



Министерство науки и высшего образования Российской Федерации
Федеральное государственное автономное образовательное учреждение
высшего образования
«Московский государственный технический университет имени
Н.Э. Баумана (национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления»
КАФЕДРА «Компьютерные системы и сети (ИУ-6)»

ОТЧЕТ О ВЫПОЛНЕНИИ ДОМАШНЕЙ РАБОТЫ
по дисциплине «Математические методы анализа данных и
принятия решений»

Студент: Козлов Владимир Михайлович
Группа: ИУ6-13М
Тип задания: домашняя работа
Тема: Байесовский классификатор

Студент

Козлов В.М.

подпись, дата

Фамилия, И.О.

Преподаватель

подпись, дата

Фамилия, И.О.

Москва, 2025

бла бла

Оглавление

Введение	4
1 Основная часть	5
1.1 Постановка задачи распределения вершин графа	5
1.2 Streaming Graph Partitioning	5
1.3 Алгоритм Fennel	6
1.3.1 Объективная функция	6
1.3.2 Инкрементальное вычисление и формула для δg	7
1.4 Программная реализация	7
1.4.1 Структура проекта и основные зависимости	7
1.4.2 Классы для представления графовых структур (graph.hpp)	8
1.4.3 Структура класса FennelStorage	10
1.5 Тестирование	12
Заключение	15
Список использованных источников	16

Введение

Современный этап развития информационных технологий характеризуется экспоненциальным ростом объемов данных и усложнением связей между ними. Графовые модели данных становятся стандартом де-факто для представления информации в таких областях, как социальные сети, рекомендательные системы, биоинформатика, телекоммуникации и интернет вещей. Эффективность работы с графами большого размера напрямую зависит от способности системы хранить и обрабатывать их, используя распределенные ресурсы.

Ключевым компонентом любой распределенной графовой базы данных (БД) является стратегия распределения вершин графа по узлам хранения (шардам). От того, насколько удачно вершины сгруппированы в шарды, зависит производительность выполнения основных операций, прежде всего — поиска пути между вершинами. Если вершины, часто участвующие в одном запросе, оказываются в разных хранилищах, это приводит к значительным сетевым издержкам и росту задержек. Таким образом, задача минимизации количества межшардовых соединений (мощности разрезов) при сохранении баланса нагрузки на узлах является критически важной для обеспечения высокой производительности распределенных графовых систем.

В рамках данной практики разрабатывается модуль оптимального распределения вершин для графовой БД. В качестве базового алгоритма выбран метод Fennel, относящийся к классу алгоритмов потокового распределения графов. Целью работы является не только теоретическое изучение современных подходов к партиционированию графов, но и практическая реализация алгоритма Fennel, способного эффективно распределять поступающий поток вершин в режиме реального времени.

1 Основная часть

1.1 Постановка задачи распределения вершин графа

В контексте распределенных графовых баз данных задача распределения вершин (графового партиционирования) формулируется следующим образом. Пусть дан неориентированный граф $G = (V, E)$, где V — множество вершин, $|V| = n$, а E — множество рёбер, $|E| = m$. Требуется найти такое разбиение (распределение) $P = \{S_1, S_2, \dots, S_k\}$ множества вершин V на k непересекающихся подмножеств (шардов) S_i , что:

- $\bigcup_{i=1}^k S_i = V$.
- $S_i \cap S_j = \emptyset$ для всех $i \neq j$.

Качество распределения оценивается по двум основным критериям, которые необходимо оптимизировать:

1. **Минимизация разрезов (Edge Cut).** Мощность разреза $\partial e(P)$ определяется как количество рёбер, концы которых оказались в разных шардах. Формально:

$$|\partial e(P)| = \sum_{i=1}^k |e(S_i, V \setminus S_i)|$$

Минимизация этого параметра напрямую влияет на снижение сетевого трафика при выполнении распределенных запросов.

2. **Балансировка нагрузки (Load Balancing).** Неравномерное распределение вершин приводит к перегрузке одних узлов и простоте других. Для количественной оценки используется показатель нормализованной максимальной нагрузки ρ , который необходимо минимизировать:

$$\rho = \frac{\max_{i=1}^k (|S_i|)}{n/k}$$

Идеально сбалансированное распределение соответствует $\rho = 1$.

Таким образом, целевая задача заключается в поиске такого распределения P^* , которое минимизирует мощность разрезов $|\partial e(P)|$ при максимально допустимом значении дисбаланса $\rho \leq \tau$, где τ — заданный порог.

1.2 Streaming Graph Partitioning

Традиционные методы оптимизации распределения, такие как METIS, требуют полного знания структуры графа для выполнения разбиения и работают в пакетном (offline) режиме. Однако во многих современных приложениях графы динамически изменяются: новые вершины и рёбра поступают непрерывно. Для таких сценариев разработаны методы потокового распределения (online partitioning).

В модели потокового распределения [1] вершины графа поступают на вход системы последовательно (в виде потока). Алгоритм должен принять решение о размещении каждой новой вершины v в одном из k шардов немедленно, основываясь только на информации об уже распределенных вершинах и, возможно, на знании соседей $N(v)$ новой вершины. Это накладывает жесткие ограничения на вычислительную сложность, но позволяет обрабатывать графы произвольного размера.

В работе [1] рассматривается семейство эвристик для потокового партиционирования, которые используют весовую функцию для учета текущей загруженности шардов и структуры связей. Цель эвристик — максимизировать количество соседей новой вершины в том шарде, куда она помещается. Общая формула для выбора шарда i выглядит следующим образом:

$$\text{ind} = \arg \max_{i \in \{1..k\}} \{|P^t(i) \cap N(v)| \cdot w(t, i)\}$$

где $P^t(i)$ — множество вершин в шарде i в момент времени t , а $w(t, i)$ — весовая функция, отвечающая за балансировку (например, линейная $w(t, i) = 1 - |P^t(i)|/C$ или экспоненциальная, где C — максимальная вместимость шарда). Одной из наиболее эффективных эвристик, согласно экспериментальным данным, является линейная детерминированная жадная эвристика (LDG).

1.3 Алгоритм Fennel

Алгоритм Fennel [2] представляет собой развитие идей потокового распределения графов и предлагает более формализованный подход к построению эвристики. Вместо комбинирования дискретных метрик (количество соседей) и весовых функций, Fennel вводит непрерывную объективную функцию $g(P)$, характеризующую качество текущего распределения P .

1.3.1 Объективная функция

Авторы Fennel определяют объективную функцию для k -распределения следующим образом:

$$g(P) = \sum_{i=1}^k [e(S_i, S_i) - p \binom{|S_i|}{2}]$$

где:

- $e(S_i, S_i)$ — количество рёбер, оба конца которых лежат внутри шарда S_i (внутренние рёбра).
- $p = 2\alpha = m \frac{k^{\gamma-1}}{n^\gamma}$, $1 \leq \gamma \leq 2$.
- α — параметр, регулирующий важность балансировки.
- γ — параметр, определяющий форму штрафа за размер шарда. В оригинальной работе рекомендуется $\gamma = \frac{3}{2}$.

Логика функции проста: она поощряет размещение рёбер внутри одного шарда (увеличение $e(S_i, S_i)$) и штрафует за слишком большой размер шарда (член $-p(|S_i|)$), что способствует балансировке.

1.3.2 Инкрементальное вычисление и формула для δg

Ключевым преимуществом Fennel для потоковой обработки является то, что для принятия решения о размещении новой вершины v не нужно пересчитывать $g(P)$ целиком. Достаточно вычислить прирост $\delta g(v, S_i)$, который получит система, если вершина v будет добавлена в существующий шард S_i .

Тогда предположить, что соседи v уже распределены, то добавление v в S_i увеличит количество внутренних рёбер в этом шарде на число соседей v , уже находящихся в S_i , то есть на $|S_i \cap N(v)|$.

Штраф за размер шарда изменяется с $-p(|S_i|)$ на $-p(|S_i|+1)$. Таким образом, изменение $\delta g(v, S_i)$ для шарда S_i вычисляется по формуле:

$$\delta g(v, S_i) = |S_i \cap N(v)| - p\left(\binom{|S_i| + 1}{2}\right) - \binom{|S_i|}{2} \quad (1)$$

Именно это значение и используется в качестве основной эвристики. Алгоритм выбирает для вершины v тот шард i , который максимизирует $\delta g(v, S_i)$. Такой подход позволяет производить все вычисления распределенно: каждый шард может независимо рассчитать δg для себя, после чего выбирается шард с максимальным значением, либо случайный среди максимальных равных.

1.4 Программная реализация

В рамках данной курсовой работы был реализован модуль потокового распределения вершин и модифицированы реализованные ранее модули:

1. Классы имитации шины для общения компонентов
2. Класс Fennel-хранилища (FennelStorage), расширяющий обычное хранилище для поддержки алгоритма феннеля.

1.4.1 Структура проекта и основные зависимости

Языком разработки был выбран C++ в силу его низкоуровневости и скорости, а также с намерением в дальнейшем внедрить эти наработки в графовую БД на C++ научного руководителя.

Проект организован в виде набора заголовочных файлов (header files), что соответствует современным подходам разработки на C++. Основные файлы проекта:

- `graph.hpp` – содержит базовые классы для представления графовых структур.

- `interface_bus.hpp` – содержит интерфейс, описывающий основные сообщения в шине.
- `bus.hpp` – содержит простую реализацию имитации шины.
- `storage.hpp` – реализует класс хранилища для управления вершинами.
- `optimizer.hpp` – содержит реализацию оптимизатора с вычислением метрики g_v .
- `main.cpp` – демонстрационный файл с тестовым сценарием.

Ниже представлен релевантный для решения задачи практики код. Полный код представлен в приложении А.

1.4.2 Классы для представления графовых структур (`graph.hpp`)

Класс интерфейса шины (`interface_bus.hpp`)

Крайне важный интерфейс, через который происходит взаимодействие всей системы. Описывает методы добавления, удаления и запроса вершин, а также объявлений о добавлении и удалении для возможности другим хранилищам знать где находится другая вершина.

И наконец ключевые для алгоритма Кернигана-Лина методы получения граничных вершин и рёбер `ask_neighbours_to_storage` и `ask_edges_to_storage`.

Листинг 1: Базовая структура класса `IBus`

```

1 template <typename KeyType>
2 class IBus {
3 public:
4     virtual Node<KeyType> request_node(const NodeKey<KeyType>& node) = 0;
5     virtual int send_add_node(const Node<KeyType>& node) = 0;
6     virtual bool send_add_node(const Node<KeyType>& node, int storage_id) = 0;
7     virtual bool send_remove_node(const NodeKey<KeyType>& node) = 0;
8     virtual bool send_remove_node(const NodeKey<KeyType>& node, int storage_id) = 0;
9     virtual int ask_who_has(int asker_id, NodeKey<KeyType> key) = 0;
10    virtual void announce_add(NodeKey<KeyType> key, int storage_id,
11        std::set<Edge<KeyType>> edges) = 0;
12    virtual void announce_remove(NodeKey<KeyType> key, int storage_id) = 0;
13    virtual float get_streaming_euristics_change(const
14        Node<KeyType>& node, int storage_id) = 0;
15    // запрашивает у source вершины, соседствующие с target

```

```

14     virtual std::set<Node<KeyType>> ask_neighbours_to_storage(int
15         source, int target) = 0;
16     // запрашивает у source рёбра, идущие в target
17     virtual std::set<Edge<KeyType>> ask_edges_to_storage(int source,
18         int target) = 0;
19 };

```

Класс шины SimpleBus (bus.hpp)

Простая синхронная однопоточная реализация методов IBus. В данной работе важна реализация send_add_node, реализующий выбор хранилища для отправки вершины и вспомогательном метода get_streaming_euristics_change:

Листинг 2: Реализация метода get_streaming_euristics_change

```

1 int send_add_node(const Node<KeyType>& node) override {
2     std::vector<IStorage<KeyType>*> best_storages;
3     float best_euristics = -10000000;
4
5     for (typename std::map<int, IStorage<KeyType>*>::iterator it =
6         storages.begin(); it != storages.end(); ++it) {
7         float this_euristics =
8             it->second->get_streaming_euristics_change(node, total_edges);
9         if (this_euristics > best_euristics) {
10             best_storages.clear();
11             best_storages.push_back(it->second);
12             best_euristics = this_euristics;
13         } else if (this_euristics == best_euristics) {
14             best_storages.push_back(it->second);
15         }
16     }
17     if (best_storages.empty()) {
18         return -1;
19     } else {
20         std::uniform_int_distribution<int> dist{0, best_storages.size()
21 - 1};
22         IStorage<KeyType>* random_storage = best_storages[dist(gen)];
23         send_add_node(node, random_storage->get_id());
24         return random_storage->get_id();
25     }
26 }
27
28 float get_streaming_euristics_change(const Node<KeyType>& node, int
29         storage_id) override {
30     if (storages.find(storage_id) == storages.end()) {
31         return false;

```

```

28     }
29     return
30     storages[storage_id] -> get_streaming_euristics_change(node,
31     total_edges);
32 }
```

Как можно увидеть, метод опрашивает все хранилища и выбирает наилучшее. Если "лучших" окажется несколько (обычно если ни в одном хранилище нет соседей новой вершины), то выбирается случайный.

Класс хранилища (storage.hpp)

Класс `Storage` представляет собой хранилище вершин графа и реализует логику управления внутренними и внешними связями.

1.4.3 Структура класса `FennelStorage`

Для работы ФенNELь-алгоритму требуется знания о кол-ве шардов, а также балансировочное число. Для передачи этих данных используется структура `FennelParameters`, представленная в листинге 3.

Листинг 3: Структура `FennelParameters`

```

1 struct FennelParameters {
2     /*k*/int storage_number;
3     /*gamma*/float balancing_number;
4 };
```

Само же хранилище `FennelStorage` представлено в листинге 4

Листинг 4: Базовая структура класса `Storage`

```

1 template <typename KeyType>
2 class FennelStorage : public Storage<KeyType> {
3 protected:
4     typedef Node<KeyType> StorageNode;
5     typedef NodeKey<KeyType> Key;
6     FennelParameters params;
7     int number_of_nodes;
8     float streaming_euristics = 0;
9
10    float get_p_parameter(int number_of_nodes, long number_of_edges){
11        if (number_of_nodes == 0) {
12            return 0;
13        }
14        return 2 * number_of_edges * std::pow(params.storage_number ,
15            (params.balancing_number - 1)) / std::pow(number_of_nodes ,
16            params.balancing_number);
```

```

15 }
16
17 float fennel_function(const Node<KeyType>& node, long total_edges) {
18     float p = get_p_parameter(number_of_nodes, total_edges);
19     float dg = 0;
20     for (typename std::map<Key, Edge<KeyType>>::const_iterator
21         edge_it = node.edges.begin(); edge_it != node.edges.end();
22         ++edge_it) {
23         const Key& neighbor_key = edge_it->first;
24         const Edge<KeyType>& edge = edge_it->second;
25         // Ищем соседа в текущем хранилище
26         typename std::map<Key, StorageNode>::iterator it2 =
27             this->nodes.find(neighbor_key);
28         if (it2 != this->nodes.end()) {
29             ++dg;
30         }
31     }
32 public:
33
34 FennelStorage(int id, FennelParameters _params, std::map<Key,
35                 StorageNode> _nodes = std::map<Key, StorageNode>())
36     : Storage<KeyType>(id, _nodes), params(_params) {}
37
38 float get_streaming_euristics_change(const Node<KeyType>& node, long
39 total_edges) override {
40     return fennel_function(node, total_edges);
41 }
42
43 void get_add_announcement(Key key, int announcer_id,
44 std::set<Edge<KeyType>> edges) override {
45     ++number_of_nodes;
46     Storage<KeyType>::get_add_announcement(key, announcer_id, edges);
47 }
48
49 void get_remove_announcement(Key key, int announcer_id) override {
50     --number_of_nodes;
51     Storage<KeyType>::get_remove_announcement(key, announcer_id);
52 }

```

1.5 Тестирование

Для тестирования создаётся простой граф показанный на рисунке 1

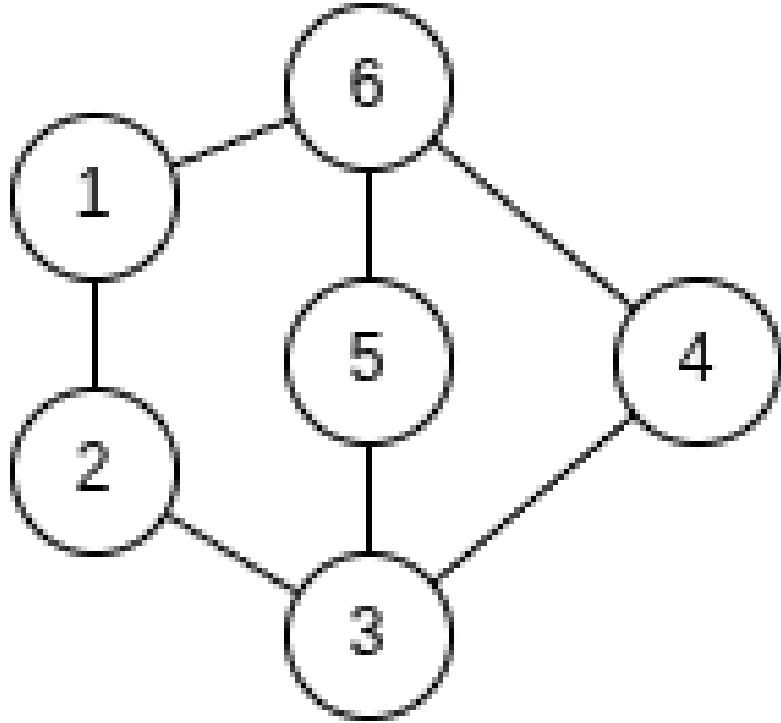


Рис. 1: Граф для тестирования

Вершины будут отправлять в порядке увеличения id, на каждом шаге вершина должна добавляться на шард, на котором у неё будет минимум разрезанных рёбер, если это не вызовет слишком сильный дисбаланс кол-ва вершин на шардах.

Листинг 5: Вывод тестовой программы

```
Adding node Node(key: 1, edges: [Edge(from: 1, to: 2, weight: 6), Edge(from: 1, to: 6, weight: 1)]), storage 1 has euristics 0, storage 2 has euristics 0
Storage 1, adding 1
Adding node Node(key: 2, edges: [Edge(from: 1, to: 2, weight: 6), Edge(from: 2, to: 3, weight: 7)]), storage 1 has euristics 1, storage 2 has euristics 0
Storage 1, adding 2
Adding node Node(key: 3, edges: [Edge(from: 2, to: 3, weight: 7), Edge(from: 3, to: 4, weight: 2), Edge(from: 3, to: 5, weight: 5)]), storage 1 has euristics -1, storage 2 has euristics 0
Storage 2, adding 3
Adding node Node(key: 4, edges: [Edge(from: 3, to: 4, weight: 2), Edge(from: 4, to: 6, weight: 3)]), storage 1 has euristics -2.17732, storage 2 has euristics -0.0886621
Storage 2, adding 4
Adding node Node(key: 5, edges: [Edge(from: 3, to: 5, weight: 5), Edge(from: 5, to: 6, weight: 6)]), storage 1 has euristics -2.12132, storage 2 has euristics -1.12132
Storage 2, adding 5
```

```

Adding node Node(key: 6, edges: [Edge(from: 1, to: 6, weight: 1), Edge(from: 4,
    ↪ to: 6, weight: 3), Edge(from: 5, to: 6, weight: 6)]), storage 1 has
    ↪ euristics -1.02386, storage 2 has euristics -1.03579
Storage 1, adding 6
Storage(id: 1, nodes: 3 {
    Node(key: 1, edges: [Edge(from: 1, to: 2, weight: 6), Edge(from: 1, to: 6,
        ↪ weight: 1)]),
    Node(key: 2, edges: [Edge(from: 1, to: 2, weight: 6), Edge(from: 2, to: 3,
        ↪ weight: 7)]),
    Node(key: 6, edges: [Edge(from: 1, to: 6, weight: 1), Edge(from: 4, to: 6,
        ↪ weight: 3), Edge(from: 5, to: 6, weight: 6)])
}, external_edges: {
    to storage 2: {
        external node 3: [
            local node 2 -> Edge(from: 2, to: 3, weight: 7)
        ],
        external node 4: [
            local node 6 -> Edge(from: 4, to: 6, weight: 3)
        ],
        external node 5: [
            local node 6 -> Edge(from: 5, to: 6, weight: 6)
        ]
    }
})
Storage(id: 2, nodes: 3 {
    Node(key: 3, edges: [Edge(from: 2, to: 3, weight: 7), Edge(from: 3, to: 4,
        ↪ weight: 2), Edge(from: 3, to: 5, weight: 5)]),
    Node(key: 4, edges: [Edge(from: 3, to: 4, weight: 2), Edge(from: 4, to: 6,
        ↪ weight: 3)]),
    Node(key: 5, edges: [Edge(from: 3, to: 5, weight: 5), Edge(from: 5, to: 6,
        ↪ weight: 6)])
}, external_edges: {
    to storage 1: {
        external node 2: [
            local node 3 -> Edge(from: 2, to: 3, weight: 7)
        ],
        external node 6: [
            local node 4 -> Edge(from: 4, to: 6, weight: 3),
            local node 5 -> Edge(from: 5, to: 6, weight: 6)
        ]
    }
})

```

В результате получаем следующий граф:

Storage 1

Storage 2

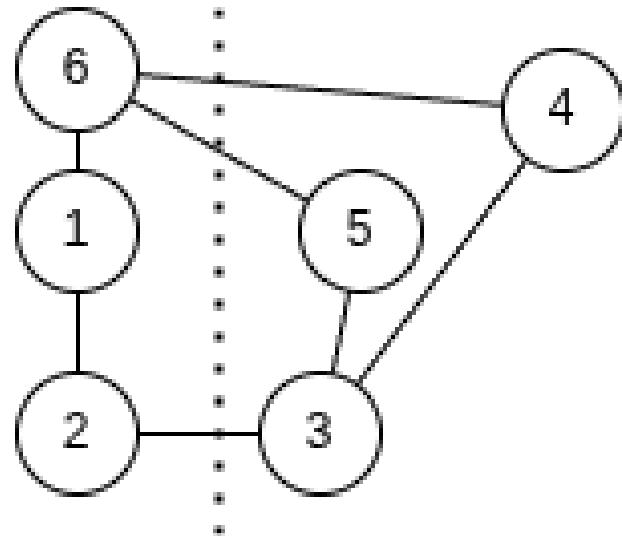


Рис. 2: Распределённый граф

Можно убедиться, что эвристика считалась верно и вершины отправлялись на шарды, которые должны.

Заключение

В ходе прохождения преддипломной практики была изучена и проанализирована предметная область, связанная с распределенным хранением графовых данных и методами оптимизации размещения вершин в условиях динамически поступающей информации. Основное внимание уделено современным подходам к потоковому партиционированию графов (Streaming Graph Partitioning), которые позволяют обрабатывать графы произвольного размера в режиме реального времени.

В процессе выполнения работы были решены следующие задачи, необходимые для подготовки выпускной квалификационной работы:

1. Проведен аналитический обзор научно-технической литературы по методам потокового распределения вершин графа, включая линейную детерминированную жадную эвристику (LDG) и алгоритм Fennel.
2. Выполнена формальная постановка задачи оптимизации распределения в контексте потоковой модели обработки, определены ключевые критерии качества: минимизация мощности разрезов (Edge Cut) и балансировка нагрузки на узлах хранения.
3. Детально изучен математический аппарат алгоритма Fennel, включая вывод целевой функции и формулы инкрементального расчета прироста качества $\delta g(v, S_i)$, что позволяет принимать решение о размещении каждой новой вершины на основе только уже распределенных данных.
4. В рамках практической реализации на языке C++ был разработан прототип модуля распределения, интегрирующий алгоритм Fennel в существующую архитектуру имитационной модели графовой базы данных. Были созданы классы ‘FennelStorage’ и модифицированы механизмы шины (‘IBus’) для сбора эвристик и принятия решений о размещении вершин.
5. Проведено тестирование разработанного прототипа на модельном графе, которое подтвердило корректность работы эвристики: вершины группируются в соответствии со структурой связей, стремясь минимизировать количество межшардовых соединений на этапе потокового распределения.

Список использованных источников

- [1] Stanton Isabelle, Kliot Gabriel. Streaming Graph Partitioning for Large Distributed Graphs. 2012. C. 1222–1230.
- [2] Fennel: Streaming Graph Partitioning for Massive Scale Graphs / Charalampos Tsourakakis, Christos Gkantsidis, Bozidar Radunovic [и др.]. 2014. C. 333–342.