Санкт-Петербургский государственный университет

Кафедра информационно-аналитических систем

Группа 23.Б08-мм

Внедрение алгоритма BBND в проект с открытым исходным кодом “Desbordante”

ЕМЕЛЬЯНОВ Максим Михайлович

Отчёт по учебной практике
в форме «Производственное задание»

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

Санкт-Петербург
2025

Оглавление

Введение

Профилирование данных [5] — это процесс получения вспомогательной информации о данных, к примеру, информации о зависимостях в них.

Одним из самых известных примеров профилирования данных является поиск функциональных зависимостей [3]. Подобные закономерности в данных называются [2] примитивами. Они используются для решения множества прикладных задач, включая оптимизацию запросов к базам данных, восстановление данных и других. Для поиска примитивов разработано множество алгоритмов поиска и верификации.

Ещё одним примером зависимостей в данных являются числовые зависимости (Numerical Dependencies, NDs) [4]. Числовая зависимость из атрибутов X в атрибуты Y с весом k ($X \xrightarrow{k} Y$) выполняется, если каждому уникальному значению из X соответствует не более k уникальных значений из Y. Числовая зависимость $X \xrightarrow{k} Y$ называется минимальной, если нет никакой другой числовой зависимости $X \xrightarrow{m} Y$, такой что $m < k$.

На текущий момент нет инструмента для анализа данных, который мог бы находить числовые зависимости. В связи с этим было принято решение реализовать алгоритм поиска числовых зависимостей [4] и внедрить его в “Desbordante” [1], инструмент с открытым исходным кодом для профилирования данных.

Автор настоящей работы занимался анализом, реализацией, и внедрением данного алгоритма в “Desbordante” совместно с Сениченковым Петром. Часть общей работы, выполненная Сениченковым Петром, представлена в его отчете по учебной практике “Реализация алгоритма поиска числовых зависимостей в «Desbordante»” [6].

1. Постановка задачи

Целью работы является реализация алгоритма для поиска числовых зависимостей на C++ для проекта с открытым исходным кодом “Desbordante”. Для её выполнения были поставлены следующие задачи:

1. подготовить обзор предметной области алгоритмов поиска числовых зависимостей;
2. реализовать следующие компоненты алгоритма BBND [4]:
   (a) базовый класс алгоритма для последующего внедрения в C++-ядро проекта;
   (b) оптимизированная очередь выдачи вариантов пути для нахождения числовой зависимости (ActiveNDPaths [4]);
3. обеспечить поддержку вызова алгоритма BBND из Python-программ в рамках платформы “Desbordante” и создать пример вызова;
4. провести апробацию получившегося алгоритма.

2. Обзор

2.1. Примитивы

На данный момент в Desbordante поддерживается множество различных примитивов. Среди них самым известным и изученным примитивом является функциональная зависимость [3] (ФЗ, Functional Dependency).

Функциональные зависимости представляют собой важный примитив, который активно применяется для нормализации схем баз данных, оптимизации запросов, очистке данных и в других областях, связанных с обработкой информации.

В случаях, когда функциональная зависимость не выполняется, полезно охарактеризовать, насколько близки данные к ее выполнению. В связи с этим было создано множество обобщений функциональных зависимостей, которые являются более гибкими примитивами, допускающими некоторые погрешности в данных и которые в частном случае сводятся к классическим функциональным зависимостям.

Одним из таких обобщений являются приближенные функциональные зависимости [8] (ПФЗ, Approximate Functional Dependency). Эти зависимости включают параметр, задающий порог допустимого отклонения от классической функциональной зависимости. Если значение параметра установлено равным нулю, то приближенная функциональная зависимость превращается в точную функциональную зависимость. Однако на практике обычно задают ненулевое значение параметра, что позволяет учитывать возможные неточности в данных. Кроме того, это позволяет выявлять зависимости, которые не являются строгими функциональными зависимостями, но представляют интерес для анализа. Такие зависимости могут быть полезны в различных областях работы с данными.

Однако приближенные функциональные зависимости — не единственное обобщение функциональных зависимостей. Другим важным обобщением являются числовые зависимости (ЧЗ, Numerical

Dependencies), которые были упомянуты ранее. Основная идея числовых зависимостей заключается в определении количества уникальных значений в наборе столбцов Y, соответствующих каждому уникальному значению из набора столбцов X. Такая информация может быть полезна для анализа данных, так как позволяет оценить взаимосвязь между атрибутами.

Поскольку выявление экземпляров примитивов вручную, особенно на больших объемах данных, представляет собой крайне сложную задачу, для этого было разработано множество алгоритмов поиска и верификации. Алгоритмы поиска позволяют на основе входных данных определить список экземпляров определенного примитива, которые выполняются на этих данных. Алгоритмы верификации, в свою очередь, не предназначены для поиска **всех** экземпляров, а используются для проверки гипотез о наличии **определенного** экземпляра в данных. Они позволяют определить, является ли предположение о экземпляре примитива верным или ложным, и иногда обосновать этот вывод.

Как следует из вышеизложенного, поиск и верификация экземпляров примитивов представляют собой важные задачи. В платформе “Desbordante” уже реализован [7] ряд алгоритмов для поиска и верификации функциональных зависимостей и приближенных функциональных зависимостей, таких как Pyro и HyFD. Однако для числовых зависимостей в настоящее время доступен только алгоритм верификации. В связи с этим была поставлена цель реализовать алгоритм поиска числовых зависимостей для платформы “Desbordante”.

2.2. Алгоритм BBND

Тема поиска числовых зависимостей в данных на данный момент является малоизученной, и количество публикаций по данной теме крайне ограничено. В частности, существует только одна статья, описывающая алгоритм поиска числовых зависимостей [4]. Этот алгоритм, называемый BBND (branch & bound algorithm for ND derivation, рус. алгоритм поиска ND методом ветвей и границ), позволяет выводить числовые

зависимости из входного набора числовых зависимостей. Для этого он использует правила $REDS$:

- **R**eflexivity (R) : $\vdash X \to X$
- **E**xtended transitivity (E) : $X \xrightarrow{k} YW \ \& \ Y \xrightarrow{l} Z \vdash X \xrightarrow{k \cdot l} YWZ$
- **D**ecomposition (D) : $X \xrightarrow{k} YZ \vdash X \xrightarrow{k} Y$
- **S**uccessor (S) : $X \xrightarrow{k} Y \vdash X \xrightarrow{k+l} Y$

Использование этих правил позволяет получить набор числовых зависимостей, которые могут быть выведены из входных данных.

Оригинальный алгоритм BBND 1 принимает на вход набор числовых зависимостей, из которых будет выводиться целевая зависимость, а также левую и правую части целевой числовой зависимости, которую необходимо вывести.

На первом этапе алгоритм выполняет предварительную обработку данных. Начальная вершина помещается в очередь активных путей (ActiveNDPaths), после чего проводится проверка возможности вывода целевой числовой зависимости. Затем из входных данных удаляются бесполезные зависимости, к примеру, повторения, с помощью функции RemoveUselessNDs 1.

После завершения предварительных процедур начинается основной этап работы алгоритма. Из очереди извлекается первая числовая зависимость, после чего рассматриваются все её “умные” расширения, то есть те, которые не будут заведомо хуже уже найденных и не являются дубликатами ранее рассмотренных зависимостей. Для каждого расширения проводится проверка: если числовая зависимость является целевой и её вес меньше текущего результата, то текущий результат обновляется. Если зависимость не является целевой, но удовлетворяет ряду условий (строка 11 на Рис. 1), она добавляется в очередь для дальнейшего расширения. Алгоритм завершает свою работу, когда в очереди активных путей не остаётся числовых зависимостей для рассмотрения.

Рис. 1: Код алгоритма метода ветвей и границ. Источник [4].

The BBND algorithm

Input G_Δ, X, Y

Output $k^{\perp}_{RED}(X, Y)$

1: $k^{\perp}_{RED}(X, Y) \leftarrow \infty$

2: $ActiveNDPaths \leftarrow \{G^x_\emptyset\}$

3: **if** $Y \not\subseteq Atrr(G_\Delta)$ **then return** $k^{\perp}_{RED}(X, Y)$

4: RemoveUselessNDs(G_Δ, X, Y)

5: **while** $ActiveNDPaths \neq \emptyset$ **do**

6: $G^X_\Pi \leftarrow$Pop($ActiveNDPaths$)

7: **for all** $G^X_{\Pi_i} \in$ SmartExtensions(G^X_Π) **do**

8: **if** $Y \subseteq Attr(G^X_{\Pi_i})$ **then**

9: **if** $\omega(G^X_{\Pi_i}) < k^{\perp}_{RED}$ **then**

10: $k^{\perp}_{RED}(X, Y) \leftarrow \omega(G^X_{\Pi_i})$

11: **end if** $\omega(G^X_{\Pi_i}) < k^{\perp}_{RED} \wedge$
$\neg$ IsDominated($G^X_{Pi_i}, ActiveNDPaths$)
$\wedge$ IsMinimal($G^X_{\Pi_i}$) **then**

12: Push($ActiveNDPaths, G^X_{\Pi_i}$)

13: **return** $k^{\perp}_{RED}(X, Y)$

Основные отличия данного алгоритма от простого перебора заключаются в использовании “умных” расширений числовых зависимостей, условий для добавления новых зависимостей в очередь активных путей и самого класса очереди активных путей. Эти элементы позволяют оптимизировать процесс вывода числовой зависимости, исключая из рассмотрения варианты, которые являются дубликатами уже рассмотренных или заведомо не являются решением, что значительно сокращает пространство поиска.

3. Описание решения

3.1. Применение алгоритма BBND

Алгоритм BBND предназначен для поиска веса **одной** конкретной числовой зависимости. Наша идея заключается в адаптации этого алгоритма для поиска **всех** числовых зависимостей в данных. Далее представлено описание использования BBND в рамках данной идеи.

На первом этапе вычисляется начальный набор числовых зависимостей с использованием уже реализованного верификатора числовых зависимостей. В этом наборе каждая зависимость имеет ровно один атрибут в левой и правой частях. Этот набор используется в качестве базового для вывода последующих числовых зависимостей.

На втором этапе применяется алгоритм BBND, который вычисляет все числовые зависимости, у которых размерность левой части равна единице, а правой — двум. Найденные таким образом числовые зависимости добавляются в начальный набор данных для последующего шага алгоритма.

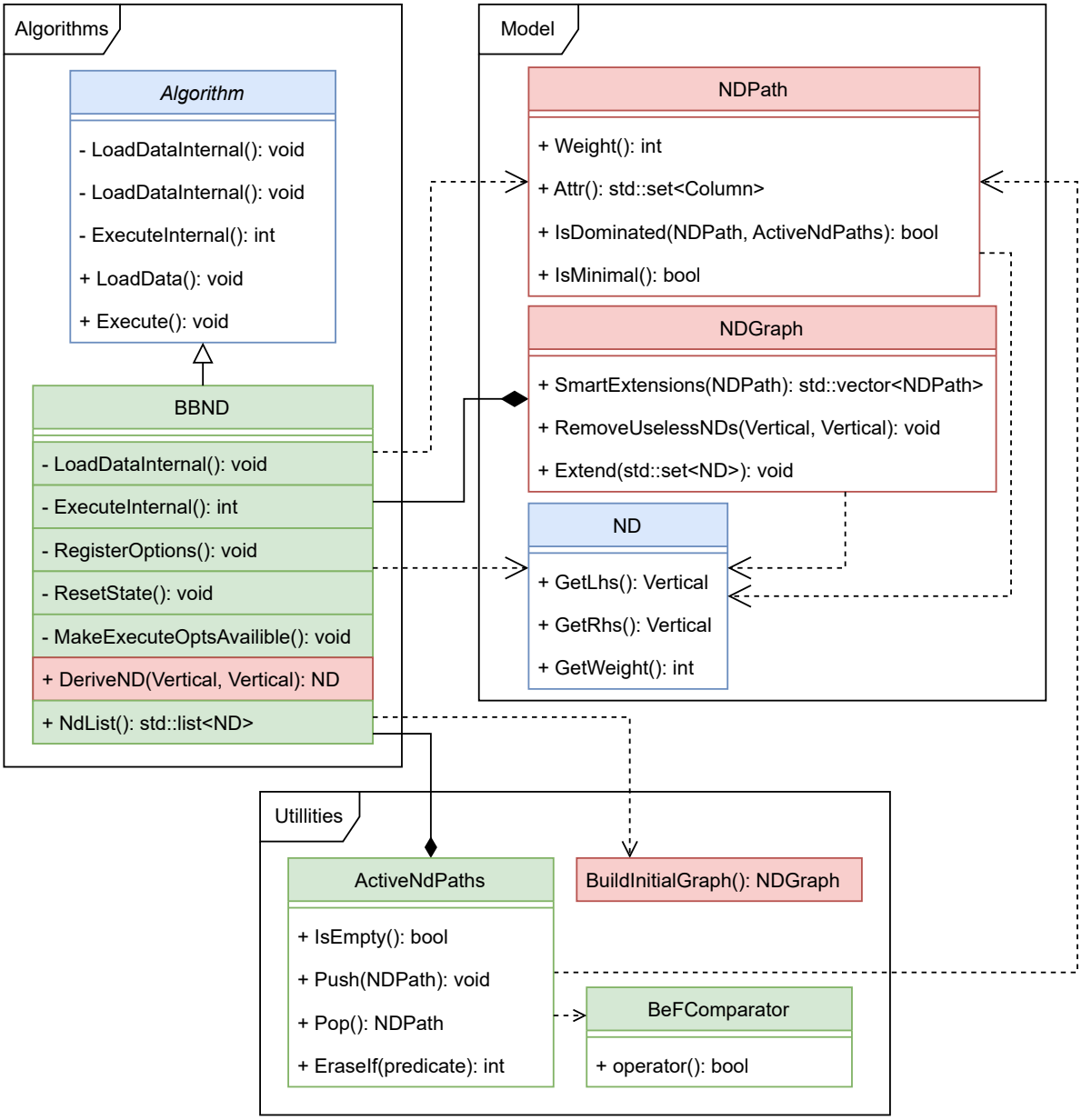
На последующих шагах алгоритм BBND последовательно запускается для числовых зависимостей с увеличивающейся размерностью правой, а затем и левой частей. Этот процесс продолжается до тех пор, пока не будут вычислены все числовые зависимости, у которых размерности левой и правой частей не превышают заданных ограничений или количества атрибутов в таблице.

3.1.1. Архитектура решения и распределение работы

Архитектура реализации алгоритма включает три основных модуля: Algorithms, Model и Utilities. Детальная иерархия классов отображена на Рисунке 2, где представлена UML-диаграмма с цветовой маркировкой:

- синим цветом обозначены компоненты, существовавшие в исходной реализации “Desbordante”;

Рис. 2: Иерархия классов BBND.



- красным — компоненты, которые были реализованы Сениченковым Петром [6];
- зеленым — новые компоненты, которые были реализованы в ходе данной работы.

В Algorithms представлены основной класс BBND, который будет вызываться для нахождения числовых зависимостей, и базовый класс алгоритма, от которого должны быть отнаследованы все классы в “Desbordante”.

В Model представлены классы, отвечающие за то, чтобы можно было представить числовую зависимость и смежные с ней сущности в виде экземпляра некоторого класса.

В Utilities представлены вспомогательные классы и функции, которые используются при запуске и работе алгоритма.

3.2. Базовый класс алгоритма

После рассмотрения теоретической части перейдем к описанию реализации. В платформе “Desbordante” все классы, предназначенные для поиска и верификации примитивов, должны наследоваться от базового класса Algorithm. Данный базовый класс предоставляет стандартный функционал для реализации алгоритмов и задает единый интерфейс взаимодействия.

Первым шагом является создание нового класса, наследуемого от Algorithm, и переопределение нескольких ключевых методов, таких как:

1. Базовый конструктор;
2. Регистрация настроек;
3. Загрузка данных;
4. Внутренний запуск алгоритма.

Упрощенная структура класса алгоритма представлена в листинге 1.

Листинг 1: Листинг кода с упрощенной структурой класса алгоритма BBND.

```
class Bbnd : public Algorithm {
private:
    util::PrimitiveCollection<model::ND> nd_collection_;
    std::shared_ptr<ColumnLayoutRelationData> relation_;

    void RegisterOptions();
```

```
    void MakeExecuteOptsAvailable() override;
    void LoadDataInternal() override;
    unsigned long long ExecuteInternal() override;

public:
    Bbnd();

    /* Returns the list of discovered NDs */
    std::list<model::ND> const& NdList() const noexcept {
        return nd_collection_.AsList();
    }
};
```

Базовый конструктор реализуется с использованием конструктора класса Algorithm. При переопределении конструктора класса алгоритма в платформе Desbordante требуется задать параметры для настройки алгоритма и определить, какие из них могут быть изменены пользователем.

RegisterOptions. Внутри данного метода определяются параметры для настройки работы алгоритма. Это реализуется с использованием существующего метода RegisterOption из базового класса и классов опций, расположенных в пространстве имен config. Данный метод вызывается в первую очередь внутри конструктора класса.

MakeOptionsAvailable. Этот метод, принадлежащий базовому классу, обеспечивает доступность параметров настройки алгоритма для пользователя. Для указания доступных параметров используются их имена, определенные в пространстве имен config::names. Метод вызывается в конструкторе после выполнения метода RegisterOptions.

LoadDataInternal. Этот метод отвечает за подготовку внутренних данных перед запуском алгоритма. В первую очередь загружается таблица с данными с использованием стандартных функций и классов таблиц, реализованных в платформе “Desbordante”. На основе загруженной таблицы формируется начальный набор числовых зависи-

мостей, у которых размерность левой и правой частей равна единице. Для построения начального набора данных используется функция BuildInitialGraph, описанная Сениченковым Петром в отчете по учебной практике “Реализация алгоритма поиска числовых зависимостей в «Desbordante»” [6]. На завершающем этапе выполняется проверка корректности данных и обновление внутренних полей класса для последующего запуска алгоритма.

ExecuteInternal. Этот метод вызывается при запуске алгоритма и содержит основную логику для поиска числовых зависимостей. После завершения выполнения метод обновляет внутренние поля класса, данные из которых могут быть получены с помощью метода NDList, возвращающего список найденных числовых зависимостей. Кроме того, метод возвращает время выполнения алгоритма в миллисекундах в виде целого числа.

3.3. ActiveNDPaths

Алгоритм BBND вычисляет вес целевой числовой зависимости $X \xrightarrow{k} Y$, используя начальный набор известных числовых зависимостей, обозначаемый δ. На основе этого набора и правил $REDS$, описанных выше, строится граф, вершинами которого являются атрибуты X и Y числовых зависимостей из δ,, а весами дуг выступают веса числовых зависимостей, связывающих эти атрибуты. Граф формируется путем последовательного добавления числовых зависимостей из набора δ до тех пор, пока в нем не будут представлены все атрибуты целевой числовой зависимости. Вес целевой числовой зависимости определяется как произведение весов всех дуг полученного графа.

Поскольку порядок добавления числовых зависимостей влияет на результат, необходимо перебрать все возможные комбинации добавления. Этот процесс можно представить в виде дерева, где каждая вершина соответствует графу, а дуги представляют добавленные числовые зависимости. Корнем дерева является вершина, содержащая только атрибуты X целевой числовой зависимости. Результатом работы алгорит-

ма является минимальный вес среди всех листьев этого дерева.

В алгоритме применяется ряд оптимизаций, направленных на сокращение пространства поиска и исключение ветвей дерева, которые заведомо не приведут к улучшению текущего результата. Для оптимизации процесса ветвления и определения, какие ветви следует продолжать, а какие — исключить, используются функции SmartExtensions, IsDominated и IsMinimal. Корректность этих функций доказана в оригинальной статье [4], а их описание было выполнено Сениченковым Петром в отчете по учебной практике "Реализация алгоритма поиска числовых зависимостей в «Desbordante»" [6].

Еще одной важной оптимизацией алгоритма является использование упорядоченной очереди для обработки вершин при поиске числовых зависимостей. Этот порядок определяет последовательность обработки вершин. Класс, отвечающий за эту функциональность, называется ActiveNDPaths. Он позволяет добавлять новые промежуточные результаты (вершины) и получать следующую вершину для обработки. Внутри класса определяется порядок выдачи вершин на основе имеющихся данных. Реализация данного функционала выполнена как расширение стандартного класса std::set с использованием специального компаратора, который передается при создании объекта.

Порядок обхода вершин определяется компаратором, который может быть реализован различными способами. В оригинальной статье [4] предложены четыре различных подхода к определению порядка обхода вершин, каждый из которых представляет собой эвристику. Как показано в статье [4], эти эвристики действительно способствуют ускорению работы алгоритма. В связи с этим было принято решение реализовать класс ActiveNDPaths как шаблонный, что позволяет передавать в него различные компараторы. Такое решение обеспечивает гибкость и упрощает исследование новых подходов к определению порядка обхода вершин.

Наилучшие результаты демонстрирует стратегия обхода, известная как best-first (BeF) [4]. Основная идея этой стратегии заключается в том, чтобы в первую очередь обрабатывать пути, которые имеют наи-

большее пересечение с атрибутами результирующей числовой зависимости (ND). Формально это означает, что значение $|\text{Attr}(Graph) \cap Y|$ максимально. Реализация данной стратегии выполнена в виде отдельной функции BeFCmpr, которая передается в качестве компаратора при создании оптимизированной очереди 2.

Листинг 2: Листинг кода с примерной структурой ActiveNDPaths и BeFCmpr.

```
bool BeFCmpr(std::pair<model::NDPath,
             std::shared_ptr<std::set<Column>>> a,
             std::pair<model::NDPath,
             std::shared_ptr<std::set<Column>>> b){
    int res = IntersectionWithEnd(a.first, a.second)
            - IntersectionWithEnd(b.first, b.second);

    if (res > 0) {
        return true;
    }
    if (res == 0 && a.first.Weight() < b.first.Weight()) {
        return true;
    }

    return false;
}


template<typename _Compare = std::less<model::NDPath>>
class ActiveNdPaths {
    private:
        std::set<std::pair<model::NDPath,
            std::shared_ptr<std::set<Column>>>,
            _Compare> queue_;
        std::shared_ptr<std::set<Column>> end_;
```

```
    public:
        ActiveNdPaths(std::set<Column> && end);

        // Changing methods
        model::NDPath Pop();
        void Push(model::NDPath&& new_path);

        // Checkout methods
        inline bool IsEmpty();

}
```

3.4. Поддержка вызова алгоритма из Python программ

Для обеспечения удобства взаимодействия с пользователем в платформе “Desbordante” реализована Python-оболочка, которая позволяет вызывать функции, написанные на C++, из Python. Для создания этой оболочки используется библиотека PyBind.

Для добавления функциональности в Python-оболочку необходимо создать функцию BindBbnd, описанную в Листинге 3. В функции реализуются методы, которые должны быть доступны из Python для нашего алгоритма. Затем эту функцию следует указать в файле *src/python_bindings/bindings.cpp* в качестве дополнительной функции. В результате для нашего алгоритма автоматически будут созданы базовые методы для создания объекта, загрузки данных и запуска алгоритма. Кроме того, необходимо реализовать метод *get_nds*, возвращающий выявленные числовые зависимости по завершении работы алгоритма.

Листинг 3: Листинг кода функции BindBbnd.

```
#include «bind_bbnd.h»

#include <pybind11/pybind11.h>
```

```
#include <pybind11/stl.h>

#include «algorithms/nd/bbnd/BBND_algorithm.h»
#include «py_util/bind_primitive.h»


void BindBbnd(pybind11::module_& main_module) {
    using namespace algos;

    auto bbnd_module =
        main_module.def_submodule('bbnd');

    BindPrimitiveNoBase<Bbnd>(bbnd_module, 'BBND')
            .def('get_nds', [](Bbnd const& bbnd) {
                return bbnd.NdList();});
}
```

Для демонстрации использования класса алгоритма был разработан пример на Python. Этот пример находится в файле *examples/basic/mining_nd/bbnd.py* 4.

Листинг 4: Листинг кода с примером использования python-оболочки. Файл examples/bbnd.py.

```
import desbordante

algo = desbordante.bbnd.algorithms.BBND()
algo.load_data(table=(VALID_PASSPORTS_TABLE, ',', True))
algo.execute()
result = algo.get_nds()
print('NDs:')
for nd in result:
    if(len(nd.lhs.indices) == 1) and (len(nd.rhs.indices) == 1):
        print(nd)
}
```

Данный код осуществляет поиск числовых зависимостей в таблице 1, расположенной в файле *examples/datasets/nd_verification_datasets/valid_passports_1.csv*. В результате работы выводятся зависимости, у которых размер левой и правой частей равен единице. Часть вывода представлена в листинге 5.

Таблица 1: valid_passports_1.csv

Name	ID	Issuing city	Entry permit	Expiration date
Kordon Kallo	375F0-KE12I	Orvech Vonor	-	05.03.2040
Nathan Cykelek	9I2-4H2	-	Kolechia	09.10.2028
Grant Baker	1GMFL-5LRD6	East Greshtin	-	28.07.2039
Kordon Kallo	7JH-35A	-	Orbistan	07.01.2027
Grant Baker	8H6-772	-	Antegria	19.11.2029
Khaled Istom	9KLA2-HH66N	East Greshtin	-	21.12.2041

Листинг 5: Часть вывода программы из листинга 4.

```
[Entry permit] —3—> [Expiration date]
[Entry permit] —2—> [Issuing city]
[Entry permit] —3—> [ID]
[Entry permit] —3—> [Name]
[Issuing city] —3—> [Expiration date]
[Issuing city] —3—> [Entry permit]
[Issuing city] —3—> [ID]
[Issuing city] —3—> [Name]
```

Из полученных данных мы можем заключить, к примеру, что разрешение на въезд выдают не более чем в двух городах, а также что в каждом из городов выдано не более трех ID. Примечательно, что первое и второе ограничения обусловлены особенностью обработки данных: пропущенные значения интерпретируются системой как идентичные элементы.

4. Эксперимент

4.1. Условия эксперимента

Все тесты выполнялись на машине со следующими характеристиками:

1. CPU Intel Core i5-10210U 1.6GHz, 4 cores, 8 threads;
2. 8 GiB RAM DDR4;
3. OS Kubuntu 24.04, Kernel version 6.8.0-55-generic, gcc 13.3.0

Для тестов были использованы таблицы размеров: 7x12, 4x626, 8x175. Мы используем нотацию вида количество столбцов x количество строк. Данные размеры были взяты, чтобы протестировать алгоритм на разном количестве строк и столбцов.

4.2. Исследовательские вопросы

Для выявления эффективности алгоритма было принято решение сравнить его с уже имеющимся в “Desbordante” верификатором, который будет находить ND посредством перебора. В связи с этим были поставлены следующие *исследовательские вопросы*:

RQ1 : Правда ли предложенный в работе алгоритм лучше простого перебора с помощью верификатора?

RQ2 : Насколько целесообразно его добавление в “Desbordante”?

4.3. Проведенные эксперименты

Согласно оригинальной статье [4], сложность алгоритма экспоненциально зависит от количества столбцов. В отличие от этого, способ с использованием верификатора зависит квадратично от количества строк, а также линейно зависит от числа сочетаний столбцов $C_n^x * C_n^y$, где x —

Таблица 2: Сравнение времени выполнения алгоритма BBND и алгоритма перебора на таблице TestND размера 7x12

Размер ЧЗ	Время выполнения BBND (с)	Время выполнения верификатора ЧЗ (мс)
$1 \to 2$	<1	1
$1 \to 3$	31	1
$1 \to 4$	114	1
$2 \to 1$	6	1
$2 \to 2$	87	1
$2 \to 3$	658	1

размер левой части числовой зависимости, y — размер правой части числовой зависимости, n — количество столбцов.

В платформе “Desbordante” после завершения выполнения алгоритма возвращается время его выполнения, которое использовалось далее. Внутренняя реализация измерения времени основана на стандартной библиотеке std::chrono. Для уменьшения ошибки измерений алгоритм и верификатор запускались три раза, и из полученных результатов вычислялось среднее значение.

4.3.1. Эксперимент 1

В первом эксперименте измерения производились на таблице размером 7x12. Результаты эксперимента представлены в таблице 2. Как видно из результатов, для подхода с использованием верификатора такое малое количество строк не является проблемой, и он выполняется за пренебрежимо малое время. В то же время, алгоритм BBND даже с таким небольшим количеством столбцов требует значительного времени для выполнения.

4.3.2. Эксперимент 2

Во втором эксперименте измерения производились на таблице размером 4x626 с целью исследования влияния увеличения количества строк и уменьшения количества столбцов на производительность алгоритмов. Результаты эксперимента представлены в таблице 3. Как видно из ре-

Таблица 3: Сравнение времени выполнения алгоритма BBND и алгоритма перебора на таблице kOdTestNormBalanceScale размера 4x626

Размер ЧЗ	Время выполнения BBND (мс)	Время выполнения верификатора ЧЗ (мс)
$2 \to 2$	2	486
$2 \to 3$	5	650
$2 \to 4$	6	652
$3 \to 3$	8	907
$3 \to 4$	8	612
$4 \to 4$	10	253

зультатов, алгоритм BBND выполняется за пренебрежимо малое время, в то время как подход с использованием верификатора требует значительно большего времени, хотя и не столь критичного, как в случае с BBND в предыдущем эксперименте.

4.3.3. Эксперимент 3

В третьем эксперименте измерения производились на таблице размером 8x175. Результаты эксперимента представлены в таблице 4. Прочерки в таблице обозначают случаи, когда время выполнения превышает 1000 с. Как видно из результатов, алгоритм BBND не справляется с вычислением числовых зависимостей (ЧЗ), у которых размерности левой и правой частей превышают три, в таблицах с количеством столбцов больше семи. При этом BBND быстрее вычисляет числовые зависимости с размером левой и правой частей, не превышающим двух, по сравнению с методом с использованием верификатора. Однако в остальных случаях метод с использованием верификатора демонстрирует более высокую производительность и способен выполнить вычисления, которые BBND не может завершить в разумные сроки.

4.4. Вывод

Таким образом, на поставленные исследовательские вопросы можно ответить следующим образом:

Таблица 4: Сравнение времени выполнения алгоритма BBND и алгоритма перебора на таблице kWdcSatellites размера 8x175

Размер ЧЗ	Время выполнения BBND (с)	Время выполнения верификатора ЧЗ (с)
$2 \to 2$	6,2	18
$2 \to 3$	53	27
$2 \to 4$	171	44
$3 \to 3$	170	68
$3 \to 4$	-	91
$4 \to 4$	-	110

- **RQ1**: *Правда ли предложенный в работе алгоритм лучше простого перебора с помощью верификатора?*

  Можно заключить, что предложенный алгоритм превосходит метод перебора только на определенных наборах данных, где количество столбцов не превышает 6-7. Однако такие наборы данных являются крайне специфичными и не часто встречаются на практике. В остальных случаях производительность алгоритма оказывается ниже, чем у метода с использованием верефикатора.

- **RQ2**: *Насколько целесообразно его добавление в "Desbordante"?*

  На основании RQ1 можно сделать вывод о нецелесообразности добавления алгоритма в текущем виде. В связи с этим необходимо провести его оптимизацию и улучшение.

5. Заключение

В ходе данной работы над реализацией и внедрением алгоритма BBND в проект “Desbordante” были достигнуты следующие результаты:

- составлен обзор предметной области алгоритмов поиска числовых зависимостей;
- были реализованы следующие компоненты алгоритма BBND [4]:
  1. базовый класс алгоритма для последующего внедрения в C++-ядро проекта;
  2. оптимизированная очередь выдачи вариантов пути для нахождения числовой зависимости (ActiveNDPaths [4]);
- была реализована поддержка вызова алгоритма BBND из Python-программ в рамках платформы “Desbordante” и создан пример вызова;
- проведена апробация получившегося алгоритма.

По итогам проведенной апробации было принято решение пока не добавлять данный алгоритм в основной репозиторий и продолжить его доработку. С кодом алгоритма можно ознакомиться на Github[1].

[1] https://github.com/Sneper-Breeze/Desbordante/tree/NDs

Список литературы

[1] 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.

[2] Chernishev George, Polyntsov Michael, Chizhov Anton et al. Desbordante: from benchmarking suite to high-performance science-intensive data profiler (preprint). — 2023. — 2301.05965.

[3] Functional dependency discovery: an experimental evaluation of seven algorithms / Thorsten Papenbrock, Jens Ehrlich, Jannik Marten et al. // Proc. VLDB Endow. — 2015. — Jun.. — Vol. 8, no. 10. — P. 1082–1093. — URL: https://doi.org/10.14778/2794367.2794377.

[4] Paolo Ciaccia Matteo Golfarelli Stefano Rizzi. Efficient derivation of numerical dependencies. — 2012. — URL: https://sci-hub.ru/https://www.sciencedirect.com/science/article/abs/pii/S0306437912001044# (дата обращения: 5 ноября 2024 г.).

[5] Song Shaoxu, Chen Lei. Introduction // Integrity Constraints on Rich Data Types. — Cham : Springer International Publishing, 2023. — P. 1–13. — ISBN: 978-3-031-27177-9. — URL: https://doi.org/10.1007/978-3-031-27177-9_1.

[6] Реализация алгоритма поиска числовых зависимостей в «Desbordante» : Rep. / Saint Petersburg State University ; Executor: Пётр Сениченков : 2024.

[7] Реализация современных алгоритмов для поиска зависимостей в базах данных : Rep. / Saint Petersburg State University ; Executor: Максим Струтовский : 2020. — URL: https://github.com/Desbordante/desbordante-core/blob/main/docs/papers/Pyro%20-%20Maxim%20Strutovsky%20-%202020%20spring.pdf.

[8] Приближённые зависимости в базах данных : Rep. / Saint Petersburg State University ; Executor: Максим Фофанов : 2024. — URL: https://github.com/Desbordante/desbordante-core/blob/main/docs/papers/Approximate%20Dependencies%20-%20Maxim%20Fofanov%20-%20autumn%202023.pdf (дата обращения: 3 июня 2025 г.).