Санкт-Петербургский государственный университет

Кафедра информационно-аналитических систем

Группа 22.Б07-мм

Улучшение существующих пользовательских сценариев в Desbordante

Бурашников Артем Максимович

Отчёт по учебной практике
в форме «Эксперимент»

Научный руководитель:
ассистент кафедры ИАС Чернышев Г. А.

Санкт-Петербург
2025

Оглавление

Введение

Чтобы получить полезные выводы из набора данных, необходимо сначала изучить его структуру, качество и выявить закономерности. Для этого применяется профилирование данных.

Профилирование данных [3] представляет собой процесс анализа и извлечения метаданных, который позволяет, например, выявлять скрытые закономерности и обнаруживать аномалии. В этом процессе могут использоваться как простые методы — например, подсчет количества пропущенных значений и расчет стандартных статистических показателей (минимум, максимум, среднее, медиана, мода, стандартное отклонение и т.п.) — так и более сложные подходы, включая обнаружение функциональных зависимостей [4] и ассоциативных правил [9].

Desbordante[1] — это проект с открытым исходным кодом, разрабатываемый с 2019 года. Он предоставляет пользователю готовый инструментарий для профилирования данных. В рамках проекта доступны несколько интерфейсов для работы: веб-приложение, библиотека для Python и командная строка.

Одной из проблем, которую ставят перед собой контрибьюторы, является привлечение пользователей и улучшение пользовательского опыта. Для этого уже существуют пользовательские сценарии[2], демонстрирующие способ работы с той или иной компонентой системы. Однако эти сценарии, реализованные в виде Python-скриптов, имеют ряд недостатков, например:

- отсутствует удобный пользовательский интерфейс;
- сложность настройки и запуска скриптов;
- отсутствие единого стандарта написания сценариев;
- недостаточная документация, что затрудняет понимание работы скриптов для новых пользователей;

[1] https://github.com/Desbordante (дата обращения: 16 марта 2025 г.)

[2] https://github.com/Desbordante/desbordante-core/tree/main/examples (дата обращения: 17 марта 2025 г.)

- отсутствие визуализации результатов, что усложняет анализ полученных данных;
- отсутствие встроенных средств логирования и отладки, что затрудняет выявление ошибок в процессе выполнения.

Одной из целей данной работы является устранение этих недостатков для повышения удобства и эффективности использования Desbordante.

Другой проблемой, над которой ведется постоянная работа[3] является ограниченный набор[4] алгоритмов профилирования данных. Участники проекта активно работают[5] над расширением функциональности: проводят экспериментальные исследования и разрабатывают новые алгоритмы. Так, Аносов Павел [13, 12] реализовал[6] алгоритм верификации запрещающих ограничений. Запрещающие ограничения определяют комбинации значений атрибутов, которые не должны встречаться в наборе данных. Помимо верификации, реализованная Павлом процедура формирует на выходе набор исключений — кортежей, нарушающих зависимости. После обнаружения таких исключений возможна их обработка для приведения данных в консистентное состояние. Очистка данных может включать модификацию, удаление или добавление записей, при этом модификация и добавление являются существенно более сложными задачами по сравнению с удалением. В работе [12] Павел предложил алгоритм минимизации множества исключений, основанный на удалении кортежей-исключений. Однако сам автор отмечает, что данный подход имеет ряд недостатков и не является оптимальным. В связи с этим одной из целей данной работы является анализ предложенного алгоритма минимизации и разработка его улучшенной версии.

[3] https://github.com/Desbordante/desbordante-core/pulls, (дата обращения: 17 марта 2025 г.)

[4] https://github.com/Desbordante/desbordante-core?tab=readme-ov-file#general (дата обращения: 4 апреля 2025 г.)

[5] https://github.com/Desbordante/desbordante-core/tree/main/docs/papers (дата обращения: 17 марта 2025 г.)

[6] https://github.com/Desbordante/desbordante-core/pull/510 (дата обращения: 17 марта 2025 г.)

1. Постановка задачи

Целью работы является повышение удобства и эффективности взаимодействия пользователей с Desbordante за счет улучшения пользовательского опыта.

Для её выполнения были поставлены следующие задачи:

1. перенести существующие примеры[7] использования примитивов в интерактивные python-ноутбуки, при этом:
   - добавить в сценарии структурированное повествование для простоты восприятия происходящего новыми пользователями;
   - добавить необходимые для понимания сценариев определения;
   - добавить пояснения к исполняемым процедурам;
   - модифицировать исходный код для исполнения в интерактивной среде;
2. проанализировать предложенный в работе [12] алгоритм очистки данных от записей, распознанных как исключения;
3. по результатам анализа предложить усовершенствованный вариант алгоритма либо его альтернативу.

[7] https://github.com/Desbordante/desbordante-core/tree/main/examples (дата обращения: 6 марта 2025 г.)

2. Предварительные сведения

В данном разделе в контексте профилирования данных представлена терминология, использованная в работе.

- **Датасет** — некоторый набор данных.
- **Примитив** — некоторое правило, действующее над данными и описанное математическими методами.
- **Инстанс примитива** — конкретное правило данного типа для конкретного датасета.
- **Алгоритм** — последовательность операций для поиска или валидации инстансов примитивов.
- **Валидация (верификация) инстанса примитива** — проверка, удерживается ли конкретный инстанс примитива на заданном датасете.
- **Поиск (майнинг) инстансов примитива** — задача по нахождению всех инстансов примитива в заданном датасете.
- **Запрещающие ограничения (DC)** [13, 12, 10].

  DC — это примитив, задающий ограничения на комбинацию значений атрибутов, которые не должны выполняться. DC позволяют находить неточности и ошибки в данных, такие как пропущенные значения, дубликаты и другие несоответствия.

3. Пользовательские сценарии и процедура очистки ЗО

Настоящая работа затрагивает два аспекта Desbordante.

1. Во-первых, в данном разделе исследован текущий формат демонстрации примеров использования Desbordante конечным пользователям. Установлено, что существующий подход ограничивается предоставлением фрагментов исходного кода на Python. Такой метод вызывает сложности у пользователей, особенно новичков: отсутствие интерактивности, неочевидность последовательности действий, невозможность быстрого тестирования гипотез.

2. Во-вторых, выполнен сравнительный анализ алгоритма удаления кортежей-исключений, возникаюших в результате верификации ЗО, реализованного в [12]. В ходе исследования были рассмотрены альтернативные алгоритмы-аналоги и выделены их ключевые характеристики.

Итогом раздела стало построение методологической базы для дальнейшей оптимизации пользовательского опыта и предоставление критериев для проведения сравнительного анализа алгоритма минимизации.

3.1. Пользовательские сценарии

Для демонстрации способов применения различных алгоритмов, реализующих тот или иной примитив, и обучению работы с проектом в целом в Desbordante имеется набор показательных сценариев. Эти сценарии подготовлены в виде исходного кода на языке Python.

У такого подхода есть недостатки:

- для реализации сценариев пользователю необходимо настраивать среду исполнения;

- отсутствуют промежуточные результаты работы кода, доступен лишь конечный результат.

Помимо этого, к возможностям при работе со сценариями предъявлены следующие требования:

- исполнять произвольные участки кода в желаемой последовательности;
- загружать и изменять данные, не затрагивая остальные участки программы;
- писать сопроводительный текст рядом с исходным кодом при помощи языка разметки `Markdown`[8].

Указанные недостатки и требования привели к рассмотрению интерактивных сред исполнения кода, которые позволяют гибко работать с примерами и анализировать промежуточные результаты. Популярные решения включают:

- **Google Colab**[9] — удобен для работы с Python-кодом в облаке, не требует локальной настройки среды и поддерживает **Markdown** для документирования.
- **Jupyter Notebook**[10] — предоставляет локальную и облачную среду исполнения, удобен для пошагового выполнения кода, но требует настройки на стороне пользователя.
- **Marimo**[11] — относительно новый инструмент, позволяющий интерактивно редактировать код и документацию, что делает его перспективным вариантом, однако он пока не получил широкого распространения, имеет ограниченную экосистему и может не поддерживать все необходимые библиотеки и функции.
- **Datalore**[12] — облачный сервис от `JetBrains`, предлагающий инструменты для анализа данных, совместной работы и интеграции

[8] https://en.wikipedia.org/wiki/Markdown (дата обращения: 2 марта 2025 г.)
[9] https://colab.research.google.com/ (дата обращения: 4 апреля 2025 г.)
[10] https://jupyter.org/ (дата обращения: 4 апреля 2025 г.)
[11] https://marimo.io/ (дата обращения: 10 апреля 2025 г.)
[12] https://www.jetbrains.com/datalore/ (дата обращения: 4 апреля 2025 г.)

с различными источниками данных. Поддерживает Python и SQL, предоставляет интерфейс и функции для работы с ноутбуками. Однако бесплатная версия имеет серьёзные ограничения, а для полноценного использования требуется платная подписка.

- **Spyder**[13] — среда разработки для научных вычислений на Python, ориентированная на работу с данными. Позволяет анализировать результаты выполнения кода и интегрируется с различными библиотеками. Однако Spyder требует установки на локальный компьютер, что создаёт дополнительные сложности при настройке окружения.

С учётом этих факторов в качестве среды исполнения был выбран **Google Colab**. Он не требует сложной настройки, поддерживает все необходимые возможности и предоставляет вычислительные ресурсы, что делает его оптимальным решением для работы с пользовательскими сценариями в Desbordante.

3.2. Алгоритм минимизации кортежей-исключений DC

В работе [12] реализована верификация *запрещающих ограничений* и обеспечен поиск кортежей, нарушающих зависимость — исключений.

Полученные пары кортежей-исключений можно представить в виде графа, где вершины соответствуют номерам записей, а рёбра — номерам пар из списка исключений. В [12] очистка данных от аномалий ограничивается только удалением рёбер, что обеспечивает выполнение примитива ЗО. Хотя в более общем случае возможны и другие подходы, такие как редактирование или добавление рёбер, Павел Аносов сознательно сосредоточился именно на стратегии удаления.

В его работе задача сформулирована следующим образом.

[13] https://www.spyder-ide.org (дата обращения: 4 апреля 2025 г.)

Узкая задача. *Дан произвольный неориентированный граф $G(V, E)$, где V — множество вершин, E — множество рёбер. Требуется удалить минимальное количество рёбер, чтобы полученный граф удовлетворял условиям примитива ЗО.*

Этот вариант учитывает только удаление рёбер и минимизирует изменения в структуре графа. Можно предложить обобщение.

Общая задача. *Дан произвольный неориентированный граф $G(V, E)$, где V — множество вершин, E — множество рёбер. Требуется найти такое подмножество $V' \subset V$, что $\forall (u, v) \in E$: $u \in V'$ или $v \in V'$.*

Эта формулировка соответствует классической задаче поиска наименьшего вершинного покрытия, которая является NP-полной [6], и для неё существует множество эвристик [1].

В работе рассмотрены несколько из них, отобранные по следующим критериям:

1. простота реализации, измеряемая в количестве строчек кода (до 50 — легкая, иначе — трудная);

2. полиномиальная асимптотическая сложность.

В таблице 1 представлены отобранные алгоритмы.

Таблица 1: Сравнение алгоритмов построения минимального вершинного покрытия

Алгоритм	**Реализация**	**Сложность**
Greedy[14], 1979 [6]	Легкая	$\mathcal{O}(\|E\| \cdot \|V\|)$
Max matching (Edmonds), 1986 [5]	Трудная	$\mathcal{O}(\|E\| \cdot \|V\|^3)$
2-approximation, 1985 [2]	Легкая	$\mathcal{O}(\|E\| \cdot \log \|V\|)$
Min-To-Min, 2020 [8]	Легкая	$\mathcal{O}(\|E\| \cdot \|V\|)$

Далее рассмотрены каждый из них.

[14]Оригинального автора найти не удалось. Указан самый старый источник, содержащий упоминание алгоритма.

3.2.1. Greedy

Этот метод не имеет конкретного, однозначно именуемого автора. Обычно его называют «жадной эвристикой для задачи минимального вершинного покрытия», и он считается элементарным (фольклорным) алгоритмом, широко известным в литературе. Такие подходы зачастую появляются независимо и без указания источника. Аносов Павел предложил его использование и реализовал в [12].

Он представляет собой следующую эвристику: на каждом шаге выбирается вершина с максимальной степенью, затем из графа удаляется она и все смежные ей рёбра. Псевдокод изложен в листинге 1.

Листинг 1 Жадный по степени вершины алгоритм поиска минимального вершинного покрытия.

```
BEGIN
  C ← ∅
  WHILE E ≠ ∅ DO
    v ← arg(max_{v∈V}(deg(v)))
    C.add(v)
    REMOVE v and all adjacent edges
END
RETURN C
```

Алгоритм корректно находит вершинное покрытие, однако оно не обязательно является минимальным. В [12] приведён контрпример. Более того, количество лишних вершин, включённых в покрытие, зависит от $|V|$ и оценивается как $\mathcal{O}(\log |V|)$ [1].

Рассмотрим шаги алгоритма подробно:

1. Подготовительный этап: цикл по всем рёбрам исходного графа, что занимает не более $\mathcal{O}(|E|)$ времени.

2. Основой ход алгоритма: функция `clean()` в своём теле выполняет проверку `has_edges()`, которая в худшем случае займёт не более $\mathcal{O}(|E|)$ времени.

3. Пока проверка возвращает значение `True`, алгоритм находит максимальный по степени вершины элемент и удаляет все смежные ему рёбра, а затем и сам элемент. Такие операции в конкретной реализации автора статьи занимают не более $\mathcal{O}(|V|)$ времени.

4. После удаления вершины с максимальной степенью, удаляются рёбра, инцидентные этой вершине. Такая процедура в сумме займет не более $\mathcal{O}(|E|)$ времени.

Таким образом, итоговая сложность алгоритма:

$$\mathcal{O}(|E|) + \mathcal{O}(|E| \cdot |V| + |E|) = \mathcal{O}(|E| \cdot |V|).$$

3.2.2. Max matching (Edmonds)

Следующим в таблице 1 представлен один из способов построения вершинного покрытия, основанный на поиске наибольшего паросочетания в графе с использованием алгоритма Эдмондса [5].
Для описания эвристики введём следующие определения.

Определение 1. *Подмножество рёбер $M \subseteq E$ называется паросочетанием, если никакие два ребра из M не имеют общих инцидентных вершин.*

Определение 2. *Максимальное паросочетание — это такое паросочетание M, которое не содержится ни в каком другом.*

Определение 3. *Наибольшее паросочетание (или максимальное по размеру паросочетание) — это такое паросочетание, которое содержит максимальное количество рёбер.*

Суть эвристики заключается в следующем:

1. алгоритм Эндмондса строит наибольшее паросочетание **M** в произвольном графе [5];

2. после нахождения **M** в вершинное покрытие **C** добавляются обе вершины каждого ребра из **M**:

$$\forall (u, v) \in M : C \cup \{u, v\};$$

3. так как **M** — максимальное паросочетание, в него нельзя добавить рёбер без нарушения его свойств. Это означит, что любое ребро исходного графа обязательно пересекается хотя бы с одним ребром из M:

$$\forall (u, v) \in E \; \exists e \in M : e \cap (u, v) = u \; \vee \; e \cap (u, v) = v;$$

4. следовательно, множество **C** действительно является вершинным покрытием.

Сам алгоритм сложен в реализации, но имеется в библиотеке для Python NetworkX[15]. В листинге 2 представлен псевдокод, взятый с ресурса `descrete.ma.tum.de`[16]. Асимптотика составляет[17] $O(|V|^3)$.

[15] https://networkx.org/ (дата обращения: 2 марта 2025 г.)

[16] https://algorithms.discrete.ma.tum.de/graph-algorithms/matchings-blossom-algorithm/index_en.html (дата обращения: 4 марта 2025 г.)

[17] https://networkx.org/documentation/stable/reference/algorithms/generated/networkx.algorithms.matching.max_weight_matching.html (дата обращения: 20 марта 2025 г.)

Листинг 2 Алгоритма Эдмондса построения максимального паросочетания в произвольном графе.

```
BEGIN
  // set of free nodes F
  WHILE F != ∅ DO
    pick r ∈ F
    // BFS queue
    queue.push(r)
    // BFS tree T
    T ← ∅
    T.add(r)
    WHILE queue != ∅
      v ← queue.pop()
      FOR ALL neighbors w of v DO
        IF w ∉ T AND w matched THEN
          T.add(w)
          T.add(mate(w))
          queue.push(mate(w))
        ELSE IF w ∈ T AND even-length cycle detected THEN
          CONTINUE
        ELSE IF w ∈ T AND odd-length cycle detected THEN
          contract cycle
        ELSE IF w ∈ F THEN
          expand all contracted nodes
          reconstruct augmenting path
          invert augmenting path
END
```

3.2.3. 2-approximation

Следующей эвристикой в таблице 1 является алгоритм 2-approximation — один из наиболее быстрых алгоритмов для аппроксимации минимального вершинного покрытия со сложностью $\mathcal{O}(|E| \cdot \log |V|)$, впервые предложенный в 1985 году [2].

Реализованная в `NetworkX`[18] на основании него процедура заключается в том, что на каждом шаге выбирается произвольное ребро графа,

[18] https://networkx.org/documentation/stable/reference/algorithms/generated/networkx.algorithms.approximation.vertex_cover.min_weighted_vertex_cover.html#networkx.algorithms.approximation.vertex_cover.min_weighted_vertex_cover (дата обращения: 19 марта 2025 г.)

а затем в вершинное покрытие добавляется вершина с меньшим весом (в нашем случае все веса вершин равны единице), а вес соседней вершины, соответственно, уменьшается. Не умаляя общности, можно считать, что из случайно выбранного ребра поочередно добавляются обе вершины.

Алгоритм жадно строит наибольшее паросочетание и даёт точную верхнюю границу для размера вершинного покрытия относительно оптимального. Действительно, пусть C_{opt} — минимальное вершинное покрытие, пусть A — множество рёбер, которые алгоритм выберет в ходе своей работы. Тогда $|C_{opt}| \geq |A|$, потому что вершины, входящие в C_{opt}, покрывают рёбра A, и при этом каждая вершина не покрывает одно и то же ребро дважды, так как иначе в A бы встретились два смежных ребра, а такое невозможно. С другой стороны, $|C| = 2 \cdot |A|$. Сравнивая оба неравенства, получаем:

$$|C| \leq 2 \cdot |C_{opt}|.$$

В листинге 3 представлен псевдокод.

Листинг 3 Жадный алгоритм поиска минимального вершинного покрытия.

```
BEGIN
  C ← ∅
  FOR (u,v) ∈ E DO
    IF u ∈ C OR v ∈ C
        CONTINUE
    IF cost(u) ≤ cost(v) THEN
        C.add(u)
        cost(v) −= cost(u)
    ELSE
        C.add(v)
        cost(u) −= cost(v)
END
RETURN C
```

3.2.4. Min-To-Min

Последний в таблице 1 алгоритм, представленный в работе [8], состоит из пяти основных шагов:

1. для каждой вершины в графе находим её степень;
2. выбираем вершину с минимальной степенью (если таких вершин несколько, выбираем первую в списке);
3. еще раз выбираем вершину с минимальной степенью среди всех вершин, смежных той, что была получена на предыдущем шаге;
4. удаляем все инцидентные кандидату рёбра;
5. добавляем кандидата в вершинное покрытие и возвращаемся к шагу 1.

Этот процесс повторяется до тех пор, пока в предоставленном графе не останется рёбер. В листинге 4 изложен псевдокод.

Листинг 4 Алгоритм Min-To-Min построения вершинного покрытия.

```
BEGIN
    C ← ∅
    WHILE E ≠ ∅
        v ← arg(min_{u∈V} deg(u))
        N(v) ← list of neighbors of v
        u ← arg(min_{w∈N(v)} deg(w))
        C ← C ∪ {u}
        REMOVE u and all adjacent edges from G
END
```

Основной цикл алгоритма выполняется не более, чем за $O(|E|)$ операций, при этом на каждой итерации больше всего времени занимает поиск соседней вершины с минимальной степенью, что ограничено сверху значением $O(|V|)$. Итоговая асимптотическая оценка:

$$O(|E| \cdot |V|).$$

3.3. Итоги обзора

В данном разделе описаны выводы, полученные во время обзора.

- Выбрана интерактивная среда для переноса примеров работы с доступными в проекте паттернами валидации и поиска.
- Отобран и проанализирован набор эвристических алгоритмов, приближённо решающих поставленную задачу поиска минимального вершинного покрытия.

В последующих разделах представлены результаты экспериментов с целью выбора лучшего из предложенных алгоритмов.

4. Описание решения

В этом разделе описываются реализованные подходы к решению поставленных задач, а также даются пояснения к произведённым изменениям и внедрённым улучшениям.

4.1. Перенос примеров

Перенос примеров из репозитория `Desbordante` в интерактивную среду исполнения не потребовал значительных программных доработок. Все примеры были интегрированы в интерактивную среду `colab.research.google.com`[19], позволяющую поэтапно выполнять программы, написанные на языке Python, прямо в окне веб-браузера. В процессе переноса выполнены следующие доработки:

- Удалены избыточные и инструкции `print()`, перегружавшие вывод.
- Часть императивной логики переписана с использованием функциональности библиотеки `pandas`, которая является де-факто стандартом [7] для анализа данных в интерактивных средах, таких как `Jupyter`[20].
- Добавлены текстовые блоки, поясняющие суть выполняемых операций.

Внимание уделялось тому, чтобы сценарий выглядел логично структурированным и понятным для конечного пользователя.

Следующие примеры были перенесены в виде отдельных интерактивных ноутбуков.

1. data_stats — базовая статистика по данным: среднее, минимум, максимум, количество уникальных значений и т.д.

[19] https://colab.research.google.com (дата обращения: 2 марта 2025 г.)

[20] https://docs.jupyter.org/en/latest/index.html#what-is-a-notebook (дата обращения: 2 марта 2025 г.)

2. mining_fd — извлечение функциональных зависимостей (ФЗ) из набора данных.

3. mining_afd — извлечение приближённых функциональных зависимостей (АФЗ).

4. verifying_fd_afd — проверка наличия как точных, так и приближённых ФЗ;

5. mining_sfd — нахождение *мягких* функциональных зависимостей.

6. mining_ind, mining_aind, verifying_ind_aind — набор примеров, связанных с зависимостями включения (в том числе приближёнными);

7. mining_ucc, mining_aucc — примеры обнаружения уникальных и приближённых уникальных комбинаций колонок.

Детали реализации доступны в соответствующем Pull Request-е репозитория[21].

4.2. Измерение эффективности алгоритмов минимизации

Для выбора наилучшего для поставленной задачи алгоритма написаны тестовые скрипты на языке Python. В листинге 5 приведена вспомогательная обёртка для измерения времени выполнения функций.

Листинг 5 Таймер-обертка для измерения времени работы функции.

```
def timer(f):
    def wrapper(*args, **kwargs):
        start = time.process_time()
        res = f(*args, **kwargs)
        finish = time.process_time()
        return (finish - start), res, f.__name__
    return wrapper
```

[21] https://github.com/Desbordante/desbordante-core/pull/552 (дата обращения: 7 апреля 2025 г.)

Для оценки корректности решений используется вспомогательная функция верификации, представленная в листинге 6. Она проверяет, является ли данное множество вершин допустимым вершинным покрытием в графе: каждое ребро должно быть инцидентно хотя бы одной вершине из покрытия.

Листинг 6 Верификация вершинного покрытия.

```
def is_vertex_cover(graph, candidate):
    for u, v in graph.edges():
        if u not in candidate and v not in candidate:
            return False
    return True
```

В качестве базовой реализации рассмотрен наивный алгоритм перебора, представленный в листинге 7. Он перебирает все возможные подмножества вершин в порядке возрастания размера и возвращает первое подходящее покрытие.

Листинг 7 Наивный алгоритм.

```
def brute_force(graph):
    nodes = list(graph.nodes)
    for r in range(1, len(nodes) + 1):
        for subset in itertools.combinations(nodes, r):
            if is_vertex_cover(graph, set(subset)):
                return set(subset)
    return set()
```

Весь исходный код и используемые наборы данных доступны в открытом репозитории на GitHub[22]. Подробности экспериментов и сравнительный анализ алгоритмов представлены в следующих разделах.

[22] https://github.com/artem-burashnikov/vertex-cover-benchmarks (дата обращения: 6 марта 2025 г.)

5. Эксперимент

В данном разделе описаны характеристики оборудования, на котором проводились эксперименты, а также представлены исследовательские гипотезы, используемые метрики и инструменты. Отдельное внимание уделено обоснованию выбора набора тестовых данных.

5.1. Характеристики оборудования

Эксперименты проводились на следующей аппаратно-программной конфигурации.

- Процессор: `AMD Ryzen` 5 4500U @ 2.3 GHz.
- Оперативная память: 8 GiB.
- Интерпретатор Python: версия 3.13.
- Среда выполнения: `CPython`.
- Операционная система: Ubuntu.
- Версия ядра Linux: 6.13.

5.2. Подготовка тестового стенда

Для нормализации полученных измерений была выполнена настройка тестовой среды. В рамках данной работы не использовались специальные библиотеки для замеров производительности, поэтому важно минимизировать влияния операционной системы на результаты экспериментов. Это включает сброс кэшей, стабилизацию частоты процессора и другие меры.

Часть команд требует наличия утилиты `cpupower`, которая может быть установлена с помощью:

```
sudo apt install linux-tools-common.
```

Команды необходимо выполнить в оболочке терминала операционной системы `Ubuntu`.

1. Отключение файла подкачки:

   `sudo swapoff -a`.

2. Установка подсистемы `CPUFreq governor` в состояние `performance`:

   `sudo cpupower frequency-set -g performance`.

3. Стабилизация частоты процессора:

   `sudo cpupower frequency-set -d 2.38GHz -u 2.38GHz`.

4. Запуск исполняемой программы на одном ядре (в данном случае ядре с порядковым номером 0):

   `taskset -c 0 python3 main.py`

5.3. Исследовательские гипотезы

Анализ поставленных задач в разделе 3.2 позволил выдвинуть следующие гипотезы.

Исследовательская гипотеза №1

Ожидается, что жадные алгоритмы *2-approximation, Min-To-Min* и *Greedy* будут работать быстрее, чем *Max matching (Edmonds)*. При этом *2-approximation* должен показать наилучшие результаты по скорости. Это предположение основано на анализе асимптотической сложности алгоритмов.

Исследовательская гипотеза №2

Алгоритм *Min-To-Min* обеспечит более качественную аппроксимацию минимального вершинного покрытия по сравнению с алгоритмом *Greedy*, то есть, вернёт покрытия меньшей или равной мощности на большинстве графов.

5.3.1. Исследовательские вопросы

Из выдвинутых гипотез возникли исследовательские вопросы.

RQ1 : Правда ли, что *Greedy* работает быстрее *Min-To-Min* на графах любого размера?

RQ2 : Действительно ли *Min-To-Min* возвращает вершинное покрытие меньшей мощности по сравнению с результатом работы *Greedy*?

5.4. Инструменты для измерений и метрики

Для измерения времени работы алгоритмов использовалась функция `time.process_time()`[23] из стандартной библиотеки Python. Эта функция позволяет измерять чистое процессорное время, исключая время, затраченное на операции ввода-вывода и простои. Дополнительно фиксировалась мощность вершинного покрытия, возвращаемого каждым алгоритмом.

5.5. Набор данных

Для фиксации исследуемых величин выбраны 8 различных неориентированных графов без меток из коллекции `SNAP` — `Stanford Large Network Dataset Collection` [11] и 5 графов малой размерности с портала `Network Repository` [14]. Графы малой размерности были добавлены для возможности сравнения с точным экспоненциальным алгорит-

[23] https://docs.python.org/3/library/time.html#time.process_time (дата обращения: 5 марта 2025 г.)

мом. Все графы были подобраны таким образом, чтобы время выполнения любого из тестируемых алгоритмов не превышало 60 секунд.

Подробная информация о тестовых данных размещена в таблице 2. Обозначения:

- $|V|$ — количество вершин;
- $|E|$ — количество рёбер без учета петель и ориентации.

Таблица 2: Информация о тестовых графах

Граф	**|V|**	**|E|**
cage3	5	7
rgg010	10	45
aves-barn	17	53
GD02-a	23	59
lap-25	25	72
road-chesapeake	39	170
scc-enron	151	9828
email-Eu	1005	16064
facebook-combined	4039	88234
ca-GrQc	5242	14484
p2p-Gnutella08	6301	20777
wiki-Vote	7115	100762
p2p-Gnutella09	8114	26013

5.6. Результаты экспериментов

Каждый алгоритм был запущен 50 раз для каждого тестового набора данных. Полученные результаты были проверены на нормальность распределения, после чего рассчитаны среднее значение и стандартное отклонение.

В таблицах 3, 4, 5, 6 представлены результаты измерений на выбранных данных, где:

- Время — среднее время работа алгоритма;

- |C| — мощность полученного вершинного покрытия;
- $|C_{opt}|$ — мощность минимального вершинного покрытия.

Метка `n/a` обозначает случаи, когда алгоритм не завершался за отведенное время (60 секунд). Зеленым цветом выделены те значения, на которых алгоритм показал наилучший результат среди других (если таких алгоритмов несколько, значения будут помечены у каждого). Красным — те значения, где алгоритм продемонстрировал наихудшее значение среди остальных.

Таблица 3: Greedy

|V|	|E|	Время	|C|	$\|C_{opt}\|$	$\frac{\|C\|}{\|C_{opt}\|}$
5	7	9.7 ± 0.5 ns	**3**	3	**1.00**
10	45	29.9 ± 0.7 ns	**9**	9	**1.00**
17	53	30.0 ± 0.7 ns	**10**	10	**1.00**
23	59	50.4 ± 0.9 ns	**10**	10	**1.00**
25	72	68.1 ± 1.1 ns	**16**	16	**1.00**
39	170	142.3 ± 1.6 ns	**23**	n/a	n/a
151	9828	6.589 ± 0.009 µs	139	n/a	n/a
1005	16064	39.6 ± 0.6 µs	591	n/a	n/a
4039	88234	0.669 ± 0.003 ms	3042	n/a	n/a
5242	14484	0.938 ± 0.008 ms	2795	n/a	n/a
6301	20777	0.881 ± 0.013 ms	2071	n/a	n/a
7115	100762	1.217 ± 0.007 ms	2358	n/a	n/a
8114	26013	1.479 ± 0.016 ms	2590	n/a	n/a

Таблица 4: 2-approximation

|V|	|E|	Время	|C|	$\|C_{opt}\|$	$\frac{\|C\|}{\|C_{opt}\|}$
5	7	8.4 ± 0.4 ns	4	3	1.33
10	45	15.4 ± 0.7 ns	**9**	9	**1.00**
17	53	25.2 ± 0.8 ns	12	10	1.20
23	59	23.3 ± 0.8 ns	15	10	1.50
25	72	25.8 ± 0.7 ns	22	16	1.38
39	170	47 ± 2 ns	36	n/a	n/a
151	9828	0.125 ± 0.015 µs	145	n/a	n/a
1005	16064	2.622 ± 0.014 µs	687	n/a	n/a
4039	88234	14.64 ± 0.04 µs	3604	n/a	n/a
5242	14484	4.7 ± 0.1 µs	3140	n/a	n/a
6301	20777	6.85 ± 0.11 µs	3322	n/a	n/a
7115	100762	20.48 ± 0.10 µs	2796	n/a	n/a
8114	26013	80.7 ± 0.8 µs	4206	n/a	n/a

Таблица 5: Max matching (Edmonds)

|V|	|E|	Время	|C|	$\|C_{opt}\|$	$\frac{\|C\|}{\|C_{opt}\|}$
5	7	0.136 ± 0.010 µs	**4**	3	**1.33**
10	45	0.108 ± 0.003 µs	**10**	9	**1.11**
17	53	3.3 ± 0.6 µs	**16**	10	**1.60**
23	59	0.489 ± 0.009 µs	**18**	10	**1.80**
25	72	0.49 ± 0.02 µs	**24**	16	**1.50**
39	170	1.347 ± 0.017 µs	**38**	n/a	n/a
151	9828	10.45 ± 0.06 µs	**146**	n/a	n/a
1005	16064	0.903 ± 0.03 ms	**958**	n/a	n/a
4039	88234	8.874 ± 0.015 ms	**3958**	n/a	n/a
5242	14484	10.57 ± 0.09 ms	**4658**	n/a	n/a
6301	20777	13.25 ± 0.11 ms	**4108**	n/a	n/a
7115	100762	36.2 ± 0.4 ms	**4498**	n/a	n/a
8114	26013	25.0 ± 0.3 ms	**5148**	n/a	n/a

Таблица 6: Min-To-Min

|V|	|E|	Время	|C|	$\|C_{opt}\|$	$\frac{\|C\|}{\|C_{opt}\|}$
5	7	25.7 ± 0.6 ns	**3**	3	**1.00**
10	45	74.9 ± 1.1 ns	**9**	9	**1.00**
17	53	0.49 ± 0.03 µs	**10**	10	**1.00**
23	59	0.1385 ± 0.014 µs	**10**	10	**1.00**
25	72	0.188 ± 0.02 µs	**16**	16	**1.00**
39	170	0.376 ± 0.003 µs	**23**	n/a	n/a
151	9828	6.88 ± 0.06 µs	**138**	n/a	n/a
1005	16064	0.148 ± 0.008 ms	**584**	n/a	n/a
4039	88234	2.755 ± 0.005 ms	**3021**	n/a	n/a
5242	14484	3.724 ± 0.003 ms	**2783**	n/a	n/a
6301	20777	5.69 ± 0.01 ms	**2054**	n/a	n/a
7115	100762	8.30 ± 0.03 ms	**2249**	n/a	n/a
8114	26013	10.98 ± 0.17 ms	**2574**	n/a	n/a

Из таблиц видно, что алгоритм *2-approximation* демонстрирует наименьшее время выполнения на всех тестовых графах, что согласуется с его низкой асимптотической сложностью. С ростом количества вершин и рёбер у графа время выполнения всех алгоритмов увеличивается, однако *2-approximation* сохраняет лидерство по скорости. Это подтверждает выдвинутую гипотезу 5.3 о том, что он является самым быстрым алгоритмом среди рассматриваемых. Алгоритм *Max matching (Edmonds)* ожидаемо оказывается самым медленным.

5.6.1. RQ1

Greedy работает сравнительное быстрее *Min-To-Min*. Это связано с особенностями реализации алгоритма *Min-To-Min* с помощью абстракций библиотеки `NetworkX`.

5.6.2. RQ2

Нельзя однозначно утверждать, что алгоритм *Min-To-Min* всегда возвращает вершинное покрытие меньшей мощности по сравнению с *Greedy*. В таблице 7 представлено отношение мощности покрытий, по-

лученных обоими алгоритмами. Максимальное превышение мощности множества, возвращаемого *Greedy*, по сравнению с *Min-To-Min*, не превышает 5%, а в большинстве случаев составляет менее 1%.

Таблица 7: Сравнение мощности вершинного покрытия, возвращаемого Greedy и Min-To-Min

|V|	|E|	$\|C_{greedy}\|$	$\|C_{mtm}\|$	$\frac{\|C_{greedy}\|}{\|C_{mtm}\|}$
5	7	3	3	1.00
10	45	9	9	1.0000
17	53	10	10	1.0000
23	59	10	10	1.0000
25	72	16	16	1.0000
39	170	23	23	1.0000
151	9828	139	138	1.0072
1005	16064	591	584	1.0120
4039	88234	3042	3021	1.0070
5242	14484	2795	2783	1.0043
6301	20777	2071	2054	1.0083
7115	100762	2358	2249	1.0485
8114	26013	2590	2574	1.0062

Таким образом, формально подтвердить гипотезу 5.3 не удалось. Это может быть обусловлено особенностями тестового набора: на реальных графах с большим числом вершин и рёбер результаты могли бы проявиться более отчётливо.

5.6.3. Общие выводы

- *2-approximation* является самым быстрым, но уступает по точности *Greedy* и *Min-To-Min*.
- *Greedy* вопреки ожиданиям, почти не уступает *Min-To-Min* по качеству результата — разница в мощности покрытия чаще всего составляет менее 1%. Это может быть связано с характером тестового набора: возможно, структуры графов способствовали тому, что жадный выбор давал близкий к оптимальному результат.

При этом в текущей реализации алгоритмов *Greedy* работает быстрее.

- *Max matching (Edmonds)* оказался наименее пригодным для задачи аппроксимации вершинного покрытия. Несмотря на его теоретическую значимость, на практике он работает значительно медленнее и при этом уступает по качеству другим эвристическим подходам. Почему так?

  – Алгоритм Эдмондса *оптимально* решает задачу **максимального паросочетания**, а не минимального вершинного покрытия напрямую. Да, по теореме Кёнига **в двудольных графах** они эквивалентны, **но в общих графах** максимальное паросочетание не обязательно приводит к хорошей аппроксимации минимального вершинного покрытия.

  – Эвристика не минимизирует покрытие — она использует построенное алгоритмом Эдмондса паросочетание как вспомогательный шаг. В отличие от *Greedy* и *Min-To-Min*, которые напрямую стремятся сократить размер покрытия.

  – Алгоритмическая сложность алгоритма Эдмондса даёт сильный удар по производительности на больших плотных графах.

  Сработал принцип «стрелять из пушки по воробьям»: мы применили мощный и сложный алгоритм не по адресу. Простые эвристики (Greedy, Min-To-Min) работают быстрее, проще и дают лучшее качество.

Таким образом, для решения поставленной в данной работе задачи рекомендуется использовать алгоритм **Min-To-Min**. Несмотря на незначительное преимущество по качеству по сравнению с *Greedy*, именно *Min-To-Min* обеспечивает наилучшее приближение к минимальному вершинному покрытию на выбранных тестовых наборах. Его производительность может быть улучшена в будущем за счёт оптимизации ре-

ализации (например, выбора другой структуры данных для представления графа) или адаптации к конкретным классам графов.

Заключение

В рамках выполнения данной работы были получены следующие результаты:

1. перенесены 11 демонстрационных примеров на языке Python в интерактивные ноутбуки на портале `colab.research.google.com`;
2. проведён детальный анализ алгоритма очистки данных, выявлены его сильные и слабые стороны;
3. предложен, реализован и экспериментально подтверждён усовершенствованный алгоритм минимизации исключений DC.

Репозиторий с экспериментами доступен на GitHub[24].
`Pull Request` с ноутбуками доступен на GitHub[25].

[24] https://github.com/artem-burashnikov/vertex-cover-benchmarks (дата обращения: 5 марта 2025 г.)

[25] https://github.com/Desbordante/desbordante-core/pull/552 (дата обращения: 7 апреля 2025 г.)

Список литературы

[1] Bansal Sangeeta, Rana Ajay. Analysis of various algorithms to solve vertex cover problem // International Journal of Innovative Technology and Exploring Engineering. — 2014. — Vol. 3, no. 12. — P. 4–6.

[2] Bar-Yehuda R., Even S. A Local-Ratio Theorem for Approximating the Weighted Vertex Cover Problem // Analysis and Design of Algorithms for Combinatorial Problems / Ed. by G. Ausiello, M. Lucertini. — North-Holland, 1985. — Vol. 109 of North-Holland Mathematics Studies. — P. 27–45. — URL: https://www.sciencedirect.com/science/article/pii/S0304020808731013.

[3] Data profiling with metanome / Thorsten Papenbrock, Tanja Bergmann, Moritz Finke et al. // Proc. VLDB Endow. — 2015. — Aug.. — Vol. 8, no. 12. — P. 1860–1863. — URL: https://doi.org/10.14778/2824032.2824086.

[4] 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.

[5] Galil Zvi. Efficient algorithms for finding maximum matching in graphs // ACM Comput. Surv. — 1986. — Mar.. — Vol. 18, no. 1. — P. 23–38. — URL: https://doi.org/10.1145/6462.6502.

[6] Garey M. R., Johnson D. S. Computers and Intractability: A Guide to the Theory of NP-Completeness (Series of Books in the Mathematical Sciences). — First edition. — W. H. Freeman, 1979. — ISBN: 0716710455. — URL: http://www.amazon.com/Computers-Intractability-NP-Completeness-Mathematical-Sciences/dp/0716710455.

[7] Gupta Pramod, Bagchi Anupam. Introduction to Pandas // Essentials of Python for Artificial Intelligence and Machine Learning. — Cham : Springer Nature Switzerland, 2024. — P. 161–196. — ISBN: 978-3-031-43725-0. — URL: https://doi.org/10.1007/978-3-031-43725-0_5.

[8] Haider Jawad, Fayaz Muhammad. A Smart Approximation Algorithm for Minimum Vertex Cover Problem based on Min-to-Min (MtM) Strategy // International Journal of Advanced Computer Science and Applications. — 2020. — Vol. 11, no. 12. — URL: http://dx.doi.org/10.14569/IJACSA.2020.0111232.

[9] Hüllermeier Eyke. Association Rules for Expressing Gradual Dependencies // Principles of Data Mining and Knowledge Discovery / Ed. by Tapio Elomaa, Heikki Mannila, Hannu Toivonen. — Berlin, Heidelberg : Springer Berlin Heidelberg, 2002. — P. 200–211.

[10] Korobitsyn Igor. FASTDC review. — URL: https://github.com/Desbordante/desbordante-core/blob/main/docs/papers/FASTDC%20review%20-%20Igor%20Korobitsyn%20-%202023%20spring.pdf (дата обращения: 11 апреля 2023 г.).

[11] Leskovec Jure, Krevl Andrej. SNAP Datasets: Stanford Large Network Dataset Collection. — http://snap.stanford.edu/data. — 2014. — Jun..

[12] Pavel Anosov. Constant DC Verification and Minimization. — URL: https://github.com/Desbordante/desbordante-core/blob/main/docs/papers/Constant%20DC%20Verification%20and%20Minimization%20-%20Anosov%20Pavel%20-%202024%20autumn.pdf (дата обращения: 1 марта 2025 г.).

[13] Pavel Anosov. DC Verification. — URL: https://github.com/Desbordante/desbordante-core/blob/main/docs/papers/DC%20Verification%20-%20Anosov%20Pavel%20-%202024%20spring.pdf (дата обращения: 3 ноября 2024 г.).

[14] Rossi Ryan A., Ahmed Nesreen K. The Network Data Repository with Interactive Graph Analytics and Visualization // AAAI. — 2015. — URL: http://networkrepository.com.