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 «Программное обеспечение ЭВМ и информационные технологии»

Лабораторная работа № 5
по дисциплине «Анализ алгоритмов»

Тема Конвейерная обработка данных

Студент Ильченко Е. А.

Группа ИУ7-54Б

Преподаватели Волкова Л.Л.

Москва, 2025

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Целью данной работы является исследование принципов конвейерной обработки данных на примере последовательной и параллельной реализаций алгоритма Дейкстры и конвейерной схемы обработки заявок.

Для достижения поставленной цели необходимо решить следующие задачи:

— разработать структуру конвейера обработки заявок и условия завершения потоков, соответствующих обслуживающим устройствам;

— реализовать конвейерную обработку заявок;

— привести схемы конвейерной и линейной обработки заявок;

— выполнить функциональное тестирование;

— провести сравнительный анализ времени выполнения конвейерной обработки заявок.

1 Аналитическая часть

1.1 Конвейерная обработка данных

Конвейер [1, 2] – организация вычислений, при которой увеличивается количество выполняемых операций за единицу времени за счёт использования принципов параллельности. Конвейеризация в общем случае основана на разделении подлежащей исполнению функции на более мелкие части, называемые ступенями, и выделении для каждой из них отдельного блока аппаратуры или программного потока.

Обработка данных разбивается на несколько этапов, организуется передача результатов от одного этапа к следующему, и становится возможным совмещение выполнения разных этапов для различных данных. Производительность при этом возрастает за счёт того, что одновременно на различных ступенях конвейера обрабатываются разные элементы данных. При этом конвейерная обработка, как правило, не сокращает время обработки одной отдельной заявки, но увеличивает пропускную способность системы в целом.

1.2 Структура конвейерной обработки заявок

В качестве объекта конвейерной обработки в данной работе выступают заявки на поиск кратчайших путей в графе. Каждая заявка содержит:

— имя файла с графом в формате DOT;

— имя стартовой вершины;

— список имён целевых вершин, для которых необходимо найти кратчайшие пути.

Для обработки набора таких заявок используется конвейер из трёх обслуживающих устройств (ОУ):

— ОУ1 загружает граф из файла, находит индексы стартовой и целевых вершин во внутреннем представлении графа и дополняет заявку этой информацией;

— ОУ2 запускает параллельный алгоритм Дейкстры для соответствующего графа и стартовой вершины, формируя массив расстояний до всех вершин и массив предков для восстановления путей;

— ОУ3 по полученным расстояниям и массиву предков восстанавливает кратчайшие пути до целевых вершин, выбирает путь минимальной длины и формирует текстовый отчёт по заявке;

Между ступенями расположены блокирующие очереди заявок: если очередь пуста, поток, обслуживающий соответствующее ОУ, блокируется до появления новой заявки. Благодаря этому одновременно могут обрабатываться несколько заявок, находящихся на разных этапах конвейера, что повышает пропускную способность системы по сравнению с линейной обработкой заявок.

Вывод

В аналитической части:

— рассмотрено понятие конвейерной обработки данных и её влияние на пропускную способность системы;

— описана структура конвейера обработки заявок на запуск алгоритма Дейкстры.

2 Конструкторская часть

2.1 Требования к реализации

К программе предъявлены следующие функциональные требования.

Входные данные

— граф в формате DOT, содержащий ориентированный взвешенный граф;

— стартовая вершина – имя вершины, от которой ищутся пути;

— целевые вершины – список имён вершин, до которых ищутся пути;

— количество заявок N – целое число, определяющее, сколько раз алгоритм будет запущен для заданного графа.

Выходные данные

— имя стартовой вершины;

— массив имён целевых вершин;

— время суммарного выполнения алгоритма для N запусков (выводится в микросекундах в стандартный поток вывода);

— объект с расстояниями до каждой целевой вершины;

— объект с информацией о кратчайшем пути, содержащий: имя целевой вершины с минимальным расстоянием, длину кратчайшего пути, массив имён вершин, составляющих путь от стартовой до целевой вершины;

— лог о работе программы.

Функциональные требования

— поддержка загрузки графов из файлов формата DOT;

— реализация последовательного алгоритма Дейкстры;

— реализация параллельного алгоритма Дейкстры с использованием нативных потоков;

— реализация конвейерной обработки набора заявок на поиск кратчайших путей с использованием трёх обслуживающих устройств и блокирующих очередей между ними;

— обработка некорректных входных данных с выводом сообщений об ошибках;

— вывод результатов в человекочитаемом текстовом формате (отчёт по каждой заявке в отдельный файл) для последующей обработки и анализа;

— замер времени выполнения алгоритмов.

Режимы работы

— линейный режим – выполнение последовательного алгоритма Дейкстры;

— конвейерный режим – последовательная обработка набора заявок на трёх обслуживающих устройствах, где на второй ступени выполняется параллельный алгоритм Дейкстры.

2.2 Разработка алгоритмов

Раздел содержит схемы алгоритмов, описывающие следующие алгоритмы: алгоритм линейной обработки 2.1, алгоритм конвейерной обработки 2.2, алгоритм обслуживающего устройства 1 2.3, алгоритм обслуживающего устройства 2 2.4, алгоритм обслуживающего устройства 3 2.5.

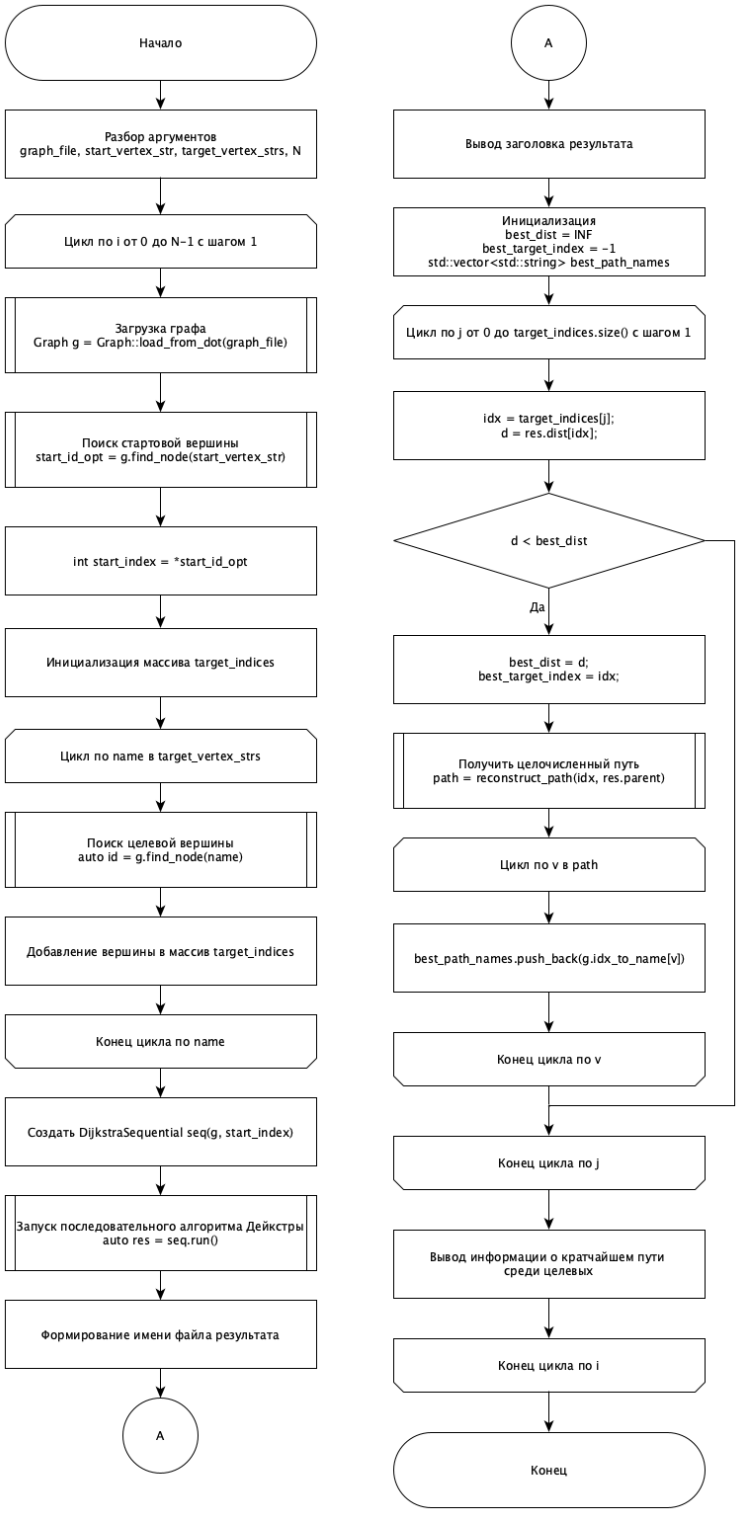

Рисунок 2.1 — Схема линейной обработки

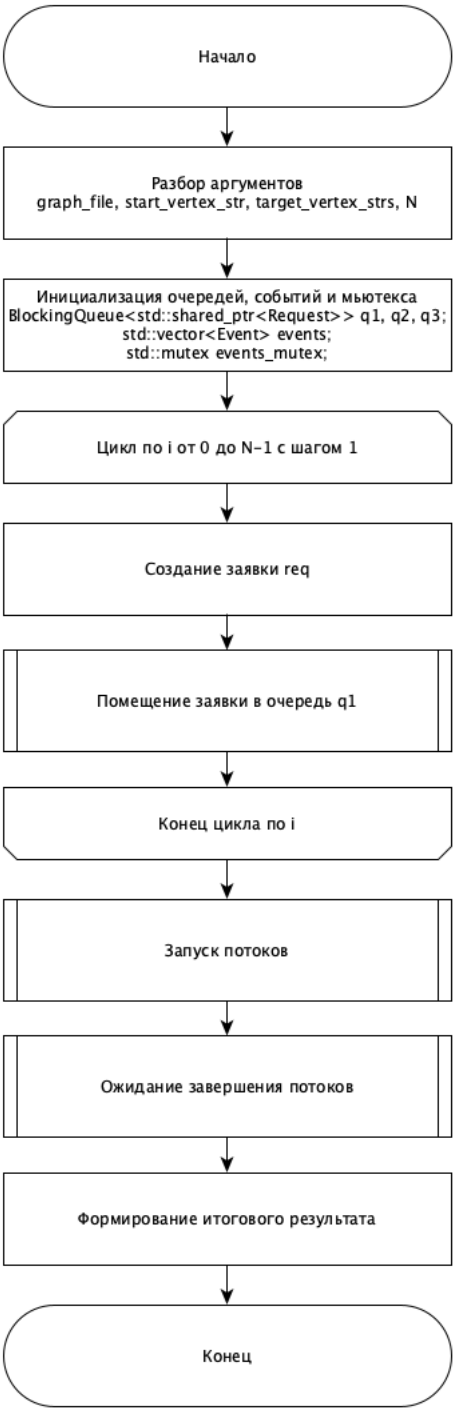

Рисунок 2.2 — Схема конвейерной обработки

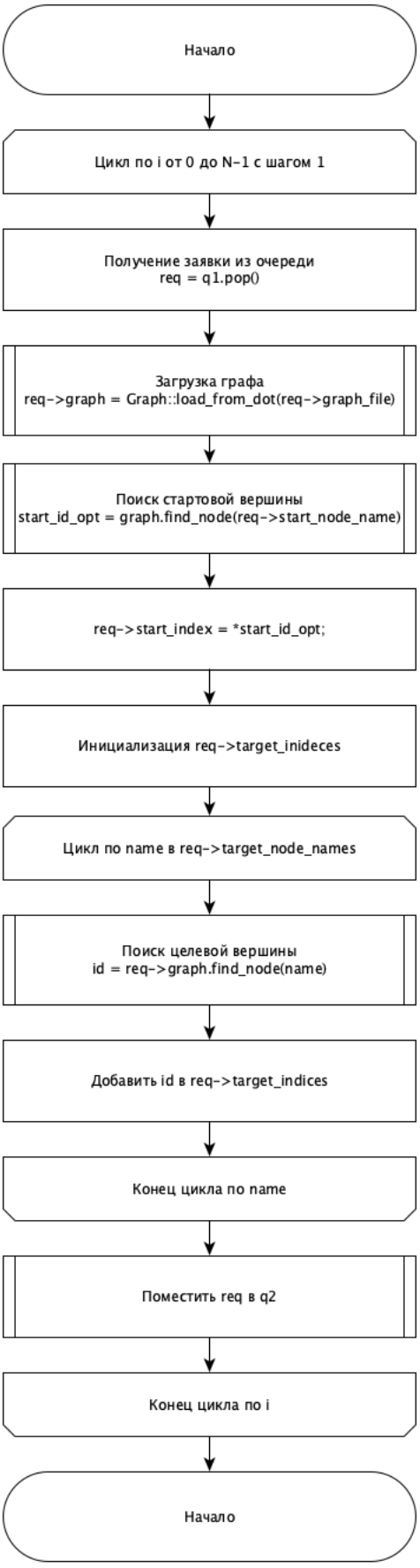

Рисунок 2.3 — Схема обслуживающего устройства 1

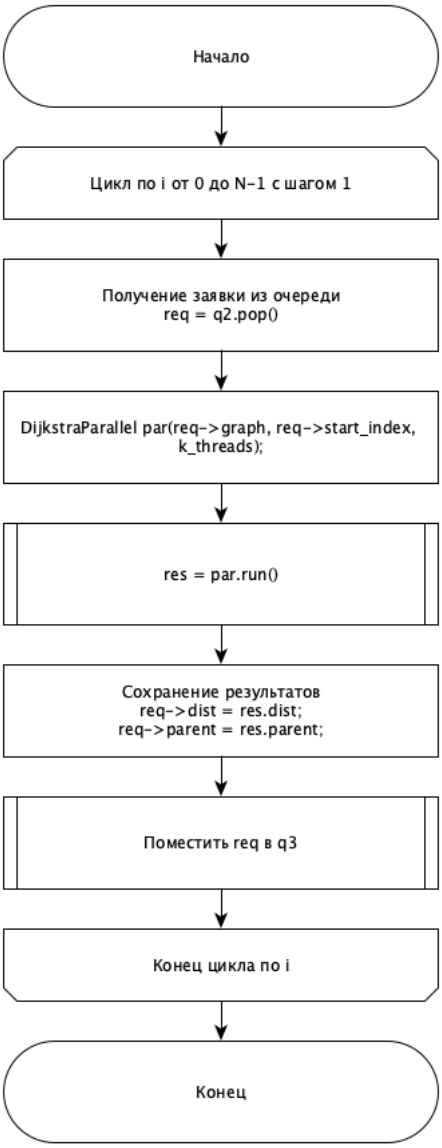

Рисунок 2.4 — Схема обслуживающего устройства 2

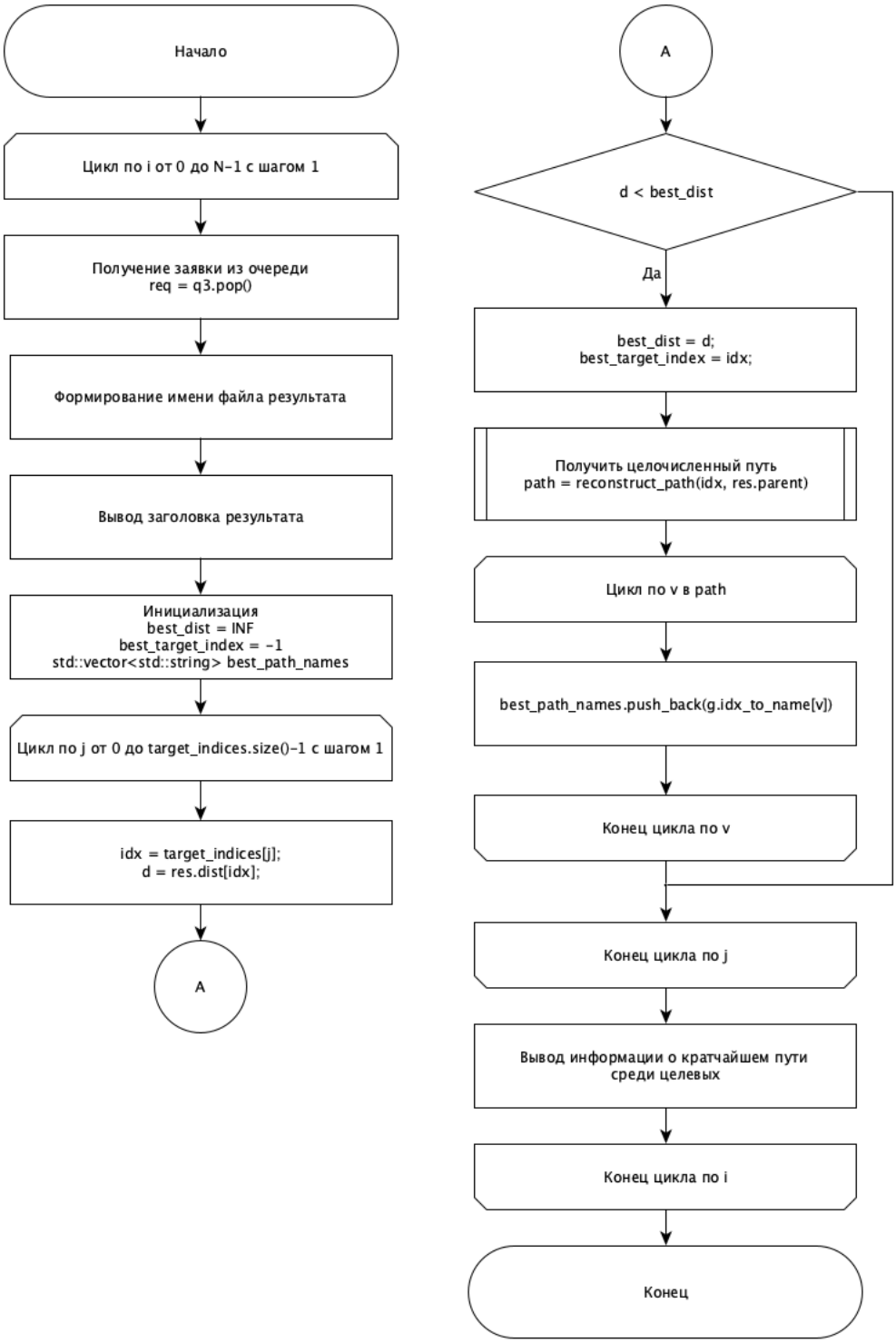

Рисунок 2.5 — Схема обслуживающего устройства 3

Вывод

В данном разделе были разработаны алгоритмы линейной, конвейерной обработки и алгоритмы обслуживающего устройства 1, обслуживающего устройства 2, обслуживающего устройства 3. Для каждого из них представлены схемы алгоритмов, описывающие логику работы.

3 Технологическая часть

3.1 Средства реализации

Для реализации алгоритмов последовательной и конвейерной обработки заявок был выбран язык программирования C++20, поскольку он соответствует требованиям лабораторной работы и предоставляет необходимые средства для работы с нативными потоками.

Измерение времени выполнения алгоритмов осуществляется с использованием стандартной библиотеки <chrono> и вспомогательного класса Timer. Разработка проводилась в среде CLion, сборка – с помощью CMake.

3.2 Инструменты разработки

3.2.1 Используемые библиотеки

Все реализации используют только стандартную библиотеку C++20:

— std::vector, std::string, std::unordered_map – хранение графа и рабочих структур;

— std::queue – реализация очередей заявок;

— std::thread – создание и управление нативными потоками для конвейерной обработки;

— std::mutex, std::lock_guard, std::unique_lock – синхронизация доступа к разделяемым данным;

— std::condition_variable – блокирующее ожидание появления заявок в очередях конвейера;

— std::filesystem – работа с путями и директориями (поиск корня проекта, создание папки результатов);

— std::chrono – замер времени работы алгоритмов.

3.3 Описание используемых структур данных

3.3.1 Структура графа

Для представления ориентированного взвешенного графа используется класс Graph, в котором вершины хранятся по индексам, а рёбра – в виде списков смежности:

Листинг 3.1 — Структура графа

```
class Graph {
public:
    std::vector<std::vector<std::pair<int, uint32_t>>> adj;
    std::unordered_map<std::string, int> name_to_idx;
    std::vector<std::string> idx_to_name;
```

```
    int ensure_node(const std::string &name);
    std::optional<int> find_node(const std::string &name) const;

    void add_edge(int u, int v, uint32_t w);

    size_t size() const { return adj.size(); }

    static Graph load_from_dot(const std::string &path);
};
```

3.3.2 Структура заявки и события конвейера

Для описания заявки, проходящей через конвейер, используется структура Request:

Листинг 3.2 — Структура Request

```
struct Request {
    int id{};
    std::string graph_file;
    std::string start_node_name;
    std::vector<std::string> target_node_names;

    Graph graph;
    int start_index{-1};
    std::vector<int> target_indices;

    std::vector<uint64_t> dist;
    std::vector<int> parent;
};
```

Для журналирования работы конвейера используется перечисление EventType и структура Event:

Листинг 3.3 — Структуры EventType и Event

```
enum class EventType {
    Start,
    End
};

struct Event {
    long long time_us{};
    int request_id{};
    int device_id{};
```

```
    EventType type{};
};
```

3.3.3 Блокирующая очередь заявок

Для связи между стадиями конвейера реализован шаблонный класс `BlockingQueue`, инкапсулирующий очередь, мьютекс и условную переменную:

Листинг 3.4 — Класс BlockingQueue

```
template<typename T>
class BlockingQueue {
public:
    void push(T value) {
        {
            std::lock_guard<std::mutex> lg(m_);
            q_.push(std::move(value));
        }
        cv_.notify_one();
    }

    T pop() {
        std::unique_lock<std::mutex> lk(m_);
        cv_.wait(lk, [&] { return !q_.empty(); });
        T v = std::move(q_.front());
        q_.pop();
        return v;
    }

private:
    std::queue<T> q_;
    std::mutex m_;
    std::condition_variable cv_;
};
```

3.4 Реализация

3.4.1 Последовательная обработка заявок

Фрагмент кода, отвечающий за многократный запуск алгоритма и формирование отчётов:

Листинг 3.5 — Последовательная обработка N заявок

```
for (int i = 0; i < N; ++i) {
```

```
    Graph g = Graph::load_from_dot(graph_file);

    auto start_id_opt = g.find_node(start_vertex_str);
    if (!start_id_opt) {
        throw std::runtime_error("Start node not found in graph");
    }
    int start_index = *start_id_opt;

    std::vector<int> target_indices;
    for (const auto &name: target_vertex_strs) {
        auto id = g.find_node(name);
        if (!id) {
            throw std::runtime_error("Target node not found in graph");
        }
        target_indices.push_back(*id);
    }

    DijkstraSequential seq(g, start_index);
    auto res = seq.run();

    // формирование имени файла и вывод результата опущены для краткост
   и
}
```

3.4.2 Конвейерная обработка заявок

Фрагмент, создающий очереди, формирующий поток заявок и запускающий три обслуживающих устройства:

Листинг 3.6 — Запуск конвейера обработки заявок

```
BlockingQueue<std::shared_ptr<Request>> q1, q2, q3;
std::vector<Event> events;
std::mutex events_mutex;

auto log_event = [&](int device_id, int request_id, EventType type) {
    const auto now = Clock::now();
    long long dt = std::chrono::duration_cast<Microseconds>(now-t0).
   count();
    Event ev{dt, request_id, device_id, type};
    std::lock_guard<std::mutex> lg(events_mutex);
    events.push_back(ev);
};
```

```
// генератор заявок
for (int i = 0; i < N; ++i) {
    auto req = std::make_shared<Request>();
    req->id = i;
    req->graph_file = graph_file;
    req->start_node_name = start_vertex_str;
    req->target_node_names = target_vertex_strs;
    q1.push(req);
}

// запуск трёх этапов конвейера (ou1, ou2, ou3)
std::thread ou1(/* обработка заявок из q1 и передача в q2 */);
std::thread ou2(/* запуск дейкстры и передача в q3 */);
std::thread ou3(/* формирование отчётов по заявкам из q3 */);

ou1.join();
ou2.join();
ou3.join();
```

После завершения конвейера программа сортирует журнал событий по времени и выводит временную диаграмму работы обслуживающих устройств, а также суммарное время работы конвейера.

3.5 Функциональные тесты

Для проверки корректности реализации были проведены функциональные тесты. Тесты выполнялись как в линейном режиме, так и в конвейерном режиме, и результаты обоих режимов сравнивались с ожидаемыми.

Таблица 3.1 — Функциональные тесты для алгоритма поиска кратчайших путей

№	Входные данные	Ожидаемый результат	Линейная обработка	Конвейерная обработка
1	Граф: A->B[2], A->C[1]; Старт: A; Цели: B, C	Расстояния: B=2, C=1; Кратчайший: C=1	Совпадает с ожидаемым	Совпадает с ожидаемым и с линейной обработкой
2	Граф: A->B[3], B->C[1], A->C[5]; Старт: A; Цели: C	Расстояние: C=4; Путь: A->B->C	Совпадает (расстояние и путь)	Совпадает (расстояние и путь)
3	Граф: A->B[1], B->C[1], C->D[1]; Старт: A; Цели: D	Расстояние: D=3; Путь: A->B->C->D	Совпадает	Совпадает
4	Граф: A->B[2], C->D[1]; Старт: A; Цели: D	Расстояние: D=∞	D недостижима (как ожидается)	D недостижима (как ожидается)
5	Граф: A->A[1]; Старт: A; Цели: A	Расстояние: A=0; Путь: A	Совпадает	Совпадает

Во всех тестах результаты линейной и конвейерной обработок совпали между собой и с ожидаемыми значениями, что подтверждает корректность реализации поиска кратчайших путей и отсутствие искажений при прохождении заявок через конвейер.

Вывод

В технологической части были описаны структуры данных, использованные для реализации последовательного и конвейерного алгоритмов поиска кратчайших путей, а также приведены ключевые фрагменты кода.

Реализованы:

— структура графа с хранением рёбер в виде списков смежности;

— классический последовательный алгоритм Дейкстры;

— конвейерная схема обработки набора заявок с тремя обслуживающими устройствами и блокирующими очередями.

Функциональные тесты показали, что результаты последовательной и конвейерной обработок совпадают на различных типах графов, что подтверждает корректность реализованных алгоритмов.

4 Исследовательская часть

4.1 Характеристики ЭВМ

Замеры проводились на устройстве со следующими характеристиками:

— процессор: Apple M4 Pro;

— количество логических ядер: 12;

— количество физических ядер: 12;

— оперативная память: 24 Гб;

— операционная система: macOS Sequoia 15.6.1.

Замеры времени проводились, когда ноутбук был загружен только системными приложениями.

4.2 Замер времени

В данном разделе представлены результаты измерения среднего времени выполнения конвейерного алгоритма обработки заявок.

Замеры проводились для различного количества заявок $N \in \{25, 50, 75, 100, 125\}$ для графов размером от 500, 800, 1000, 1500, 2000 вершин и двух конфигураций исполнения:

— линейный вариант;

— конвейерный вариант.

Для каждой конфигурации замер повторялся 5 раз, в таблице приведено среднее время выполнения в миллисекундах. Результаты приведены в таблице 4.1, графики по таблице представлен на рисунке 4.1.

Таблица 4.1 — Результаты измерения среднего времени выполнения (мс)

N	**Линейный, мс**	**Конвейерный, мс**
25	41122.748	39066.611
50	82146.180	77217.409
75	123156.066	114535.142
100	164241.412	151102.499
125	205278.415	186803.357

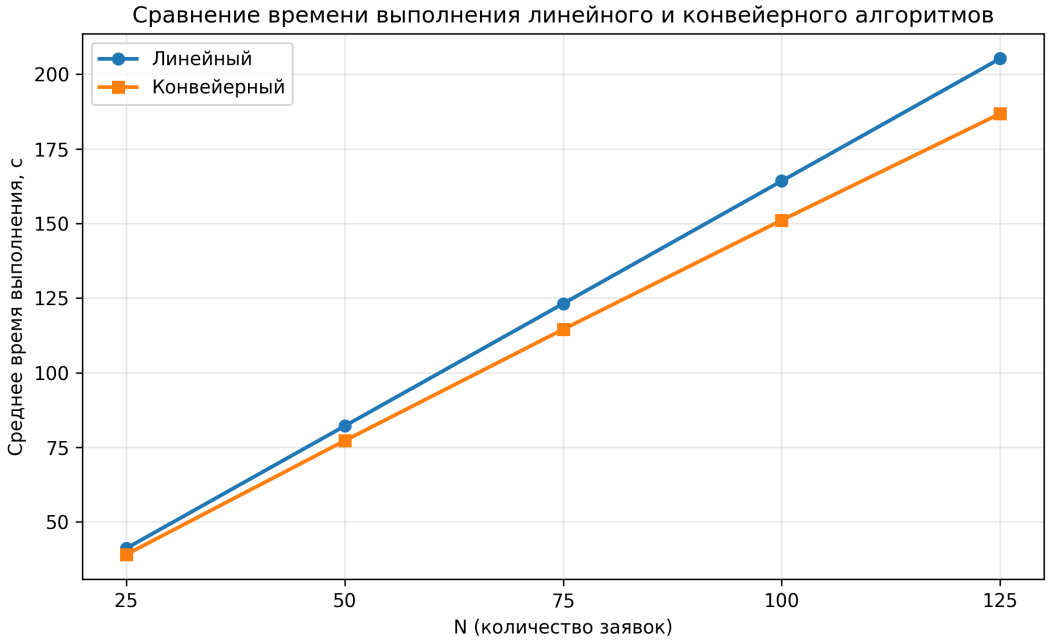

Рисунок 4.1 — Сравнение времени выполнения реализации линейного и конвейерного алгоритмов

4.3 Логирование событий

В листинге 4.1 представлен пример логирования событий для графа на 3000 вершин при количестве заявок $N = 4$. Из логов следует, что события на разных обслуживающих устройствах происходят параллельно. Например, в интервале $[4244982, 4246195]$ мкс одновременно работают ОУ1 (заявка 1) и ОУ2 (заявка 0), а в дальнейшем аналогичное перекрытие наблюдается и между другими заявками.

Анализ временных меток также показывает конвейерный характер обработки: каждая заявка последовательно проходит через ОУ1, ОУ2 и ОУ3, при этом, когда одна заявка уже обрабатывается на ОУ3, другая в это время может находиться на ОУ2, а третья – на ОУ1.

Листинг 4.1 — Логирование событий

```
[15] START | заявка#0 | оу1
[4244972] END | заявка#0 | оу1
[4244975] START | заявка#1 | оу1
[4244982] START | заявка#0 | оу2
[4246195] END | заявка#0 | оу2
[4246200] START | заявка#0 | оу3
[4247599] END | заявка#0 | оу3
[8411120] END | заявка#1 | оу1
[8411122] START | заявка#2 | оу1
[8411129] START | заявка#1 | оу2
[8412229] END | заявка#1 | оу2
[8412233] START | заявка#1 | оу3
```

```
[8413450] END | заявка#1 | оу3
[12593969] END | заявка#2 | оу1
[12593972] START | заявка#3 | оу1
[12593978] START | заявка#2 | оу2
[12595146] END | заявка#2 | оу2
[12595150] START | заявка#2 | оу3
[12595324] END | заявка#2 | оу3
[16874896] END | заявка#3 | оу1
[16874905] START | заявка#3 | оу2
[16876492] END | заявка#3 | оу2
[16876497] START | заявка#3 | оу3
[16877377] END | заявка#3 | оу3
```

Вывод

В данной части были проведены исследования производительности конвейерной архитектуры для линейного и конвейерного вариантов алгоритма обработки заявок. Результаты показывают, что конвейерная обработка обеспечивает сокращение времени выполнения по сравнению с линейной для всех исследованных значений N.

Анализ логов подтверждает параллельную работу обслуживающих устройств и конвейерный характер обработки: несколько заявок одновременно находятся на разных стадиях обслуживания.

ЗАКЛЮЧЕНИЕ

В ходе выполнения лабораторной работы поставленная цель была достигнута, а также были решены следующие задачи:

— разработана структура конвейера обработки заявок и условия завершения потоков, соответствующих обслуживающим устройствам;

— реализована конвейерная обработка заявок и линейный вариант обработки;

— приведены и проанализированы схемы конвейерной и линейной обработки заявок;

— выполнено функциональное тестирование, подтвердившее корректность работы реализованных вариантов обработки;

— проведён сравнительный анализ времени выполнения линейного и конвейерного вариантов обработки заявок, показавший преимущество конвейерной архитектуры по времени выполнения при увеличении числа заявок;

— выполнен анализ логов работы конвейера, подтвердивший параллельную работу обслуживающих устройств и конвейерный характер обработки (одновременное нахождение разных заявок на различных стадиях обработки).

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Конвейерная обработка данных [Электронный ресурс]. – Режим доступа: https://studref.com/636041/ekonomika/konveyernaya_obrabotka_dannyh (дата обращения: 15.12.2025).
2. Patterson D.A., Hennessy J.L. Computer Organization and Design: The Hardware/Software Interface. 5th Edition. – Morgan Kaufmann, 2013. – 800 p.