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

Содержание

Введение **4**
1 Теоретическая часть **5**
1.1 Постановка задачи распределения графа 5
1.2 Методы распределения вершин 5
1.3 Многоуровневая парадигма и роль алгоритма KL 6
1.4 Алгоритм Кернигана-Лина 6
1.4.1 Граничная модификация (BKL) 10
2 Практическая часть **11**
2.1 Структура проекта и основные зависимости 11
2.2 Классы для представления графовых структур (graph.hpp) 11
2.2.1 Класс NodeKey 11
2.2.2 Класс Edge 12
2.2.3 Класс Node 13
2.3 Класс хранилища (storage.hpp) 15
2.3.1 Структура класса Storage 15
2.3.2 Добавление вершин с автоматическим созданием связей 15
2.3.3 Методы для работы с граничными вершинами 16
2.3.4 Методы для расчёта суммарных весов рёбер 18
2.4 Класс оптимизатора (optimizer.hpp) 19
2.4.1 Структура класса оптимизатора 19
2.4.2 Основной метод расчёта метрик 20
2.5 Тестирование 21
Заключение **22**
Приложение А **23**

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 Теоретическая 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 $P = \{S_1, S_2, \ldots, S_k\}$, 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 $P^* = \{S_1^*, S_2^*, \ldots, S_k^*\}$, которое:

1. Минимизирует общую мощность разрезов:

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

   или относительную величину:

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

2. Минимизирует нормализованную максимальную нагрузку (максимизирует балансировку):

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

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

1.2 Методы распределения вершин

В рамках исследования рассматривались две принципиально различные категории методов решения задачи распределения:

1. **Методы потокового распределения (online partitioning)**: Методы, распределяющие вершину исходя из данных только об уже распределённых вершинах. Эти методы могут быть использованы при изначальном наполнении базы данных.

2. **Методы оптимизации распределения (offline partitioning)**: Методы распределения графа целиком, которые могут применяться для перераспределения графа по шардам, оптимизируя распределение, полученное методами первой категории.

Алгоритм Кернигана-Лина относится ко второй категории и является методом *уточнения* (refinement) разбиения. Как отмечается в исследовании, методы второй категории более точны и дают лучшие результаты за счёт знания всей структуры графа, но требуют больше вычислительных ресурсов.

В контексте библиотеки METIS (MEtis Tournament Inspired Strategy), которая рассматривается в работе как промышленный стандарт для задач разбиения графов, алгоритм Кернигана-Лина используется на этапе уточнения разбиения в рамках многоуровневой парадигмы.

1.3 Многоуровневая парадигма и роль алгоритма KL

Многоуровневая парадигма, лежащая в основе METIS, состоит из трёх этапов:

1. **Свёртка (Coarsening)**: Последовательное уменьшение графа путём схлопывания вершин.
2. **Распределение (Partitioning)**: Нахождение начального разбиения для сильно свёрнутого графа.
3. **Развёртка и уточнение (Uncoarsening and Refinement)**: Последовательное восстановление исходного графа с одновременным улучшением разбиения.

На третьем этапе алгоритм Кернигана-Лина играет ключевую роль.

1.4 Алгоритм Кернигана-Лина

Алгоритм Кернигана-Лина (Kernighan-Lin, KL) — это эвристический алгоритм для задачи разбиения графов на две равные по размеру части с минимальным весом разреза.

Постановка задачи

Пусть дан неориентированный взвешенный граф $G = (V, E)$ с весовой функцией $w : E \to R^+$. Требуется разбить множество вершин V на два непересекающихся подмножества A и B таких, что:

- $|A| = |B| = n/2$ (предполагаем, что $n = |V|$ чётно)
- Вес разреза минимален: $\min \sum_{a \in A, b \in B, (a,b) \in E} w(a, b)$

Основные определения

- $P[v]$ — разбиение, содержащее вершину v
- Для вершины v определим **внешнюю стоимость**:

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

- **Внутреннюю стоимость**:

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

- **Функция улучшения** для вершины v (разность между внешней и внутренней стоимостью):

$$g_v = E_v - I_v = \sum_{\substack{(v,u)\in E \\ P[v]\neq P[u]}} w(v,u) - \sum_{\substack{(v,u)\in E \\ P[v]=P[u]}} w(v,u) \tag{4}$$

Положительное значение g_v означает, что перемещение вершины v в противоположное разбиение уменьшит вес разреза.

Algorithm 1 Алгоритм Кернигана-Лина

Require: Граф $G = (V, E)$ с весами рёбер, начальное разбиение (A, B)
Ensure: Улучшенное разбиение (A, B)

Инициализация: улучшение $\leftarrow$ true
while улучшение = true **do**
    $A' \leftarrow A$, $B' \leftarrow B$ ▷ Рабочие копии
    Рассчитать g_v для всех $v \in V$
    for $k \leftarrow 1$ to $n/2$ **do**
        Найти пару (a_k, b_k) с максимальным выигрышем:

$$g_k = \max_{\substack{a \in A' \\ b \in B'}} [g_a + g_b - 2w(a, b)]$$

где $w(a, b) = 0$, если $(a, b) \notin E$
        Пометить a_k и b_k как "заблокированные"(больше не участвуют в выборе на этой итерации)
        Обновить значения g_v для соседей a_k и b_k:
        for каждого соседа x вершины a_k **do**
            if x не заблокирован **then**
                $g_x \leftarrow g_x + 2w(a_k, x)$ если $P[x] = P[a_k]$, иначе $g_x \leftarrow g_x - 2w(a_k, x)$
            end if
        end for
        for каждого соседа y вершины b_k **do**
            if y не заблокирован **then**
                $g_y \leftarrow g_y + 2w(b_k, y)$ если $P[y] = P[b_k]$, иначе $g_y \leftarrow g_y - 2w(b_k, y)$
            end if
        end for
    end for
    Найти m, максимизирующий частичную сумму:

$$G_m = \max_{1 \le m \le n/2} \sum_{i=1}^{m} g_i$$

    if $G_m > 0$ **then**
        Произвести обмены первых m пар: $(a_1, b_1), \ldots, (a_m, b_m)$
        $A \leftarrow (A \setminus \{a_1, \ldots, a_m\}) \cup \{b_1, \ldots, b_m\}$
        $B \leftarrow (B \setminus \{b_1, \ldots, b_m\}) \cup \{a_1, \ldots, a_m\}$
    else
        улучшение $\leftarrow$ false ▷ Улучшение не найдено
    end if
end while
return (A, B)

Выигрыш от обмена пары вершин При обмене вершин $a \in A$ и $b \in B$:

$$\Delta g(a, b) = g_a + g_b - 2w(a, b)$$

где:

- $g_a + g_b$ — выигрыш от индивидуального перемещения вершин
- $-2w(a, b)$ — коррекция, так как ребро между a и b (если существует) изменит свой статус: если оно было внутренним, станет внешним, и наоборот

Обновление g_v для соседей После выбора пары (a_k, b_k) и их блокировки, для соседей этих вершин значения g обновляются:

- Для соседа x вершины a_k:

$$g_x \leftarrow \begin{cases} g_x + 2w(a_k, x), & \text{если } x \text{ находится в том же разбиении, что и } a_k \\ g_x - 2w(a_k, x), & \text{если } x \text{ находится в противоположном разбиении} \end{cases}$$

- Аналогично для соседей b_k

Сложность алгоритма
При наивной реализации сложность составляет $O(n^3)$:

- Внешний цикл: O(итераций) (Экспериментально было установлено, что хорошая связность достигается уже после 5-10 итераций)
- Внутренний цикл по k: $O(n/2)$
- Поиск оптимальной пары на каждом шаге: $O(n^2)$
- Обновление значений g: O(степень вершины)

С оптимизациями можно достичь сложности $O(n^2 \log n)$.
Пример работы
Рассмотрим простой граф из 6 вершин. Начальное разбиение: $A = \{1, 2, 3\}$, $B = \{4, 5, 6\}$.

1. Рассчитываем g_v для всех вершин
2. Находим пару с максимальным $\Delta g(a, b)$
3. Блокируем выбранные вершины, обновляем g для их соседей
4. Повторяем шаги 2-3, пока все вершины не будут заблокированы
5. Находим m, максимизирующее частичную сумму выигрышей
6. Если максимальная сумма положительна, производим обмены

1.4.1 Граничная модификация (BKL)

Особый интерес представляет граничная модификация алгоритма (Boundary KL - BKL), которая применяет KL только к "граничным"вершинам - тем, которые смежны с вершинами из другого шарда. Как отмечается в работе: "Так как итерации прерываются, большинство вычислений тратятся впустую, особенно в KL(1), поэтому предлагается применять KL только к граничным вершинам. Это позволяет сильно уменьшить вычислительные затраты."

Результаты эксперимента (Таблица 4) показывают, что BKL(1) демонстрирует лучшее соотношение время/качество:

- BCSSTK31: $\lambda = 8.08\%$, время выполнения $= 0.76$
- BCSSTK32: $\lambda = 7.31\%$, время выполнения $= 0.96$

2 Практическая часть

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

1. Классы для представления графовых структур (вершины, рёбра, ключи)
2. Класс хранилища (Storage) для управления вершинами и их связями
3. 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:

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

2.1 Структура проекта и основные зависимости

Языком разработки был выбран C++ в силу его низкоуровневости и скорости, а также с намереньем в дальнейшим 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
- `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 курсовой работы код. Полный код представлен в приложении А.

2.2 Классы для представления графовых структур (graph.hpp)

2.2.1 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

```
template <typename KeyType>
class NodeKey {
public:
    KeyType key_value;
    NodeKey(): key_value(KeyType()) {};
    NodeKey(const KeyType& key): key_value(key) {};

    bool operator<(const KeyType& other){
        return key_value < other;
    };

    bool operator>(const KeyType& other){
        return key_value > other;
    };

    bool operator==(const KeyType& other){
        return key_value == other;
    };

    NodeKey<KeyType>& operator=(const NodeKey<KeyType>& other) {
        key_value = other.key_value;
        return *this;
    };
};
```

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.

2.2.2 Класс Edge

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

Листинг 2: Класс Edge в graph.hpp

```
class Edge {
public:
    int weight;
    std::map<std::string, Parameter> parameters;

    Edge& operator=(const Edge& other){
        weight = other.weight;
        parameters = other.parameters;
        return *this;
```

```
10      };
11  };
```

2.2.3 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  public:
4      typedef std::pair<NodeKey<KeyType>,Edge> Neighbour;
5
6      Node(): key(NodeKey<KeyType>()) {};
7      Node(KeyType _key): key(NodeKey<KeyType>(_key)) {};
8      Node(NodeKey<KeyType> _key): key(_key) {};
9      Node(NodeKey<KeyType> _key, std::vector<Neighbour>
       _this_storage_neighbours):
10         key(_key),
       this_storage_neighbours(_this_storage_neighbours) {};
11     Node(NodeKey<KeyType> _key, std::vector<Neighbour>
       _this_storage_neighbours,
12          std::map<int,std::vector<Neighbour>>
       _other_storages_neighbours)
13         : key(_key),
14           this_storage_neighbours(_this_storage_neighbours),
15           other_storages_neighbours(_other_storages_neighbours)
       {};
16
17     // Конструктор копирования
18     Node(const Node& other)
19         : key(other.key),
20           parameters(other.parameters),
21           this_storage_neighbours(other.this_storage_neighbours),
22
       other_storages_neighbours(other.other_storages_neighbours) {}
```

Класс имеет несколько конструкторов для удобного создания вершин с различными конфигурациями связей. Внутренние связи хранятся в `this_storage_neighbours`, а внешние связи (в другие хранилища) – в `other_storages_neighbours`.

Листинг 4: Класс Node в graph.hpp (часть 2)

```
1      NodeKey<KeyType> key;
2      std::map<std::string, Parameter> parameters;
3
```

```
4     std::vector<Neighbour> this_storage_neighbours;
5     std::map<int, std::vector<Neighbour>>
      other_storages_neighbours;
6
7     void set_this_storage_neighbours(std::vector<Neighbour>
      neighbours) const {
8         this_storage_neighbours = neighbours;
9     };
10
11    void
      set_other_storages_neighbours(std::map<int, std::vector<Neighbour>>
      neighbours) const {
12        other_storages_neighbours = neighbours;
13    };
```

Структура данных организована так, чтобы эффективно разделять внутренние и внешние связи. Внешние связи организованы в виде отображения идентификатора хранилища на список соседей в этом хранилище.

Листинг 5: Методы класса Node для расчёта весов рёбер

```
1     // Получить сумму весов всех внутренних рёбер
2     int get_internal_edges_weight_sum() const {
3         int total_weight = 0;
4
5         typename std::vector<Neighbour>::const_iterator it;
6         for (it = this_storage_neighbours.begin();
7              it != this_storage_neighbours.end(); ++it) {
8             total_weight += it->second.weight;
9         }
10
11        return total_weight;
12    }
13
14    // Получить сумму весов внешних рёбер в конкретное хранилище
15    int get_external_edges_weight_sum_to_storage(int
      target_storage_id) const {
16        int total_weight = 0;
17
18        typename std::map<int,
      std::vector<Neighbour>>::const_iterator map_it;
19        map_it =
      other_storages_neighbours.find(target_storage_id);
20
21        if (map_it != other_storages_neighbours.end()) {
22            const std::vector<Neighbour>& edges = map_it->second;
23
```

```
24              typename std::vector<Neighbour>::const_iterator
    edge_it;
25              for (edge_it = edges.begin(); edge_it != edges.end();
    ++edge_it) {
26                  total_weight += edge_it->second.weight;
27              }
28          }
29
30          return total_weight;
31      }
```

Методы `get_internal_edges_weight_sum()` и `get_external_edges_weight_sum_to_storage()` являются ключевыми для реализации алгоритма Кернигана-Лина, так как они непосредственно вычисляют суммы весов рёбер, необходимые для расчёта метрики g_v.

2.3 Класс хранилища (storage.hpp)

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

2.3.1 Структура класса Storage

Листинг 6: Базовая структура класса Storage

```
1  template <typename KeyType>
2  class Storage {
3  private:
4      typedef Node<KeyType> StorageNode;
5      int storage_id;
6      std::unordered_map<KeyType, StorageNode> nodes;
7
8  public:
9      Storage(int id) : storage_id(id) {}
10     Storage(int id, std::unordered_map<KeyType, StorageNode>
       _nodes)
11         : storage_id(id), nodes(_nodes) {}
12
13     int get_id() const {
14         return storage_id;
15     }
```

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

2.3.2 Добавление вершин с автоматическим созданием связей

Листинг 7: Метод add_node класса Storage

```
1     bool add_node(const StorageNode& node) {
2         KeyType key = node.key.key_value;
3         typename std::unordered_map<KeyType,
    StorageNode>::iterator it =
4             nodes.find(key);
5         if (it != nodes.end()) {
6             return false;
7         }
8         nodes[key] = node;
9
10        // Проходим по всем указанным соседям
11        for (size_t i = 0; i <
    node.this_storage_neighbours.size(); ++i) {
12            KeyType neighbor_key =
    node.this_storage_neighbours[i].first.key_value;
13            Edge edge = node.this_storage_neighbours[i].second;
14
15            // Пропускаем петли (ребра к самому себе)
16            if (neighbor_key == key) {
17                continue;
18            }
19
20            typename StorageNode::Neighbour
    neighbor_to(NodeKey<KeyType>(key), edge);
21            if (has_node(neighbor_key)) {
22                StorageNode* neighbor_node =
    get_node(neighbor_key);
23
    neighbor_node->this_storage_neighbours.push_back(neighbor_to);
24            }
25        }
26        return true;
27    }
```

Метод `add_node` не только добавляет вершину в хранилище, но и автоматически создает обратные связи с её соседями, что обеспечивает целостность графовой структуры. Это требуется для потокового распределения.

2.3.3 Методы для работы с граничными вершинами

Листинг 8: Методы для получения граничных вершин

```
1     // Получить подграф узлов, имеющих соседей в указанном хранилище (копии)
2     std::vector<StorageNode>
    get_nodes_with_neighbors_in_storage_copy(
```

```
3           int target_storage_id) const {
4           std::vector<StorageNode> result;
5
6           typename std::unordered_map<KeyType,
    StorageNode>::const_iterator it;
7           for (it = nodes.begin(); it != nodes.end(); ++it) {
8               const StorageNode& node = it->second;
9
10              typename std::map<int, std::vector<typename
    StorageNode::Neighbour>>::const_iterator map_it;
11              map_it =
    node.other_storages_neighbours.find(target_storage_id);
12
13              if (map_it != node.other_storages_neighbours.end()) {
14                  if (!map_it->second.empty()) {
15                      result.push_back(node);
16                  }
17              }
18          }
19
20          return result;
21      }
22
23      // Получить подграф узлов, имеющих соседей в указанном хранилище (копии)
24      std::unordered_map<KeyType, StorageNode>
25      get_nodes_with_neighbors_in_storage_map_copy(int
    target_storage_id) const {
26          std::unordered_map<KeyType, StorageNode> result;
27
28          typename std::unordered_map<KeyType,
    StorageNode>::const_iterator it;
29          for (it = nodes.begin(); it != nodes.end(); ++it) {
30              const StorageNode& node = it->second;
31
32              typename std::map<int, std::vector<typename
    StorageNode::Neighbour>>::const_iterator map_it;
33              map_it =
    node.other_storages_neighbours.find(target_storage_id);
34
35              if (map_it != node.other_storages_neighbours.end()) {
36                  if (!map_it->second.empty()) {
37                      result.insert(*it);
38                  }
39              }
40          }
41
```

```
42          return result;
43      }
```

Эти методы реализуют важную оптимизацию алгоритма Кернигана-Лина – работу только с граничными вершинами (Boundary KL Refinement). Это значительно снижает вычислительную сложность, так как исключает из рассмотрения вершины, не имеющие внешних связей.

2.3.4 Методы для расчёта суммарных весов рёбер

Листинг 9: Методы для расчёта весов рёбер в хранилище

```
1       // Подсчет суммы весов всех внутренних ребер в хранилище
2       int get_internal_edges_weight_sum() const {
3           int total_weight = 0;
4           int duplicate_weight = 0;
5
6           typename std::unordered_map<KeyType,
    StorageNode>::const_iterator node_it;
7           for (node_it = nodes.begin(); node_it != nodes.end();
    ++node_it) {
8               const StorageNode& node = node_it->second;
9
10              typename std::vector<typename
    StorageNode::Neighbour>::const_iterator neighbour_it;
11              for (neighbour_it =
    node.this_storage_neighbours.begin();
12                   neighbour_it !=
    node.this_storage_neighbours.end();
13                   ++neighbour_it) {
14
15                  // пропускаем петли
16                  if (node.key.key_value ==
    neighbour_it->first.key_value) {
17                      continue;
18                  }
19
20                  total_weight += neighbour_it->second.weight;
21                  if (has_node(neighbour_it->first.key_value)) {
22                      duplicate_weight +=
    neighbour_it->second.weight;
23                  }
24              }
25          }
26
27          return total_weight - duplicate_weight / 2;
```

```
28      }
29
30      // Подсчет суммы весов ребер из текущего хранилища в целевое хранилище
31      int get_external_edges_weight_sum_to_storage(int
    target_storage_id) const {
32          int total_weight = 0;
33
34          typename std::unordered_map<KeyType,
    StorageNode>::const_iterator node_it;
35          for (node_it = nodes.begin(); node_it != nodes.end();
    ++node_it) {
36              const StorageNode& node = node_it->second;
37
38              typename std::map<int, std::vector<typename
    StorageNode::Neighbour>>::const_iterator map_it;
39              map_it =
    node.other_storages_neighbours.find(target_storage_id);
40
41              if (map_it != node.other_storages_neighbours.end()) {
42                  const std::vector<typename
    StorageNode::Neighbour>& edges = map_it->second;
43
44                  typename std::vector<typename
    StorageNode::Neighbour>::const_iterator edge_it;
45                  for (edge_it = edges.begin(); edge_it !=
    edges.end(); ++edge_it) {
46                      total_weight += edge_it->second.weight;
47                  }
48              }
49          }
50
51          return total_weight;
52      }
```

2.4 Класс оптимизатора (optimizer.hpp)

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

2.4.1 Структура класса оптимизатора

Листинг 10: Класс StorageOptimizer

```
1 template <typename KeyType>
2 class StorageOptimizer {
```

```
3 private:
4     const Storage<KeyType>& storage1;
5     const Storage<KeyType>& storage2;
6
7 public:
8     StorageOptimizer(const Storage<KeyType>& s1, const
    Storage<KeyType>& s2)
9         : storage1(s1), storage2(s2) {}
```

2.4.2 Основной метод расчёта метрик

Листинг 11: Методы расчёта g_v

```
1     int calculate_gv(const Node<KeyType>& node, int
    other_storage_id) const {
2         int internal_edges_weight =
    node.get_internal_edges_weight_sum();
3         int external_edges_weight =
    node.get_external_edges_weight_sum_to_storage(
4             other_storage_id);
5
6         return internal_edges_weight - external_edges_weight;
7     }
8     void calculate_gvs() const {
9         // Получаем граничные вершины для обоих хранилищ
10        std::unordered_map<KeyType, Node<KeyType>>
    boundary_nodes1 =
11
    storage1.get_nodes_with_neighbors_in_storage_map_copy(storage2.get_id());
12        std::unordered_map<KeyType, Node<KeyType>>
    boundary_nodes2 =
13
    storage2.get_nodes_with_neighbors_in_storage_map_copy(storage1.get_id());
14
15        typename std::unordered_map<KeyType,
    Node<KeyType>>::const_iterator it;
16        for (it = boundary_nodes1.begin(); it !=
    boundary_nodes1.end(); ++it) {
17            const Node<KeyType>& node = it->second;
18
19            std::cout << "Metric for node " << node.key.key_value
    << " is "
20                      << calculate_gv(node, storage2.get_id()) <<
    std::endl;
21        }
22
```

```
23          for (it = boundary_nodes2.begin(); it !=
    boundary_nodes2.end(); ++it) {
24              const Node<KeyType>& node = it->second;
25
26              std::cout << "Metric for node " << node.key.key_value
    << " is "
27                        << calculate_gv(node, storage1.get_id()) <<
    std::endl;
28          }
29      }
```

Метод `calculate_gvs()` демонстрирует практическую реализацию оптимизации 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 рассчётом метрики

2.5 Тестирование

Для тестирования создаётся простой граф показанный на рисунке 1

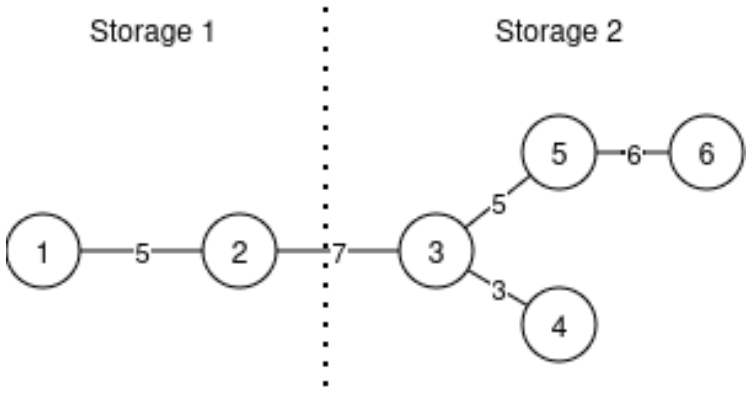

Рис. 1: Граф для тестирования

Модуль оптимизатора должен правильно посчитать метрики только для 2-х вершин: 2 и 3, так как они граничные. Их значения соответсвенно равны -2 и 1.

Листинг 12: Вывод тестовой программы

```
Metric for node 2 is -2
Metric for node 3 is 1
```

Результат верен, он показывает, что перемещение вершины 2 повысит связность, но по графу очевидно, что это внесёт только больший дисбаланс в количестве вершин между хранилищами, поэтому вершины и должны меняться местами.

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.

Приложение А

Полные листинги файлов программы.

Листинг 13: graph.hpp

```
#ifndef VKR_COURSE_GRAPH
#define VKR_COURSE_GRAPH

#include <stdlib.h>
#include <map>
#include <string>
#include <vector>
#include <numeric>
#include <utility>
#include <stdexcept>

template <typename KeyType>
class NodeKey {
    public:
        KeyType key_value;
        NodeKey(): key_value(KeyType()) {};
        NodeKey(const KeyType& key): key_value(key) {};

        bool operator<(const KeyType& other){
            return key_value < other;
        };

        bool operator>(const KeyType& other){
            return key_value > other;
        };

        bool operator==(const KeyType& other){
            return key_value == other;
        };

        NodeKey<KeyType>& operator=(const NodeKey<KeyType>&
    other) {
            key_value = other.key_value;
            return *this;
        };
};


class Parameter{
    private:
        std::vector<std::byte> data;
```

```
41      public:
42          void set_data(const std::vector<std::byte>& _data) {
43              data = _data;
44          }
45          std::vector<std::byte> get_data() {
46              return data;
47          };
48  };
49
50  class Edge {
51      public:
52          int weight;
53          std::map<std::string, Parameter> parameters;
54
55          Edge& operator=(const Edge& other){
56              weight = other.weight;
57              parameters = other.parameters;
58              return *this;
59          };
60  };
61
62  template <typename KeyType>
63  class Node {
64      public:
65          typedef std::pair<NodeKey<KeyType>,Edge> Neighbour;
66
67          Node(): key(NodeKey<KeyType>()) {};
68          Node(KeyType _key): key(NodeKey<KeyType>(_key)) {};
69          Node(NodeKey<KeyType> _key): key(_key) {};
70          Node(NodeKey<KeyType> _key, std::vector<Neighbour>
      _this_storage_neighbours): key(_key),
      this_storage_neighbours(_this_storage_neighbours) {};
71          Node(NodeKey<KeyType> _key, std::vector<Neighbour>
      _this_storage_neighbours, std::map<int,std::vector<Neighbour>>
      _other_storages_neighbours)
72              : key(_key),
73              this_storage_neighbours(_this_storage_neighbours),
74              other_storages_neighbours(_other_storages_neighbours)
      {};
75          // Конструктор копирования
76          Node(const Node& other)
77              : key(other.key),
78              parameters(other.parameters),
79
      this_storage_neighbours(other.this_storage_neighbours),
80
```

```
    other_storages_neighbours(other.other_storages_neighbours) {}
81
82          NodeKey<KeyType> key;
83          std::map<std::string, Parameter> parameters;
84
85          std::vector<Neighbour> this_storage_neighbours;
86          std::map<int,std::vector<Neighbour>>
    other_storages_neighbours;
87
88          void set_this_storage_neighbours(std::vector<Neighbour>
    neighbours) const {this_storage_neighbours = neighbours;};
89          void
    set_other_storages_neighbours(std::map<int,std::vector<Neighbour>>
    neighbours) const {other_storages_neighbours = neighbours;};
90
91          int get_edge_num_to_others() const {
92              return std::accumulate(
93                  other_storages_neighbours.begin(),
94                  other_storages_neighbours.end(),
95                  0,
96                  [](int sum, const auto& pair) {
97                      return sum + pair.second.size();
98                  }
99              );
100         };
101
102         // Получить сумму весов всех внутренних рёбер
103         int get_internal_edges_weight_sum() const {
104             int total_weight = 0;
105
106             typename std::vector<Neighbour>::const_iterator it;
107             for (it = this_storage_neighbours.begin(); it !=
    this_storage_neighbours.end(); ++it) {
108                 total_weight += it->second.weight;
109             }
110
111             return total_weight;
112         }
113
114         // Получить сумму весов внешних рёбер в конкретное хранилище
115         int get_external_edges_weight_sum_to_storage(int
    target_storage_id) const {
116             int total_weight = 0;
117
118             typename std::map<int,
    std::vector<Neighbour>>::const_iterator map_it;
```

```
119             map_it =
    other_storages_neighbours.find(target_storage_id);
120
121             if (map_it != other_storages_neighbours.end()) {
122                 const std::vector<Neighbour>& edges =
    map_it->second;
123
124                 typename std::vector<Neighbour>::const_iterator
    edge_it;
125                 for (edge_it = edges.begin(); edge_it !=
    edges.end(); ++edge_it) {
126                     total_weight += edge_it->second.weight;
127                 }
128             }
129
130             return total_weight;
131         }
132
133         friend void swap(Node& first, Node& second) noexcept {
134             using std::swap;
135
136             swap(first.parameters, second.parameters);
137             swap(first.this_storage_neighbours,
    second.this_storage_neighbours);
138             swap(first.other_storages_neighbours,
    second.other_storages_neighbours);
139         }
140
141         Node& operator=(const Node& other) {
142             if (this != &other) {
143                 key = other.key;
144                 parameters = other.parameters;
145                 this_storage_neighbours =
    other.this_storage_neighbours;
146                 other_storages_neighbours =
    other.other_storages_neighbours;
147             }
148             return *this;
149         }
150 };
151
152 #endif // VKR_COURSE_GRAPH
```

Листинг 14: storage.hpp

```
1 #ifndef VKR_COURSE_STORAGE
```

```
2 #define VKR_COURSE_STORAGE
3
4 #include "graph.hpp"
5 #include <unordered_map>
6 #include <unordered_set>
7 #include <vector>
8 #include <algorithm>
9 #include <iostream>
10
11 template <typename KeyType>
12 class Storage {
13 private:
14     typedef Node<KeyType> StorageNode;
15     int storage_id;
16     std::unordered_map<KeyType, StorageNode> nodes;
17
18 public:
19     Storage(int id) : storage_id(id) {}
20     Storage(int id, std::unordered_map<KeyType, StorageNode>
    _nodes) : storage_id(id), nodes(_nodes) {}
21
22     int get_id() const {
23         return storage_id;
24     }
25
26     // добавляет узел и создает двусторонние ребра с указанными соседями
27     bool add_node(const StorageNode& node) {
28         KeyType key = node.key.key_value;
29         typename std::unordered_map<KeyType,
    StorageNode>::iterator it = nodes.find(key);
30         if (it != nodes.end()) {
31             return false;
32         }
33         nodes[key] = node;
34
35         // Проходим по всем указанным соседям
36         for (size_t i = 0; i <
    node.this_storage_neighbours.size(); ++i) {
37             KeyType neighbor_key =
    node.this_storage_neighbours[i].first.key_value;
38             Edge edge = node.this_storage_neighbours[i].second;
39
40             // Пропускаем петли (ребра к самому себе)
41             if (neighbor_key == key) {
42                 continue;
43             }
```

```
44
45              typename StorageNode::Neighbour
    neighbor_to(NodeKey<KeyType>(key), edge);
46              if (has_node(neighbor_key)) {
47                  StorageNode* neighbor_node =
    get_node(neighbor_key);
48
    neighbor_node->this_storage_neighbours.push_back(neighbor_to);
49              }
50          }
51          return true;
52      }
53
54      // Самый простой remove_node: удаляет узел и все связанные с ним ребра
55      bool remove_node(const KeyType& key) {
56          if (!has_node(key)) {
57              return false;
58          }
59
60          // Получаем узел для удаления
61          Node<KeyType>* node_to_remove = get_node(key);
62          if (node_to_remove == nullptr) {
63              return false;
64          }
65
66          // Проходим по всем соседям удаляемого узла
67          for (size_t i = 0; i <
    node_to_remove->this_storage_neighbours.size(); ++i) {
68              KeyType neighbor_key =
    node_to_remove->this_storage_neighbours[i].first.key_value;
69
70              // Ищем и удаляем обратное ребро у соседа
71              Node<KeyType>* neighbor = get_node(neighbor_key);
72              if (neighbor != nullptr) {
73                  for (size_t j = 0; j <
    neighbor->this_storage_neighbours.size(); ++j) {
74                      if
    (neighbor->this_storage_neighbours[j].first.key_value == key) {
75
    neighbor->this_storage_neighbours.erase(neighbor->this_storage_neighbours.b
    + j);
76                          break;
77                      }
78                  }
79              }
80          }
```

```
81
82          // Удаляем сам узел из хранилища
83          nodes.erase(key);
84          return true;
85      }
86
87      StorageNode* get_node(const KeyType& key) {
88          typename std::unordered_map<KeyType,
    StorageNode>::iterator it = nodes.find(key);
89          if (it != nodes.end()) {
90              return &(it->second);
91          }
92          return nullptr;
93      }
94
95      bool has_node(const KeyType& key) const {
96          typename std::unordered_map<KeyType,
    StorageNode>::const_iterator it = nodes.find(key);
97          return it != nodes.end();
98      }
99
100     const std::unordered_map<KeyType, StorageNode>&
    get_all_nodes() const {
101         return nodes;
102     }
103
104     size_t size() const {
105         return nodes.size();
106     }
107
108     void clear() {
109         nodes.clear();
110     }
111
112     //
113     // Подсчёты сумм весов
114     //
115
116     // Подсчет суммы весов всех внутренних ребер в хранилище
117     int get_internal_edges_weight_sum() const {
118         // Потенциально будет использоваться с использованием только
    "граничных"вершин, поэтому нужно отдельно записывать дупликаты, так есть
    недублирующиеся, которые идут "вглубь"хранилища
119         int total_weight = 0;
120         int duplicate_weight = 0;
121
```

```
122         typename std::unordered_map<KeyType,
    StorageNode>::const_iterator node_it;
123         for (node_it = nodes.begin(); node_it != nodes.end();
    ++node_it) {
124             const StorageNode& node = node_it->second;
125 
126             typename std::vector<typename
    StorageNode::Neighbour>::const_iterator neighbour_it;
127             for (neighbour_it =
    node.this_storage_neighbours.begin();
128                 neighbour_it !=
    node.this_storage_neighbours.end();
129                 ++neighbour_it) {
130 
131                 // пропускаем петли
132                 if (node.key.key_value ==
    neighbour_it->first.key_value) {
133                     continue;
134                 }
135 
136                 std::cout << "Edge from " << node.key.key_value
    << " to " << neighbour_it->first.key_value << std::endl;
137 
138                 total_weight += neighbour_it->second.weight;
139                 if (has_node(neighbour_it->first.key_value)) {
140                     duplicate_weight +=
    neighbour_it->second.weight;
141                 }
142             }
143         }
144 
145         return total_weight - duplicate_weight / 2;
146     }
147 
148     // Подсчет суммы весов ребер из текущего хранилища в целевое хранилище
149     int get_external_edges_weight_sum_to_storage(int
    target_storage_id) const {
150         int total_weight = 0;
151 
152         typename std::unordered_map<KeyType,
    StorageNode>::const_iterator node_it;
153         for (node_it = nodes.begin(); node_it != nodes.end();
    ++node_it) {
154             const StorageNode& node = node_it->second;
155 
156             typename std::map<int, std::vector<typename
```

```
    StorageNode::Neighbour>>::const_iterator map_it;
157             map_it =
    node.other_storages_neighbours.find(target_storage_id);
158
159             if (map_it != node.other_storages_neighbours.end()) {
160                 const std::vector<typename
    StorageNode::Neighbour>& edges = map_it->second;
161
162                 typename std::vector<typename
    StorageNode::Neighbour>::const_iterator edge_it;
163                 for (edge_it = edges.begin(); edge_it !=
    edges.end(); ++edge_it) {
164                     total_weight += edge_it->second.weight;
165                 }
166             }
167         }
168
169         return total_weight;
170     }
171
172     //
173     // Получение наборов связанных с другим хранилищем
174     //
175
176     // Получить подграф узлов, имеющих соседей в указанном хранилище (копии)
177     std::vector<StorageNode>
    get_nodes_with_neighbors_in_storage_copy(int
    target_storage_id) const {
178         std::vector<StorageNode> result;
179
180         typename std::unordered_map<KeyType,
    StorageNode>::const_iterator it;
181         for (it = nodes.begin(); it != nodes.end(); ++it) {
182             const StorageNode& node = it->second;
183
184             typename std::map<int, std::vector<typename
    StorageNode::Neighbour>>::const_iterator map_it;
185             map_it =
    node.other_storages_neighbours.find(target_storage_id);
186
187             if (map_it != node.other_storages_neighbours.end()) {
188                 if (!map_it->second.empty()) {
189                     result.push_back(node);
190                 }
191             }
192         }
```

```
193
194         return result;
195     }
196
197     // Получить подграф узлов, имеющих соседей в указанном хранилище (копии)
198     std::unordered_map<KeyType, StorageNode>
    get_nodes_with_neighbors_in_storage_map_copy(int
    target_storage_id) const {
199         std::unordered_map<KeyType, StorageNode> result;
200
201         typename std::unordered_map<KeyType,
    StorageNode>::const_iterator it;
202         for (it = nodes.begin(); it != nodes.end(); ++it) {
203             const StorageNode& node = it->second;
204
205             typename std::map<int, std::vector<typename
    StorageNode::Neighbour>>::const_iterator map_it;
206             map_it =
    node.other_storages_neighbours.find(target_storage_id);
207
208             if (map_it != node.other_storages_neighbours.end()) {
209                 if (!map_it->second.empty()) {
210                     result.insert(*it);
211                 }
212             }
213         }
214
215         return result;
216     }
217
218     std::vector<typename StorageNode::Neighbour>
    get_all_edges_to_storage(int target_storage_id) const {
219         std::vector<typename StorageNode::Neighbour> all_edges;
220
221         typename std::unordered_map<KeyType,
    StorageNode>::const_iterator it;
222         for (it = nodes.begin(); it != nodes.end(); ++it) {
223             const StorageNode& node = it->second;
224
225             typename std::map<int, std::vector<typename
    StorageNode::Neighbour>>::const_iterator map_it;
226             map_it =
    node.other_storages_neighbours.find(target_storage_id);
227
228             if (map_it != node.other_storages_neighbours.end()) {
229                 const std::vector<typename
```

```
    StorageNode::Neighbour>& neighbors = map_it->second;
230                 all_edges.insert(all_edges.end(),
    neighbors.begin(), neighbors.end());
231             }
232         }
233
234         return all_edges;
235     }
236
237
238     int get_total_edges_to_storage(int target_storage_id) const {
239         int total = 0;
240
241         typename std::unordered_map<KeyType,
    StorageNode>::const_iterator it;
242         for (it = nodes.begin(); it != nodes.end(); ++it) {
243             const StorageNode& node = it->second;
244
245             typename std::map<int, std::vector<typename
    StorageNode::Neighbour>>::const_iterator map_it;
246             map_it =
    node.other_storages_neighbours.find(target_storage_id);
247
248             if (map_it != node.other_storages_neighbours.end()) {
249                 total += map_it->second.size();
250             }
251         }
252
253         return total;
254     }
255
256     //
257     // операции над вершинами
258     //
259
260     bool add_internal_edge(const KeyType& from_key,
261                            const KeyType& to_key,
262                            const Edge& edge) {
263         StorageNode* from_node = get_node(from_key);
264         StorageNode* to_node = get_node(to_key);
265
266         if (from_node == nullptr || to_node == nullptr) {
267             return false;
268         }
269
270         typename StorageNode::Neighbour
```

```
    neighbor(NodeKey<KeyType>(to_key), edge);
271         from_node->this_storage_neighbours.push_back(neighbor);
272         return true;
273     }
274 
275     bool add_external_edge(const KeyType& from_key,
276                            int target_storage_id,
277                            const KeyType& to_key,
278                            const Edge& edge) {
279         StorageNode* from_node = get_node(from_key);
280 
281         if (from_node == nullptr) {
282             return false;
283         }
284 
285         typename StorageNode::Neighbour
    neighbor(NodeKey<KeyType>(to_key), edge);
286 
    from_node->other_storages_neighbours[target_storage_id].push_back(neighbor)
287         return true;
288     }
289 
290     // Удалить внутреннее ребро
291     bool remove_internal_edge(const KeyType& from_key, const
    KeyType& to_key) {
292         StorageNode* from_node = get_node(from_key);
293         if (from_node == nullptr) {
294             return false;
295         }
296 
297         // Ищем ребро в this_storage_neighbours
298         typename std::vector<typename
    StorageNode::Neighbour>::iterator it;
299         for (it = from_node->this_storage_neighbours.begin();
300              it != from_node->this_storage_neighbours.end();
301              ++it) {
302             if (it->first.key_value == to_key) {
303                 from_node->this_storage_neighbours.erase(it);
304                 return true;
305             }
306         }
307 
308         return false;
309     }
310 
311     // Удалить внешнее ребро (конкретное ребро в конкретное хранилище)
```

```
312     bool remove_external_edge(const KeyType& from_key,
313                               int target_storage_id,
314                               const KeyType& to_key) {
315         StorageNode* from_node = get_node(from_key);
316         if (from_node == nullptr) {
317             return false;
318         }
319 
320         typename std::map<int, std::vector<typename
    StorageNode::Neighbour>>::iterator storage_it;
321         storage_it =
    from_node->other_storages_neighbours.find(target_storage_id);
322         if (storage_it ==
    from_node->other_storages_neighbours.end()) {
323             return false;
324         }
325 
326         std::vector<typename StorageNode::Neighbour>& neighbors =
    storage_it->second;
327         typename std::vector<typename
    StorageNode::Neighbour>::iterator it;
328         for (it = neighbors.begin(); it != neighbors.end(); ++it)
    {
329             if (it->first.key_value == to_key) {
330                 neighbors.erase(it);
331 
332                 // Если после удаления вектор стал пустым, удаляем запись из map
333                 if (neighbors.empty()) {
334 
    from_node->other_storages_neighbours.erase(storage_it);
335                 }
336 
337                 return true;
338             }
339         }
340 
341         return false;
342     }
343 
344     typename std::unordered_map<KeyType, StorageNode>::iterator
    begin() {
345         return nodes.begin();
346     }
347 
348     typename std::unordered_map<KeyType, StorageNode>::iterator
    end() {
```

```
349         return nodes.end();
350     }
351 
352     typename std::unordered_map<KeyType,
    StorageNode>::const_iterator begin() const {
353         return nodes.begin();
354     }
355 
356     typename std::unordered_map<KeyType,
    StorageNode>::const_iterator end() const {
357         return nodes.end();
358     }
359 
360     typename std::unordered_map<KeyType,
    StorageNode>::const_iterator cbegin() const {
361         return nodes.cbegin();
362     }
363 
364     typename std::unordered_map<KeyType,
    StorageNode>::const_iterator cend() const {
365         return nodes.cend();
366     }
367 
368     bool empty() const {
369         return nodes.empty();
370     }
371 
372     // Проверить, есть ли узел с рёбрами в указанное хранилище
373     bool has_node_with_edges_to_storage(const KeyType& node_key,
    int target_storage_id) const {
374         const StorageNode* node = get_node(node_key);
375         if (node == nullptr) {
376             return false;
377         }
378 
379         typename std::map<int, std::vector<typename
    StorageNode::Neighbour>>::const_iterator it;
380         it =
    node->other_storages_neighbours.find(target_storage_id);
381 
382         if (it != node->other_storages_neighbours.end()) {
383             return !it->second.empty();
384         }
385 
386         return false;
387     }
```

```
388
389     // Получить вес всех рёбер в указанное хранилище
390     int get_total_weight_to_storage(int target_storage_id) const {
391         int total_weight = 0;
392
393         typename std::unordered_map<KeyType,
    StorageNode>::const_iterator it;
394         for (it = nodes.begin(); it != nodes.end(); ++it) {
395             const StorageNode& node = it->second;
396
397             typename std::map<int, std::vector<typename
    StorageNode::Neighbour>>::const_iterator map_it;
398             map_it =
    node.other_storages_neighbours.find(target_storage_id);
399
400             if (map_it != node.other_storages_neighbours.end()) {
401                 const std::vector<typename
    StorageNode::Neighbour>& edges = map_it->second;
402                 for (size_t i = 0; i < edges.size(); ++i) {
403                     total_weight += edges[i].second.weight;
404                 }
405             }
406         }
407
408         return total_weight;
409     }
410
411 };
412
413 #endif // VKR_COURSE_STORAGE
```

Листинг 15: optimizer.hpp

```
1 #ifndef VKR_COURSE_OPTIMIZER
2 #define VKR_COURSE_OPTIMIZER
3
4 #include "storage.hpp"
5 #include <unordered_set>
6
7 template <typename KeyType>
8 class StorageOptimizer {
9 private:
10     const Storage<KeyType>& storage1;
11     const Storage<KeyType>& storage2;
12
13     int calculate_gv(const Node<KeyType>& node, int
```

```
    other_storage_id) const {
14          int internal_edges_weight =
    node.get_internal_edges_weight_sum();
15          int external_edges_weight =
    node.get_external_edges_weight_sum_to_storage(other_storage_id);
16
17          return internal_edges_weight - external_edges_weight;
18      }
19
20  public:
21      StorageOptimizer(const Storage<KeyType>& s1, const
    Storage<KeyType>& s2)
22          : storage1(s1), storage2(s2) {}
23
24      void calculate_gvs() const {
25          // Получаем граничные вершины для обоих хранилищ
26          std::unordered_map<KeyType, Node<KeyType>>
    boundary_nodes1 =
    storage1.get_nodes_with_neighbors_in_storage_map_copy(storage2.get_id());
27          std::unordered_map<KeyType, Node<KeyType>>
    boundary_nodes2 =
    storage2.get_nodes_with_neighbors_in_storage_map_copy(storage1.get_id());
28
29          typename std::unordered_map<KeyType,
    Node<KeyType>>::const_iterator it;
30          for (it = boundary_nodes1.begin(); it !=
    boundary_nodes1.end(); ++it) {
31              const Node<KeyType>& node = it->second;
32
33              std::cout << "Metric for node " << node.key.key_value
    << " is " << calculate_gv(node, storage2.get_id()) <<
    std::endl;
34          }
35
36          for (it = boundary_nodes2.begin(); it !=
    boundary_nodes2.end(); ++it) {
37              const Node<KeyType>& node = it->second;
38
39              std::cout << "Metric for node " << node.key.key_value
    << " is " << calculate_gv(node, storage1.get_id()) <<
    std::endl;
40          }
41      }
42  };
43
44  #endif // VKR_COURSE_OPTIMIZER
```

Листинг 16: main.cpp (Демонтстрация работы)

```cpp
#include "include/graph.hpp"
#include "include/storage.hpp"
#include "include/optimizer.hpp"

#include <stdlib.h>
#include <iostream>
#include <vector>
#include <utility>

int main() {
    // Создаём два хранилища
    Storage<int> storage1(1);
    Storage<int> storage2(2);

    // Создаём рёбра
    Edge edge1{5};   // Ребро между 1 и 2 в storage1
    Edge edge2{3};   // Ребро между 3 и 4 в storage2
    Edge edge3{7};   // Ребро между 2 (storage1) и 3 (storage2)

    // 1. Вершина 1 в storage1
    NodeKey<int> key1(1);
    std::vector<typename Node<int>::Neighbour> n1;
    storage1.add_node(Node<int>(key1, n1));

    // 2. Вершина 2 в storage1 (внутреннее ребро к 1 + внешнее к 3)
    NodeKey<int> key2(2);
    std::vector<typename Node<int>::Neighbour> n2;
    n2.push_back(std::make_pair(NodeKey<int>(1), edge1));

    std::map<int, std::vector<typename Node<int>::Neighbour>>
    ext2;
    std::vector<typename Node<int>::Neighbour> ext_n2;
    ext_n2.push_back(std::make_pair(NodeKey<int>(3), edge3));
    ext2[2] = ext_n2;   // storage2 имеет ID=2

    storage1.add_node(Node<int>(key2, n2, ext2));

    // 3. Вершина 3 в storage2 (внешнее к 2)
    NodeKey<int> key3(3);
    std::vector<typename Node<int>::Neighbour> n3;
    //n3.push_back(std::make_pair(NodeKey<int>(4), edge2));

    std::map<int, std::vector<typename Node<int>::Neighbour>>
```

```
    ext3;
43      std::vector<typename Node<int>::Neighbour> ext_n3;
44      ext_n3.push_back(std::make_pair(NodeKey<int>(2), edge3));
45      ext3[1] = ext_n3;   // storage1 имеет ID=1
46
47      storage2.add_node(Node<int>(key3, n3, ext3));
48
49      // 4. Вершина 4 в storage2 (только внутреннее ребро к 3)
50      NodeKey<int> key4(4);
51      std::vector<typename Node<int>::Neighbour> n4;
52      n4.push_back(std::make_pair(NodeKey<int>(3), edge2));
53      storage2.add_node(Node<int>(key4, n4));
54
55      // 5. Вершина 5 в storage2 (только внутреннее ребро к 5)
56      Edge edge4{5};   // Ребро между 4 и 5 (storage2)
57      NodeKey<int> key5(5);
58      std::vector<typename Node<int>::Neighbour> n5;
59      n5.push_back(std::make_pair(NodeKey<int>(3), edge4));
60      storage2.add_node(Node<int>(key5, n5));
61
62      // 6. Вершина 6 в storage2 (только внутреннее ребро к 5)
63      Edge edge5{6};   // Ребро между 6 и 5 (storage2)
64      NodeKey<int> key6(6);
65      std::vector<typename Node<int>::Neighbour> n6;
66      n5.push_back(std::make_pair(NodeKey<int>(5), edge5));
67      storage2.add_node(Node<int>(key6, n6));
68
69      StorageOptimizer optimizer(storage1, storage2);
70
71      optimizer.calculate_gvs();
72
73      return 0;
74  }
```