



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

подпись, дата

Козлов В.М.

Фамилия, И.О.

Преподаватель

подпись, дата

Фамилия, И.О.

Москва, 2024

бла бла

Оглавление

| | |
|--|-----------|
| Введение | 4 |
| 1 Основная часть | 5 |
| 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 | 12 |
| 1.3.4 Класс оптимизатора (optimizer.hpp) | 17 |
| 1.4 Тестирование | 20 |
| Заключение..... | 23 |
| Список использованных источников..... | 25 |
| Приложение А | 26 |

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 один мастер и множество хранилищ, соединённых общей шиной, через которую происходит общение.

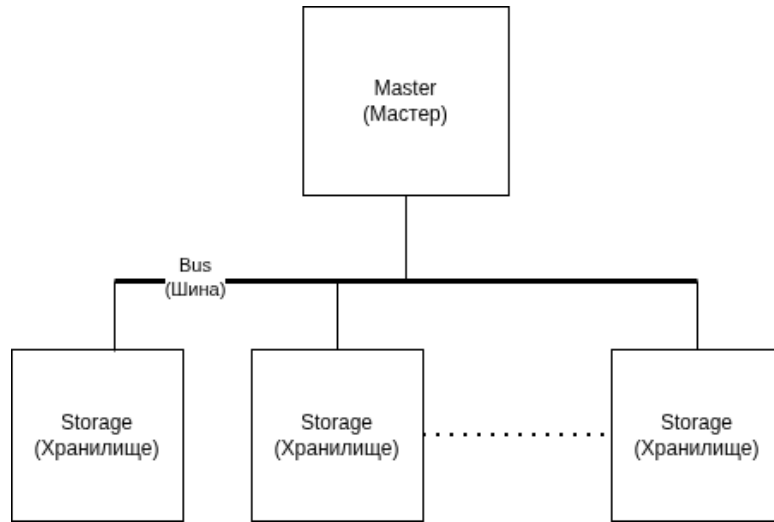


Рис. 1: Упрощённая архитектура

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)
24    const {
25        if (node == to) {
26            return from;
27        } else if (node == from) {
28            return to;
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>>
19 new_edges) {
20         edges.merge(new_edges);
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
29         edges.erase(removing_edge.get_other(current_key));
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 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, 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. Релевант-

ный 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

Листинг 6: Методы добавления вершин класса `Storage`

```
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;
30                external_edges_to_announce.insert(edge);
31            }

```

```

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_in_storage` используя мапу 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,
        Edge<KeyType>>>>::const_iterator storage_it =
        external_edges.find(target_storage_id);
4     if (storage_it == external_edges.end()) {
5         return result;
6     }
7
8     const std::map<Key, std::map<Key, Edge<KeyType>>>&
        node_edges = storage_it->second;
9     for (typename std::map<Key, std::map<Key,
        Edge<KeyType>>>::const_iterator node_edges_it =

```

```

node_edges.begin(); node_edges_it != node_edges.end();
++node_edges_it) {
10     const std::map<Key, Edge<KeyType>>& local_node_edges =
node_edges_it->second;
11     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 листинге 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,
    std::set<Edge<KeyType>> boundary_edges) const {
3     NodeKey<KeyType> this_key = node.get_key();
4     float internal_edges_weight = 0;
5     float external_edges_weight = 0;

```

```

6
7     std::map<NodeKey<KeyType>, Edge<KeyType>>
    boundary_edges_map;
8     for (typename std::set<Edge<KeyType>>::const_iterator
    edge_it = boundary_edges.begin();
9         edge_it != boundary_edges.end(); ++edge_it) {
10         NodeKey<KeyType> other = edge_it->get_other(this_key);
11         if (!(other == this_key)) {
12             boundary_edges_map[other] = *edge_it;
13         }
14     }
15
16     for (typename std::map<NodeKey<KeyType>,
    Edge<KeyType>>::const_iterator edge_it = node.edges.begin();
17         edge_it != node.edges.end(); ++edge_it) {
18         const NodeKey<KeyType>& neighbor_key = edge_it->first;
19         const Edge<KeyType>& edge = edge_it->second;
20         if (boundary_edges_map.find(neighbor_key) !=
    boundary_edges_map.end()) {
21             external_edges_weight += edge.get_weight();
22         } else {
23             internal_edges_weight += edge.get_weight();
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

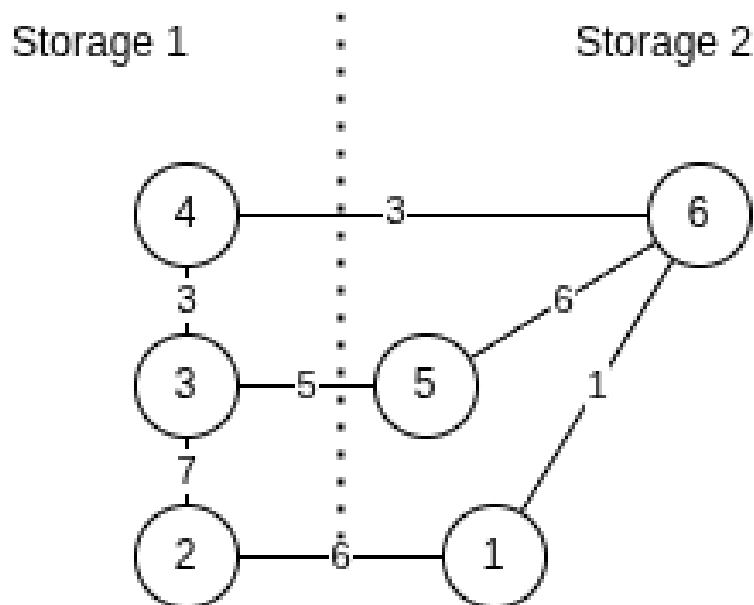


Рис. 2: Граф для тестирования

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:

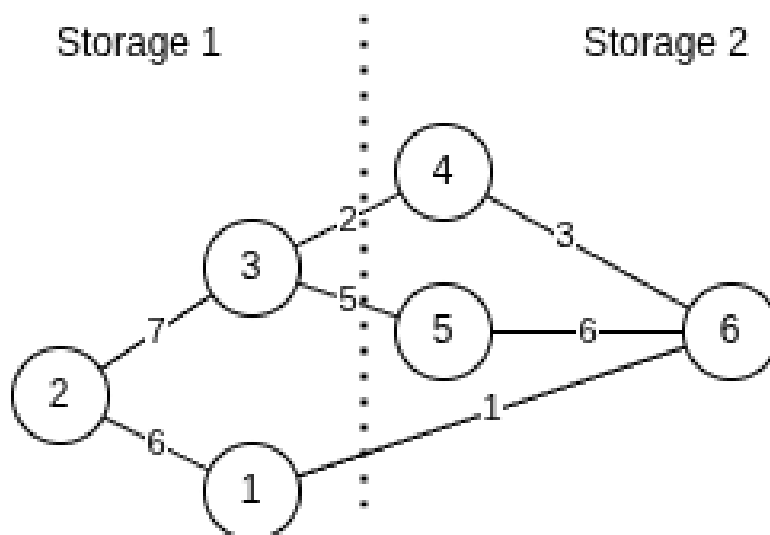


Рис. 3: Оптимизированный граф

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 листинги кода программы.

Листинг 14: graph.hpp

```
1 #ifndef VKR_GRAPH
2 #define VKR_GRAPH
3
4 #include <stdlib.h>
5 #include <map>
6 #include <string>
7 #include <vector>
8 #include <numeric>
9 #include <utility>
10 #include <stdexcept>
11 #include <iostream>
12 #include <sstream>
13
14 template <typename KeyType>
15 class NodeKey {
16 public:
17     KeyType key_value;
18
19     NodeKey(): key_value(KeyType()) {};
20     NodeKey(const KeyType& key): key_value(key) {};
21
22     // Оператор присваивания
23     NodeKey<KeyType>& operator=(const NodeKey<KeyType>& other) {
24         if (this != &other) {
25             key_value = other.key_value;
26         }
27         return *this;
28     };
29
30     // Дружественные операторы сравнения
31     friend bool operator<(const NodeKey<KeyType>& lhs, const
        NodeKey<KeyType>& rhs) {
32         return lhs.key_value < rhs.key_value;
33     }
34
35     friend bool operator>(const NodeKey<KeyType>& lhs, const
        NodeKey<KeyType>& rhs) {
```

```

36     return rhs < lhs;
37 }
38
39 friend bool operator==(const NodeKey<KeyType>& lhs, const
    NodeKey<KeyType>& rhs) {
40     return lhs.key_value == rhs.key_value;
41 }
42
43 friend bool operator!=(const NodeKey<KeyType>& lhs, const
    NodeKey<KeyType>& rhs) {
44     return !(lhs == rhs);
45 }
46 };
47
48
49 class Parameter{
50 private:
51     std::vector<std::byte> data;
52 public:
53     void set_data(const std::vector<std::byte>& _data) {
54         data = _data;
55     }
56     std::vector<std::byte> get_data() {
57         return data;
58     };
59 };
60
61 template <typename KeyType>
62 class Edge {
63 private:
64     float weight;
65     bool directional;
66     std::map<std::string, Parameter> parameters;
67     NodeKey<KeyType> from;
68     NodeKey<KeyType> to;
69 public:
70     Edge() : weight(1), directional(false) {}
71     Edge(const NodeKey<KeyType>& from_node, const
        NodeKey<KeyType>& to_node)
72         : weight(1), directional(false), from(from_node),
        to(to_node) {}

```

```

73
74     Edge(const NodeKey<KeyType>& from_node, const
NodeKey<KeyType>& to_node, int edge_weight)
75         : weight(edge_weight), directional(false),
from(from_node), to(to_node) {}
76
77     Edge(const NodeKey<KeyType>& from_node, const
NodeKey<KeyType>& to_node,
78         float edge_weight, bool is_directional)
79         : weight(edge_weight), directional(is_directional),
from(from_node), to(to_node) {}
80
81     Edge(const NodeKey<KeyType>& from_node, const
NodeKey<KeyType>& to_node,
82         float edge_weight, bool is_directional, const
std::map<std::string, Parameter>& params)
83         : weight(edge_weight), directional(is_directional),
parameters(params),
84         from(from_node), to(to_node) {}
85
86     Edge(const Edge& other)
87         : weight(other.weight), directional(other.directional),
88         parameters(other.parameters), from(other.from),
to(other.to) {}
89
90     float get_weight() const {
91         return weight;
92     }
93     bool is_directional() const {
94         return directional;
95     }
96     const std::map<std::string, Parameter>& get_parameters()
const {
97         return parameters;
98     }
99     const NodeKey<KeyType>& get_from() const {
100         return from;
101     }
102     const NodeKey<KeyType>& get_to() const {
103         return to;
104     }

```

```

105
106     Edge& get_reverted() {
107         return Edge<KeyType>(from, to, weight, directional,
108     parameters);
109     }
110
111     Edge& operator=(const Edge& other) {
112         if (this != &other) {
113             weight = other.weight;
114             directional = other.directional;
115             parameters = other.parameters;
116             from = other.from;
117             to = other.to;
118         }
119         return *this;
120     }
121
122     bool operator<(const Edge& other) const {
123         if (from != other.from) return from < other.from;
124         if (to != other.to) return to < other.to;
125         if (weight != other.weight) return weight <
126     other.weight;
127         return directional < other.directional;
128     }
129
130     // doesn't check extra parameters
131     bool operator==(const Edge& other) {
132         if (other.directional != directional) return false;
133
134         if (directional) {
135             if (other.to != to || other.from != from) return
136     false;
137         } else {
138             if ((other.to != to && other.from != from) ||
139     (other.from != to && other.to != from)) return
140     false;
141         }
142
143         if (other.weight != weight) return false;
144
145         return true;

```

```

142     }
143
144     NodeKey<KeyType> get_other(const NodeKey<KeyType>& node)
const {
145         if (node == to) {
146             return from;
147         } else if (node == from) {
148             return to;
149         }
150
151         return node;
152     }
153
154     NodeKey<KeyType> get_other(NodeKey<KeyType>* node) const {
155         if (*node == to) {
156             return from;
157         } else if (*node == from) {
158             return to;
159         } else {
160             return *node;
161         }
162     }
163
164     friend std::ostream& operator<<(std::ostream& os, const
Edge<KeyType>& edge) {
165         os << "Edge(from: " << edge.get_from().key_value
166         << ", to: " << edge.get_to().key_value
167         << ", weight: " << edge.get_weight();
168
169         if (edge.is_directional()) {
170             os << ", directional";
171         }
172
173         // Добавляем информацию о параметрах, если они есть
174         const std::map<std::string, Parameter>& params =
edge.get_parameters();
175         if (!params.empty()) {
176             os << ", parameters: {";
177             typename std::map<std::string,
Parameter>::const_iterator it = params.begin();
178             if (it != params.end()) {

```

```

179         os << it->first;
180         ++it;
181     }
182     for (; it != params.end(); ++it) {
183         os << ", " << it->first;
184     }
185     os << "}";
186 }
187
188 os << ")";
189 return os;
190 }
191 };
192
193 template <typename KeyType>
194 class Node {
195 private:
196     NodeKey<KeyType> key;
197 public:
198     std::map<std::string, Parameter> parameters;
199     std::map<NodeKey<KeyType>, Edge<KeyType>> edges;
200
201     Node():
202         key(NodeKey<KeyType>()) {};
203
204     Node(KeyType _key):
205         key(NodeKey<KeyType>(_key)) {};
206
207     Node(NodeKey<KeyType> _key):
208         key(_key) {};
209
210     Node(NodeKey<KeyType> _key, std::map<NodeKey<KeyType>,
Edge<KeyType>> _edges):
211         key(_key), edges(_edges) {};
212
213     Node(const Node& other)
214         : key(other.key),
215         parameters(other.parameters),
216         edges(other.edges) {};
217
218     NodeKey<KeyType> get_key() const {

```

```

219         return key;
220     }
221
222     void add_edge(Edge<KeyType> new_edge) {
223         NodeKey<KeyType> current_key = this->get_key();
224         edges[new_edge.get_other(current_key)] = new_edge;
225     }
226
227     void add_edges(std::map<NodeKey<KeyType>, Edge<KeyType>>
new_edges) {
228         edges.merge(new_edges);
229     }
230
231     bool remove_edge_to(NodeKey<KeyType> neighbour_key) {
232         return edges.erase(neighbour_key);
233     }
234
235     bool remove_edge(Edge<KeyType> removing_edge) {
236         NodeKey<KeyType> current_key = this->get_key();
237         return
edges.erase(removing_edge.get_other(current_key));
238     }
239
240     Node& operator=(const Node& other) {
241         if (this != &other) {
242             key = other.key;
243             parameters = other.parameters;
244             edges = other.edges;
245         }
246         return *this;
247     }
248
249     bool operator<(const Node<KeyType>& other) const {
250         return key < other.key;
251     }
252
253     friend std::ostream& operator<<(std::ostream& os, const
Node<KeyType>& node) {
254         os << "Node(key: " << node.get_key().key_value;
255
256         // Выводим параметры узла

```



```

257         if (!node.parameters.empty()) {
258             os << " , parameters: {";
259             typename std::map<std::string,
Parameter>::const_iterator param_it =
node.parameters.begin();
260             if (param_it != node.parameters.end()) {
261                 os << param_it->first;
262                 ++param_it;
263             }
264             for (; param_it != node.parameters.end();
++param_it) {
265                 os << " , " << param_it->first;
266             }
267             os << "}";
268         }
269
270         // Выводим ребра узла
271         if (!node.edges.empty()) {
272             os << " , edges: [";
273             typename std::map<NodeKey<KeyType>,
Edge<KeyType>>::const_iterator edge_it = node.edges.begin();
274             if (edge_it != node.edges.end()) {
275                 os << edge_it->second;
276                 ++edge_it;
277             }
278             for (; edge_it != node.edges.end(); ++edge_it) {
279                 os << " , " << edge_it->second;
280             }
281             os << "]";
282         }
283
284         os << " )";
285         return os;
286     }
287 };
288
289 #endif // VKR_GRAPH

```

Листинг 15: interface_bus.hpp

```

1 #ifndef VKR_IBUS
2 #define VKR_IBUS

```

```

3
4 #include "graph.hpp"
5 #include <set>
6
7 template <typename KeyType>
8 class IBus {
9 public:
10     virtual Node<KeyType> request_node(const NodeKey<KeyType>&
        node) = 0;
11     virtual int send_add_node(const Node<KeyType>& node) = 0;
12     virtual bool send_add_node(const Node<KeyType>& node, int
        storage_id) = 0;
13     virtual bool send_remove_node(const NodeKey<KeyType>& node)
        = 0;
14     virtual bool send_remove_node(const NodeKey<KeyType>& node,
        int storage_id) = 0;
15     virtual int ask_who_has(int asker_id, NodeKey<KeyType> key)
        = 0;
16     virtual void announce_add(NodeKey<KeyType> key, int
        storage_id, std::set<Edge<KeyType>> edges) = 0;
17     virtual void announce_remove(NodeKey<KeyType> key, int
        storage_id) = 0;
18     // запрашивает у source вершины, соседствующие с target
19     virtual std::set<Node<KeyType>>
        ask_neighbours_to_storage(int source, int target) = 0;
20     // запрашивает у source рёбра, идущие в target
21     virtual std::set<Edge<KeyType>> ask_edges_to_storage(int
        source, int target) = 0;
22 };
23
24 #endif // VKR_IBUS

```

Листинг 16: 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
        = storages.begin(); it != storages.end(); ++it) {
27             Node<KeyType*>* node_pointer =
        it->second->get_node(node);
28             if (node_pointer != nullptr) {
29                 return Node<KeyType>(*node_pointer);
30                 break;
31             };
32         }
33         return Node<KeyType>();
34     };
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
52         = storages.begin(); it != storages.end(); ++it) {
53         if (it—>second—>remove_node_and_announce(node)) {
54             success = true;
55             break;
56         };
57     }
58     return success;
59 };
60 bool send_remove_node(const NodeKey<KeyType>& node, int
61     storage_id) override {
62     if (storages.find(storage_id) == storages.end()) {
63         return false;
64     }
65     return storages[storage_id]—>remove_node_and_announce(node);
66 };
67 int ask_who_has(int asker_id, NodeKey<KeyType> key) override{
68     for (typename std::map<int, Storage<KeyType>*>::iterator it
69         = storages.begin(); it != storages.end(); ++it) {
70         if (it—>second != nullptr && it—>second—>has_node(key))
71         {
72             return it—>first;
73         };
74     }
75     return -1;
76 };
77 void announce_add(NodeKey<KeyType> key, int storage_id,
78     std::set<Edge<KeyType>> edges) override {
79     for (typename std::map<int, Storage<KeyType>*>::iterator it
80         = storages.begin(); it != storages.end(); ++it) {
81         if (it—>second != nullptr) {
82             it—>second—>get_add_announcement(key, storage_id,
83                 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

```

Листинг 17: storage.hpp

```

1 #ifndef VKR_COURSE_STORAGE
2 #define VKR_COURSE_STORAGE

```

```

3
4 #include "graph.hpp"
5 #include "interface_bus.hpp"
6 #include <set>
7 #include <optional>
8 #include <vector>
9 #include <algorithm>
10 #include <iostream>
11 #include <map>
12
13 template <typename KeyType>
14 class Storage {
15 private:
16 typedef Node<KeyType> StorageNode;
17 typedef NodeKey<KeyType> Key;
18 int storage_id;
19 std::map<Key, StorageNode> nodes;
20 // external_edges[storage edge go to][external node][local node] = edge
21 std::map<int, std::map<Key, std::map<Key, Edge<KeyType>>>>
    external_edges;
22 IBus<KeyType> *bus = nullptr;
23
24 public:
25 Storage(int id)
26     :storage_id(id) {}
27 Storage(int id, std::map<Key, StorageNode> _nodes)
28     :storage_id(id), nodes(_nodes) {}
29
30 int get_id() const { return storage_id; };
31 void connect_to_bus(IBus<KeyType>* _bus) { bus = _bus; };
32
33 void get_add_announcement(Key key, int announcer_id,
    std::set<Edge<KeyType>> edges) {
34     if (announcer_id == storage_id) return;
35
36     for (typename std::set<Edge<KeyType>>::iterator edge_it =
    edges.begin(); edge_it != edges.end(); ++edge_it) {
37         Key other_key = edge_it->get_other(key);
38         typename std::map<Key, StorageNode>::iterator it =
    nodes.find(other_key);
39         if (it != nodes.end()) {

```

```

40         it->second.add_edge(*edge_it);
41         typename std::map<Key, std::map<Key,
Edge<KeyType>>>>::iterator it2 =
external_edges[announcer_id].find(key);
42         if (it2 == external_edges[announcer_id].end()){
43             external_edges[announcer_id][key] =
std::map<Key, Edge<KeyType>>>();
44         }
45         external_edges[announcer_id][key][other_key] =
*edge_it;
46     }
47 }
48 };
49
50 void get_remove_announcement(Key key, int announcer_id) {
51     if (announcer_id == storage_id) return;
52
53     typename std::map<int, std::map<Key, std::map<Key,
Edge<KeyType>>>>::iterator storage_it =
54         external_edges.find(announcer_id);
55     if (storage_it == external_edges.end()) {
56         return;
57     }
58
59     std::map<Key, std::map<Key, Edge<KeyType>>>>& external_map =
storage_it->second;
60     typename std::map<Key, std::map<Key,
Edge<KeyType>>>>::iterator external_node_it =
61         external_map.find(key);
62     if (external_node_it == external_map.end()) {
63         return;
64     }
65
66     std::map<Key, Edge<KeyType>>>& local_to_ext_map =
external_node_it->second;
67
68     // Удаляем рёбра из локальных узлов
69     for (typename std::map<Key, Edge<KeyType>>>::iterator
local_nodes_it = local_to_ext_map.begin();
70         local_nodes_it != local_to_ext_map.end();
++local_nodes_it) {

```

```

71         Key local_key = local_nodes_it->first;
72         typename std::map<Key, StorageNode>::iterator node =
nodes.find(local_key);
73         if (node != nodes.end()) {
74             node->second.remove_edge_to(key);
75         }
76     }
77
78     // Удаляем запись из external_edges
79     external_map.erase(external_node_it);
80
81     // Если для этого хранилища не осталось внешних узлов, удаляем запись
82     if (external_map.empty()) {
83         external_edges.erase(storage_it);
84     }
85 }
86
87 std::optional<std::set<Edge<KeyType>>> add_node(const
StorageNode& node){
88     Key key = node.get_key();
89     std::cout << storage_id << " adding " << key.key_value <<
std::endl;
90     typename std::map<Key, StorageNode>::iterator it =
nodes.find(key);
91     if (it != nodes.end()) {
92         return std::nullopt;
93     }
94     nodes[key] = node;
95
96     std::set<Edge<KeyType>> external_edges_to_announce;
97     for (typename std::map<Key, Edge<KeyType>>::const_iterator
edge_it = node.edges.begin(); edge_it != node.edges.end();
++edge_it) {
98         const Key& neighbor_key = edge_it->first;
99         const Edge<KeyType>& edge = edge_it->second;
100         std::cout << storage_id << " looks for " <<
neighbor_key.key_value << std::endl;
101         // Ищем соседа в текущем хранилище
102         typename std::map<Key, StorageNode>::iterator it2 =
nodes.find(neighbor_key);
103         if (it2 != nodes.end()) {

```



```

104         std::cout << storage_id << " node " <<
neighbor_key.key_value << " is inside, no asking" <<
std::endl;
105         // Сосед найден в этом же хранилище - добавляем обратное ребро
106         it2->second.add_edge(edge);
107     } else {
108         std::cout << storage_id << " node " <<
neighbor_key.key_value << " is outside, asking" <<
std::endl;
109         // Сосед не найден - спрашиваем у шины, где он находится
110         int neighbours_storage_id =
bus->ask_who_has(storage_id, neighbor_key);
111         std::cout << storage_id << " asked for " <<
neighbor_key.key_value << " answer: " <<
neighbours_storage_id << std::endl;
112         if (neighbours_storage_id != -1) {
113             // Сохраняем внешнее ребро
114
external_edges[neighbours_storage_id][neighbor_key][key] =
edge;
115             external_edges_to_announce.insert(edge);
116         }
117         // Если сосед нигде не найден - игнорируем (возможно, будет
добавлен позже)
118     }
119 }
120
121 return external_edges_to_announce;
122 };
123
124 bool add_node_and_announce(const StorageNode& node) {
125     if (bus == nullptr) {
126         return false;
127     };
128     std::optional<std::set<Edge<KeyType>>> external_edges =
add_node(node);
129     if (!external_edges.has_value()) {
130         return false;
131     }
132     bus->announce_add(node.get_key(), storage_id,
external_edges.value());
133     return true;

```

```

134 };
135
136
137 bool remove_node(const Key& key) {
138     if (!has_node(key)) {
139         return false;
140     };
141     StorageNode node = nodes.find(key)→second;
142     nodes.erase(key);
143
144     for (typename std::map<NodeKey<KeyType>,
Edge<KeyType>>>::iterator edge_it = node.edges.begin();
edge_it != node.edges.end(); ++edge_it) {
145         Key other_key = edge_it→second.get_other(key);
146         typename std::map<Key, StorageNode>::iterator it =
nodes.find(other_key);
147         if (it != nodes.end()) {
148             it→second.remove_edge_to(key);
149         }
150     }
151
152     // Удаляем все внешние рёбра, связанные с этим узлом
153     typename std::map<int, std::map<Key, std::map<Key,
Edge<KeyType>>>>::iterator ext_storage_it;
154     for (ext_storage_it = external_edges.begin();
ext_storage_it != external_edges.end(); ) {
155         std::map<Key, std::map<Key, Edge<KeyType>>>& ext_nodes
= ext_storage_it→second;
156
157         // Удаляем записи, где наш узел является локальным
158         typename std::map<Key, std::map<Key,
Edge<KeyType>>>::iterator ext_node_it;
159         for (ext_node_it = ext_nodes.begin(); ext_node_it !=
ext_nodes.end(); ) {
160             std::map<Key, Edge<KeyType>>& local_edges =
ext_node_it→second;
161
162             // Удаляем все рёбра, где локальный ключ - наш удаляемый узел
163             typename std::map<Key, Edge<KeyType>>::iterator
edge_it = local_edges.find(key);
164             if (edge_it != local_edges.end()) {

```

```

165         local_edges.erase(edge_it);
166     }
167
168     // Если после удаления для этого внешнего узла не осталось рёбер,
удаляем запись
169     if (local_edges.empty()) {
170         ext_nodes.erase(ext_node_it++);
171     } else {
172         ++ext_node_it;
173     }
174 }
175
176 // Если для этого хранилища не осталось внешних узлов, удаляем запись
177 if (ext_nodes.empty()) {
178     external_edges.erase(ext_storage_it++);
179 } else {
180     ++ext_storage_it;
181 }
182 }
183
184 // Удаляем сам узел
185 nodes.erase(node.get_key());
186
187 return true;
188 };
189
190 bool remove_node_and_announce(const Key& key) {
191     if (remove_node(key)) {
192         bus->announce_remove(key, storage_id);
193         return true;
194     }
195     return false;
196 };
197
198 StorageNode* get_node(const Key& key) {
199     typename std::map<Key, StorageNode>::iterator it =
        nodes.find(key);
200     if (it != nodes.end()) {
201         return &(it->second);
202     }
203     return nullptr;
204 };

```

```

205
206
207 bool has_node(const Key& key) const {
208     return nodes.find(key) != nodes.end();
209 }
210
211 const std::map<Key, StorageNode>& get_all_nodes() const {
212     return nodes;
213 }
214
215 size_t size() const {
216     return nodes.size();
217 }
218
219 void clear() {
220     nodes.clear();
221 }
222
223 //
224 // Получение наборов связанных с другим хранилищем
225 //
226
227 // Получить набор узлов, имеющих соседей в указанном хранилище (копии)
228 std::set<StorageNode> get_nodes_with_neighbors_in_storage(int
    target_storage_id) const {
229     std::set<StorageNode> result;
230
231     typename std::map<int, std::map<Key, std::map<Key,
    Edge<KeyType>>>>::const_iterator storage_it =
    external_edges.find(target_storage_id);
232     if (storage_it == external_edges.end()) {
233         return result;
234     }
235
236     const std::map<Key, std::map<Key, Edge<KeyType>>>>&
    node_edges = storage_it->second;
237     for (typename std::map<Key, std::map<Key,
    Edge<KeyType>>>>::const_iterator node_edges_it =
    node_edges.begin(); node_edges_it != node_edges.end();
    ++node_edges_it) {
238         const std::map<Key, Edge<KeyType>>& local_node_edges =

```

```

node_edges_it->second;
239     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) {
240         Key local_key = local_node_edges_it->first;
241         typename std::map<Key, StorageNode>>::const_iterator
node_it = nodes.find(local_key);
242         if (node_it != nodes.end()) {
243             result.insert(node_it->second);
244         }
245     }
246 }
247
248 return result;
249 }
250
251 std::set<Edge<KeyType>> get_all_edges_to_storage(int
target_storage_id) const {
252     std::set<Edge<KeyType>> result;
253     typename std::map<int, std::map<Key, std::map<Key,
Edge<KeyType>>>>::const_iterator storage_it =
external_edges.find(target_storage_id);
254     if (storage_it == external_edges.end()) {
255         return std::set<Edge<KeyType>>();
256     } else {
257         const std::map<Key, std::map<Key, Edge<KeyType>>>&
external_nodes_map = storage_it->second;
258         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) {
259             const std::map<Key, Edge<KeyType>>& node_edge_map =
node_edges_it->second;
260             for (typename std::map<Key,
Edge<KeyType>>>::const_iterator edges_it =
node_edge_map.begin(); edges_it != node_edge_map.end();
++edges_it) {
261                 result.insert(edges_it->second);
262             }
263         }

```

```

264     }
265     return result;
266 }
267
268 typename std::map<Key, StorageNode>::iterator begin() {
269     return nodes.begin();
270 }
271
272 typename std::map<Key, StorageNode>::iterator end() {
273     return nodes.end();
274 }
275
276 typename std::map<Key, StorageNode>::const_iterator begin()
277     const {
278     return nodes.begin();
279 }
280 typename std::map<Key, StorageNode>::const_iterator end() const
281     {
282     return nodes.end();
283 }
284
285 typename std::map<Key, StorageNode>::const_iterator cbegin()
286     const {
287     return nodes.cbegin();
288 }
289
290 typename std::map<Key, StorageNode>::const_iterator cend()
291     const {
292     return nodes.cend();
293 }
294
295 bool empty() const {
296     return nodes.empty();
297 }
298
299 friend std::ostream& operator<<(std::ostream& os, const
    Storage<KeyType>& storage) {
300     os << "Storage(id: " << storage.get_id();
301
302     if (storage.empty()) {

```

```

300         os << " , empty";
301     } else {
302         os << " , nodes: " << storage.size() << " {";
303
304         typename std::map<Key, StorageNode>::const_iterator it
= storage.begin();
305         if (it != storage.end()) {
306             os << std::endl << " " << it->second;
307             ++it;
308         }
309         for (; it != storage.end(); ++it) {
310             os << " ," << std::endl << " " << it->second;
311         }
312         os << std::endl << "}";
313     }
314
315     // Вывод внешних рёбер
316     if (!storage.external_edges.empty()) {
317         os << " , external_edges: {";
318
319         typename std::map<int, std::map<Key, std::map<Key,
Edge<KeyType>>>>::const_iterator storage_it =
320             storage.external_edges.begin();
321
322         if (storage_it != storage.external_edges.end()) {
323             os << std::endl << " to storage " <<
storage_it->first << ": {";
324
325             const std::map<Key, std::map<Key, Edge<KeyType>>>&
ext_nodes = storage_it->second;
326             typename std::map<Key, std::map<Key,
Edge<KeyType>>>::const_iterator node_it = ext_nodes.begin();
327
328             if (node_it != ext_nodes.end()) {
329                 os << std::endl << " external node " <<
node_it->first.key_value << ": [";
330
331                 const std::map<Key, Edge<KeyType>>& local_edges
= node_it->second;
332                 typename std::map<Key,
Edge<KeyType>>::const_iterator edge_it = local_edges.begin();

```

```

333
334         if (edge_it != local_edges.end()) {
335             os << std::endl << "          local node " <<
edge_it->first.key_value
336                 << " -> " << edge_it->second;
337             ++edge_it;
338         }
339         for (; edge_it != local_edges.end(); ++edge_it)
{
340             os << " ," << std::endl << "          local node
" << edge_it->first.key_value
341                 << " -> " << edge_it->second;
342         }
343         os << std::endl << "      ]";
344         ++node_it;
345     }
346     for (; node_it != ext_nodes.end(); ++node_it) {
347         os << " ," << std::endl << "          external node "
<< node_it->first.key_value << " : [";
348
349         const std::map<Key, Edge<KeyType>>& local_edges
= node_it->second;
350         typename std::map<Key,
Edge<KeyType>>::const_iterator edge_it = local_edges.begin();
351
352         if (edge_it != local_edges.end()) {
353             os << std::endl << "          local node " <<
edge_it->first.key_value
354                 << " -> " << edge_it->second;
355             ++edge_it;
356         }
357         for (; edge_it != local_edges.end(); ++edge_it)
{
358             os << " ," << std::endl << "          local node
" << edge_it->first.key_value
359                 << " -> " << edge_it->second;
360         }
361         os << std::endl << "      ]";
362     }
363     os << std::endl << "  }";
364     ++storage_it;

```



```

365     }
366
367     for (; storage_it != storage.external_edges.end();
++storage_it) {
368         os << ", " << std::endl << "    to storage " <<
storage_it->first << ": {";
369
370         const std::map<Key, std::map<Key, Edge<KeyType>>>&
ext_nodes = storage_it->second;
371         typename std::map<Key, std::map<Key,
Edge<KeyType>>>::const_iterator node_it = ext_nodes.begin();
372
373         if (node_it != ext_nodes.end()) {
374             os << std::endl << "        external node " <<
node_it->first.key_value << ": [";
375
376             const std::map<Key, Edge<KeyType>>& local_edges
= node_it->second;
377             typename std::map<Key,
Edge<KeyType>>::const_iterator edge_it = local_edges.begin();
378
379             if (edge_it != local_edges.end()) {
380                 os << std::endl << "            local node " <<
edge_it->first.key_value
381                 << " -> " << edge_it->second;
382                 ++edge_it;
383             }
384             for (; edge_it != local_edges.end(); ++edge_it)
{
385                 os << ", " << std::endl << "            local node
" << edge_it->first.key_value
386                 << " -> " << edge_it->second;
387             }
388             os << std::endl << "        ]";
389             ++node_it;
390         }
391         for (; node_it != ext_nodes.end(); ++node_it) {
392             os << ", " << std::endl << "        external node "
<< node_it->first.key_value << ": [";
393
394             const std::map<Key, Edge<KeyType>>& local_edges

```

```

    = node_it->second;
395         typename std::map<Key,
Edge<KeyType>>::const_iterator edge_it = local_edges.begin();
396
397         if (edge_it != local_edges.end()) {
398             os << std::endl << "          local node " <<
edge_it->first.key_value
399                 << " -> " << edge_it->second;
400             ++edge_it;
401         }
402         for (; edge_it != local_edges.end(); ++edge_it)
{
403             os << ", " << std::endl << "          local node
" << edge_it->first.key_value
404                 << " -> " << edge_it->second;
405         }
406         os << std::endl << "      ]";
407     }
408     os << std::endl << "  }";
409 }
410
411     os << std::endl << "}";
412 }
413
414     os << ")";
415     return os;
416 }
417 };
418
419 #endif // VKR_COURSE_STORAGE

```

Листинг 18: optimizer.hpp

```

1 #ifndef VKR_OPTIMIZER
2 #define VKR_OPTIMIZER
3
4 #include "interface_bus.hpp"
5
6 #include <stdlib.h>
7 #include <iostream>
8 #include <map>
9

```

```

10 template <typename KeyType>
11 class ExternalStorageOptimizer {
12 private:
13 IBus<KeyType>* bus;
14
15 float calculate_gv(const Node<KeyType>& node,
16                   std::set<Edge<KeyType>> boundary_edges) const {
17     NodeKey<KeyType> this_key = node.get_key();
18     float internal_edges_weight = 0;
19     float external_edges_weight = 0;
20
21     std::map<NodeKey<KeyType>, Edge<KeyType>>
22     boundary_edges_map;
23     for (typename std::set<Edge<KeyType>>::const_iterator
24     edge_it = boundary_edges.begin();
25     edge_it != boundary_edges.end(); ++edge_it) {
26         NodeKey<KeyType> other = edge_it->get_other(this_key);
27         if (!(other == this_key)) {
28             boundary_edges_map[other] = *edge_it;
29         }
30     }
31
32     for (typename std::map<NodeKey<KeyType>,
33     Edge<KeyType>>::const_iterator edge_it = node.edges.begin();
34     edge_it != node.edges.end(); ++edge_it) {
35         const NodeKey<KeyType>& neighbor_key = edge_it->first;
36         const Edge<KeyType>& edge = edge_it->second;
37         if (boundary_edges_map.find(neighbor_key) !=
38         boundary_edges_map.end()) {
39             external_edges_weight += edge.get_weight();
40             std::cout << "node: " << node.get_key().key_value
41             << " external neighbour: " << neighbor_key.key_value <<
42             std::endl;
43         } else {
44             internal_edges_weight += edge.get_weight();
45         }
46     }
47
48     return internal_edges_weight - external_edges_weight;
49 }
50
51

```

```

43 std::map<int, std::set<Node<KeyType>>>
    get_negative_gvs(std::map<int, std::map<Node<KeyType>,
float>>> full_map) {
44     std::map<int, std::set<Node<KeyType>>> result;
45     typename std::map<int, std::map<Node<KeyType>,
float>>>::const_iterator it;
46     for (it = full_map.begin(); it != full_map.end(); ++it) {
47         result[it->first] = std::set<Node<KeyType>>();
48         const std::map<Node<KeyType>, float>& nodes =
it->second;
49         typename std::map<Node<KeyType>, float>::const_iterator
it2;
50         for (it2 = nodes.begin(); it2 != nodes.end(); ++it2) {
51             std::cout << "storage: " << it->first << " node: "
<< it2->first << " gv: " << it2->second << std::endl;
52             if (it2->second < 0) {
53                 result[it->first].insert(it2->first);
54             }
55         }
56         if (result[it->first].empty()) {
57             result.erase(it->first);
58         }
59     }
60     return result;
61 };
62
63 public:
64 ExternalStorageOptimizer(IBus<KeyType>* _bus)
65     : bus(_bus) {}
66
67 std::map<int, std::map<Node<KeyType>, float>>> calculate_gvs(int
storage1, int storage2) const {
68     std::map<int, std::map<Node<KeyType>, float>>> result;
69     // Получаем граничные вершины для обоих хранилищ
70     std::set<Node<KeyType>> boundary_nodes1 =
bus->ask_neighbours_to_storage(storage1, storage2);
71     std::set<Node<KeyType>> boundary_nodes2 =
bus->ask_neighbours_to_storage(storage2, storage1);
72
73     std::set<Edge<KeyType>> boundary_edges1 =
bus->ask_edges_to_storage(storage1, storage2);

```

```

74     std::set<Edge<KeyType>> boundary_edges2 =
bus->ask_edges_to_storage(storage2, storage1);

75
76     result[storage1] = std::map<Node<KeyType>, float>();
77     typename std::set<Node<KeyType>>::const_iterator it;
78     for (it = boundary_nodes1.begin(); it !=
boundary_nodes1.end(); ++it) {
79         const Node<KeyType>& node = *it;
80         result[storage1][node.get_key()] = calculate_gv(node,
boundary_edges1);
81     }
82
83     result[storage2] = std::map<Node<KeyType>, float>();
84     for (it = boundary_nodes2.begin(); it !=
boundary_nodes2.end(); ++it) {
85         const Node<KeyType>& node = *it;
86         result[storage2][node.get_key()] = calculate_gv(node,
boundary_edges2);
87     }
88     return result;
89 };
90
91 void optimize(int storage1, int storage2, int iterations_limit
= 5) {
92     if (iterations_limit == 0) return;
93
94     int iteration = 0;
95     std::map<int, std::set<Node<KeyType>>> negative_gvs =
get_negative_gvs(calculate_gvs(storage1, storage2));
96     do {
97         std::cout << "iteration " << iteration << std::endl;
98         typename std::map<int,
std::set<Node<KeyType>>>::iterator map_it;
99         for (map_it = negative_gvs.begin(); map_it !=
negative_gvs.end(); ++map_it) {
100             int this_storage = map_it->first;
101             int other_storage = this_storage == storage1 ?
storage2 : storage1;
102             std::set<Node<KeyType>>& nodes = map_it->second;
103             typename std::set<Node<KeyType>>::const_iterator
set_it;

```

```

104         for (set_it = nodes.begin(); set_it != nodes.end();
++set_it) {
105             const Node<KeyType>& node = *set_it;
106             Node<int> removed =
bus->request_node(node.get_key());
107             bus->send_remove_node(removed.get_key());
108             bus->send_add_node(removed, other_storage);
109         }
110     }
111     ++iteration;
112     negative_gvs = get_negative_gvs(calculate_gvs(storage1,
storage2));
113 } while (iteration < iterations_limit &&
!negative_gvs.empty());
114 }
115 };
116
117 #endif // VKR_OPTIMIZER

```

Листинг 19: main.cpp (Демонстрация работы)

```

1 #include "include/graph.hpp"
2 #include "include/bus.hpp"
3 #include "include/storage.hpp"
4 #include "include/optimizer.hpp"
5
6 #include <stdlib.h>
7 #include <iostream>
8 #include <vector>
9 #include <utility>
10
11 int main() {
12     SimpleBus<int> bus;
13     Storage<int> storage1(1);
14     Storage<int> storage2(2);
15
16     bus.connect_storage(&storage1);
17     bus.connect_storage(&storage2);
18
19     Node<int> node1(1);
20     Node<int> node2(2);
21     Node<int> node3(3);

```

```

22     Node<int> node4(4);
23     Node<int> node5(5);
24     Node<int> node6(6);
25
26     Edge<int> edge1_2(1, 2, 6); node1.add_edge(edge1_2);
    node2.add_edge(edge1_2);
27     Edge<int> edge1_6(1, 6, 1); node1.add_edge(edge1_6);
    node6.add_edge(edge1_6);
28     Edge<int> edge2_3(2, 3, 7); node2.add_edge(edge2_3);
    node3.add_edge(edge2_3);
29     Edge<int> edge3_4(3, 4, 2); node3.add_edge(edge3_4);
    node4.add_edge(edge3_4);
30     Edge<int> edge4_6(4, 6, 3); node4.add_edge(edge4_6);
    node6.add_edge(edge4_6);
31     Edge<int> edge3_5(3, 5, 5); node3.add_edge(edge3_5);
    node5.add_edge(edge3_5);
32     Edge<int> edge5_6(5, 6, 6); node5.add_edge(edge5_6);
    node6.add_edge(edge5_6);
33
34     bus.send_add_node(node1, 2);
35     bus.send_add_node(node2, 1);
36     bus.send_add_node(node3, 1);
37     bus.send_add_node(node4, 1);
38     bus.send_add_node(node5, 2);
39     bus.send_add_node(node6, 2);
40
41     std::cout << storage1 << std::endl;
42     std::cout << storage2 << std::endl;
43
44     std::cout << "Neighbours of 1:" << std::endl;
45     for (const auto& element : bus.ask_neighbours_to_storage(1,
46     2)) {
47         std::cout << element << " ";
48     }
49     std::cout << std::endl;
50
51     std::cout << "Neighbours of 2:" << std::endl;
52     for (const auto& element : bus.ask_neighbours_to_storage(2,
53     1)) {
54         std::cout << element << " ";
55     }

```

```

54     std::cout << std::endl;
55
56     std::cout << "edges between 1 to 2:" << std::endl;
57     for (const auto& element : bus.ask_edges_to_storage(1, 2)) {
58         std::cout << element << " ";
59     }
60     std::cout << std::endl;
61
62     std::cout << "edges between 2 to 1:" << std::endl;
63     for (const auto& element : bus.ask_edges_to_storage(2, 1)) {
64         std::cout << element << " ";
65     }
66     std::cout << std::endl;
67
68     Node<int> removed = bus.request_node(2);
69     bus.send_remove_node(2);
70
71     std::cout << storage1 << std::endl;
72     std::cout << storage2 << std::endl;
73
74     bus.send_add_node(removed, 2);
75
76     std::cout << storage1 << std::endl;
77     std::cout << storage2 << std::endl;
78
79     ExternalStorageOptimizer<int> optimizer(&bus);
80
81     optimizer.optimize(1, 2, 10);
82
83     std::cout << storage1 << std::endl;
84     std::cout << storage2 << std::endl;
85
86     return 0;
87 }

```
