



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	_____	<u>Козлов В.М.</u>
	подпись, дата	Фамилия, И.О.
Преподаватель	_____	_____
	подпись, дата	Фамилия, И.О.

Москва, 2024

бла бла

# Оглавление

<b>Введение</b> .....	<b>4</b>
<b>1 Основная часть</b> .....	<b>5</b>
1.1 Постановка задачи распределения графа .....	5
1.2 Общая архитектура .....	5
1.3 Программная реализация .....	6
1.3.1 Структура проекта и основные зависимости . . . . .	6
1.3.2 Классы для представления графовых структур (graph.hpp) . . . . .	7
1.3.3 Структура класса Storage . . . . .	11
1.3.4 Класс оптимизатора (optimizer.hpp) . . . . .	16
1.4 Тестирование .....	19
<b>Заключение</b> .....	<b>21</b>
<b>Список использованных источников</b> .....	<b>22</b>
<b>Приложение А</b> .....	<b>23</b>

# Введение

Современные распределённые графовые базы данных сталкиваются с фундаментальной проблемой эффективного распределения вершин графа по узлам хранения (шардам). Оптимальное распределение становится критически важным для производительности систем, где основной операцией является поиск путей между вершинами, которые могут находиться в разных шардах. Неэффективное распределение приводит к значительным задержкам при выполнении запросов и избыточным сетевым коммуникациям между узлами.

Актуальность данной работы обусловлена стремительным ростом объёмов графовых данных в таких областях, как социальные сети, рекомендательные системы, биоинформатика и интернет вещей. Традиционные подходы к распределению данных демонстрируют ограниченную эффективность при работе с графами, требующими учёта структурных особенностей и связности вершин.

В рамках исследования проводится сравнительный анализ двух принципиально различных подходов к распределению графов: потоковых методов (online partitioning), работающих в реальном времени по мере поступления данных, и методов оптимизации распределения (offline partitioning), требующих полного знания структуры графа. Особое внимание уделяется алгоритмам библиотеки METIS, представляющей собой промышленный стандарт для задач разбиения графов, и современным потоковым алгоритмам, таким как Fennel и Streaming Graph Partitioning.

Целью работы является исследование и сравнение эффективности различных методов распределения вершин графа по гомогенным хранилищам, а также разработка предложений по комбинированию подходов для достижения оптимального баланса между качеством разбиения и вычислительной эффективностью в условиях реальной эксплуатации распределённых графовых баз данных.

# 1 Основная часть

## 1.1 Постановка задачи распределения графа

Формальная постановка задачи, рассматриваемой в работе, может быть сформулирована следующим образом. Дан граф  $G = (V, E)$ , где  $|V| = n$  - количество вершин,  $|E| = m$  - количество рёбер. Распределение графа представляет собой разбиение  $P = \{S_1, S_2, \dots, S_k\}$ , где  $S_i$  - набор вершин (шард) такой, что  $S_i \cap S_j = \emptyset$  для  $i \neq j$  и  $\bigcup_{i=1}^k S_i = V$ .

Требуется найти такое распределение  $P^* = \{S_1^*, S_2^*, \dots, S_k^*\}$ , которое:

1. Минимизирует общую мощность разрезов:

$$|\partial e(P)| = \left| \bigcup_{i=1}^k e(S_i^*, V \setminus S_i^*) \right| \rightarrow \min \quad (1)$$

или относительную величину:

$$\lambda = \frac{|\partial e(P)|}{m} \times 100\% \rightarrow \min \quad (2)$$

2. Минимизирует нормализованную максимальную нагрузку (максимизирует балансировку):

$$\rho = \frac{\max_{i=1..k} (|S_i^*|)}{\frac{n}{k}} \rightarrow \min \quad (3)$$

Эта задача относится к классу NP-сложных задач [1], что обуславливает необходимость использования эвристических подходов, среди которых алгоритм Кернигана-Лина занимает важное место [2].

## 1.2 Общая архитектура

Упрощённая архитектура представлена на рисунке 1. БД представляет собой состоит из одного мастера и множество хранилищ, соединённых общей шиной, через которую происходит общение.

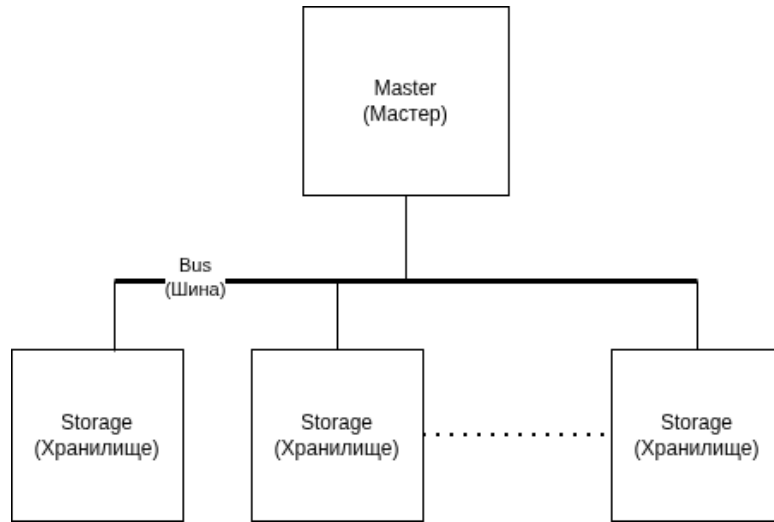


Рис. 1: Упрощённая архитектура

### 1.3 Программная реализация

В рамках данной курсовой работы была реализована вычислительная модель для расчёта и оптимизации метрики улучшения распределения  $g_v$  для граничных вершин графа согласно алгоритму Кернигана-Лина. Реализация включает в себя четыре основных компонента:

1. Классы для представления графовых структур (вершины, рёбра, ключи).
2. Классы имитации шины для общения компонентов
3. Класс хранилища (Storage) для управления вершинами и их связями.
4. Класс оптимизатора (StorageOptimizer) для расчёта метрики  $g_v$ .

Основной задачей практической части являлось создание инфраструктуры для вычисления функции улучшения распределения, определенной в алгоритме Кернигана-Лина:

$$g_v = \sum_{\substack{(v,u) \in E \\ P[v] \neq P[u]}} w(v,u) - \sum_{\substack{(v,u) \in E \\ P[v] = P[u]}} w(v,u)$$

#### 1.3.1 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.

### 1.3.2 Классы для представления графовых структур (`graph.hpp`)

#### Класс `NodeKey`

Класс `NodeKey` представляет собой обёртку для ключа вершины графа. Он обеспечивает типобезопасность и возможность использования различных типов данных в качестве ключей (целые числа, строки и т.д.).

Листинг 1: Класс `NodeKey` в `graph.hpp`

---

```

1 template <typename KeyType>
2 class NodeKey {
3 public:
4 KeyType key_value;
5
6 NodeKey(): key_value(KeyType()) {};
7 NodeKey(const KeyType& key): key_value(key) {};
8
9 // Оператор присваивания
10 NodeKey<KeyType>& operator=(const NodeKey<KeyType>& other) {
11     if (this != &other) {
12         key_value = other.key_value;
13     }
14     return *this;
15 };
16
17 // Дружественные операторы сравнения
18 friend bool operator<(const NodeKey<KeyType>& lhs, const
    NodeKey<KeyType>& rhs) {
19     return lhs.key_value < rhs.key_value;
20 }
21
22 friend bool operator>(const NodeKey<KeyType>& lhs, const
    NodeKey<KeyType>& rhs) {
```

```

23     return rhs < lhs;
24 }
25
26 friend bool operator==(const NodeKey<KeyType>& lhs, const
    NodeKey<KeyType>& rhs) {
27     return lhs.key_value == rhs.key_value;
28 }
29
30 friend bool operator!=(const NodeKey<KeyType>& lhs, const
    NodeKey<KeyType>& rhs) {
31     return !(lhs == rhs);
32 }
33 };

```

---

Класс `NodeKey` является шаблонным, что позволяет использовать различные типы данных в качестве ключей вершин. Это важно для обеспечения гибкости при работе с различными типами графовых данных. Известно, что в конечной реализации используются строковые ключи, но была добавлена гибкость для потенциального использования пользовательских гибридных ключей.

### Класс `Edge`

Класс `Edge` представляет ребро графа с весом и дополнительными параметрами.

Листинг 2: Класс `Edge` в `graph.hpp`

---

```

1 template <typename KeyType>
2 class Edge {
3 private:
4     float weight;
5     bool directional;
6     std::map<std::string, Parameter> parameters;
7     NodeKey<KeyType> from;
8     NodeKey<KeyType> to;
9 public:
10    float get_weight() const {
11        return weight;
12    }
13    bool is_directional() const {
14        return directional;
15    }
16    const NodeKey<KeyType>& get_from() const {
17        return from;
18    }
19    const NodeKey<KeyType>& get_to() const {
20        return to;
21    }

```



```

22
23     NodeKey<KeyType> get_other(const NodeKey<KeyType>& node) const {
24         if (node == to) {
25             return from;
26         } else if (node == from) {
27             return to;
28         }
29
30         return node;
31     }
32
33     NodeKey<KeyType> get_other(NodeKey<KeyType>* node) const {
34         if (*node == to) {
35             return from;
36         } else if (*node == from) {
37             return to;
38         } else {
39             return *node;
40         }
41     }
42 };

```

---

## Класс Node

Класс Node представляет вершину графа и содержит всю информацию о её связях с другими вершинами.

Листинг 3: Класс Node в graph.hpp (часть 1)

```

1 template <typename KeyType>
2 class Node {
3 private:
4     NodeKey<KeyType> key;
5 public:
6     std::map<std::string, Parameter> parameters;
7     std::map<NodeKey<KeyType>, Edge<KeyType>> edges;
8 }
9     NodeKey<KeyType> get_key() const {
10         return key;
11     }
12
13     void add_edge(Edge<KeyType> new_edge) {
14         NodeKey<KeyType> current_key = this->get_key();
15         edges[new_edge.get_other(current_key)] = new_edge;
16     }
17

```

```

18     void add_edges(std::map<NodeKey<KeyType>, Edge<KeyType>>
new_edges) {
19         edges.merge(new_edges);
20     }
21
22     bool remove_edge_to(NodeKey<KeyType> neighbour_key) {
23         return edges.erase(neighbour_key);
24     }
25
26     bool remove_edge(Edge<KeyType> removing_edge) {
27         NodeKey<KeyType> current_key = this->get_key();
28         return edges.erase(removing_edge.get_other(current_key));
29     }
30 };

```

---

Класс имеет несколько конструкторов для удобного создания вершин с различными конфигурациями связей. Все рёбра хранятся в вершине, от ссылок было решено отказаться, так как в конечной реализации хранилища должны находиться на разных машинах, а в вершины периодически перемещаться между ними.

#### 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 4: 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 NodeKey<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,
std::set<Edge<KeyType>> edges) = 0;
11    virtual void announce_remove(NodeKey<KeyType> key, int

```

```

    storage_id) = 0;
12    // запрашивает у source вершины, соседствующие с target
13    virtual std::set<Node<KeyType>> ask_neighbours_to_storage(int
    source, int target) = 0;
14    // запрашивает у source рёбра, идущие в target
15    virtual std::set<Edge<KeyType>> ask_edges_to_storage(int source,
    int target) = 0;
16 };

```

---

### Класс шины SimpleBus (bus.hpp)

Простая синхронная однопоточная реализация методов IBus. Релевантный код слишком длинный для вставки в основную часть, поэтому представлен в приложении А.

### Класс хранилища (storage.hpp)

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

#### 1.3.3 Структура класса Storage

Листинг 5: Базовая структура класса Storage

---

```

1 template <typename KeyType>
2 class Storage {
3 private:
4     typedef Node<KeyType> StorageNode;
5     typedef NodeKey<KeyType> Key;
6     int storage_id;
7     std::map<Key, StorageNode> nodes;
8     // external_edges[storage edge go to][external node][local node] = edge
9     std::map<int, std::map<Key, std::map<Key, Edge<KeyType>>>>
        external_edges;
10 IBus<KeyType> *bus = nullptr;
11
12 public:
13 Storage(int id)
14     : storage_id(id) {}
15 Storage(int id, std::map<Key, StorageNode> _nodes)
16     : storage_id(id), nodes(_nodes) {}
17
18 int get_id() const { return storage_id; };
19 void connect_to_bus(IBus<KeyType>* _bus) { bus = _bus; };

```

---

Хранилище идентифицируется уникальным `storage_id` и содержит вершины в виде хэш-таблицы для обеспечения быстрого доступа по ключу.

### Добавление вершин с автоматическим созданием связей

```

1 std::optional<std::set<Edge<KeyType>>> add_node(const StorageNode&
    node){
2     Key key = node.get_key();
3     std::cout << storage_id << " adding " << key.key_value <<
    std::endl;
4     typename std::map<Key, StorageNode>::iterator it =
    nodes.find(key);
5     if (it != nodes.end()) {
6         return std::nullopt;
7     }
8     nodes[key] = node;
9
10    std::set<Edge<KeyType>> external_edges_to_announce;
11    for (typename std::map<Key, Edge<KeyType>>::const_iterator
    edge_it = node.edges.begin(); edge_it != node.edges.end();
    ++edge_it) {
12        const Key& neighbor_key = edge_it->first;
13        const Edge<KeyType>& edge = edge_it->second;
14        std::cout << storage_id << " looks for " <<
    neighbor_key.key_value << std::endl;
15        // Ищем соседа в текущем хранилище
16        typename std::map<Key, StorageNode>::iterator it2 =
    nodes.find(neighbor_key);
17        if (it2 != nodes.end()) {
18            std::cout << storage_id << " node " <<
    neighbor_key.key_value << " is inside, no asking" << std::endl;
19            // Сосед найден в этом же хранилище - добавляем обратное ребро
20            it2->second.add_edge(edge);
21        } else {
22            std::cout << storage_id << " node " <<
    neighbor_key.key_value << " is outside, asking" << std::endl;
23            // Сосед не найден - спрашиваем у шины, где он находится
24            int neighbours_storage_id = bus->ask_who_has(storage_id,
    neighbor_key);
25            std::cout << storage_id << " asked for " <<
    neighbor_key.key_value << " answer: " << neighbours_storage_id <<
    std::endl;
26            if (neighbours_storage_id != -1) {
27                // Сохраняем внешнее ребро
28
29            external_edges[neighbours_storage_id][neighbor_key][key] = edge;
    external_edges_to_announce.insert(edge);

```

```

30         }
31         // Если сосед нигде не найден - игнорируем (возможно, будет добавлен
    позже)
32     }
33 }
34
35     return external_edges_to_announce;
36 };
37
38 bool add_node_and_announce(const StorageNode& node) {
39     if (bus == nullptr) {
40         return false;
41     };
42     std::optional<std::set<Edge<KeyType>>> external_edges =
    add_node(node);
43     if (!external_edges.has_value()) {
44         return false;
45     }
46     bus->announce_add(node.get_key(), storage_id,
    external_edges.value());
47     return true;
48 };

```

---

Метод `add_node` не только добавляет вершину в хранилище, но и автоматически создает связи с её соседями, что обеспечивает целостность графовой структуры. Это требуется для будущей реализации потокового распределения. Также метод `add_node_and_announce` позволяет при добавлении объявить о добавлении новой вершины.

### Удаление вершин с автоматическим удалением связей

Листинг 7: Метод удаления вершин класса `Storage`

---

```

1 bool remove_node(const Key& key) {
2     if (!has_node(key)) {
3         return false;
4     };
5     StorageNode node = nodes.find(key)->second;
6     nodes.erase(key);
7
8     for (typename std::map<NodeKey<KeyType>,
    Edge<KeyType>>::iterator edge_it = node.edges.begin(); edge_it !=
    node.edges.end(); ++edge_it) {
9         Key other_key = edge_it->second.get_other(key);
10        typename std::map<Key, StorageNode>::iterator it =
    nodes.find(other_key);
11        if (it != nodes.end()) {

```

```

12         it->second.remove_edge_to(key);
13     }
14 }
15
16     return true;
17 };
18
19 bool remove_node_and_announce(const Key& key) {
20     if (remove_node(key)) {
21         bus->announce_remove(key, storage_id);
22         return true;
23     }
24     return false;
25 };

```

---

Методы `remove_node` и `remove_node_and_announce` аналогично их `add` версиям автоматически следят за целостностью графа как локально, так и внешне. Это значительно снижает вычислительную сложность, так как исключает из рассмотрения вершины, не имеющие внешних связей.

### Методы для работы с граничными вершинами

Для реализации граничного алгоритма Кернигана-Лина требуется уметь получать вершины, граничащие с другим хранилищем, чем занимается метод `get_nodes_with_neighbors` используя карту внешних соседей.

---

Листинг 8: Метод для получения граничных вершин

---

```

1     std::set<StorageNode> result;
2
3     typename std::map<int, std::map<Key, std::map<Key,
4         Edge<KeyType>>>>::const_iterator storage_it =
5         external_edges.find(target_storage_id);
6     if (storage_it == external_edges.end()) {
7         return result;
8     }
9
10    const std::map<Key, std::map<Key, Edge<KeyType>>>& node_edges =
11        storage_it->second;
12
13    for (typename std::map<Key, std::map<Key,
14        Edge<KeyType>>>::const_iterator node_edges_it =
15        node_edges.begin(); node_edges_it != node_edges.end();
16        ++node_edges_it) {
17        const std::map<Key, Edge<KeyType>>& local_node_edges =
18            node_edges_it->second;
19        for (typename std::map<Key, Edge<KeyType>>::const_iterator

```

```

    local_node_edges_it = local_node_edges.begin();
    local_node_edges_it != local_node_edges.end();
    ++local_node_edges_it) {
12         Key local_key = local_node_edges_it->first;
13         typename std::map<Key, StorageNode>::const_iterator
    node_it = nodes.find(local_key);
14         if (node_it != nodes.end()) {
15             result.insert(node_it->second);
16         }
17     }
18 }
19
20 return result;

```

---

В листниге 9 представлен метод получения рёбер между хранилищами, что позволяет несколько упростить дальнейшие вычисления.

Листинг 9: Метод получения рёбер в другое хранилище

---

```

1 std::set<Edge<KeyType>> get_all_edges_to_storage(int
    target_storage_id) const {
2     std::set<Edge<KeyType>> result;
3     typename std::map<int, std::map<Key, std::map<Key,
    Edge<KeyType>>>>::const_iterator storage_it =
    external_edges.find(target_storage_id);
4     if (storage_it == external_edges.end()) {
5         return std::set<Edge<KeyType>>();
6     } else {
7         const std::map<Key, std::map<Key, Edge<KeyType>>>&
    external_nodes_map = storage_it->second;
8         for (typename std::map<Key, std::map<Key,
    Edge<KeyType>>>>::const_iterator node_edges_it =
    external_nodes_map.begin(); node_edges_it !=
    external_nodes_map.end(); ++node_edges_it) {
9             const std::map<Key, Edge<KeyType>>& node_edge_map =
    node_edges_it->second;
10            for (typename std::map<Key,
    Edge<KeyType>>>::const_iterator edges_it = node_edge_map.begin();
    edges_it != node_edge_map.end(); ++edges_it) {
11                result.insert(edges_it->second);
12            }
13        }
14    }
15    return result;
16 }

```

---

### 1.3.4 Класс оптимизатора (optimizer.hpp)

Класс `ExternalStorageOptimizer` является центральным компонентом реализации алгоритма Кернигана-Лина и отвечает за расчёт метрики улучшения  $g_v$ .

#### Структура класса оптимизатора

Листинг 10: Класс `StorageOptimizer`

---

```
1 template <typename KeyType>
2 class ExternalStorageOptimizer {
3 private:
4     IBus<KeyType>* bus;
5 public:
6     ExternalStorageOptimizer(IBus<KeyType>* _bus)
7         : bus(_bus) {}
8 }
```

---

#### Основной метод расчёта метрик

Метод `calculate_gvs()` демонстрирует практическую реализацию итерации Boundary KL (алгоритм работы только с граничными вершинами). Он получает граничные вершины из обоих хранилищ и вычисляет для каждой из них метрику улучшения  $g_v$ , а метод `calculate_gv` занимается непосредственно расчётом метрики.

Листинг 11: Методы расчёта  $g_v$

---

```
1 private:
2 float calculate_gv(const Node<KeyType>& node,
3     std::set<Edge<KeyType>> boundary_edges) const {
4     NodeKey<KeyType> this_key = node.get_key();
5     float internal_edges_weight = 0;
6     float external_edges_weight = 0;
7
8     std::map<NodeKey<KeyType>, Edge<KeyType>> boundary_edges_map;
9     for (typename std::set<Edge<KeyType>>::const_iterator edge_it =
10         boundary_edges.begin();
11         edge_it != boundary_edges.end(); ++edge_it) {
12         NodeKey<KeyType> other = edge_it->get_other(this_key);
13         if (!(other == this_key)) {
14             boundary_edges_map[other] = *edge_it;
15         }
16     }
17
18     for (typename std::map<NodeKey<KeyType>,
19         Edge<KeyType>>::const_iterator edge_it = node.edges.begin();
20         edge_it != node.edges.end(); ++edge_it) {
21         const NodeKey<KeyType>& neighbor_key = edge_it->first;
```



```

18         const Edge<KeyType>& edge = edge_it->second;
19         if (boundary_edges_map.find(neighbor_key) !=
boundary_edges_map.end()) {
20             external_edges_weight += edge.get_weight();
21         } else {
22             internal_edges_weight += edge.get_weight();
23         }
24     }
25
26     return internal_edges_weight - external_edges_weight;
27 }
28
29 public:
30 std::map<int, std::map<Node<KeyType>, float>> calculate_gvs(int
storage1, int storage2) const {
31     std::map<int, std::map<Node<KeyType>, float>> result;
32     // Получаем граничные вершины для обоих хранилищ
33     std::set<Node<KeyType>> boundary_nodes1 =
bus->ask_neighbours_to_storage(storage1, storage2);
34     std::set<Node<KeyType>> boundary_nodes2 =
bus->ask_neighbours_to_storage(storage2, storage1);
35
36     std::set<Edge<KeyType>> boundary_edges1 =
bus->ask_edges_to_storage(storage1, storage2);
37     std::set<Edge<KeyType>> boundary_edges2 =
bus->ask_edges_to_storage(storage2, storage1);
38
39     result[storage1] = std::map<Node<KeyType>, float>();
40     typename std::set<Node<KeyType>>::const_iterator it;
41     for (it = boundary_nodes1.begin(); it != boundary_nodes1.end();
++it) {
42         const Node<KeyType>& node = *it;
43         result[storage1][node.get_key()] = calculate_gv(node,
boundary_edges1);
44     }
45
46     result[storage2] = std::map<Node<KeyType>, float>();
47     for (it = boundary_nodes2.begin(); it != boundary_nodes2.end();
++it) {
48         const Node<KeyType>& node = *it;
49         result[storage2][node.get_key()] = calculate_gv(node,
boundary_edges2);
50     }

```

```
51     return result;
52 };
```

---

### Метод оптимизации

Метод `optimize()` демонстрирует практическую реализацию оптимизации KL. Он итеративно пользуется алгоритмом Кернигана-Лина, получает вершины с негативным  $g_v$ , после чего перемещает их в другое хранилище.

Листинг 12: Метод оптимизации

---

```
1 void optimize(int storage1, int storage2, int iterations_limit = 5) {
2     if (iterations_limit == 0) return;
3
4     int iteration = 0;
5     std::map<int, std::set<Node<KeyType>>> negative_gvs =
        get_negative_gvs(calculate_gvs(storage1, storage2));
6     do {
7         typename std::map<int, std::set<Node<KeyType>>>::iterator
            map_it;
8         for (map_it = negative_gvs.begin(); map_it !=
            negative_gvs.end(); ++map_it) {
9             int this_storage = map_it->first;
10            int other_storage = this_storage == storage1 ? storage2
                : storage1;
11            std::set<Node<KeyType>>& nodes = map_it->second;
12            typename std::set<Node<KeyType>>::const_iterator set_it;
13            for (set_it = nodes.begin(); set_it != nodes.end();
                ++set_it) {
14                const Node<KeyType>& node = *set_it;
15                Node<int> removed =
                    bus->request_node(node.get_key());
16                bus->send_remove_node(removed.get_key(),
                    this_storage);
17                bus->send_add_node(removed, other_storage);
18            }
19        }
20        ++iteration;
21        negative_gvs = get_negative_gvs(calculate_gvs(storage1,
            storage2));
22    } while (iteration < iterations_limit && !negative_gvs.empty());
23 }
```

---

## 1.4 Тестирование

Для тестирования создаётся простой граф показанный на рисунке 2

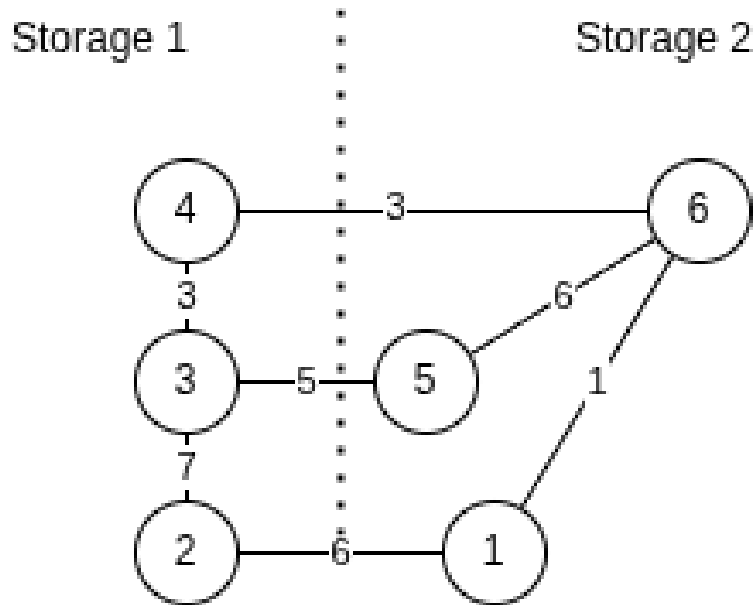


Рис. 2: Граф для тестирования

После оптимизации должно оказаться, что у всех вершин  $g_v > 0$

Листинг 13: Вывод тестовой программы

```
storage: 1 node: Node(key: 1) gv: 5
storage: 1 node: Node(key: 3) gv: 0
storage: 2 node: Node(key: 4) gv: 1
storage: 2 node: Node(key: 5) gv: 1
storage: 2 node: Node(key: 6) gv: 8
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: 3, edges: [Edge(from: 2, to: 3, weight: 7), Edge(from: 3, to: 4,
    ↪ weight: 2), Edge(from: 3, to: 5, weight: 5)])
}, external_edges: {
  to storage 2: {
    external node 4: [
      local node 3 -> Edge(from: 3, to: 4, weight: 2)
    ],
    external node 5: [
      local node 3 -> Edge(from: 3, to: 5, weight: 5)
    ],
    external node 6: [
      local node 1 -> Edge(from: 1, to: 6, weight: 1)
```

```

    ]
  }
})
Storage(id: 2, nodes: 3 {
  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)]),
  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 1: {
    external node 1: [
      local node 6 -> Edge(from: 1, to: 6, weight: 1)
    ],
    external node 3: [
      local node 4 -> Edge(from: 3, to: 4, weight: 2),
      local node 5 -> Edge(from: 3, to: 5, weight: 5)
    ]
  }
})

```

---

В результате получаем следующий граф:

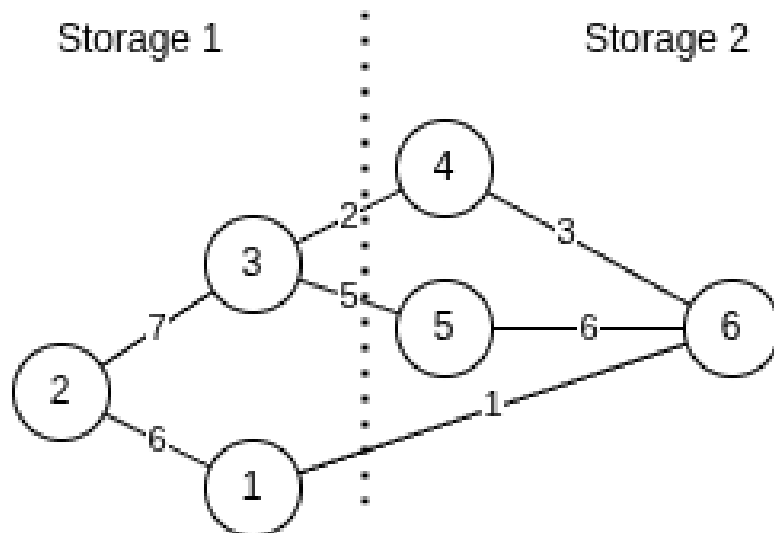


Рис. 3: Оптимизированный граф

Можно убедиться, что  $g_v$  действительно неотрицателен у всех и расчёты программы верны, граф успешно оптимизирован.

## Заключение

В ходе выполнения курсового проекта была успешно исследована задача распределения вершин графа по гомогенным хранилищам и реализован ключевой компонент алгоритма Кернигана-Лина для оптимизации такого распределения.

На теоретическом уровне проведён анализ современных подходов к разбиению графов, включая как потоковые методы (Fennel, Streaming Graph Partitioning), так и методы оптимизации распределения (библиотека METIS). Установлено, что алгоритм Кернигана-Лина, несмотря на свою классическую природу, остаётся эффективным инструментом для уточнения разбиения графов, особенно в его оптимизированной граничной версии (BKL), которая значительно снижает вычислительные затраты.

На практическом уровне реализована вычислительная модель для расчёта метрики улучшения распределения  $g_v$ , являющейся основой алгоритма Кернигана-Лина. Разработаны:

- Гибкая система представления графовых структур с поддержкой различных типов ключей вершин
- Классы для управления вершинами и их связями в рамках отдельных хранилищ
- Оптимизатор, вычисляющий метрику  $g_v$  только для граничных вершин (реализация подхода Boundary KL)
- Демонстрационный пример, подтверждающий корректность работы реализованных компонентов

Ключевым достижением работы является реализация оптимизации Boundary KL, позволяющей работать только с вершинами, имеющими внешние связи, что значительно снижает вычислительную сложность алгоритма при сохранении качества оптимизации.

Полученные результаты подтверждают эффективность комбинированного подхода, при котором потоковые методы используются для начального распределения вершин, а алгоритм Кернигана-Лина — для последующей оптимизации. Такая стратегия позволяет достичь оптимального баланса между скоростью распределения и качеством разбиения графа, что особенно важно для распределённых графовых баз данных, работающих с большими объёмами данных в реальном времени.

Реализованная система может быть расширена для поддержки большего количества хранилищ, добавления полного цикла итераций алгоритма Кернигана-Лина и интеграции с реальными системами управления графовыми базами данных.

## **Список использованных источников**

- [1] Garey Michael R., Johnson David S., Stockmeyer Larry. Some simplified NP-complete problems. 1974. С. 47–63.
- [2] Fennel: Streaming Graph Partitioning for Massive Scale Graphs / Charalampos Tsourakakis, Christos Gkantsidis, Bozidar Radunovic [и др.]. 2014. С. 333–342.

# Приложение А

Полные листинг кода реализации шины.

Листинг 14: bus.hpp

---

```
1 #ifndef VKR_BUS
2 #define VKR_BUS
3
4 #include "interface_bus.hpp"
5 #include "storage.hpp"
6 #include <map>
7 #include <set>
8
9 template <typename KeyType>
10 class SimpleBus : public IBus<KeyType> {
11 private:
12     std::map<int, Storage<KeyType*>*> storages;
13 public:
14     SimpleBus(): storages() {};
15
16     int connect_storage(Storage<KeyType*>* storage) {
17         if (storages.find(storage->get_id()) != storages.end()) {
18             return -1;
19         }
20         storages[storage->get_id()] = storage;
21         storage->connect_to_bus(this);
22         return storage->get_id();
23     };
24
25     Node<KeyType> request_node(const NodeKey<KeyType>& node) override {
26         for (typename std::map<int, Storage<KeyType*>*>::iterator it =
27             storages.begin(); it != storages.end(); ++it) {
28             Node<KeyType*>* node_pointer = it->second->get_node(node);
29             if (node_pointer != nullptr) {
30                 return Node<KeyType*>(*node_pointer);
31                 break;
32             };
33         }
34         return Node<KeyType*>();
35
36     int send_add_node(const Node<KeyType*>& node) override {
37         (void)node;
38         // No autosending
```

```

39     return -1;
40 };
41
42 bool send_add_node(const Node<KeyType>& node, int storage_id)
    override {
43     if (storages.find(storage_id) == storages.end()) {
44         return false;
45     }
46     return storages[storage_id]→add_node_and_announce(node);
47 };
48
49 bool send_remove_node(const NodeKey<KeyType>& node) override {
50     bool success = false;
51     for (typename std::map<int, Storage<KeyType>*>::iterator it =
        storages.begin(); it != storages.end(); ++it) {
52         if (it→second→remove_node_and_announce(node)) {
53             success = true;
54             break;
55         };
56     }
57     return success;
58 };
59
60 bool send_remove_node(const NodeKey<KeyType>& node, int storage_id)
    override {
61     if (storages.find(storage_id) == storages.end()) {
62         return false;
63     }
64     return storages[storage_id]→remove_node_and_announce(node);
65 };
66
67 int ask_who_has(int asker_id, NodeKey<KeyType> key) override{
68     for (typename std::map<int, Storage<KeyType>*>::iterator it =
        storages.begin(); it != storages.end(); ++it) {
69         if (it→second != nullptr && it→second→has_node(key)) {
70             return it→first;
71         };
72     }
73     return -1;
74 };
75
76 void announce_add(NodeKey<KeyType> key, int storage_id,
    std::set<Edge<KeyType>> edges) override {

```



```

77     for (typename std::map<int, Storage<KeyType>*>::iterator it =
storages.begin(); it != storages.end(); ++it) {
78         if (it->second != nullptr) {
79             it->second->get_add_announcement(key, storage_id, edges);
80         };
81     }
82 };
83
84 void announce_remove(NodeKey<KeyType> key, int storage_id) override {
85     for (typename std::map<int, Storage<KeyType>*>::iterator it =
storages.begin(); it != storages.end(); ++it) {
86         if (it->second != nullptr) {
87             it->second->get_remove_announcement(key, storage_id);
88         };
89     }
90 };
91
92 // запрашивает у source вершины, соседствующие с target
93 std::set<Node<KeyType>> ask_neighbours_to_storage(int source, int
target) {
94     if (storages.find(source) == storages.end()) {
95         return std::set<Node<KeyType>>();
96     }
97     return
storages[source]->get_nodes_with_neighbors_in_storage(target);
98 }
99
100 // запрашивает у source рёбра, идущие в target
101 std::set<Edge<KeyType>> ask_edges_to_storage(int source, int target)
{
102     if (storages.find(source) == storages.end()) {
103         return std::set<Edge<KeyType>>();
104     }
105     return storages[source]->get_all_edges_to_storage(target);
106 }
107
108 };
109
110 #endif // VKR_BUS

```

---