Санкт-Петербургский государственный университет

Кафедра информационно-аналитических систем

Группа 22.Б08-мм

Интеграция алгоритма поиска UCC в рамках платформы Desbordante

Синельников Михаил Алексеевич

Отчёт по учебной практике
в форме «Решение»

Научный руководитель:
ассистент кафедры ИАС Г. А. Чернышев

Санкт-Петербург
2024

Оглавление

Введение

В современном мире наблюдается глобальный рост количества данных. Их роль становится всё более важной, поэтому умение извлекать из них полезную информацию становится всё более востребованным. Информация, получаемая из данных, называется метаданными, а процесс их получения — профилированием данных. Примером метаданных для табличных данных может служить информация о числе его строк и колонок, количестве незаданных значений.

Немаловажную роль в работе с данными играет наукоёмкое профилирование. В отличие от обычного профилирования данных, оно ориентировано на нахождение нетривиальных зависимостей, позволяющих делать более осмысленные выводы о свойствах изучаемых данных.

Одни из наиболее известных неочевидных закономерностей в таблицах — функциональные зависимости (FD). Говорят, что функциональная зависимость $X \to Y$ (X, Y — множества атрибутов таблицы) удерживается, если для любой пары строк в таблице из совпадения их значений на атрибутах X следует совпадение их значений на атрибутах Y.

С каждым годом размеры данных растут, что создаёт проблему поддержания эффективности наукоёмкого профилирования. Для этого разрабатываются всё более производительные алгоритмы поиска различных нетривиальных закономерностей.

Desbordante[1] [1] — наукоёмкий профилировщик данных с открытым исходным кодом, предоставляющий пользователю возможность искать различные паттерны в своих данных, в том числе функциональные зависимости, а также некоторые другие виды зависимостей.

Другим примером нетривиальных закономерностей являются уникальные комбинации колонок (Unique Column Combination, UCC) — множества атрибутов, при проекции на которые все кортежи остаются уникальными. Данный вид зависимостей используется для поиска ключей в реляционных базах данных, поиска аномалий в данных и других

[1] https://github.com/Desbordante/desbordante-core/

задач управления данными [3].

На данный момент в Desbordante реализован[2] алгоритм HyUCC поиска UCC. В 2020 году в статье [3] был представлен алгоритм HPIValid. В секции экспериментов данной статьи была проанализирована производительность HPIValid в сравнении с HyUCC и было показано, что HPIValid работает быстрее. Однако для эксперимента использовалась реализация HyUCC на языке программирования Java, а HPIValid — на языке C++, что ставит под сомнение результаты данного эксперимента. В связи с этим возникла необходимость проверить производительность HPIValid в сравнении с реализованной в Desbordante на языке C++ версии HyUCC и интегрировать код алгоритма в Desbordante для расширения его функциональности и повышения эффективности поиска UCC.

[2] https://github.com/Desbordante/desbordante-core/tree/main/src/core/algorithms/ucc/hyucc

1. Постановка задачи

Целью работы является интеграция в платформу Desbordante и тестирование алгоритма HPIValid поиска UCC. Для достижения цели были поставлены следующие задачи:

- провести обзор предметной области и алгоритма HPIValid;
- интегрировать алгоритм в платформу Desbordante;
- произвести сравнение интегрированной и оригинальной версий алгоритма и сравнить производительность алгоритмов HPIValid и HyUCC.

2. Обзор

2.1. Основные определения

2.1.1. UCC

Все определения в данном подразделе взяты из работы [3].

Определение 2.1. Реляционная схема R — множество всех атрибутов в таблице. r — отношение (множество кортежей, размер которых равен числу атрибутов).

Определение 2.2. Набор расхождений (difference set) кортежей $r_1, r_2 \in r$ — множество атрибутов на которых значения кортежей различны: $\{a \in R \mid r_1[a] \neq r_2[a]\}$.

Определение 2.3. Уникальная комбинация колонок (UCC) — подмножество $S \subseteq R : \forall r_1, r_2 \in r \ \exists a \in S : r_1[a] \neq r_2[a]$.

Если $S \subseteq R$ — UCC, $S_1 \subseteq R$, $S_1 \supseteq S$, то S_1 — UCC.

Определение 2.4. UCC S называется минимальной, если не существует UCC $S_1 : S_1 \subset S$.

Справедливо следующее утверждение: S является UCC тогда и только тогда, когда S имеет непустое пересечение с наборами расхождений для каждой пары строк.

Действительно, если S является UCC, то любая пара строк отличается хотя бы на одном атрибуте $a \in S$, поэтому a принадлежит набору расхождений для данной пары строк, а значит, и его пересечению с S. Если же подмножество $S \subseteq R$ имеет непустое пересечение с наборами расхождений для каждой пары строк, то для любой пары строк существует атрибут $a \in S$, на котором эта пара строк отличается, а значит, S является UCC.

Пример 2.1. В качестве примера рассмотрим таблицу 1. Множество $\{Time, Course\}$ является UCC, так как единственная пара строк, совпадающая на атрибуте $Time$ (t_3, t_4), различается на атрибуте $Course$. Если

Таблица 1: Пример таблицы (взят из работы [2])

	Room_Nr	**Time**	**Course**	**Lecturer**
t_1	101	Wed 10:00 am	Programming	Miako
t_2	101	Wed 02:00 pm	Databases	Daniel
t_3	102	Fri 02:00 pm	Programming	Miako
t_4	101	Fri 02:00 pm	Databases	Saurabh

исключить из UCC атрибут *Course*, то полученное множество перестанет быть UCC из-за равенства строк t_3, t_4 на атрибуте *Time*. Атрибут *Time* также не может быть исключён из UCC, так как строки t_1, t_3 совпадают на атрибуте *Course*. Значит, данная UCC минимальна.

Множество, состоящее из наборов расхождений для каждой пары строк, имеет вид:
$\{\{Time, Course, Lecturer\}$,
$\{RoomNr, Time\}$,
$\{Time, Course, Lecturer\}$,
$\{RoomNr, Time, Course, Lecturer\}$,
$\{Time, Lecturer\}$,
$\{RoomNr, Course, Lecturer\}\}$.
Заметим, что $\{Time, Course\}$ имеет непустое пересечение со всеми наборами расхождений.

2.1.2. Гиперграфы

Определения в данном подразделе взяты из работ [3] и [4].

Определение 2.5. Гиперграф $\mathcal{F}$ — множество подмножеств множества вершин V ($F = \{F_1, \ldots, F_m\}, F_i \subseteq V$). F_i — рёбра гиперграфа. $|F|$ — число рёбер гиперграфа.

Определение 2.6. Вершинное покрытие гиперграфа $\mathcal{F}$ (transversal, hitting set) — подмножество V, имеющее непустое пересечение с каждым ребром $\mathcal{F}$.

Определение 2.7. Вершинное покрытие S гиперграфа $\mathcal{F}$ называется минимальным, если не существует вершинного покрытия $S_1 : S_1 \subset S$.

Определение 2.8. Обозначим за $\mathcal{F}(v)$ множество рёбер гиперграфа $\mathcal{F}$, включающих в себя вершину $v \in V$: $\mathcal{F}(v) = \{F \mid F \in \mathcal{F}, v \in F\}$.

Определение 2.9. Пусть $S \subseteq V$. Обозначим за $uncov(S)$ множество рёбер гиперграфа $\mathcal{F}$, не пересекающихся с S: $uncov(S) = \{F \mid F \in \mathcal{F}, S \cap F = \emptyset\}$.

$S \subseteq V$ является вершинным покрытием тогда и только тогда, когда $uncov(S) = \emptyset$.

Определение 2.10. Пусть $S \subseteq V, v \in S$. Ребро F гиперграфа $\mathcal{F}$ называется критическим для v, если $S \cap F = \{v\}$. Обозначим за $crit(v, S)$ множество всех критических для v рёбер: $crit(v, S) = \{F \mid F \in \mathcal{F}, S \cap F = \{v\}\}$.

$S \subseteq V$ является минимальным вершинным покрытием тогда и только тогда, когда $uncov(S) = \emptyset$ и $\forall v \in S\ crit(v, S) \neq \emptyset$. Действительно, если $\exists v \in S : crit(v, S) = \emptyset$, то любое ребро гиперграфа пересекается с S не только по v, откуда следует, что $S \setminus \{v\}$ останется вершинным покрытием.

Пример 2.2. Пусть $\mathcal{F} = \{\{1, 2\}, \{1, 3\}, \{2, 3, 4\}\}$. $S = \{1, 3, 4\}$ является вершинным покрытием гиперграфа $\mathcal{F}$. $crit(1, S) = \{\{1, 2\}\}$; $crit(3, S) = \emptyset$; $crit(4, S) = \emptyset$. Таким образом, S не является минимальным вершинным покрытием, и вершины 3 или 4 могут быть удалены из S. Вершинное покрытие $S_1 = S \setminus \{4\} = \{1, 3\}$ является минимальным, так как $crit(1, S) = \{\{1, 2\}\}$; $crit(3, S) = \{\{2, 3, 4\}\}$.

2.2. Алгоритм HPIValid

2.2.1. Принцип работы алгоритма

Алгоритм HPIValid предназначен для нахождения всех минимальных UCC в таблице. Основная идея алгоритма — сведение задачи поиска всех минимальных UCC к задаче поиска всех минимальных вершинных покрытий гиперграфа.

Возьмём в качестве множества вершин множество атрибутов, а в качестве гиперграфа — множество, состоящее из наборов расхождений

для каждой пары строк таблицы. Как было указано в подразделе 2.1.1, $S \subseteq R$ является UCC тогда и только тогда, когда оно имеет непустое пересечение со всеми наборами расхождений, что в точности означает, что S является вершинным покрытием для построенного гиперграфа.

Наивная реализация алгоритма предполагает построение гиперграфа путём вычисления наборов расхождений для каждой пары строк и дальнейшего нахождения его всех минимальных вершинных покрытий при помощи одного из существующих алгоритмов. В качестве такого алгоритма был выбран MMCS, представленный в статье [4] и являющийся наиболее быстрым. Однако этап формирования гиперграфа в данной реализации занимает $O(n^2)$ времени, что делает её неэффективной.

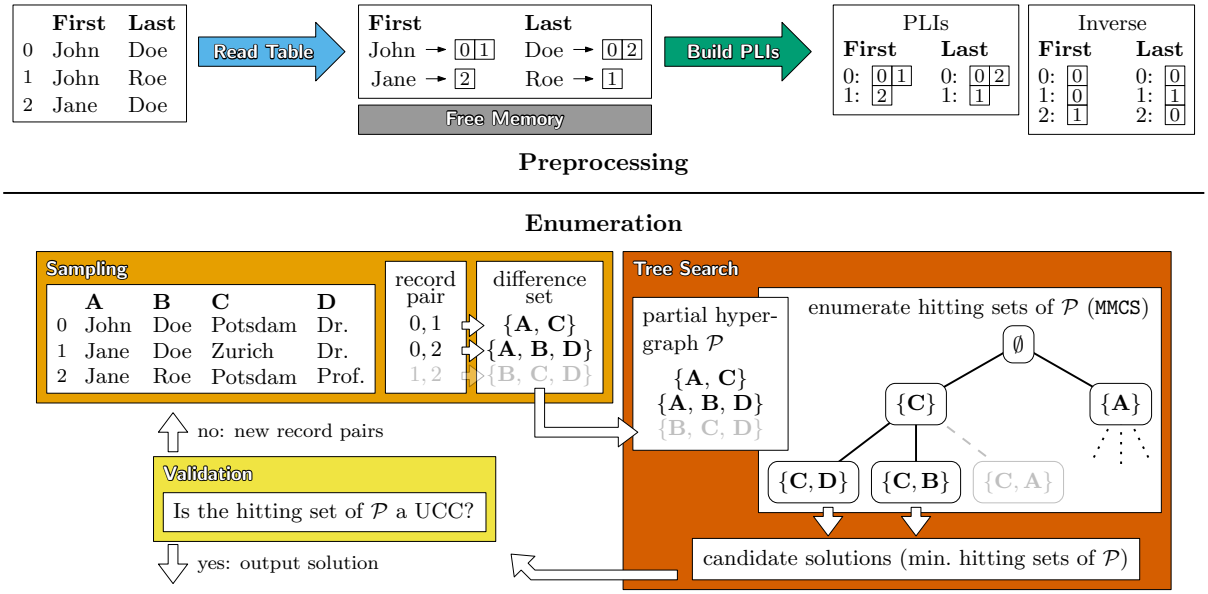

Рис. 1: Схема алгоритма HPIValid (взято из работы [3])

Продвинутая реализация HPIValid (рисунок 1) состоит из четырёх этапов:

1. Препроцессинг
2. Сэмплинг
3. Поиск
4. Валидация

На этапе препроцессинга алгоритм считывает таблицу и формирует структуры данных, необходимые для эффективной валидации UCC.

На этапе сэмплинга из множества всех пар строк выбирается некоторая его часть, для каждой пары вычисляется её набор расхождений и записывается в формируемый гиперграф. Таким образом решается проблема неэффективности перебора всех пар строк. Однако при этом появляется другая проблема: полученный гиперграф является неполным, вследствие чего последующий поиск может выдавать неверные результаты.

На этапе поиска для сформированного неполного гиперграфа запускается алгоритм MMCS, находящий минимальные вершинные покрытия в гиперграфе. При этом некоторые вершинные покрытия могут оказаться неверными для полного гиперграфа: в неполном гиперграфе отсутствуют некоторые рёбра, с которыми вершинное покрытие может не пересекаться, из-за чего оно не будет являться вершинным покрытием в полном гиперграфе, а значит и UCC.

На этапе валидации для каждого полученного вершинного покрытия проверяется, действительно ли оно является UCC. Если валидация успешна, то UCC добавляется к списку результатов. Если же полученное вершинное покрытие не является UCC, то существуют пары строк, совпадающие на вершинном покрытии. В таком случае происходит возвращение алгоритма на стадию сэмплинга, и к гиперграфу добавляются наборы расхождений для тех пар строк, из-за которых не прошла валидация. При этом поиск продолжается с того момента, на котором он остановился при нахождении очередного вершинного покрытия. В работе [3] приведено доказательство корректности работы MMCS для неполных гиперграфов с добавлением рёбер по ходу работы.

2.2.2. Алгоритм MMCS

Алгоритм основан на обходе дерева, в узлы которого являются наборами атрибутов. В вершине дерева находится пустой набор, а на каждом следующем слое число атрибутов в узле увеличивается на один. Во время работы алгоритма поддерживаются следующие переменные:

- S — текущий набор атрибутов;
- $uncov$ — множество рёбер гиперграфа, не пересекающихся с S ($uncov(S)$);
- $crit$ — множество, состоящее из множеств критических рёбер для каждой вершины из S ($crit[u] = crit(u, S)$);
- $CAND$ — множество кандидатов на добавление к S.

Как было указано в подразделе 2.1.2, вершинное покрытие S минимально тогда и только тогда, когда $uncov(S) = \emptyset$ и $\forall v \in S\ crit(v, S) \neq \emptyset$. В терминах введённых переменных условие выгладит так: $uncov = \emptyset$ и $\forall u \in S\ crit[u] \neq \emptyset$. Вторая часть условия называется условием минимальности. Поиск минимальных вершинных покрытий начинается с $S = \emptyset$ и продолжается до тех пор, пока кандидаты удовлетворяют условию минимальности.

```
global variable: crit[u] (initially ∅ for each u),
    uncov (initially F), CAND (initially V)
ALGORITHM MMCS (S)
1. if uncov = ∅ then output S ; return
2. choose a hyperedge F from uncov;
3. C := CAND ∩ F; CAND := CAND \ C
4. for each v ∈ C do
5.    call Update_crit_uncov (v, crit[], uncov)
6.    if crit(f, S ∪ v) ≠ ∅ for each f ∈ S then
          call MMCS(S ∪ v); CAND := CAND ∪ v
7.    recover the change to crit[] and uncov done in 5
8. end for
```

Рис. 2: Схема алгоритма MMCS (взято из работы [4])

На рисунке 2 представлен принцип работы алгоритма. При каждом вызове $MMCS(S)$ к текущему кандидату S (текущему узлу дерева) добавляются некоторые вершины v из множества кандидатов $CAND$ ($S \cup \{v\}$ — дети текущего узла, находящиеся в следующем слое). Затем для каждой такой вершины происходит рекурсивный вызов алгоритма (то есть выполняется поиск в глубину).

Инвариант алгоритма заключается в том, что S удовлетворяет условию минимальности, поэтому, если $uncov = \emptyset$ для текущего кандидата

S, то S является минимальным вершинным покрытием (строка 1 алгоритма). Поиск в данной ветви дерева на этом заканчивается, так как при добавлении любой вершины к S вершинное покрытие перестаёт быть минимальным.

Далее происходит выбор ребра $F \in uncov : |F \cap CAND|$ минимально. Чтобы S было минимальным вершинным покрытием, оно в том числе должно иметь непустое пересечение с F. Поскольку $F \in uncov$, $S \cap F = \emptyset$. Чтобы новое множество $S_1 \supseteq S$ было минимальным вершинным покрытием, нужно, чтобы оно как минимум содержало $v \in F$. Значит, для продолжения поиска можно рассматривать не всё множество $CAND$, а $C = CAND \cap F$. Для уменьшения числа рекурсивных вызовов выбирается такое F, что C минимально (такой выбор можно сделать эффективно).

На следующем этапе происходит перебор вершин из пересечения. Для каждой вершины $v \in C$ формируется новый кандидат $S \cup \{v\}$ и, если $S \cup \{v\}$ удовлетворяет условию минимальности, производится рекурсивный вызов. При этом для нового кандидата обновляются $crit$, $uncov$ и $CAND$.

3. Метод

Алгоритм HPIValid имеет открытую реализацию[3], сделанную авторами статьи [3].

3.1. Реализация HPIValid

Основные классы и структуры оригинальной реализации:

- Класс `HPIValid`;
- Структура `PLITable`;
- Класс `ResultCollector`;
- Структура `Config`;
- Класс `Hypergraph`;
- Класс `TreeSearch`.

Класс `HPIValid` — главный класс алгоритма. В конструкторе класса происходит вся работа алгоритма. Параметрами конструктора являются путь к таблице, а также ссылки на экземпляры вспомогательных классов `ResultCollector` и `Config`.

Структура `PLITable` предназначена для хранения всей информации о таблице, необходимой для работы алгоритма. Данные в структуру записываются на этапе препроцессинга (рисунок 1), после чего алгоритм не обращается к исходной таблице.

Класс `ResultCollector` предназначен для хранения всей статистики алгоритма, в том числе числа найденных UCC. В оригинальной реализации присутствует также его класс-наследник `ResultCollectorOutput`, позволяющий выводить найденные UCC в файл.

Структура `Config` — структура, хранящая параметры запуска алгоритма с установленными значениями по умолчанию.

[3] https://hpi.de/friedrich/docs/gitlab/thomas.blaesius/HPIValid

Класс `Hypergraph` — класс, хранящий информацию о гиперграфе и позволяющий производить основные действия с ним, например, добавление или удаление ребра.

Класс `TreeSearch` — класс, производящий основную часть работы алгоритма: сэмплинг, поиск и валидацию. Параметры его конструктора — ссылки на экземпляры классов `PLITable`, `ResultCollector` и `Config`. Алгоритм вызывается из конструктора класса `HPIValid` с помощью метода `TreeSearch::run()`.

Кроме того, в оригинальной реализации алгоритма присутствуют вспомогательные классы для работы со входной таблицей, использующиеся только на этапе препроцессинга: структура `ETable`, класс `Table`.

3.2. Интеграция алгоритма

В проекте Desbordante любой класс, реализующий алгоритм, должен наследоваться от класса `Algorithm` и переопределять некоторые общие методы. Все алгоритмы, реализующие поиск UCC, должны наследоваться от класса `UCCAlgorithm`, который наследуется от `Algorithm`, и переопределять все нужные методы.

В результате был создан класс `HPIValid`, наследующийся от класса `UCCAlgorithm`. В метод `ExecuteInternal`, который является переопределением метода класса `Algorithm`, был перенесён код, выполняющийся в конструкторе класса `HPIValid` оригинальной реализации.

В оригинальной реализации алгоритма все опции передаются через консольный ввод. В Desbordante требуемые алгоритмом опции указываются внутри переопределяемого метода `MakeOptionsAvailable` класса `Algorithm` с помощью метода `RegisterOption`. В классе `UCCAlgorithm` переопределён метод `MakeOptionsAvailable` с указанием нужных опций.

Поскольку большая часть опций оригинальной реализации относилась к особенностям чтения таблицы, а все опции, влияющие на работу самого алгоритма и хранящиеся в структуре `Config`, имели значения по умолчанию, подобранные авторами статьи для наиболее быстрого ис-

полнения алгоритма, было решено, что дополнительные опции, помимо указанных в классе `UCCAlgorithm`, не будут требоваться.

Для чтения входной таблицы в Desbordante используется реализованный там класс `InputTable`, а для хранения полученной информации — класс `ColumnLayoutRelationData`. Чтение таблицы выделено в отдельный от исполнения алгоритма метод — переопределяемый метод `LoadDataInternal` класса `Algorithm`. Кроме того, для получения структуры `PLITable`, используемой в оригинальной реализации, существуют реализованные в Desbordante методы, работающие с экземплярами класса `ColumnLayoutRelationData`.

Таким образом, этап препроцессинга в интегрированном алгоритме сводится к вызову нескольких уже реализованных функций. Для полного соответствия структуры `PLITable` получаемым в результате их выполнения структурам типы полей `PLITable` были немного изменены, что почти не повлияло на использование структуры в остальном алгоритме.

Результирующий список UCC для любого алгоритма их поиска в Desbordante должен храниться в поле `ucc_collection_`, определённом в классе `UCCAlgorithm` вместе с методами его получения. В оригинальной реализации каждая найденная UCC обрабатывается в специальном методе `ResultCollector::ucc_found()`. При этом в реализации этого метода информация о UCC не используется и подсчитывается только количество найденных UCC. В переопределении этого метода классом-наследником `ResultCollectorOutput` информация о UCC записывается в файл.

В результате в интегрированную версию алгоритма был включён только класс `ResultCollector`, а метод `ResultCollector::ucc_found()` был изменён с целью добавления найденных UCC в `ucc_collection_`.

4. Эксперимент

Для тестирования интегрированного алгоритма на корректность было решено сравнить его результаты с результатами уже реализованных в Desbordante алгоритмов поиска UCC. Для этого был изменён файл `src/test/test_ucc_algorithms.cpp` с целью добавления проверки нового алгоритма на представленных там тестах (все тесты в Desbordante используют библиотеку GoogleTest). В результате запуска тестов результаты интегрированного алгоритма HPIValid совпали с результатами ранее реализованного алгоритма HyUCC.

Кроме того, была проведена проверка качества кода. Для этого были использованы статические анализаторы кода PVS-Studio[4] и CPPLint[5], позволяющие найти большое число различных ошибок.

PVS-Studio позволяет найти уязвимости кода, ошибки приведения типов, неиспользуемые переменные, а также проверяет код на соответствие стандарту MISRA. CPPLint позволяет проверить код на соответствие Google C++ Style Guide. В таблицах 2 и 3 представлены некоторые ошибки, встретившиеся в коде оригинального и интегрированного алгоритмов. Таким образом, качество кода интегрированного алгоритма незначительно улучшилось.

Для проверки производительности интегрированного алгоритма в сравнении с оригинальной реализацией были выбраны наборы данных с различным числом строк и колонок. На каждом из них оба алгоритма были запущены по 10 раз, после чего было посчитано среднее время их работы.

Результаты тестирования представлены в таблице 4. Из результатов видно, что на одних наборах данных время работы интегрированной версии алгоритма осталось примерно таким же, как и время работы оригинальной версии, на других интегрированная версия стала работать быстрее, а на третьих — медленнее.

Для сравнения производительности алгоритма HPIValid и алгорит-

[4] https://pvs-studio.com/en/

[5] https://github.com/cpplint/cpplint

Таблица 2: Ошибки, выявленные PVS-Studio

Название ошибки	**Число появлений в оригинальном алгоритме**	**Число появлений в интегрированном алгоритме**
V820. The variable is not used after copying. Copying can be replaced with move/swap for optimization.	1	0
V730. Not all members of a class are initialized inside the constructor.	1	0
V2507. MISRA. The body of a loop or conditional statement should be enclosed in braces.	8	0
V2506. MISRA. A function should have a single point of exit at the end.	5	2

Таблица 3: Ошибки, выявленные CPPLint

Название ошибки	**Число появлений в оригинальном алгоритме**	**Число появлений в интегрированном алгоритме**
Add missing #include.	4	0
Constructors callable with one argument should be marked explicit.	1	0
At least two spaces is best between code and comments.	2	0
Line ends in whitespace. Consider deleting these extra spaces.	2	0

ма HyUCC использовались реализованные в Desbordante на языке C++ версии алгоритмов. Таким образом, сравнение получилось более объективным, чем сравнение Java-реализации HyUCC с C++-реализацией HPIValid в статье [3]. Данное сравнение проводилось на тех же наборах

данных, что и предыдущее.

Таблица 4: Среднее время работы алгоритмов на разных наборах данных

Название набора данных	**Время работы оригинальной версии HPIValid, с**	**Время работы интегрированной версии HPIValid, с**	**Время работы HyUCC, с**
ncvoter_allc	1.82	2.01	1031.97
fd-reduced-30	3.57	1.34	71.02
DITAG_FEATURE	3.74	2.37	19.73
SG_BIOENTRY	0.09	0.01	0.31
SG_LOCATION	0.72	0.03	1.64
flight_r1001_c109	0.08	0.11	3.33
EpicMeds	0.63	0.64	4.56
EpicVitals	0.35	0.41	3.01
iowa1kk	0.40	0.34	7.20

Результаты сравнения представлены в таблице 4. Данные результаты показывают, что алгоритм HPIValid действительно более эффективен, чем HyUCC, причём на некоторых наборах данных время работы уменьшилось значительно. На маленьких наборах данных преимущество незначительно и может быть списано на погрешность измерений, а на больших HPIValid работает в несколько раз быстрее, чем HyUCC.

Заключение

В ходе работы были выполнены следующие задачи:

- сделан обзор предметной области и алгоритма HPIValid;
- алгоритм был интегрирован в Desbordante;
- проведено сравнение производительности оригинальной реализации алгоритма, его интегрированной версии и алгоритма HyUCC.

Исходный код алгоритма доступен на GitHub[6].

[6] https://github.com/MichaelS239/Desbordante/tree/hpivalid (дата обращения: 17 июня 2024 г.).

Список литературы

[1] Desbordante: from benchmarking suite to high-performance science-intensive data profiler (preprint) / George A. Chernishev, Michael Polyntsov, Anton Chizhov et al. // CoRR. — 2023. — Vol. abs/2301.05965. — arXiv : 2301.05965.

[2] Discovering Functional Dependencies through Hitting Set Enumeration / Tobias Bleifuß, Thorsten Papenbrock, Thomas Bläsius et al. // Proc. ACM Manag. Data. — 2024. — mar. — Vol. 2, no. 1. — 24 p. — URL: https://doi.org/10.1145/3639298.

[3] Hitting set enumeration with partial information for unique column combination discovery / Johann Birnick, Thomas Bläsius, Tobias Friedrich et al. // Proc. VLDB Endow. — 2020. — jul. — Vol. 13, no. 12. — P. 2270–2283. — URL: https://doi.org/10.14778/3407790.3407824.

[4] Murakami Keisuke, Uno Takeaki. Efficient algorithms for dualizing large-scale hypergraphs // Discrete Appl. Math. — 2014. — jun. — Vol. 170. — P. 83–94. — URL: https://doi.org/10.1016/j.dam.2014.01.012.