi_{next}. Динамический порядок определяется следующим образом: $P >_{next} Q$ если $Cov(P, Evi_{next}) > Cov(Q, Evi_{next})$, где $Cov(P, Evi_{next}) = |\{E \in Evi_{next} | P \in E\}|$. Если же $Cov(P, Evi_{next}) = Cov(Q, Evi_{next})$, то $P >_{next} Q$ если $P >_{curr} Q$;

4. Рекурсивно вызываем $searchMinimalCovers()$

5. Убираем из $\mathbf{X}$ *предикат* P;

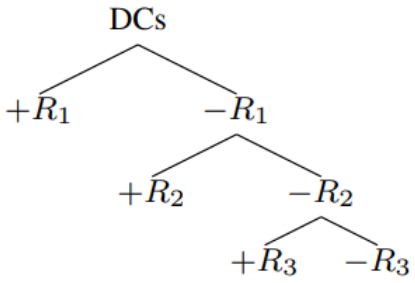


Рис. 3: Таксономия *DC* для базы данных с тремя *предикатами* [5]

3.4.4. Разделение пространства *DC*

Определим *Evidence set* от I по модулю P как $Evi_I^P = \{E - \{P\} | E \in Evi_I, P \in E\}$

Заметим, что $\neg(\overline{X_1} \wedge ... \wedge \overline{X_n} \wedge P)$ — валидный *минимальный DC*, **содержащий предикат** P, тогда и только тогда, когда $\{X_1 \wedge ... \wedge X_n\}$ — минимальное покрытие Evi_I^P.

Теперь мы можем построить таксономию всех возможных *DC*. Для начала как-то упорядочим *предикаты*. Теперь возьмём первый *предикат*, и рассмотрим те *DC*, в которых он есть, что мы можем сделать, пользуясь определением Evi_I^P. Теперь перейдём к тем *DC*, в которых нет первого *предиката*, и сделаем ту же саму развилку уже со вторым, и так далее. На примере массива данных, в котором только три *предиката* — R_1, R_2, R_3 — такая таксономия будет выглядеть как на рисунке 3, где "+" означает присутствие *предиката* в *DC*, а "–" — его отсутствие.

Теперь, чтобы исключить повторный перебор уже рассмотренных *DC*, авторы статьи предлагают искать *DC* поднимаясь по дереву снизу вверх. На примере с тремя *предикатами*, сперва мы рассматриваем *DC*, содержащие R_3 и больше никакие другие предикаты. Потом — содержащие R_2, но не R_1. Если какое-то из обнаруженных покрытий $Evi_I^{R_2}$ дублирует какие-то из *DC*, обнаруженных на предыдущей итерации, значит, это покрытие не приведёт к **глобально** *минимальному DC*, **что и проверяется в строчках 12-14 алгоритма 4**.

3.5. Метрика *Interestingness*

Метрика $Interestingness$ для DC φ вычисляется по формуле $Inter(\varphi) = a \times Coverage(\varphi) + (1-a) \times Succinctness(\varphi)$. $Coverage(\varphi)$ измеряет насколько φ подтверждается массивом данных, а $Succinctness(\varphi)$ — краткость, лаконичность φ. Таким образом, $Inter(\varphi)$ учитывает и то, и другое. И $Succinctness(\varphi)$, и $Coverage(\varphi)$ лежат в интервале от 0 до 1. На основании экспериментов, данные которых приведены в разделе 4.6, авторы статьи пришли к выводу [5], что оптимальное значение $a = 0.6$.

3.5.1. Coverage

Метрика $Coverage$ вычисляется по формуле:

$$Coverage(\varphi) = \frac{\sum_{k=0}^{|\varphi.Pres|-1} |kE| \times w(k)}{\sum_{k=0}^{|\varphi.Pres|-1} |kE|}$$

Где $|kE|$ — количество пар строк в массиве данных, для которых выполняется k из *предикатов DC*. Сами же такие строки по отдельности обозначаются как kE. $w(k)$ — это вес kE, который вычисляется по формуле $w(k) = \frac{k+1}{|\varphi.Pres|}$. Нетрудно заметить, что т.к. $w(k)$ всегда на интервале от 0 до 1, то и про $Coverage(\varphi)$ можно сказать то же самое.

3.5.2. Succinctness

Метрика $Succinctness$ вычисляется по формуле:

$$Succinctness(\varphi) = \frac{Min(\{Len(\phi)|\forall\phi\})}{Len(\varphi)}$$

Иначе говоря, $Succinctness(\varphi)$ — это частное от деления минимального возможного размера DC на размер φ. Это обеспечивает, что результат будет в диапазоне от 0 до 1.

В качестве функции размера DC $Len(\varphi)$ авторы статьи предлагают использовать количество уникальных символов в DC, принадлежащих

алфавиту $\mathbb{A} = \{t_\alpha, t_\beta, \mathbb{U}, \mathbb{B}, Cons\}$, где $\mathbb{U}$ — названия столбцов, $\mathbb{B}$ — операторы, а $Cons$ — это константы.

3.5.3. Отсеивание на основе *Interestingness*

Функция $Succinctness(\varphi)$ является антимонотонной, т.е. при добавлении новых *предикатов* она становится меньше. Если бы то же самое можно было сказать про $Coverage(\varphi)$, проблема необходимости перебора всех DC вида $\neg(\overline{\mathbf{X}} \wedge \mathbf{W})$ решилась бы сразу (вместо φ подставляем $\neg(\overline{\mathbf{X}})$), но это не так.

Впрочем, можно заметить, что если мы найдём предельное верхнее значение $w(k)$ (которое обозначим как W), то его можно вынести из под суммы и обнаружить, что:

$$Coverage(\varphi) = \frac{\sum_{k=0}^{|\varphi.Pres|-1} |kE| \times w(k)}{\sum_{k=0}^{|\varphi.Pres|-1} |kE|} < \frac{\sum_{k=0}^{|\varphi.Pres|-1} |kE|}{\sum_{k=0}^{|\varphi.Pres|-1} |kE|} \times W = W$$

Рассмотрим $w(k) = \frac{k+1}{|\varphi.Pres|} = 1 - \frac{l}{|\varphi.Pres|}$, где l — число *предикатов* в φ, которые не удовлетворяются данным kE. Заметим, что [5]:

1. l больше или равно числу предикатов в $\overline{\mathbf{X}}$, которые не удовлетворяются данным kE;
2. $|\varphi.Pres|$ меньше, чем $\frac{|\mathbf{P}|}{2}$.

Таким образом, $\exists Z : \frac{l}{|\varphi.Pres|} \geq Z \implies 1 - \frac{l}{|\varphi.Pres|}$ ограничен сверху. Однако, чтобы найти W, всё ещё необходимо перебрать kE. К счастью, можно перебрать не все: если верить авторам [5], 1000 должно хватить.

Таким образом, мы решили проблему с $Coverage(\varphi)$.

3.6. Модификации алгоритма FASTDC

3.6.1. A-FASTDC

A-FASTDC — это алгоритм приближённого поиска DC. Он полезен на массивах данных, в которых могут быть ошибки, а также позволя-

ет уменьшить проблему оверфиттинга. В нём мы будем считать, что DC валиден, если доля пар строк, которые ему противоречат, не выше определённого порога ϵ. Для этого общее число нарушений надо разделить на количество возможных пар строк, т.е. $|I|(|I|-1)$, и сравнить получившееся число с ϵ.

Чтобы всё работало, необходимо переопределить минимальные покрытия. Для этого для каждого множества E в Evi_I введём значение $count(E)$ — количество пар строк $\langle t_x, t_y \rangle$ таких, что $SAT(\langle t_x, t_y \rangle)$.

Теперь можно определить *ϵ-минимальное* покрытие для Evi_I как $\mathbf{X} \subseteq \mathbf{P}$ такое, что $\sum(count(E)) \leq \epsilon|I|(|I|-1)$ для таких $E \in Evi_I$, что $\mathbf{X} \cap E = \emptyset$, и при у $\mathbf{X}$ нет подмножества, обладающего теми же свойствами.

Минимальные приближённые DC переопределяются аналогичным образом.

Для того, чтобы искать *минимальные приближённые DC*, алгоритм 4 надо изменить в двух местах:

1. При динамическом перерасчёте порядка *предикатов* функцию $Cov()$ переопределить как $Cov(P, Evi) = \sum_{E \in \{E \in Evi | P \in E\}} count(E)$;
2. При проверке того, стоит ли завершать поиск, опираться не на то, пусто ли множество Evi_{curr}, а на то, больше ли в нём элементов, чем $\epsilon|I|(|I|-1)$.

3.6.2. C-FASTDC

C-FASTDC — это алгоритм поиска *константных DC*.

Процесс составления пространства *предикатов* в чём-то похож на таковой для обычных *DC*. Т.к. простанство константных *предикатов* гораздо больше, чем пространство обычных, мы не можем просто добавить обнаруженные константные *предикаты* в $\mathbf{P}$ и выполнить FASTDC для всех *предикатов* сразу. Эта проблема решается тем, что алгоритм отсеивает предикаты, которые встречаются реже некоторой частоты τ.

Algorithm 5 C-FASTDC

Ввод: Таблица I, схема этой таблицы R и минимальное требование к частоте τ

Вывод: *Константные DC* Γ.

Составление пространства предикатов:
Пусть $\mathbf{Q} \leftarrow \emptyset$ будет пространством константных предикатов
for all $A \in R$ **do**
    for all $c \in A$ **do**
        $\mathbf{Q} \leftarrow \mathbf{Q} + t_\alpha.A\theta c$ где $\theta \in \{=, \neq\}$
        if A — численный тип **then**
            $\mathbf{Q} \leftarrow \mathbf{Q} + t_\alpha.A\theta c$ где $\theta \in \{>, <, \geq, \leq\}$
        end if
    end for
end for
Для каждого предиката составляется список строк, на которых он выполняется:
for all $t \in I$ **do**
    if t удовлетворяет Q **then**
        $sup(Q, I) \leftarrow sup(Q, I) + t$
    end if
end for
Составляется список частых предикатов:
Пусть L_1 будет множеством частых предикатов
for all $Q \in \mathbf{Q}$ **do**
    if $|sup(Q, I)| = 0$ **then**
        $\Gamma \leftarrow \Gamma + \neg(Q)$
    else if $\frac{|sup(Q,I)|}{|I|} \geq \tau$ **then**
        $L_1 \leftarrow L_1 + \{Q\}$
    end if
end for
Составляются множества предикатов размера > 1:
$m \leftarrow 2$
while $L_{m-1} \neq \emptyset$ **do**
    for all $c \in L_{m-1}$ **do**

30: $\Sigma \leftarrow FASTDC(sup(c, I), R)$
31: **for all** $\varphi \in \Sigma$ **do**
32: $\Gamma \leftarrow \Gamma + \phi$, предикаты которого взяты у c и φ
33: **end for**
34: **end for**
35: $C_m = \{c | c = a \cup b \wedge a \in L_{m-1} \wedge b \in \bigcup L_{k-1} \wedge b \notin a\}$
36: **for all** $c \in C_m$ **do**
37: Просканировать массив данных чтобы наполнить $sup(c, I)$
38: **if** $|sup(c, I)| = 0$ **then**
39: $\Gamma \leftarrow \Gamma + \phi$, предикаты которого взяты у c
40: **else if** $\frac{|sup(c,I)|}{|I|} \geq \tau$ **then**
41: $L_m \leftarrow L_m + c$
42: **end if**
43: **end for**
$m \leftarrow m + 1$
44: **end while**

Теперь внимательнее разберём стадии алгоритма:

1. **Составление пространтсва предикатов:** как уже было сказано, во многом похоже на аналогичный процесс для обычных DC;

2. **Для каждого предиката составлятся список строк, на которых он выполняется:** понадобится при оценке частоты *предикатов*;

3. **Составляется список частых предикатов:** если для какого-то из *предикатов* $sup(Q, I) = 0$, значит, этот *предикат* не выполняется ни на одной строке. Следовательно, $\neg(Q)$ — валидный DC. Если же $\frac{|sup(Q,I)|}{|I|} \geq \tau$, значит, этот *предикат* прошёл проверку на частоту, и его можно записать в L_1;

4. **Составляются множества предикатов размера > 1:** если нашлось множество таких предикатов, которое по частоте преодолело порог τ, то только тогда мы вызываем $FASTDC(sup(c, I), R)$, и

добавляем в Γ найденные *DC*. Дальше генерируются новые множества *предикатов* большего размера, после чего они проверяются на частоту (перед этим сканируется массив данных, чтобы наполнить $sup(c, I)$). Как и с единичными *предикатами* с шага 3, если $sup(c, I) = 0$, это значит, что мы нашли *DC*.

4. Эксперименты

4.1. Условия эксперимента

Авторы статьи [5] указали следующие технические данные:

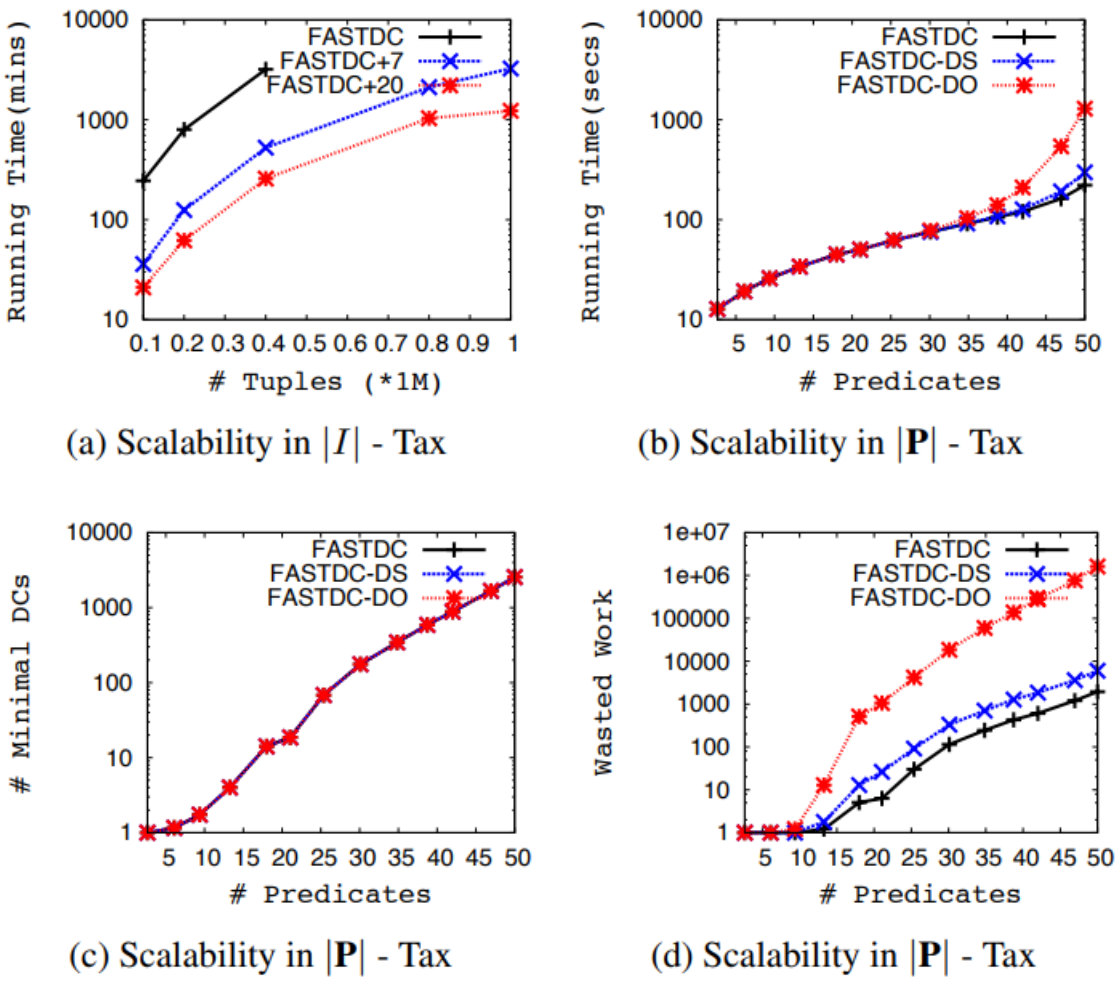
- четырёхъядерный процессор 3.4 GHz;
- 4 GB оперативной памяти;
- Windows 7;
- Java.

Использовался один синтетический массив данных, *Tax*, и два реальных: *Hospital* (115 тысяч строк) и *SP Stock* (123 тысячи строк). *Tax* уже был рассмотрен ранее — из него и были взяты примеры — *Hospital* содержит 17 строковых атрибутов, а *SP Stock* — информацию о биржевых торгах. В основном в экспериментах используется *Tax*, т.к. в нём удобнее менять количество строк и столбцов.

4.2. Обозначения

Авторы статьи вводят ряд обозначений:

- FASTDC+M — алгоритм FASTDC, запущенный на M машинах;
- FASTDC-DO — алгоритм FASTDC без динамического порядка *предикатов*;
- FASTDC-DS — FASTDC без разделения пространства DC как в разделе 3.4.4.;
- Σ_s — DC, найденные алгоритмом;
- Σ_g — DC, обнаруженные экспертами;
- “золотые” DC — Σ_g и те, которые ими подразумеваются;



(a) Scalability in $|I|$ - Tax

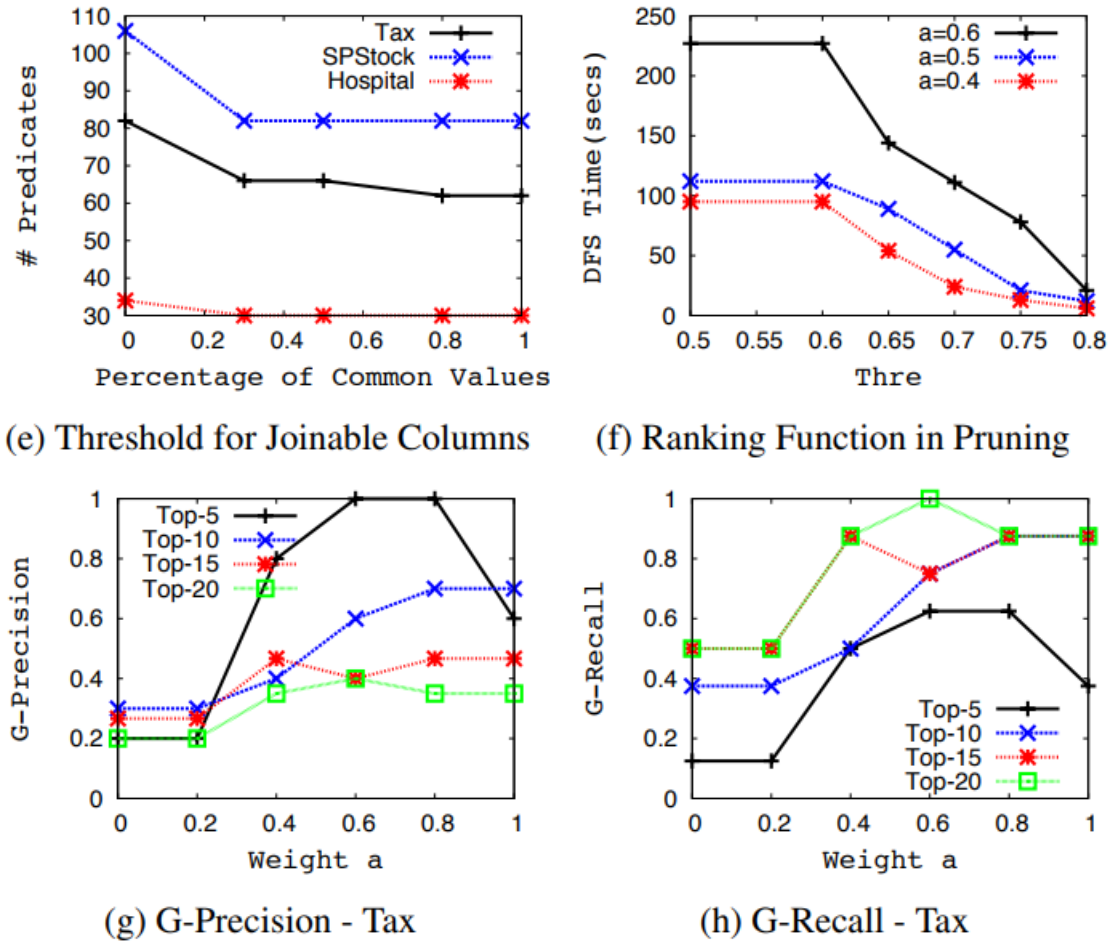
(b) Scalability in $|\mathbf{P}|$ - Tax

(c) Scalability in $|\mathbf{P}|$ - Tax

(d) Scalability in $|\mathbf{P}|$ - Tax

Рис. 4: Зависимость времени работы от количества машин и числа строк (a), а также зависимость времени работы (b), числа DC (c) и объёма лишней работы (d) от числа *предикатов* и наличия оптимизаций [5]

- G-Precision — процент "золотых" DC среди Σ_s;
- G-Recall — число "золотых" DC среди Σ_s разделить на общее число "золотых" DC;
- G-F-Measure — среднее гармоническое G-Precision и G-Recall.



(e) Threshold for Joinable Columns (f) Ranking Function in Pruning

(g) G-Precision - Tax (h) G-Recall - Tax

Рис. 5: Зависимость числа предикатов от минимума совпадающих значений для объединимых столбцов (e), времени поиска в глубину от значения a и порога $Inter()$ (f), а также зависимость G-Presision и G-Recall от значения a (g и h) [5]

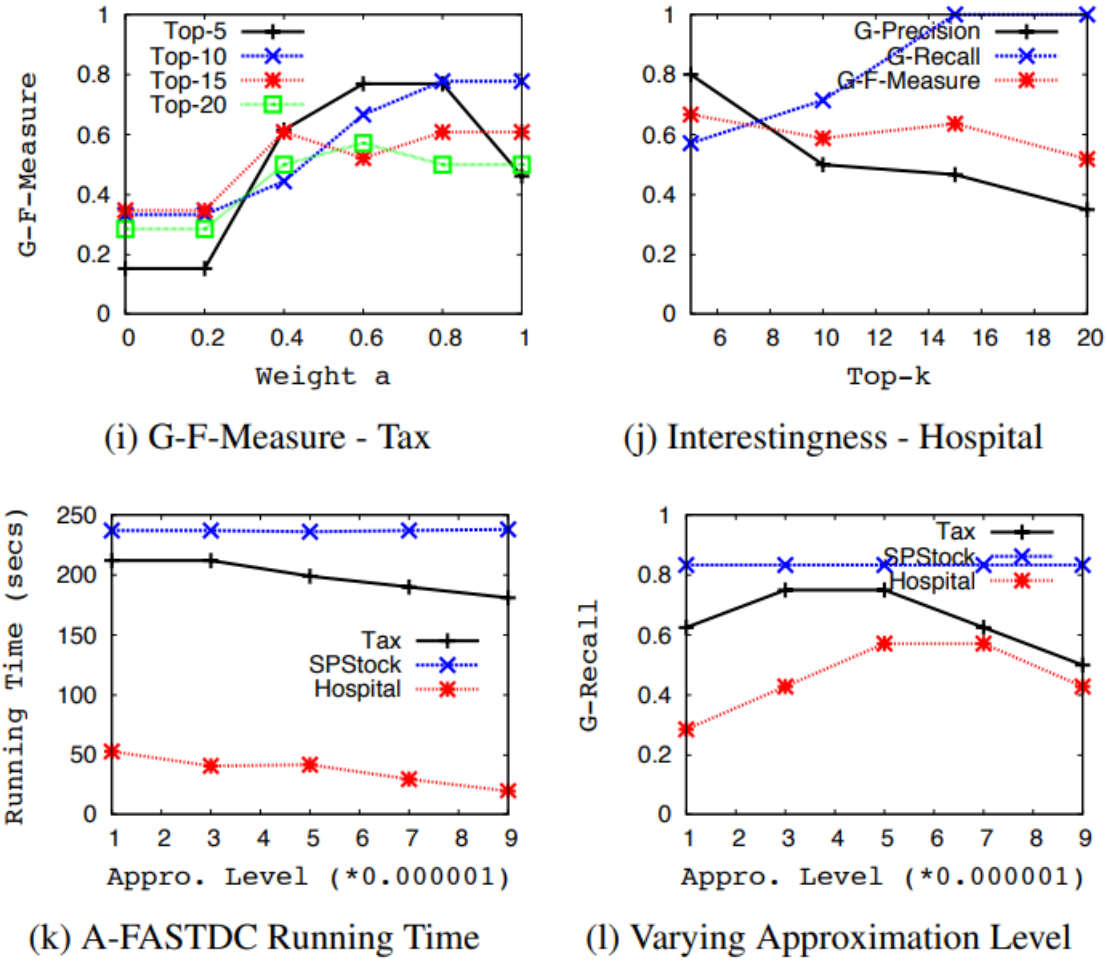


Рис. 6: G-F-Measure в зависимости от a (i), G-Presision, G-Recall и G-F-Measure для k наиболее "интересных" DC на массиве данных *Hospital* (j), время работы A-FASTDC (k), G-Recall для A-FASTDC в зависимости от уровня точности (l) [5]

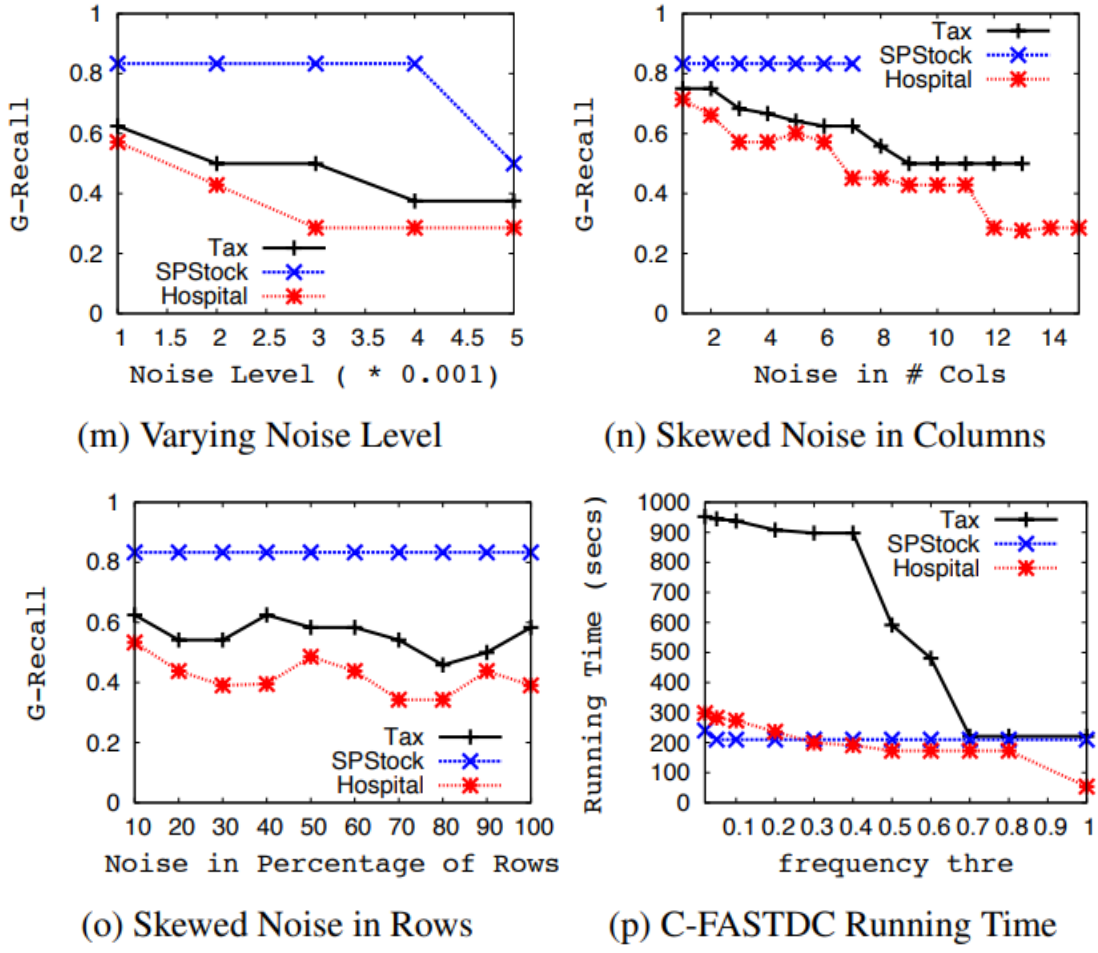


Рис. 7: G-Recall для A-FASTDC в зависимости от уровня шума (m), от распределения шума по столбцам и по строкам (n и o), а также зависимость времени работы C-FASTDC от τ (p) [5]

4.3. Эффективность распараллеливания построения *Evi*

Как видно на графике 4 (a), на котором шкала времени логорифмическая, ускорение при распараллеливании алгоритма почти линейное: на одном миллионе строк FASTDC+7 работал 3257 минут, а FASTDC+20 — 1228. В этом эксперименте **P** зафиксирован и имеет размер 50.

4.4. Эффективность динамического порядка *предикатов* и разделения пространства *DC*

Как видно на графике 4 (d), и динамический порядок, и разделение пространства уменьшают количество "потраченной работы", которая здесь озачает количество раз, которые алгоритму пришлось прервать поиск не из-за того, что все множества в *Evi* на данной ветке оказались покрыты, а потому, что больше не было *предикатов*, которые бы можно было добавить в покрытие. График 4 (b) показывает, что выигрыш по времени есть тоже. На графике 4 (c) видно, что эти оптимизации не влияют на число *DC*, которые найдёт алгоритм.

4.5. Тридцатипроцентный порог для объединимых столбцов

Как видно на графике 5 (e), при необходимом проценте общих значений больше 30% *предикаты* почти перестают отсеиваться.

4.6. Оптимальное значение *a* в формуле метрики *Inter()*

Как видно на графике 5 (g, h) и графике 6 (i), примерно при значении в 0.6 достигается максимальная близость к "золотым" *DC*, если судить по G-Precision, G-Recall и G-F-Measure.

4.7. A-FASTDC

Как видно на графике 6 (l), по началу при росте ϵ растёт и G-Recall, однако после определённого значения начинается обратный тренд. Связано это скорее всего с тем, что в какой-то момент начинают "обнаруживаться" *DC*, *предикаты* которых являются подмножеством $\varphi.Pres$ для какого-то $\Sigma_g \models \varphi$, и из-за этого сам φ отсекается.

График 7 (m) показывает, что хоть при фиксированном ϵ и возрастающем шуме точность и падает, A-FASTDC всё ещё способен находить "золотые" *DC*.

График 7 (n) показывает, что если зафиксировать шум на определённом значении, но распределять его по столбцам неравномерно, то он имеет меньше влияния на точность. Аналогичный эффект со строками (график 7 (o)) не наблюдается.

4.8. C-FASTDC

На графике 7 (p) видно, что при увеличении τ падает и время работы, что не удивительно, но не равномерно на всех массивах данных. Можно заметить, что время работы C-FASTDC *SP Stock* не меняется в завимости от τ, и что сильнее всего разница видна на *Tax*.

4.9. Оценка времени работы

На графике 4 (a) время растёт квадратично с ростом числа строк, что логично: перебор всех пар имеет сложность $O(|\mathbf{P}| \times |I|^2)$. Как видно по графику 4 (b и c), и число *минимальных DC*, и время работы растут экспоненциально с ростом числа *предикатов*.

4.10. Итоги

- Способ распараллеливания генерации Evi_I, предложенный авторами статьи, действительно даёт линейный выигрыш по времени;

- Динамический порядок *предикатов* действительно позволяет выиграть время и вычисления, как и разделение пространства DC;
- A-FASTDC может обнаруживать DC даже при наличии шума;
- Время работы FASTDC квадратично зависимо от числа строк в экпоненциально от числа *предикатов*.

Заключение

Целью данной работы являлось изучение алгоритма поиска DC с целью дальнейшей его реализации. Для достижения данной цели были выполнены следующие задачи:

1. Рассмотрен язык DC и аксиомы, которые к нему применимы;
2. Проведён обзор алгоритма FASTDC и его принципа работы;
3. Разобраны оптимизации алгоритма, предложенные авторами статьи;
4. Приведены и проанализированы результаты экспериментов авторов статьи;
5. Сделаны выводы об эффективности предложенных оптимизаций алгоритма.

Список литературы

[1] Abiteboul S., Hull R., Vianu V. Foundations of Databases. — 1995. — URL: https://wiki.epfl.ch/provenance2011/documents/foundations+of+databases-abiteboul-1995.pdf (online; accessed: 2023-06-02).

[2] Baudinet M., Chomicki J., Wolper P. Constraint-Generating Dependencies. — 1999. — URL: https://www.sciencedirect.com/science/article/pii/S002200009991632X (online; accessed: 2023-06-03).

[3] Bishop C. M. Pattern Recognition and Machine Learning (Information Science and Statistics). — 2006. — URL: https://www.microsoft.com/en-us/research/uploads/prod/2006/01/Bishop-Pattern-Recognition-and-Machine-Learning-2006.pdf (online; accessed: 2023-10-29).

[4] Bleifuß Tobias, et al. Approximate Discovery of Functional Dependencies for Large Datasets. — 2016. — URL: https://hpi.de/fileadmin/user_upload/fachgebiete/naumann/publications/PDFs/2016_bleifuß_approximate.pdf (online; accessed: 2021-12-11).

[5] Chu Xu, Ilyas Ihab F., Papotti Paolo. Discovering Denial Constraints. — 2013. — URL: http://www.vldb.org/pvldb/vol6/p1498-papotti.pdf (online; accessed: 2023-05-31).

[6] Mining Database Structure; Or, How to Build a Data Quality Browser / T. Dasu, T. Johnson, S. Muthukrishnan, V. Shkapenyuk. — 2002. — P. 240–251. — URL: https://www.academia.edu/6168918/Mining_Database_Structure_Or_How_to_Build_a_Data_Quality_Browser (online; accessed: 2023-11-07).