



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, 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 графиками, 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 предствалаляет собой состоят из одного мастера и множество хранилищ, соединённых общей шиной, через которую происходит общение.

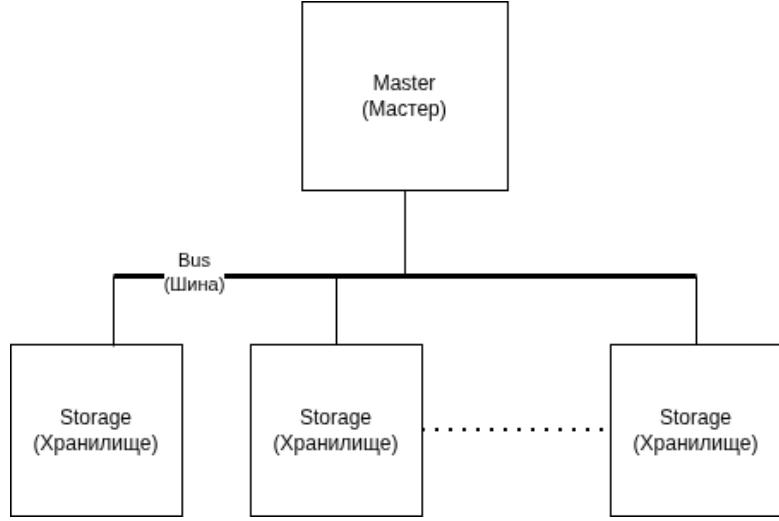


Рис. 1: Упрощённая архитектура

1.3 Программная реализация

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

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

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

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

1.3.1 Структура проекта и основные зависимости

Языком разработки был выбран C++ в силу его низкоуровневости и скорости, а также с намерением в дальнейшем внедрить эти наработки в

графовую БД на С++ научного руководителя.

Проект организован в виде набора заголовочных файлов (header files), что соответствует современным подходам разработки на С++. 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
19     NodeKey<KeyType>& rhs) {
20     return lhs.key_value < rhs.key_value;
21 }
22 friend bool operator>(const NodeKey<KeyType>& lhs, const
23     NodeKey<KeyType>& rhs) {
24     return rhs < lhs;
25 }
26 friend bool operator==(const NodeKey<KeyType>& lhs, const
27     NodeKey<KeyType>& rhs) {
28     return lhs.key_value == rhs.key_value;
29 }
30 friend bool operator!=(const NodeKey<KeyType>& lhs, const
31     NodeKey<KeyType>& rhs) {
32     return !(lhs == rhs);
33 };

```

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

Класс Edge

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

Листинг 2: Класс `Edge` в `graph.hpp`

```
1 template <typename KeyType>
```

```

2 class Edge {
3 private:
4     float weight;
5     bool directional;
6     std::map<std::string, Parameter> parameters;
7     NodeKey<KeyType> from;
8     NodeKey<KeyType> to;
9 public:
10    float get_weight() const {
11        return weight;
12    }
13    bool is_directional() const {
14        return directional;
15    }
16    const NodeKey<KeyType>& get_from() const {
17        return from;
18    }
19    const NodeKey<KeyType>& get_to() const {
20        return to;
21    }
22
23    NodeKey<KeyType> get_other(const NodeKey<KeyType>& node) const {
24        if (node == to) {
25            return from;
26        } else if (node == from) {
27            return to;
28        }
29
30        return node;
31    }
32
33    NodeKey<KeyType> get_other(NodeKey<KeyType>* node) const {
34        if (*node == to) {
35            return from;
36        } else if (*node == from) {
37            return to;
38        } else {
39            return *node;
40        }
41    }

```

42 };

Класс Node

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

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

```
1 template <typename KeyType>
2 class Node {
3 private:
4     NodeKey<KeyType> key;
5 public:
6     std::map<std::string, Parameter> parameters;
7     std::map<NodeKey<KeyType>, Edge<KeyType>> edges;
8 }
9     NodeKey<KeyType> get_key() const {
10         return key;
11     }
12
13     void add_edge(Edge<KeyType> new_edge) {
14         NodeKey<KeyType> current_key = this->get_key();
15         edges[new_edge.get_other(current_key)] = new_edge;
16     }
17
18     void add_edges(std::map<NodeKey<KeyType>, Edge<KeyType>>
19 new_edges) {
20         edges.merge(new_edges);
21     }
22
23     bool remove_edge_to(NodeKey<KeyType> neighbour_key) {
24         return edges.erase(neighbour_key);
25     }
26
27     bool remove_edge(Edge<KeyType> removing_edge) {
28         NodeKey<KeyType> current_key = this->get_key();
29         return
30         edges.erase(removeing_edge.get_other(current_key));
31     }
32 }
```

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>&
5                                         node) = 0;
6     virtual int send_add_node(const Node<KeyType>& node) = 0;
7     virtual bool send_add_node(const Node<KeyType>& node, int
8                                storage_id) = 0;
9     virtual bool send_remove_node(const NodeKey<KeyType>& node)
10        = 0;
11    virtual bool send_remove_node(const NodeKey<KeyType>& node,
12                                  int storage_id) = 0;
13    virtual int ask_who_has(int asker_id, NodeKey<KeyType> key)
14        = 0;
15    virtual void announce_add(NodeKey<KeyType> key, int
16                             storage_id, std::set<Edge<KeyType>> edges) = 0;
17    virtual void announce_remove(NodeKey<KeyType> key, int
18                                 storage_id) = 0;
19    // запрашивает у source вершины, соседствующие с target
20    virtual std::set<Node<KeyType>>
21    ask_neighbours_to_storage(int source, int target) = 0;
22    // запрашивает у source рёбра, идущие в target
23    virtual std::set<Edge<KeyType>> ask_edges_to_storage(int
24                                              source, int target) = 0;
25};
```

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>>>>
10    external_edges;
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
2     StorageNode& node){
3     Key key = node.get_key();
4     std::cout << storage_id << " adding " << key.key_value <<
5     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
external_edges[neighbours_storage_id][neighbor_key][key] =
edge;
29          external_edges_to_announce.insert(edge);
30      }
}

```

```

31             // Если сосед нигде не найден - игнорируем (возможно, будет
32             // добавлен позже)
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 =
43     add_node(node);
44     if (!external_edges.has_value()) {
45         return false;
46     }
47     bus->announce_add(node.get_key(), storage_id,
48     external_edges.value());
49     return true;
50 };

```

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>,
9          Edge<KeyType>>::iterator edge_it = node.edges.begin();
10        edge_it != node.edges.end(); ++edge_it) {
11         Key other_key = edge_it->second.get_other(key);
12     }
13 }

```

```

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 mapу 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) {
    const std::map<Key, Edge<KeyType>>& local_node_edges =
node_edges_it->second;
    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) {
        Key local_key = local_node_edges_it->first;
        typename std::map<Key, StorageNode>::const_iterator
node_it = nodes.find(local_key);
        if (node_it != nodes.end()) {
            result.insert(node_it->second);
        }
    }
}
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) {  

        result.insert(edges_it->second);  

    }  

}  

}  

return result;  

}

```

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 работает 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.

Листинг 11: Методы расчёта g_v

```

1 private:
2 float calculate_gv(const Node<KeyType>& node,
3                     std::set<Edge<KeyType>> boundary_edges) const {
4     NodeKey<KeyType> this_key = node.get_key();
5     float internal_edges_weight = 0;
6     float external_edges_weight = 0;

```

```

6
7     std :: map<NodeKey<KeyType>, Edge<KeyType>>
8         boundary_edges_map;
9     for (typename std :: set<Edge<KeyType>>::const_iterator
10        edge_it = boundary_edges.begin();
11        edge_it != boundary_edges.end(); ++edge_it) {
12         NodeKey<KeyType> other = edge_it->get_other(this_key);
13         if (!(other == this_key)) {
14             boundary_edges_map[other] = *edge_it;
15         }
16     }
17
18     for (typename std :: map<NodeKey<KeyType>,
19           Edge<KeyType>>::const_iterator edge_it = node.edges.begin();
20           edge_it != node.edges.end(); ++edge_it) {
21         const NodeKey<KeyType>& neighbor_key = edge_it->first;
22         const Edge<KeyType>& edge = edge_it->second;
23         if (boundary_edges_map.find(neighbor_key) !=
24             boundary_edges_map.end()) {
25             external_edges_weight += edge.get_weight();
26         } else {
27             internal_edges_weight += edge.get_weight();
28         }
29     }
30
31     return internal_edges_weight - external_edges_weight;
32 }
33
34 public:
35     std :: map<int, std :: map<Node<KeyType>, float>> calculate_gvs(int
36         storage1, int storage2) const {
37         std :: map<int, std :: map<Node<KeyType>, float>> result;
38         // Получаем граничные вершины для обоих хранилищ
39         std :: set<Node<KeyType>> boundary_nodes1 =
40             bus->ask_neighbours_to_storage(storage1, storage2);
41         std :: set<Node<KeyType>> boundary_nodes2 =
42             bus->ask_neighbours_to_storage(storage2, storage1);
43
44         std :: set<Edge<KeyType>> boundary_edges1 =
45             bus->ask_edges_to_storage(storage1, storage2);
46         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,
44                                boundary_edges1);
45    }
46
47    result[storage2] = std::map<Node<KeyType>, float>();
48    for (it = boundary_nodes2.begin(); it != boundary_nodes2.end(); ++it) {
49        const Node<KeyType>& node = *it;
50        result[storage2][node.get_key()] = calculate_gv(node,
51                                boundary_edges2);
52    }
53    return result;
54};

```

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 =
6         get_negative_gvs(calculate_gvs(storage1, storage2));
7     do {
8         typename std::map<int,
9             std::set<Node<KeyType>>>::iterator map_it;
10        for (map_it = negative_gvs.begin(); map_it != negative_gvs.end(); ++map_it) {
11            int this_storage = map_it->first;
12            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
13         set_it;
14         for (set_it = nodes.begin(); set_it != nodes.end();
15             ++set_it) {
16             const Node<KeyType>& node = *set_it;
17             Node<int> removed =
18                 bus->request_node(node.get_key());
19             bus->send_remove_node(removed.get_key(),
20             this_storage);
21             bus->send_add_node(removed, other_storage);
22         }
23     }
24     ++iteration;
25     negative_gvs = get_negative_gvs(calculate_gvs(storage1,
26         storage2));
27     } while (iteration < iterations_limit &&
28         !negative_gvs.empty());
29 }

```

1.4 Тестирование

Для тестирования создаётся простой граф показанный на рисунке 2

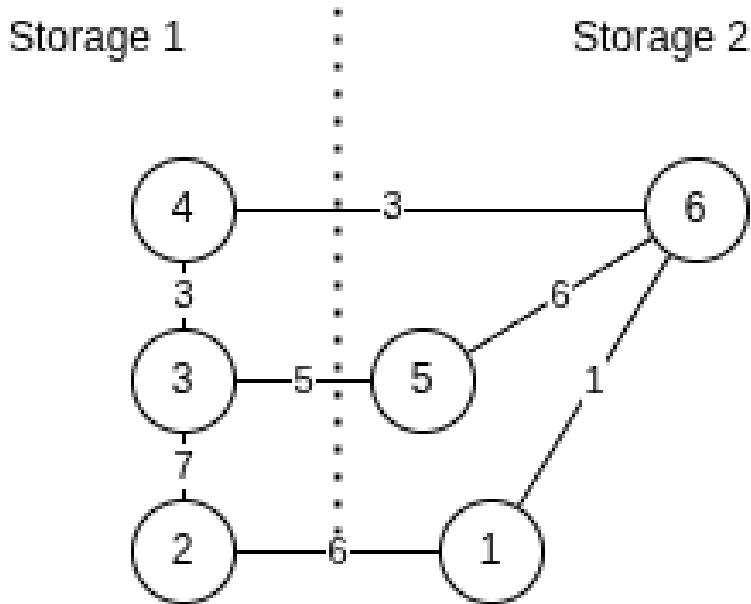


Рис. 2: Граф для тестирования

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:

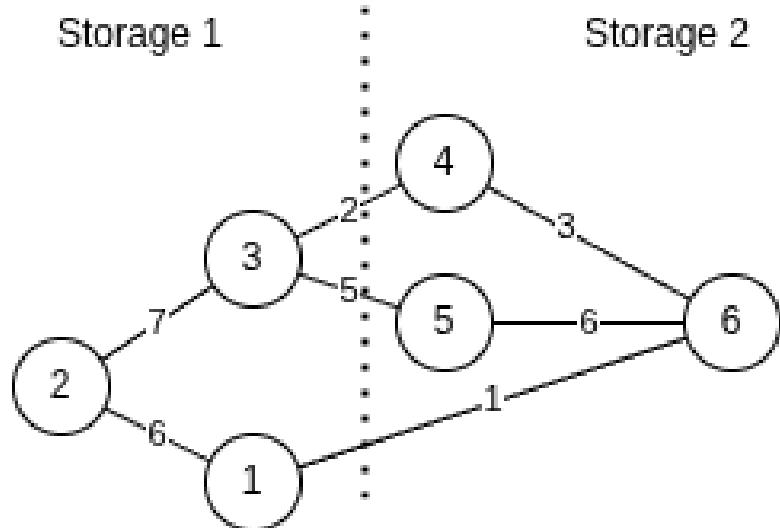


Рис. 3: Оптимизированный граф

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 объёмаами 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. C. 47–63.
- [2] Fennel: Streaming Graph Partitioning for Massive Scale Graphs / Charalampos Tsourakakis, Christos Gkantsidis, Bozidar Radunovic [и др.]. 2014. C. 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
32     NodeKey<KeyType>& rhs) {
33         return lhs.key_value < rhs.key_value;
34     }
35     friend bool operator>(const NodeKey<KeyType>& lhs, const
36     NodeKey<KeyType>& rhs) {
```

```

36     return rhs < lhs;
37 }
38
39 friend bool operator==(const NodeKey<KeyType>& lhs, const
40     NodeKey<KeyType>& rhs) {
41     return lhs.key_value == rhs.key_value;
42 }
43
44 friend bool operator!=(const NodeKey<KeyType>& lhs, const
45     NodeKey<KeyType>& rhs) {
46     return !(lhs == rhs);
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
72         NodeKey<KeyType>& to_node)
73         : weight(1), directional(false), from(from_node),
74         to(to_node) {}

```

```

73
74     Edge(const NodeKey<KeyType>& from_node, const
75         NodeKey<KeyType>& to_node, int edge_weight)
76         : weight(edge_weight), directional(false),
77         from(from_node), to(to_node) {}
78
79     Edge(const NodeKey<KeyType>& from_node, const
80         NodeKey<KeyType>& to_node,
81         float edge_weight, bool is_directional)
82         : weight(edge_weight), directional(is_directional),
83         from(from_node), to(to_node) {}
84
85     Edge(const NodeKey<KeyType>& from_node, const
86         NodeKey<KeyType>& to_node,
87         float edge_weight, bool is_directional, const
88         std::map<std::string, Parameter>& params)
89         : weight(edge_weight), directional(is_directional),
90         parameters(params),
91         from(from_node), to(to_node) {}
92
93     Edge(const Edge& other)
94         : weight(other.weight), directional(other.directional),
95         parameters(other.parameters), from(other.from),
96         to(other.to) {}
97
98     float get_weight() const {
99         return weight;
100    }
101    bool is_directional() const {
102        return directional;
103    }
104    const std::map<std::string, Parameter>& get_parameters()
105    const {
106        return parameters;
107    }
108    const NodeKey<KeyType>& get_from() const {
109        return from;
110    }
111    const NodeKey<KeyType>& get_to() const {
112        return to;
113    }
114}

```

```

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)
145     const {
146         if (node == to) {
147             return from;
148         } else if (node == from) {
149             return to;
150         }
151     }
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
165     Edge<KeyType>& edge) {
166         os << "Edge(from: " << edge.get_from().key_value
167         << ", to: " << edge.get_to().key_value
168         << ", weight: " << edge.get_weight();
169
170         if (edge.is_directional()) {
171             os << ", directional";
172         }
173
174         // Добавляем информацию о параметрах, если они есть
175         const std::map<std::string, Parameter>& params =
176         edge.get_parameters();
177         if (!params.empty()) {
178             os << ", parameters: {";
179             typename std::map<std::string,
Parameter>::const_iterator it = params.begin();
180             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>,
211           Edge<KeyType>> _edges ) :
212         key( _key ) , edges( _edges ) {};
213
214     Node( const Node& other )
215         : key( other .key ) ,
216           parameters( other .parameters ) ,
217           edges( other .edges ) {};
218
219     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>>
228 new_edges) {
229         edges.merge(new_edges);
230     }
231
232     bool remove_edge_to(NodeKey<KeyType> neighbour_key) {
233         return edges.erase(neighbour_key);
234     }
235
236     bool remove_edge(Edge<KeyType> removing_edge) {
237         NodeKey<KeyType> current_key = this->get_key();
238         return
239             edges.erase(remove_edge.get_other(current_key));
240     }
241
242     Node& operator=(const Node& other) {
243         if (this != &other) {
244             key = other.key;
245             parameters = other.parameters;
246             edges = other.edges;
247         }
248         return *this;
249     }
250
251     bool operator<(const Node<KeyType>& other) const {
252         return key < other.key;
253     }
254
255     friend std :: ostream& operator<<(std :: ostream& os, const
256 Node<KeyType>& node) {
257         os << "Node(key: " << node.get_key().key_value;
258
259         // 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>&
11        node) = 0;
12    virtual int send_add_node(const Node<KeyType>& node) = 0;
13    virtual bool send_add_node(const Node<KeyType>& node, int
14        storage_id) = 0;
15    virtual bool send_remove_node(const NodeKey<KeyType>& node)
16        = 0;
17    virtual bool send_remove_node(const NodeKey<KeyType>& node,
18        int storage_id) = 0;
19    virtual int ask_who_has(int asker_id, NodeKey<KeyType> key)
20        = 0;
21    virtual void announce_add(NodeKey<KeyType> key, int
22        storage_id, std::set<Edge<KeyType>> edges) = 0;
23    virtual void announce_remove(NodeKey<KeyType> key, int
24        storage_id) = 0;
25    // запрашивает у source вершины, соседствующие с target
26    virtual std::set<Node<KeyType>>
27        ask_neighbours_to_storage(int source, int target) = 0;
28    // запрашивает у source рёбра, идущие в target
29    virtual std::set<Edge<KeyType>> ask_edges_to_storage(int
30        source, int target) = 0;
31 };
32
33
34 #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
61 bool send_remove_node(const NodeKey<KeyType>& node, int
62 storage_id) override {
63     if (storages.find(storage_id) == storages.end()) {
64         return false;
65     }
66     return storages[storage_id]→remove_node_and_announce(node);
67 }
68
69 int ask_who_has(int asker_id, NodeKey<KeyType> key) override{
70     for (typename std::map<int, Storage<KeyType*>*>::iterator it
71         = storages.begin(); it != storages.end(); ++it) {
72         if (it→second != nullptr && it→second→has_node(key))
73         {
74             return it→first;
75         }
76     }
77     return -1;
78 }
79
80 void announce_add(NodeKey<KeyType> key, int storage_id,
81 std::set<Edge<KeyType>> edges) override {
82     for (typename std::map<int, Storage<KeyType*>*>::iterator it
83         = storages.begin(); it != storages.end(); ++it) {
84         if (it→second != nullptr) {
85             it→second→get_add_announcement(key, storage_id,
86             edges);

```

```

80         };
81     }
82 };
83
84 void announce_remove(NodeKey<KeyType> key, int storage_id)
85     override {
86     for (typename std::map<int, Storage<KeyType*>*>::iterator it
87         = storages.begin(); it != storages.end(); ++it) {
88         if (it->second != nullptr) {
89             it->second->get_remove_announcement(key,
90             storage_id);
91         };
92     };
93 };
94
95 // запрашивает у source вершины, соседствующие с target
96 std :: set<Node<KeyType>> ask_neighbours_to_storage(int source,
97 int target) {
98     if (storages.find(source) == storages.end()) {
99         return std :: set<Node<KeyType>>();
100    }
101    return
102        storages[source]->get_nodes_with_neighbors_in_storage(target);
103 }
104
105 // запрашивает у source рёбра, идущие в target
106 std :: set<Edge<KeyType>> ask_edges_to_storage(int source, int
107 target) {
108     if (storages.find(source) == storages.end()) {
109         return std :: set<Edge<KeyType>>();
110    }
111    return storages[source]->get_all_edges_to_storage(target);
112 }
113
114 #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>>>>
22         external_edges;
23     IBus<KeyType> *bus = nullptr;
24
25 public:
26     Storage(int id)
27         : storage_id(id) {}
28     Storage(int id, std::map<Key, StorageNode> _nodes)
29         : storage_id(id), nodes(_nodes) {}
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,
34                               std::set<Edge<KeyType>> edges) {
35         if (announcer_id == storage_id) return;
36
37         for (typename std::set<Edge<KeyType>>::iterator edge_it =
38               edges.begin(); edge_it != edges.end(); ++edge_it) {
39             Key other_key = edge_it->get_other(key);
40             typename std::map<Key, StorageNode>::iterator it =
41               nodes.find(other_key);
42             if (it != nodes.end()) {

```

```

40         it->second.add_edge(*edge_it);
41         typename std::map<Key, std::map<Key,
42             Edge<KeyType>>>::iterator it2 =
43             external_edges[announcer_id].find(key);
44             if (it2 == external_edges[announcer_id].end()){
45                 external_edges[announcer_id][key] =
46                     std::map<Key, Edge<KeyType>>();
47             }
48             external_edges[announcer_id][key][other_key] =
49                 *edge_it;
50     };
51 }
52
53 void get_remove_announcement(Key key, int announcer_id) {
54     if (announcer_id == storage_id) return;
55
56     typename std::map<int, std::map<Key, std::map<Key,
57         Edge<KeyType>>>>::iterator storage_it =
58         external_edges.find(announcer_id);
59     if (storage_it == external_edges.end()) {
60         return;
61     }
62
63     std::map<Key, std::map<Key, Edge<KeyType>>>& external_map =
64     storage_it->second;
65     typename std::map<Key, std::map<Key,
66         Edge<KeyType>>>::iterator external_node_it =
67         external_map.find(key);
68     if (external_node_it == external_map.end()) {
69         return;
70     }
71
72     std::map<Key, Edge<KeyType>>& local_to_ext_map =
73     external_node_it->second;
74
75     // Удаляем рёбра из локальных узлов
76     for (typename std::map<Key, Edge<KeyType>>::iterator
77         local_nodes_it = local_to_ext_map.begin();
78         local_nodes_it != local_to_ext_map.end();
79         ++local_nodes_it) {

```

```

71         Key local_key = local_nodes_it->first;
72         typename std::map<Key, StorageNode>::iterator node =
73             nodes.find(local_key);
74         if (node != nodes.end()) {
75             node->second.remove_edge_to(key);
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
88 StorageNode& node){
89     Key key = node.get_key();
90     std::cout << storage_id << " adding " << key.key_value <<
91     std::endl;
92     typename std::map<Key, StorageNode>::iterator it =
93         nodes.find(key);
94     if (it != nodes.end()) {
95         return std::nullopt;
96     }
97     nodes[key] = node;
98
99     std::set<Edge<KeyType>> external_edges_to_announce;
100    for (typename std::map<Key, Edge<KeyType>>::const_iterator
101        edge_it = node.edges.begin(); edge_it != node.edges.end();
102        ++edge_it) {
103        const Key& neighbor_key = edge_it->first;
104        const Edge<KeyType>& edge = edge_it->second;
105        std::cout << storage_id << " looks for " <<
106        neighbor_key.key_value << std::endl;
107        // Ищем соседа в текущем хранилище
108        typename std::map<Key, StorageNode>::iterator it2 =
109        nodes.find(neighbor_key);
110        if (it2 != nodes.end()) {

```

```

104         std::cout << storage_id << " node " <<
105         neighbor_key.key_value << " is inside, no asking" <<
106         std::endl;
107         // Сосед найден в этом же хранилище - добавляем обратное ребро
108         it2->second.add_edge(edge);
109     } else {
110         std::cout << storage_id << " node " <<
111         neighbor_key.key_value << " is outside, asking" <<
112         std::endl;
113         // Сосед не найден - спрашиваем у шины, где он находится
114         int neighbours_storage_id =
115         bus->ask_who_has(storage_id, neighbor_key);
116         std::cout << storage_id << " asked for " <<
117         neighbor_key.key_value << " answer: " <<
118         neighbours_storage_id << std::endl;
119         if (neighbours_storage_id != -1) {
120             // Сохраняем внешнее ребро
121             external_edges[neighbours_storage_id][neighbor_key][key] =
122             edge;
123             external_edges_to_announce.insert(edge);
124         }
125         // Если сосед нигде не найден - игнорируем (возможно, будет
126         // добавлен позже)
127     }
128 }
129
130     return external_edges_to_announce;
131 };
132
133 bool add_node_and_announce(const StorageNode& node) {
134     if (bus == nullptr) {
135         return false;
136     };
137     std::optional<std::set<Edge<KeyType>>> external_edges =
138     add_node(node);
139     if (!external_edges.has_value()) {
140         return false;
141     }
142     bus->announce_add(node.get_key(), storage_id,
143     external_edges.value());
144
145     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>,
145           Edge<KeyType>>::iterator edge_it = node.edges.begin();
146           edge_it != node.edges.end(); ++edge_it) {
147         Key other_key = edge_it->second.get_other(key);
148         typename std::map<Key, StorageNode>::iterator it =
149             nodes.find(other_key);
150         if (it != nodes.end()) {
151             it->second.remove_edge_to(key);
152         }
153     }
154
155     // Удаляем все внешние ребра, связанные с этим узлом
156     typename std::map<int, std::map<Key, std::map<Key,
157     Edge<KeyType>>>::iterator ext_storage_it;
158     for (ext_storage_it = external_edges.begin();
159           ext_storage_it != external_edges.end(); ) {
160         std::map<Key, std::map<Key, Edge<KeyType>>& ext_nodes =
161             ext_storage_it->second;
162
163         // Удаляем записи, где наш узел является локальным
164         typename std::map<Key, std::map<Key,
165         Edge<KeyType>>>::iterator ext_node_it;
166         for (ext_node_it = ext_nodes.begin(); ext_node_it !=
167               ext_nodes.end(); ) {
168             std::map<Key, Edge<KeyType>>& local_edges =
169             ext_node_it->second;
170
171             // Удаляем все ребра, где локальный ключ - наш удаляемый узел
172             typename std::map<Key, Edge<KeyType>>::iterator
173             edge_it = local_edges.find(key);
174             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,
169         // удаляем запись
170         if (local_edges.empty()) {
171             ext_nodes.erase(ext_node_it++);
172         } else {
173             ++ext_node_it;
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 =
200     nodes.find(key);
201     if (it != nodes.end()) {
202         return &(it->second);
203     }
204     return nullptr;
205 };

```

```

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
229     target_storage_id) const {
230     std::set<StorageNode> result;
231
232     typename std::map<int, std::map<Key, std::map<Key,
233         Edge<KeyType>>>>::const_iterator storage_it =
234         external_edges.find(target_storage_id);
235     if (storage_it == external_edges.end()) {
236         return result;
237     }
238
239     const std::map<Key, std::map<Key, Edge<KeyType>>>&
240     node_edges = storage_it->second;
241     for (typename std::map<Key, std::map<Key,
242         Edge<KeyType>>>::const_iterator node_edges_it =
243         node_edges.begin(); node_edges_it != node_edges.end();
244         ++node_edges_it) {
245         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
281 typename std::map<Key, StorageNode>::const_iterator end() const
282     {
283     return nodes.end();
284 }
285
286 typename std::map<Key, StorageNode>::const_iterator cbegin()
287     const {
288     return nodes.cbegin();
289 }
290
291 typename std::map<Key, StorageNode>::const_iterator cend()
292     const {
293     return nodes.cend();
294 }
295
296 friend std::ostream& operator<<(std::ostream& os, const
297     Storage<KeyType>& storage) {
298     os << "Storage(id: " << storage.get_id();
299     if (storage.empty()) {

```

```

300         os << ", empty";
301     } else {
302         os << ", nodes: " << storage.size() << " {";
303
304         typename std::map<Key, StorageNode>::const_iterator it
305 = storage.begin();
306         if (it != storage.end()) {
307             os << std::endl << " " << it->second;
308             ++it;
309         }
310         for (; it != storage.end(); ++it) {
311             os << " " << std::endl << " " << it->second;
312         }
313         os << std::endl << "}";
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,
320 Edge<KeyType>>>::const_iterator storage_it =
321         storage.external_edges.begin();
322
323     if (storage_it != storage.external_edges.end()) {
324         os << std::endl << " to storage " <<
325         storage_it->first << ": {";
326
327         const std::map<Key, std::map<Key, Edge<KeyType>>>&
328         ext_nodes = storage_it->second;
329
330         typename std::map<Key, std::map<Key,
331 Edge<KeyType>>>::const_iterator node_it = ext_nodes.begin();
332
333         if (node_it != ext_nodes.end()) {
334             os << std::endl << "      external node " <<
335             node_it->first.key_value << ": [";
336
337             const std::map<Key, Edge<KeyType>>& local_edges
338 = node_it->second;
339
340             typename std::map<Key,
341 Edge<KeyType>>::const_iterator edge_it = local_edges.begin();

```

```

333
334         if (edge_it != local_edges.end()) {
335             os << std::endl << "      local node " <<
336             edge_it->first.key_value
337             << " -> " << edge_it->second;
338             ++edge_it;
339         }
340         for (; edge_it != local_edges.end(); ++edge_it)
341     {
342             os << ", " << std::endl << "      local node "
343             << edge_it->first.key_value
344             << " -> " << edge_it->second;
345         }
346         os << std::endl << "    ] ";
347         ++node_it;
348     }
349     const std::map<Key, Edge<KeyType>>& local_edges
350     = node_it->second;
351     typename std::map<Key,
352     Edge<KeyType>>::const_iterator edge_it = local_edges.begin();
353
354         if (edge_it != local_edges.end()) {
355             os << std::endl << "      local node " <<
356             edge_it->first.key_value
357             << " -> " << edge_it->second;
358             ++edge_it;
359         }
360         for (; edge_it != local_edges.end(); ++edge_it)
361     {
362             os << ", " << std::endl << "      local node "
363             << edge_it->first.key_value
364             << " -> " << edge_it->second;
365         }
366         os << std::endl << "    ] ";
367         ++storage_it;

```

```

365         }
366
367         for ( ; storage_it != storage.external_edges.end() ;
368             ++storage_it) {
368             os << "," << std::endl << " to storage " <<
369             storage_it->first << ":" {";
370
370             const std::map<Key, std::map<Key, Edge<KeyType>>>&
371             ext_nodes = storage_it->second;
372             typename std::map<Key, std::map<Key,
373             Edge<KeyType>>>::const_iterator node_it = ext_nodes.begin();
374
374             if (node_it != ext_nodes.end()) {
375                 os << std::endl << "     external node " <<
375                 node_it->first.key_value << ":" [";
376
376             const std::map<Key, Edge<KeyType>>& local_edges
377             = node_it->second;
378             typename std::map<Key,
379             Edge<KeyType>>::const_iterator edge_it = local_edges.begin();
380
380             if (edge_it != local_edges.end()) {
381                 os << std::endl << "         local node " <<
381                 edge_it->first.key_value
382                     << " -> " << edge_it->second;
383                     ++edge_it;
384             }
384             for ( ; edge_it != local_edges.end(); ++edge_it)
385             {
385                 os << "," << std::endl << "         local node "
386                 << 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 "
392             << 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           {
404               os << ", " << std::endl << "      local node
" << edge_it->first.key_value
405                           << " -> " << edge_it->second;
406               }
407               os << std::endl << "    ] ";
408           }
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 =
49             it -> second ;
50         typename std :: map<Node<KeyType> , float >::const_iterator
51             it2 ;
52         for ( it2 = nodes . begin () ; it2 != nodes . end () ; ++it2 ) {
53             std :: cout << "storage: " << it -> first << " node: "
54             << it2 -> first << " gv: " << it2 -> second << std :: endl ;
55             if ( it2 -> second < 0 ) {
56                 result [ it -> first ]. insert ( it2 -> first );
57             }
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
68     storage1 , int storage2 ) const {
69     std :: map<int , std :: map<Node<KeyType> , float >> result ;
70     // Получаем граничные вершины для обоих хранилищ
71     std :: set<Node<KeyType>> boundary_nodes1 =
72         bus -> ask_neighbours_to_storage ( storage1 , storage2 );
73     std :: set<Node<KeyType>> boundary_nodes2 =
74         bus -> ask_neighbours_to_storage ( storage2 , storage1 );
75
76     std :: set<Edge<KeyType>> boundary_edges1 =
77         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();
105             ++set_it) {
106             const Node<KeyType>& node = *set_it;
107             Node<int> removed =
108                 bus->request_node(node.get_key());
109             bus->send_remove_node(removed.get_key());
110             bus->send_add_node(removed, other_storage);
111         }
112     ++iteration;
113     negative_gvs = get_negative_gvs(calculate_gvs(storage1,
114     storage2));
115 } while (iteration < iterations_limit &&
116 !negative_gvs.empty());
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);
27     node2.add_edge(edge1_2);
28     Edge<int> edge1_6(1, 6, 1); node1.add_edge(edge1_6);
29     node6.add_edge(edge1_6);
30     Edge<int> edge2_3(2, 3, 7); node2.add_edge(edge2_3);
31     node3.add_edge(edge2_3);
32     Edge<int> edge3_4(3, 4, 2); node3.add_edge(edge3_4);
33     node4.add_edge(edge3_4);
34     Edge<int> edge4_6(4, 6, 3); node4.add_edge(edge4_6);
35     node6.add_edge(edge4_6);
36     Edge<int> edge3_5(3, 5, 5); node3.add_edge(edge3_5);
37     node5.add_edge(edge3_5);
38     Edge<int> edge5_6(5, 6, 6); node5.add_edge(edge5_6);
39     node6.add_edge(edge5_6);
40
41     bus.send_add_node(node1, 2);
42     bus.send_add_node(node2, 1);
43     bus.send_add_node(node3, 1);
44     bus.send_add_node(node4, 1);
45     bus.send_add_node(node5, 2);
46     bus.send_add_node(node6, 2);
47
48     std::cout << storage1 << std::endl;
49     std::cout << storage2 << std::endl;
50
51     std::cout << "Neighbours of 1:" << std::endl;
52     for (const auto& element : bus.ask_neighbours_to_storage(1,
53         2)) {
54         std::cout << element << " ";
55     }
56     std::cout << std::endl;
57
58     std::cout << "Neighbours of 2:" << std::endl;
59     for (const auto& element : bus.ask_neighbours_to_storage(2,
60         1)) {
61         std::cout << element << " ";
62     }

```

```

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 }
```
