

Санкт-Петербургский государственный университет

Кафедра информационно-аналитических систем

Группа 24.Б42-мм

Реализация поиска часто встречающихся подграфов с помощью алгоритма gSpan в Desbordante

Малышев Максим Владимирович

Отчёт по учебной практике
в форме «Решение»

Научный руководитель:
ассистент кафедры ИАС, Чернышёв Г. А.

Санкт-Петербург
2026

Оглавление

Введение	3
1. Постановка задачи	4
2. Обзор	5
2.1. Задача поиска частых подграфов	5
2.2. Основные понятия	5
2.3. Алгоритм gSpan	9
3. Описание решения	13
3.1. Используемые структуры	13
3.2. Особенности реализации	14
3.3. Python-привязки и пример	16
4. Эксперимент	17
Заключение	19
Список литературы	20

Введение

В современном мире объемы данных растут экспоненциально, и всё большую роль играют данные, имеющие сложную структуру. Графовые модели данных находят широкое применение в самых разных областях: от биоинформатики и хемоинформатики (анализ химических соединений и белков) до анализа социальных сетей, рекомендательных систем и анализа программного кода. Одной из фундаментальных задач интеллектуального анализа графовых данных является поиск частых подграфов (Frequent Subgraph Mining, FSM). Эта задача заключается в выявлении подструктур, которые встречаются в наборе графов чаще определенного порога.

Обнаружение частых подграфов позволяет выявлять скрытые закономерности, характерные для класса объектов, классифицировать сложные структуры и индексировать графовые базы данных. Однако высокая вычислительная сложность, связанная с проблемой изоморфизма подграфов, требует применения высокоэффективных алгоритмов, таких как gSpan [2].

Desbordante [1] — высокопроизводительная C++ библиотека для профилирования данных и поиска закономерностей. Изначально спроектированная для эффективного анализа таблиц (поиск функциональных зависимостей, уникальных комбинаций колонок и др.), библиотека постепенно расширяет область своего применения и на другие структуры данных.

На текущий момент возможности Desbordante по работе с графами ограничены: в библиотеке реализованы алгоритмы по поиску и валидации графовых функциональных зависимостей (Graph Functional Dependencies, GFD). Отсутствие фундаментального алгоритма для поиска частых подграфов не позволяет использовать Desbordante как полноценный инструмент для графовой аналитики. Интеграция алгоритма gSpan позволяет заполнить этот пробел, дополнив возможности анализа зависимостей возможностями структурного анализа.

1. Постановка задачи

Целью работы является расширение функциональности Desbordante по работе с графовыми данными, а именно реализация алгоритма gSpan для поиска частых подграфов. Для этого были поставлены следующие задачи:

- сделать обзор алгоритма gSpan, включая его теоретические основы и принципы работы;
- интегрировать алгоритм в проект;
- разработать модульные тесты для проверки корректности реализации;
- добавить возможность запуска алгоритма из Python, сделать простой пример, иллюстрирующий работу;
- сравнить производительность с реализацией в SPMF.

2. Обзор

Все определения, свойства, теоремы, а также псевдокод алгоритмов, приведённые в данном разделе, взяты из работы [2].

2.1. Задача поиска частых подграфов

Пусть задана база данных графов $D = \{G_1, G_2, \dots, G_n\}$, состоящая из n связных помеченных графов. Каждый граф $G_i = (V_i, E_i, L_i, l_i)$ представляет собой четвёрку: множество вершин V_i , множество рёбер $E_i \subseteq V_i \times V_i$, множество меток L_i и функция разметки $l_i : V_i \cup E_i \rightarrow L_i$.

Требуется найти все связные помеченные подграфы g , для которых поддержка (support) в базе данных D не меньше заданного порога min_sup . Поддержка подграфа g определяется как число графов из D , содержащих g как подграф:

$$support(g) = |\{G_i \in D \mid g \sqsubseteq G_i\}|,$$

где $g \sqsubseteq G_i$ означает, что g изоморфно (т.е. существует биекция между вершинами, сохраняющая метки и структуру связей) некоторому подграфу G_i .

2.2. Основные понятия

Определение 1. DFS-дерево (Depth-First Search tree) — это дерево, полученное путём обхода графа в глубину. Для одного графа можно построить несколько различных DFS-деревьев, начиная с разных вершин и выбирая разные рёбра, по которым продолжается обход.

Формально, построение DFS-дерева T для связного графа G осуществляется следующим образом:

1. Выбирается произвольная стартовая вершина $v_0 \in V(G)$, которая становится корнем дерева T .

2. Из вершины v (изначально $v = v_0$) рассматриваются все инцидентные ей рёбра $(v, u) \in E(G)$:

- Если вершина u ещё не посещена, ребро (v, u) добавляется в дерево T , после чего алгоритм рекурсивно вызывается для u .
- Если вершина u уже посещена, ребро не добавляется в дерево T .

3. Процесс завершается, когда все вершины, достижимые из v_0 , посещены.

Относительно конкретного DFS-дерева все рёбра исходного графа классифицируются на два типа:

- **Прямое** (forward edge) — соединяет текущую вершину с новой, ещё не посещённой (входит в DFS-дерево);
- **Обратное** (backward edge) — соединяет текущую вершину с уже посещённой, но не её прямым родителем (не входит в DFS-дерево).

Определение 2. Пусть в результате обхода в глубину графа G построено DFS-дерево T . Каждой вершине v присвоим индекс — целое число, равное порядковому номеру её первого посещения (моменту «открытия») в процессе обхода. Для любых двух вершин u, v с индексами i, j соответственно выполняется:

$$i < j \iff u \text{ была обнаружена раньше } v$$

Вершина с минимальным индексом (v_0) называется **корнем** дерева T . Вершина с максимальным индексом (v_n) называется **крайней правой вершиной** (rightmost vertex).

Определение 3. **Крайним правым путём** (rightmost path) в DFS-дереве T называется простой путь от корня v_0 до крайней правой вершины v_n .

Свойство 1 (Характеристика рёбер через индексы). Для любого ребра (v_i, v_j) относительно построенного DFS-дерева T с назначенными индексами вершин справедливы следующие соотношения:

- Если (v_i, v_j) — **прямое** ребро, то $i < j$.
- Если (v_i, v_j) — **обратное** ребро, то $i > j$.

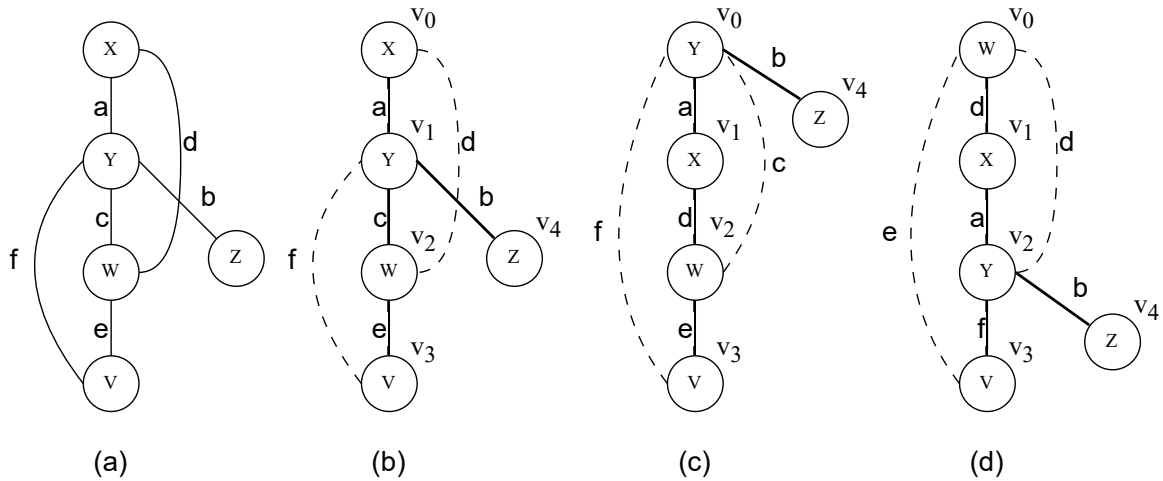


Рис. 1: DFS-деревья (адаптированы из [2])

Пример 1. На рисунке 1 представлены три изоморфных графа (b)–(d) для графа (a). Выделенные жирным рёбра представляют различные DFS-деревья для графа (a). Пунктирной линией выделены обратные рёбра.

Определение 4. Пусть каждое ребро, пройденное при обходе в глубину, представлено упорядоченной парой (i, j) , где i и j — номера вершин, присвоенные в соответствии с порядком их обнаружения. Упорядочим рёбра по следующему правилу.

Пусть $e_1 = (i_1, j_1)$, $e_2 = (i_2, j_2)$, тогда:

- (i) если $i_1 = i_2$ и $j_1 < j_2$, то $e_1 < e_2$;
- (ii) если $i_1 < j_1$ и $j_1 = i_2$, то $e_1 < e_2$;
- (iii) если $e_1 < e_2$ и $e_2 < e_3$, то $e_1 < e_3$.

Интуитивно, одно ребро меньше другого, если по нему «прошли» раньше в процессе DFS.

Определение 5. Пусть T — некоторое DFS-дерево для графа G . **DFS-кодом** (DFS-code) называется последовательность рёбер $code(G, T) = (e_1, e_2, \dots, e_m)$, упорядоченная по правилу выше, то есть в порядке их прохождения при данном обходе в глубину.

Для учёта меток каждое ребро расширим до пятёрки:

$$e = (i, j, l(v_i), l(e_{ij}), l(v_j)),$$

где $l(v_i), l(v_j)$ — метки вершин, $l(e_{ij})$ — метка ребра. Предполагается, что на множестве меток L задан некоторый линейный порядок \prec_L . Тогда на множестве расширенных рёбер, представленных пятёрками, лексикографический порядок определяется как комбинация порядка из определения 4 и порядка \prec_L на метках.

Определение 6 (Лексикографический порядок DFS-кодов). Пусть $\alpha = (a_1, \dots, a_m)$ и $\beta = (b_1, \dots, b_n)$ — DFS-коды. Тогда $\alpha \leq \beta$, если выполняется хотя бы одно из условий:

- (i) $\exists 0 \leq t \leq \min(m, n) : a_k = b_k$ для $k < t$ и $a_t < b_t$;
- (ii) $a_k = b_k$ для $0 \leq k \leq m$ и $n \geq m$.

Таблица 1: DFS-коды для графов, представленных на рис. 1(b)–(d)

ребро	(Рис. 1b) α	(Рис 1c) β	(Рис. 1d) γ
0	(0, 1, X, a, Y)	(0, 1, Y, a, X)	(0, 1, W, d, X)
1	(1, 2, Y, c, W)	(1, 2, X, d, W)	(1, 2, X, a, Y)
2	(2, 0, W, d, X)	(2, 0, W, c, Y)	(2, 0, Y, d, W)
3	(2, 3, W, e, V)	(2, 3, W, e, V)	(2, 3, Y, f, V)
4	(3, 1, V, f, Y)	(3, 0, V, f, Y)	(3, 0, V, e, W)
5	(1, 4, Y, b, Z)	(0, 4, Y, b, Z)	(2, 4, Y, b, Z)

Пример 2. В таблице 1 представлены DFS-коды для трёх изоморфных графов, изображённых на рисунке 1 под буквами (b)–(d). Используя определённый выше лексикографический порядок, нетрудно установить, что $\gamma < \alpha < \beta$.

Определение 7. Минимальным DFS-кодом графа G называется минимальный элемент множества всех его DFS-кодов относительно лексикографического порядка $Z(G)$. Обозначается $\min(Z(G))$.

Теорема 1. Два графа G_1 и G_2 изоморфны тогда и только тогда, когда их минимальные DFS-коды совпадают:

$$G_1 \cong G_2 \iff \min(Z(G_1)) = \min(Z(G_2)).$$

Таким образом, задача поиска частых подграфов сводится к поиску минимальных DFS-кодов.

Определение 8 (Отношение родитель-потомок для DFS-кодов). Пусть $\alpha = (a_1, a_2, \dots, a_m)$ и $\beta = (a_1, a_2, \dots, a_m, b)$ — валидные DFS-коды, где β получен из α добавлением одного нового ребра b , инцидентного вершине на **крайнем правом пути** кода α . Тогда α называется **родителем** β , а β — **потомком** α .

Определение 9. На основе отношения родитель-потомок для DFS-кодов может быть построено **дерево DFS-кодов**, в котором:

- Каждая вершина дерева соответствует некоторому валидному DFS-коду.
- Корень дерева соответствует пустому DFS-коду.
- Отношение порядка между потомками одной вершины соответствует лексикографическому порядку их DFS-кодов.

Обход данного дерева в глубину (pre-order traversal) порождает DFS-коды в лексикографическом порядке.

2.3. Алгоритм gSpan

Наивные подходы к поиску частых подграфов имеют фундаментальные проблемы: во-первых, они генерируют чрезмерное количество

заведомо нечастых подграфов-кандидатов; во-вторых, требуется выполнять проверку изоморфизма графов для устранения дубликатов среди кандидатов, что является вычислительно трудоёмкой задачей.

Алгоритм **gSpan** предлагает решение обеих проблем. Вместо генерации кандидатов используется стратегия последовательного роста подграфов из уже найденных частых фрагментов. Благодаря введённому ранее понятию минимального DFS-кода как канонической формы графа, проверка изоморфизма сводится к простому сравнению DFS-кодов на равенство. Алгоритм состоит из двух частей:

- Подготовка и предобработка графов;
- Рекурсивный рост подграфов и проверка их поддержки.

2.3.1. Основная процедура: **GraphSet_Projection**

Основная процедура, представленная в алгоритме 1, выполняет предварительную обработку данных и запускает рекурсивный поиск.

Algorithm 1 **GraphSet_Projection**(\mathbb{D}, \mathbb{S})

Input: A graph database \mathbb{D} , minimum support threshold $minSup$.

Output: A set of frequent subgraphs \mathbb{S} .

- 1: sort the labels in \mathbb{D} by their frequency;
 - 2: remove infrequent vertices and edges;
 - 3: relabel the remaining vertices and edges;
 - 4: $\mathbb{S}^1 \leftarrow$ all frequent 1-edge graphs in \mathbb{D} ;
 - 5: sort \mathbb{S}^1 in DFS lexicographic order;
 - 6: $\mathbb{S} \leftarrow \mathbb{S}^1$;
 - 7: **for** each edge $e \in \mathbb{S}^1$ **do**
 - 8: initialize s with e , set $s.D$ by graphs which contains e ;
 - 9: **Subgraph_Mining**($\mathbb{D}, \mathbb{S}, s$);
 - 10: $\mathbb{D} \leftarrow \mathbb{D} - e$;
 - 11: **if** $|\mathbb{D}| < minSup$ **then**
 - 12: **break**;
 - 13: **end if**
 - 14: **end for**
-

Шаги 1–6 выполняют предобработку: удаляются редкие метки, которые не могут входить в частые подграфы, а оставшиеся метки пере-

нумеровываются для обеспечения лексикографического порядка. Множество \mathbb{S}^1 частых однорёберных подграфов служит набором «семян» (seeds) для последующего роста.

Шаги 7–14 реализуют основной цикл поиска. Для каждого семени e формируется его проекция $s.D$ — подмножество графов из D , содержащих ребро e . Затем вызывается рекурсивная процедура **Subgraph_Mining**, которая строит всё поддереву дерева DFS-кодов, корнем которого является e . После обработки поддерева ребро e удаляется из всех графов в D (проекция набора данных), что сокращает размер данных для последующих итераций.

2.3.2. Рекурсивная процедура: Subgraph_Mining

Процедура **Subgraph_Mining** (алгоритм 2) реализует рекурсивный рост подграфов. Она проверяет, является ли текущий DFS-код s минимальным (строка 1). Если нет, подграф отсекается как дубликат. В противном случае s добавляется в множество результатов (строка 4). Затем для каждого графа в проекции \mathbb{D} перечисляются все вхождения s и подсчитывается поддержка его потомков (строка 5). Для каждого потомка c с достаточной поддержкой рекурсивно вызывается **Subgraph_Mining** (строки 6–8), продолжая рост подграфа.

Algorithm 2 Subgraph_Mining($\mathbb{D}, \mathbb{S}, s$)

Input: A graph database \mathbb{D} , a set of frequent subgraphs \mathbb{S} , a candidate subgraph s , and a threshold $minSup$.

Output: An updated set of frequent subgraphs \mathbb{S} .

```
1: if  $s \neq min(s)$  then
2:   return;
3: end if
4:  $\mathbb{S} \leftarrow \mathbb{S} \cup \{s\}$ ;
5: enumerate  $s$  in each graph in  $\mathbb{D}$  and count its children;
6: for each  $c, c$  is  $s$ ' child do
7:   if  $support(c) \geq minSup$  then
8:      $s \leftarrow c$ ;
9:     Subgraph_Mining( $\mathbb{D}_s, \mathbb{S}, s$ );
10:  end if
11: end for
```

2.3.3. Ключевые оптимизации

- **Отсечение дубликатов:** проверка $s \neq min(s)$ предотвращает обработку уже найденных подграфов. Вместо генерации всех возможных DFS-кодов для одного и того же графа используется следующий метод: выполняется попытка построить DFS-код для того же графа, но при этом на каждом шаге добавляемого ребра производится сравнение с соответствующим ребром кода s . Если добавляемое ребро оказывается лексикографически меньше, то проверка прекращается — s не является минимальным.
- **Проекция данных:** постепенное удаление обработанных рёбер из набора D сокращает объём данных для последующих итераций, ускоряя работу алгоритма.

3. Описание решения

Для интеграции алгоритма `gSpan` в платформу `Desbordante` за основу была взята реализация на `Java` из библиотеки с открытым исходным кодом `SPMF`. Поскольку `Desbordante` разработана на `C++` и имеет собственную архитектуру для работы с данными и алгоритмами, потребовалась адаптация исходного кода, включающая перевод на `C++` и перепроектирование в соответствии с внутренней архитектурой.

3.1. Используемые структуры

Для реализации алгоритма в проекте `Desbordante` были созданы следующие классы и структуры:

- Класс `GSpan`;
- Структура `ExtendedEdge`;
- Класс `DFSCode`;
- Тип `FrequentSubgraph`;
- Класс `SparseTriangularMatrix`;

Рассмотрим назначение и особенности реализации каждого компонента подробнее.

Класс `GSpan` — основной модуль для интеграции алгоритма. Является наследником класса `Algorithm`. Выполняет предобработку данных и осуществляет поиск частых подграфов.

Структура `ExtendedEdge` представляет собой расширенное ребро графа, используемое в процессе генерации кандидатов.

Класс `DFSCode` — ключевой в алгоритме. Реализован как `std::vector<ExtendedEdge>` с дополнительными модификациями, например полем `rightmost` для хранения крайней правой вершины и методом `On-RightMostPath`, позволяющим проверить, лежит ли вершина на крайнем правом пути.

Тип FrequentSubgraph — структура-контейнер, предназначенная для хранения результатов работы алгоритма. Она хранит найденный подграф в виде `DFSCode`, вычисленное для него значение поддержки и список идентификаторов графов, которые содержат данный подграф.

Класс SparseTriangularMatrix — вспомогательная структура данных, реализованная для оптимизации использования памяти. Алгоритму необходимо хранить информацию о частоте встречаемости пар меток ребер и вершин на ранних этапах. Поскольку количество уникальных меток может быть велико, но граф связей разрежен, использование полной матрицы смежности неэффективно. Использование двух вложенных хеш-таблиц (`std::unordered_map`) обеспечивает быстрый доступ и эффективное использование памяти.

Для наглядности на рисунке 2 приведена упрощённая UML-диаграмма классов.

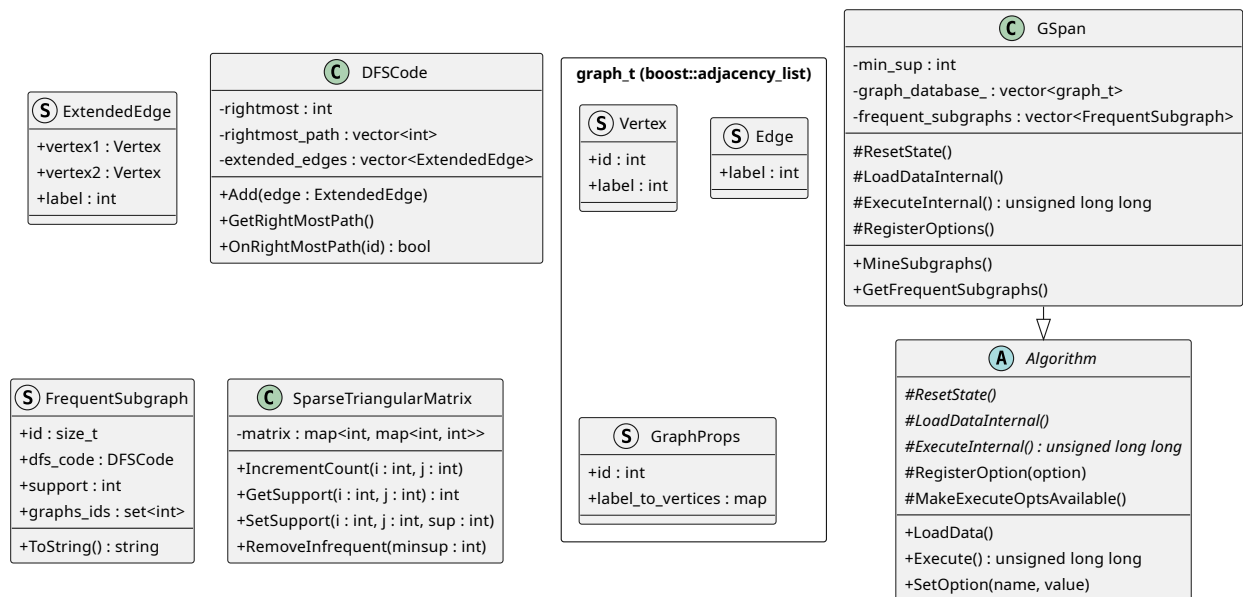


Рис. 2: Диаграмма классов

3.2. Особенности реализации

Несмотря на наличие в Desbordante модуля GFD, его инфраструктура не подходит для `gSpan`: в GFD вершины хранят словарь строковых атрибутов, тогда как `gSpan` оперирует единственной целочисленной

меткой. Кроме того, `gSpan` требует индекса «метка \rightarrow список вершин с этой меткой» для эффективного поиска. В связи с этим введён отдельный тип `graph_t` на основе `boost::adjacency_list`. В вершинах и рёбрах хранятся метки, а в свойствах графа — его идентификатор и требуемый индекс.

Парсер также реализован отдельно, поскольку `gSpan` работает с базой графов (множество графов в одном файле), а не с единственным графом. Формат входных данных задаётся строками вида `t # <id>` (начало графа), `v <vertex_id> <label>` (вершина), `e <v1> <v2> <label>` (ребро). При чтении парсер сразу строит необходимые индексы.

Для управления работой алгоритма были добавлены параметры запуска:

- `minsup` — минимальная поддержка как доля от числа графов (значение в диапазоне $(0, 1]$);
- `output_single_vertices` — выводить ли частые подграфы из одной вершины;
- `max_number_of_edges` — ограничение на максимальный размер паттерна по числу рёбер;
- `output_path` — путь к файлу для сохранения результата (если не задан, результат не сохраняется).

Результаты сохраняются в виде списка объектов `FrequentSubgraph`. При необходимости они записываются в файл в текстовом формате, включающем описание вершин и рёбер паттерна, а также строку со списком идентификаторов графов, в которых он встречается.

Для проверки корректности реализации в проект были добавлены модульные тесты. Они включают тестирование базовых структур (`DFSCode`, `ExtendedEdge`, `FrequentSubgraph`), тесты корректности чтения графовой базы и тесты запуска алгоритма на небольших синтетических примерах. Также проверяются граничные случаи параметров запуска (например, корректность диапазона `minsup`).

3.3. Python-привязки и пример

Для использования алгоритма из Python были добавлены Python bindings на основе `pybind11`. Пользователь может создать объект алгоритма, загрузить данные с указанием `graph_database` и `minsup`, выполнить поиск и получить список найденных подграфов. Кроме того, были подготовлены небольшие Python-примеры, демонстрирующие типовой сценарий запуска, один из которых приведён ниже.

```
1 import desbordante
2 GRAPH = 'examples/datasets/fsm/gspan_test_simple.txt'
3 algo = desbordante.gspan.algorithms.GSpan()
4 algo.load_data(graph_database=GRAPH, minsup = 0.9)
5 algo.execute()
6 result = algo.get_frequent_subgraphs()
7 print(len(result), 'frequent subgraphs found with minsup = 0.9:\n')
8 print('Frequent Subgraphs:')
9 for sg in result:
10     print(sg)
```

В этом примере демонстрируется базовое использование Python bindings для алгоритма `gSpan`. Код импортирует модуль `desbordante` (строка 1), инициализирует объект `GSpan`, загружает графовый датасет с минимальной поддержкой 0.9 (строки 2–4), выполняет поиск частых подграфов (строка 5) и выводит результаты (строки 6–10). Это простой демонстрационный пример, предназначенный для проверки работоспособности привязок в рамках текущего пуллреквеста; для финального релиза планируется его расширение с целью повышения пригодности для обучения пользователей.

4. Эксперимент

Для оценки корректности и эффективности реализации алгоритма gSpan в платформе Desbordante были проведены эксперименты, направленные на сравнение её производительности с эталонной реализацией из библиотеки SPMF.

Таблица 2: Характеристики тестовых датасетов

Датасет	Графов	Ср. вершин	Ср. рёбер
Mutag	188	17.93	19.79
Enzymes	600	32.63	62.14
DandD	1,178	284.31	715.66

Замеры производительности проводились на трёх общедоступных¹ датасетах, представленных в таблице 2. Результаты усреднялись по пяти запускам.

Таблица 3: Результаты измерений

Датасет	minsup	Desbordante (сек)	SPMF (сек)	Ускорение
Mutag	0.6	12	4	0.33
Mutag	0.4	89	31	0.35
Mutag	0.2	1198	330	0.28
Enzymes	0.8	11	9	0.82
Enzymes	0.7	102	80	0.78
Enzymes	0.6	984	691	0.7
DandD	0.6	6	6	1
DandD	0.4	17	18	1.06
DandD	0.2	89	95	1.07

Как видно из таблицы 3, при снижении порога minsup время работы алгоритма резко растёт для всех датасетов. Это объясняется тем, что более низкий порог заставляет алгоритм учитывать значительно большее количество частых подграфов, что приводит к экспоненциальному расширению пространства поиска.

При этом видно, что с ростом размера и сложности датасета Desbordante работает эффективнее. Если на маленьких графах быстрее оказывается SPMF, то на самом крупном датасете DandD реализация

¹<https://www.philippe-fournier-viger.com/spmf/index.php?link=datasets.php>

на C++ выходит вперед и показывает лучшее время при низких значениях `minsup`. Полученные результаты могут послужить основой для дальнейшей работы по оптимизации имеющейся реализации.

Заключение

В ходе работы были выполнены следующие задачи:

- создан обзор алгоритма gSpan, включая теоретические основы и принципы работы;
- реализована интеграция алгоритма в Desbordante;
- добавлены модульные тесты к алгоритму;
- добавлен Python-интерфейс для использования алгоритма, сделан демонстрационный пример;
- проведено сравнение производительности с SPMF.

Исходный код алгоритма доступен на GitHub².

²<https://github.com/Desbordante/desbordante-core/pull/666>

Список литературы

- [1] Desbordante: from benchmarking suite to high-performance science-intensive data profiler / George Chernishev, Michael Polyntsov, Anton Chizhov et al. // Proceedings of the 8th International Conference on Data Science and Management of Data (12th ACM IKDD CODS and 30th COMAD). — CODS-COMAD '24. — New York, NY, USA : Association for Computing Machinery, 2025. — P. 234–243. — URL: <https://doi.org/10.1145/3703323.3703725>.
- [2] Yan Xifeng, Han Jiawei. gSpan: Graph-Based Substructure Pattern Mining // Proceedings of the 2002 IEEE International Conference on Data Mining. — ICDM '02. — USA : IEEE Computer Society, 2002. — P. 721.