Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное 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»

ОТЧЕТ

по лабораторной работе № 7

по курсу «Анализ алгоритмов»

на тему: «Алгоритмы поиска»

Студент ИУ7-51Б _____ Д. В. Шубенина
(Группа) (Подпись, дата) (И. О. Фамилия)

Преподаватель _____ Л. Л. Волкова
(Подпись, дата) (И. О. Фамилия)

Преподаватель _____ Ю. В. Строганов
(Подпись, дата) (И. О. Фамилия)

2023 г.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Целью данной лабораторной работы является исследование лучших и худших случаев работы алгоритмов поиска.

Для достижения поставленной цели необходимо решить следующие задачи:

1) описать используемые алгоритмы поиска;
2) описать худший и лучший случаи работы рассмотренных алгоритмов;
3) определить средства программной реализации;
4) реализовать данные алгоритмы поиска;
5) проанализировать алгоритмы по количеству сравнений.

1 Аналитическая часть

В данном разделе приведена информация, касающаяся алгоритма бинарного поиска.

1.1 Алгоритм бинарного поиска

Алгоритм бинарного поиска является алгоритмом поиска значения в отсортированном массиве данных.

Алгоритм бинарного поиска начинает сравнение целевого значения с элементом в середине массива. Если они не равны, половина массива, в которой целевое значение не может находиться, удаляется, и поиск продолжается в оставшейся половине. Затем цикл повторяется: снова берется элемент в середине оставшейся половины, сравнивается с целевым значением, и процесс продолжается до тех пор, пока не будет найдено целевое значение. Если поиск завершается с пустой оставшейся половиной, значит, целевое значение отсутствует в массиве.

В лучшем случае искомое значение находится в середине массива — тогда алгоритм отрабатывает за 1 итерацию.

В худшем случае искомый элемент отсутствует в массиве — тогда количество итераций равно $ceil(\log_2(n) + 1)$.

2 Конструкторская часть

В данном разделе будут представлены схемы последовательной и параллельной работы стадий конвейера.

2.1 Требования к программному обеспечению

К программному обеспечению предъявлен ряд требований:

1) наличие интерфейса для выбора действий;
2) возможность обработки файлов MIDI;
3) возможность выбора линейной или конвейерной реализаций алгоритма.

2.2 Разработка алгоритмов

На рисунке 2.1 представлена схема последовательного нечеткого алгоритма кластеризации c-средних.

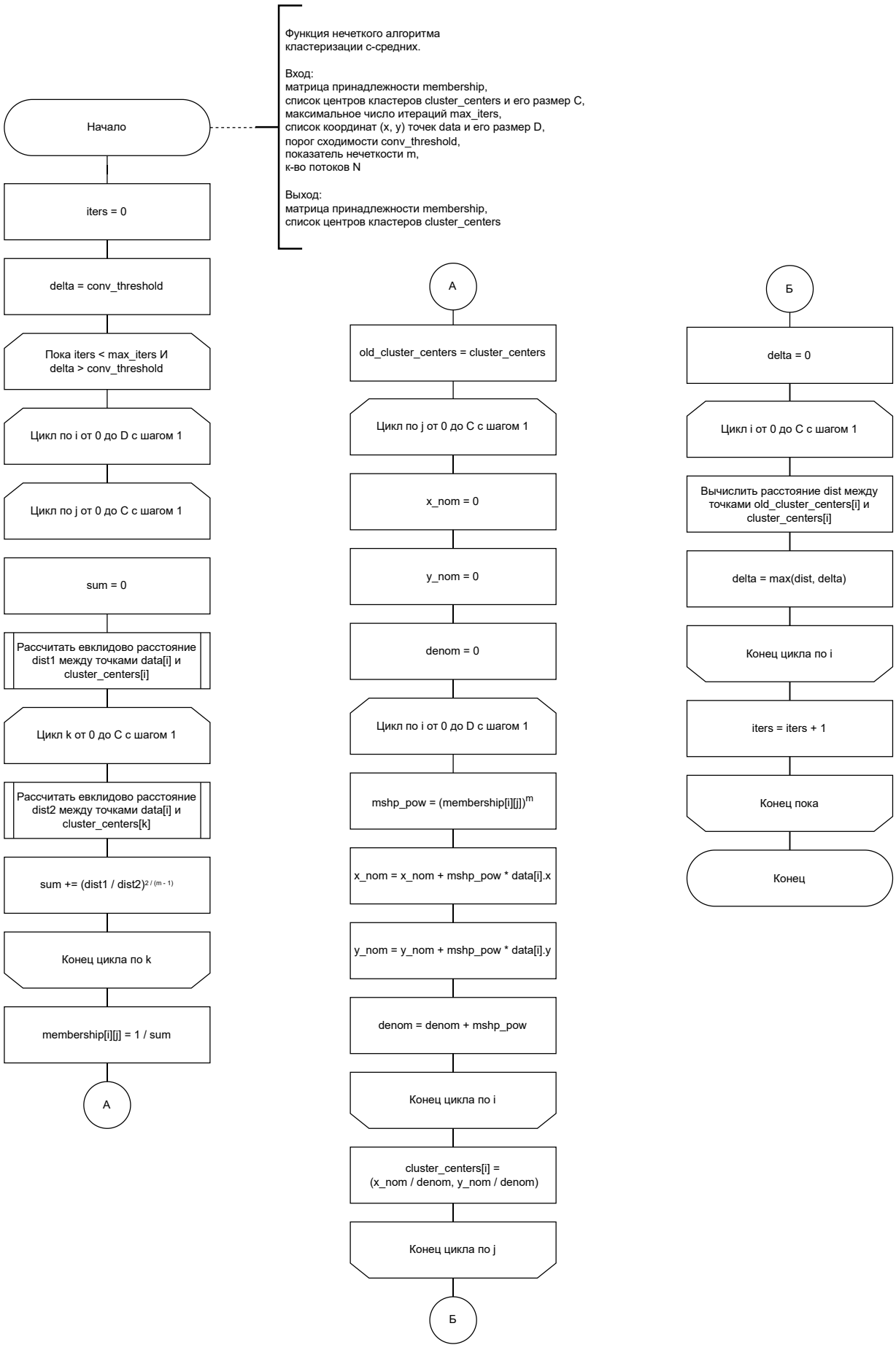

Рисунок 2.1 – Схема последовательного нечеткого алгоритма c-средних

На рисунке 2.2 представлена схема алгоритма линейной обработки датасетов файлов MIDI.

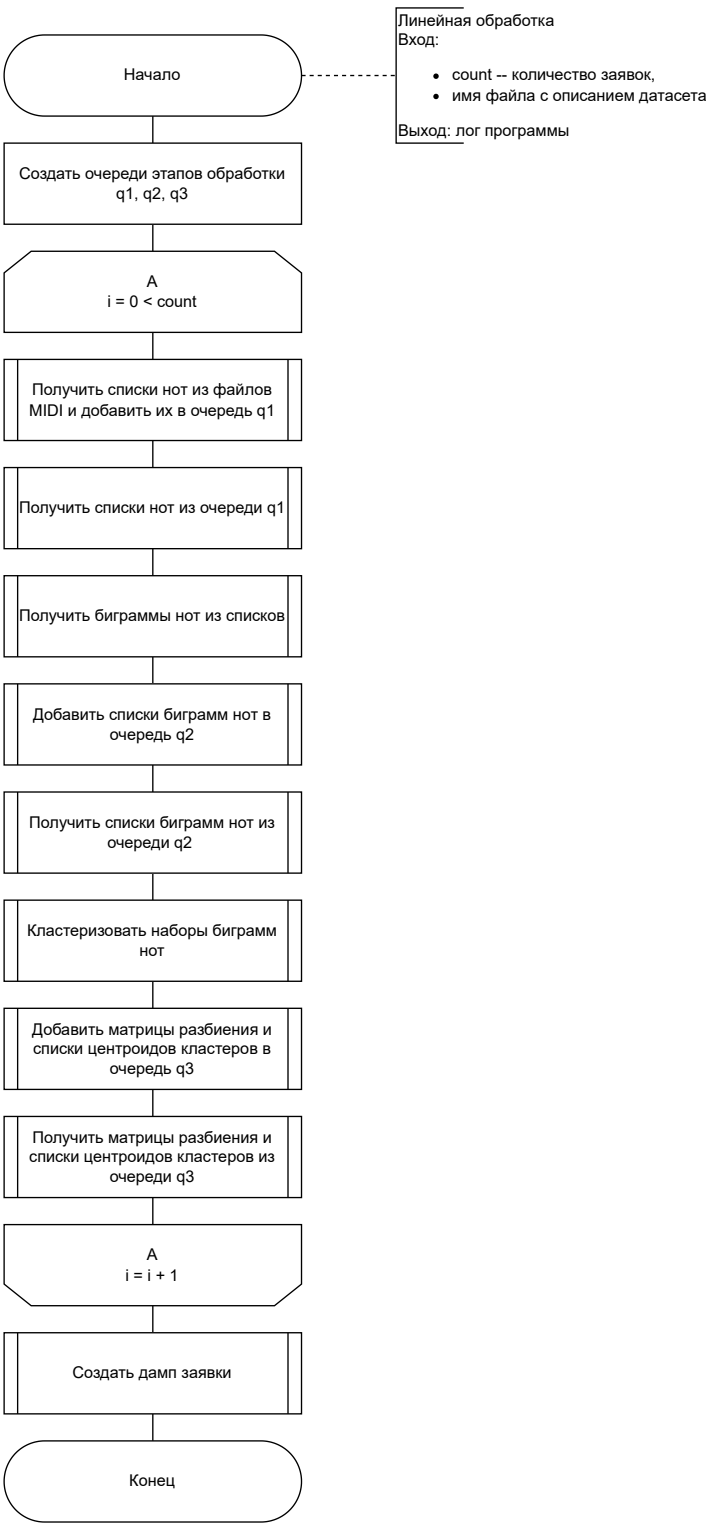

Рисунок 2.2 – Схема алгоритма линейной обработки датасетов файлов MIDI

На рисунке 2.3 представлена схема алгоритма главного потока конвейерной обработки датасетов файлов MIDI.

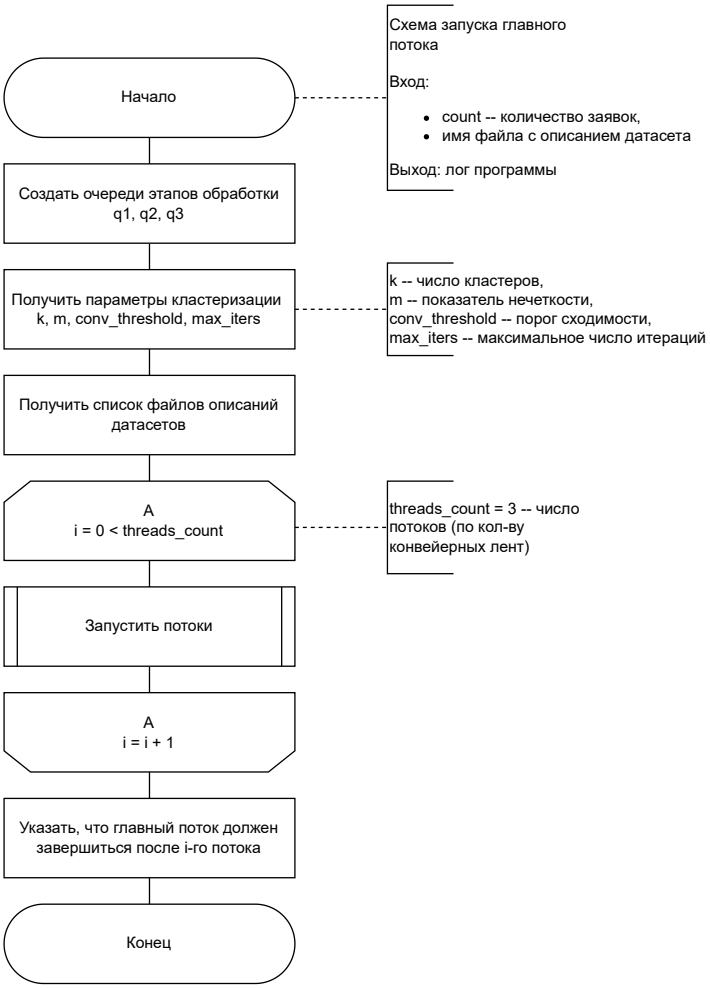

Рисунок 2.3 – Схема алгоритма главного потока конвейерной обработки датасетов файлов MIDI

На рисунке 2.4 представлена схема алгоритма потока, выполняющего извлечение нот из MIDI файлов.

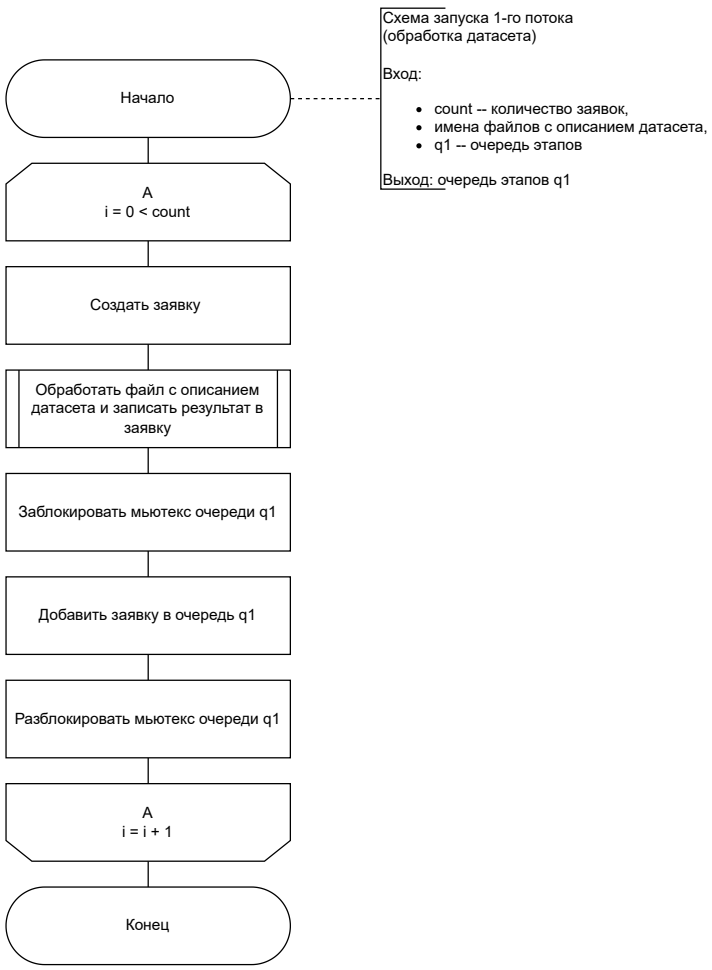

Рисунок 2.4 – Схема алгоритма потока, выполняющего извлечение нот из MIDI файлов

На рисунке 2.5 представлена схема алгоритма потока, выполняющего извлечение биграмм нот из обработанных на предыдущем этапе файлов.

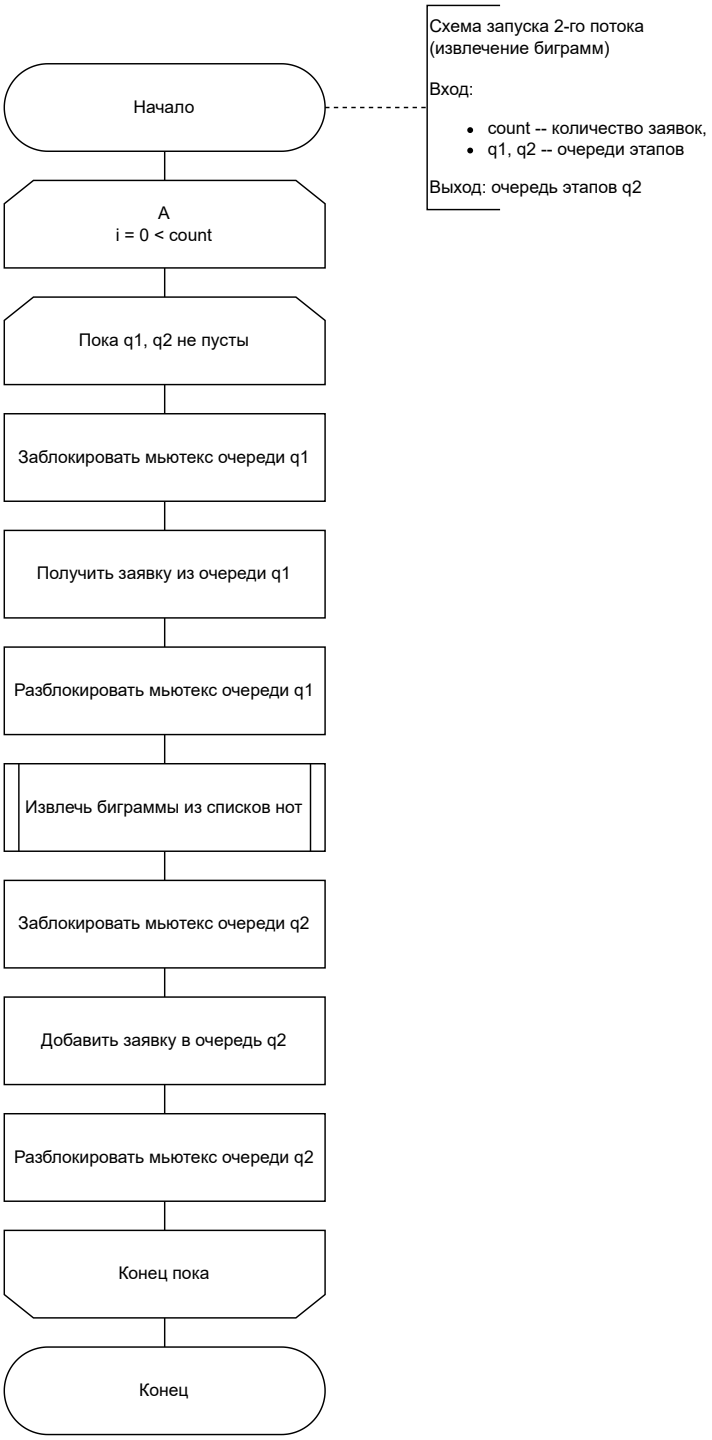

Рисунок 2.5 – Схема алгоритма потока, выполняющего извлечение биграмм нот из обработанных на предыдущем этапе файлов

На рисунке 2.6 представлена схема алгоритма потока, выполняющего кластеризацию биграмм нот.

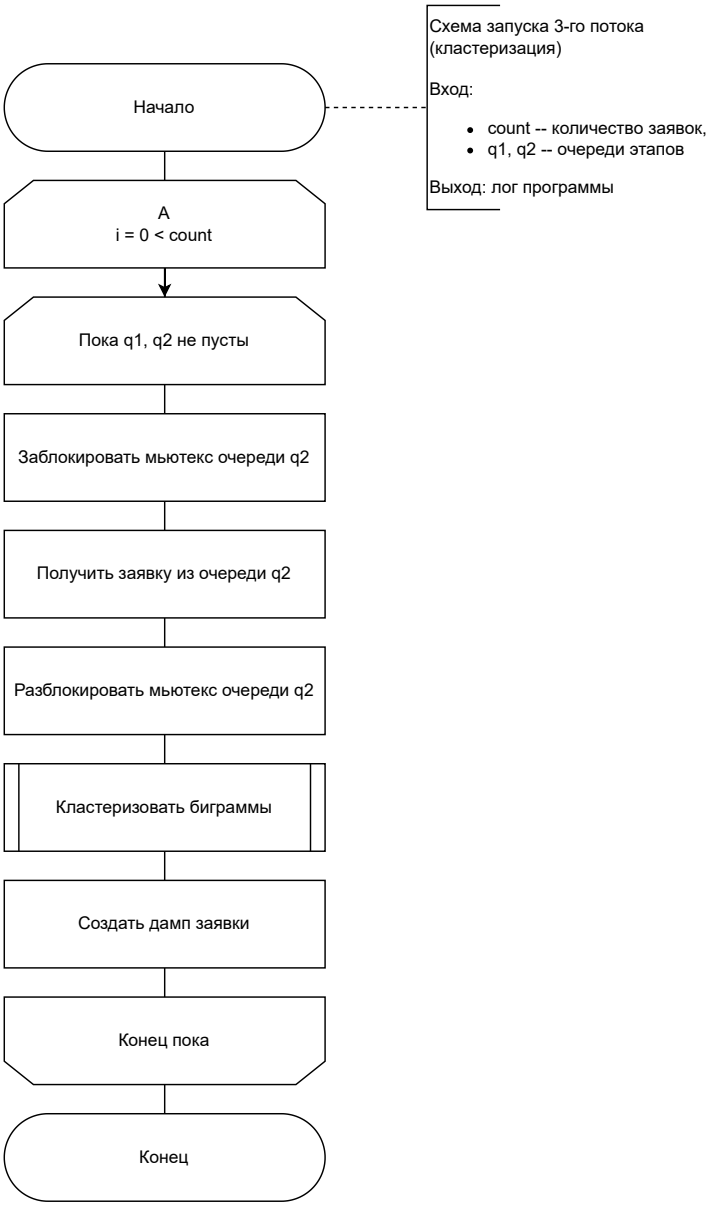

Рисунок 2.6 – Схема алгоритма потока, выполняющего кластеризацию биграмм нот

Вывод

В данном разделе были перечислены требования к программному обеспечению и построены схемы рассматриваемых алгоритмов.

3 Технологическая часть

В данном разделе описаны средства реализации программного обеспечения, а также листинги и функциональные тесты.

3.1 Средства реализации

В качестве языка программирования, используемого при написании данной лабораторной работы, был выбран C++ [1], так как в нем имеется контейнер `std::vector`, представляющий собой динамический массив данных произвольного типа, и библиотека `<ctime>` [**cpp-ctime**], позволяющая производить замеры процессорного времени. Также выбранный язяк программирования предоставляет возможность работы с:

— потоками (класс `thread` [**cpp-thread**]);

— мьютексами (класс `mutex` [**cpp-mutex**]);

Обработка файлов MIDI производилась с помощью библиотеки MidiFile [**midifile**].

3.2 Сведения о модулях программы

Данная программа разбита на следующие модули:

— `main.cpp` — файл, содержащий точку входа в программу;

— `algorithms.cpp` — файл, содержащий реализации алгоритмов, используемых на различных стадиях конвейера;

— `ts_queue.cpp` — файл, содержащий реализацию потокобезопасной очереди;

— `pipeline.cpp` — файл, содержащий функции конвейерной обработки;

— `utils.cpp` — файл, содержащий вспомогательные функции;

— `measure.cpp` — файл, содержащий функции, замеряющие процессорное время выполнения реализуемых алгоритмов.

3.3 Реализация алгоритмов

На листинге 3.1 представлены реализации алгоритмов извлечения нот из файлов MIDI, извлечения биграмм и кластеризации c-средних.

Листинг 3.1 – Реализации алгоритмов извлечения нот из файлов MIDI, извлечения биграмм и кластеризации c-средних

```
std::vector<note_vector_t> read_dataset(const std::string
   &descr_file)
{
    auto [id, n, filenames] = read_descr_file(descr_file);
    std::vector<note_vector_t> notes;
    for (const auto &filename : filenames)
    {
        smf::MidiFile midifile;
        midifile.read(filename);

        note_vector_t temp_notes;
        for (int track = 0; track < midifile.getTrackCount();
           track++)
        {
            for (int event = 0; event < midifile[track].size();
               event++)
            {
                if (midifile[track][event].isNoteOn())
                {
                    temp_notes.push_back(
                        midifile[track][event][1]);
                }
            }
        }
        notes.emplace_back(std::move(temp_notes));
    }
    return notes;
}

std::vector<point_vec_t> extract_bigrams(const
   std::vector<note_vector_t> &notes)
{
    std::vector<point_vec_t> bigrams;
    for (const auto &note_set : notes)
    {
```

```
32          point_vec_t temp_bigrams;
33          for (size_t i = 1; i < note_set.size(); ++i)
34          {
35              std::vector<double> bigram{
36                  (double)note_set[i - 1], (double)note_set[i]};
37              temp_bigrams.emplace_back(bigram);
38          }
39          bigrams.emplace_back(temp_bigrams);
40      }
41      return bigrams;
42  }
43
44  void c_means(
45          membership_t &membership, point_vec_t &cluster_centers,
46          const point_vec_t &data,
47          double m, double conv_threshold, int max_iters)
48  {
49      int iters = 0;
50      double delta = conv_threshold + 1.0;
51      while (iters < max_iters && delta > conv_threshold)
52      {
53          for (size_t i = 0; i < data.size(); ++i)
54          {
55              for (size_t j = 0; j < cluster_centers.size(); ++j)
56              {
57                  double sum = 0.0;
58                  double dist1 = sqrt(pow(data[i][0] -
                        cluster_centers[j][0], 2) +
59                                      pow(data[i][1] -
                                          cluster_centers[j][1],
                                          2));
60                  for (size_t k = 0; k < cluster_centers.size();
                        ++k)
61                  {
62                      double dist2 = sqrt(pow(data[i][0] -
                            cluster_centers[k][0], 2) +
63                                          pow(data[i][1] -
                                              cluster_centers[k][1],
                                              2));
64                      sum += pow(dist1 / dist2, 2.0 / (m - 1.0));
65                  }
```

```
66                  membership[i][j] = 1.0 / sum;
67              }
68          }
69          auto old_cluster_centers = cluster_centers;
70          for (size_t j = 0; j < cluster_centers.size(); ++j)
71          {
72              double x_nom = 0.0, y_nom = 0.0, denom = 0.0;
73              for (size_t i = 0; i < data.size(); ++i)
74              {
75                  double membership_pow_m = pow(membership[i][j],
                        m);
76                  x_nom += membership_pow_m * data[i][0];
77                  y_nom += membership_pow_m * data[i][1];
78                  denom += membership_pow_m;
79              }
80              cluster_centers[j] = {x_nom / denom, y_nom / denom};
81          }
82          delta = 0.0;
83          for (size_t i = 0; i < cluster_centers.size(); ++i)
84          {
85              double distance = sqrt(pow(old_cluster_centers[i][0]
                    - cluster_centers[i][0], 2) +
86                                     pow(old_cluster_centers[i][1]
                                        - cluster_centers[i][1],
                                        2));
87              if (distance > delta)
88                  delta = distance;
89          }
90          ++iters;
91      }
92  }
```

На листинге 3.2 представлена реализация линейного алгоритма обработки набора файлов MIDI.

Листинг 3.2 – Реализация линейного алгоритма обработки набора файлов MIDI

```
1  void consequent(
2      int req_cnt,
3      const std::vector<std::string> &datasets,
4      std::tuple<int, double, double, int> cls_params,
5      bool verbose)
```

```
{
    auto [k, m, conv_threshold, max_iters] = cls_params;
    std::vector<std::unique_ptr<stages_t>> pool;
    for (int i = 0; i < req_cnt; ++i)
    {
        stages_t *s = new stages_t;

        clock_gettime(CLOCK_REALTIME, &s->parsed.op_start);
        s->parsed.notes = read_dataset(datasets[i]);
        clock_gettime(CLOCK_REALTIME, &s->parsed.op_end);

        clock_gettime(CLOCK_REALTIME, &s->embedded.op_start);
        s->embedded.embeddings =
           extract_bigrams(s->parsed.notes);
        clock_gettime(CLOCK_REALTIME, &s->embedded.op_end);

        clock_gettime(CLOCK_REALTIME, &s->clusterized.op_start);
        for (const auto &embed : s->embedded.embeddings)
        {
            auto [mshp, cc] = init_structures(embed.size(), k);
            c_means(mshp, cc, embed, m, conv_threshold,
               max_iters);
            s->clusterized.results.emplace_back(std::move(mshp),
               std::move(cc));
        }
        clock_gettime(CLOCK_REALTIME, &s->clusterized.op_end);
        if (verbose)
            pool.emplace_back(s);
    }
    if (verbose)
        dump_pool(pool, "cons.txt");
}
```

На листинге 3.3 представлена реализация конвейерного алгоритма обработки набора файлов MIDI.

Листинг 3.3 – Реализация конвейерного алгоритма обработки набора файлов MIDI

```
static void service_01(
    int req_cnt,
    const std::vector<std::string> &datasets,
    ts_queue<stages_t *> &q1)
```

```
{
    for (int i = 0; i < req_cnt; ++i)
    {
        stages_t *s = new stages_t();
        clock_gettime(CLOCK_REALTIME, &s->parsed.op_start);
        s->parsed.notes = read_dataset(datasets[i]);
        clock_gettime(CLOCK_REALTIME, &s->parsed.op_end);
        q1.push(s);
    }
}

static void service_02(
    int req_cnt,
    ts_queue<stages_t *> &q1,
    ts_queue<stages_t *> &q2)
{
    for (int i = 0; i < req_cnt; ++i)
    {
        stages_t *s = q1.pop();

        clock_gettime(CLOCK_REALTIME, &s->embedded.op_start);
        s->embedded.embeddings =
           extract_bigrams(s->parsed.notes);
        clock_gettime(CLOCK_REALTIME, &s->embedded.op_end);
        q2.push(s);
    }
}

static void service_03(
    int req_cnt,
    std::tuple<int, double, double, int> cls_params,
    ts_queue<stages_t *> &q2,
    std::vector<std::unique_ptr<stages_t>> &pool,
    bool verbose)
{
    for (int i = 0; i < req_cnt; ++i)
    {
        stages_t *s = q2.pop();
        auto [k, m, conv_threshold, max_iters] = cls_params;

        clock_gettime(CLOCK_REALTIME, &s->clusterized.op_start);
```

```
45          for (const auto &embed : s->embedded.embeddings)
46          {
47              auto [mshp, cc] = init_structures(embed.size(), k);
48              c_means(mshp, cc, embed, m, conv_threshold,
                    max_iters);
49              s->clusterized.results.emplace_back(std::move(mshp),
                    std::move(cc));
50          }
51          clock_gettime(CLOCK_REALTIME, &s->clusterized.op_end);
52          pool.emplace_back(s);
53      }
54  }
55
56  void concurrent(
57      int req_cnt,
58      const std::vector<std::string> &datasets,
59      std::tuple<int, double, double, int> cls_params,
60      bool verbose)
61  {
62      std::vector<std::unique_ptr<stages_t>> pool;
63      ts_queue<stages_t *> q1;
64      ts_queue<stages_t *> q2;
65
66      std::thread t_01(service_01, req_cnt, std::cref(datasets),
            std::ref(q1));
67      std::thread t_02(service_02, req_cnt, std::ref(q1),
            std::ref(q2));
68      std::thread t_03(service_03, req_cnt, cls_params,
            std::ref(q2), std::ref(pool), verbose);
69
70      t_01.join();
71      t_02.join();
72      t_03.join();
73
74      if (verbose)
75          dump_pool(pool, "conc.txt");
76  }
```

Вывод

В данном разделе были рассмотрены средства реализации, а также представлен листинг реализаций линейного и конвейерного алгоритмов обработки набора файлов MIDI.

4 Исследовательская часть

В данном разделе приведены технические характеристики устройства, на котором проводилось измерение времени работы программного обеспечения, а также результаты замеров времени.

4.1 Технические характеристики

Технические характеристики устройства, на котором выполнялись замеры по времени:

— процессор: AMD Ryzen 7 5800X @ 3.800 ГГц, 8 физ. ядер, 16 лог. ядер;

— оперативная память: 32 ГБайт.

— операционная система: Manjaro Linux x86_64 (версия ядра Linux 6.5.12-1-MANJARO).

Измерения проводились на стационарном компьютере. Во время проведения измерений устройство было нагружено только системными приложениями.

4.2 Демонстрация работы программы

На рисунке 4.1 продемонстрирована работа программы для случая, когда пользователь выбрал пункт 2 «Запустить последовательную обработку датасета файлов MIDI», выбрал файл с описанием датасета под номером 1 и ввел следующие значения:

— количество заявок — 10,

— число кластеров — 3,

— значение показателя нечеткости — 2,

— значение порога сходимости — 1,

— максимальное к-во итераций — 10.

```
                Меню
1. Запустить последовательную обработку датасета файлов MIDI.
2. Запустить конвейерную обработку датасета файлов MIDI.
3. Произвести замеры по времени реализуемых алгоритмов.
0. Выход.

Выберите опцию (0-3): 2

Введите количество заявок: 10
Выберите файлы описания датасета:
    1. "/home/daria/Документы/bmstu-aa/lab5/prog/descriptions/ds_02.txt"
    2. "/home/daria/Документы/bmstu-aa/lab5/prog/descriptions/ds_01.txt"
Номера файлов (0 для выхода): 2 1 2 2 1 2 2 2 1 2
Введите количество кластеров: 3
Введите показатель нечеткости: 2
Введите порог сходимости: 1
Введите максимальное количество итераций: 10

                Меню
1. Запустить последовательную обработку датасета файлов MIDI.
2. Запустить конвейерную обработку датасета файлов MIDI.
3. Произвести замеры по времени реализуемых алгоритмов.
0. Выход.

Выберите опцию (0-3): 0
```

Рисунок 4.1 – Демонстрация работы программы

4.3 Временные характеристики

Исследование временных характеристик реализуемых алгоритмов производилось 2 раза:

1) при изменении числа заявок от 5 до 50 с шагом 5 для датасета, состоящего из двух файлов MIDI;

2) при изменения размера датасета от 1 до 10 файлов MIDI при 10 заявках.

Для кластеризации были выбраны следующие параметры:

— число кластеров — 5,

— значение показателя нечеткости — 2,

— значение порога сходимости — 1,

— максимальное к-во итераций — 1000.

На рисунке 4.2 приведены результаты измерения времени работы алгоритмов линейной и конвейерной обработки датасетов файлов MIDI при варьировании числа заявок; таблица 4.1 содержит данные, по которой был построен данный график.

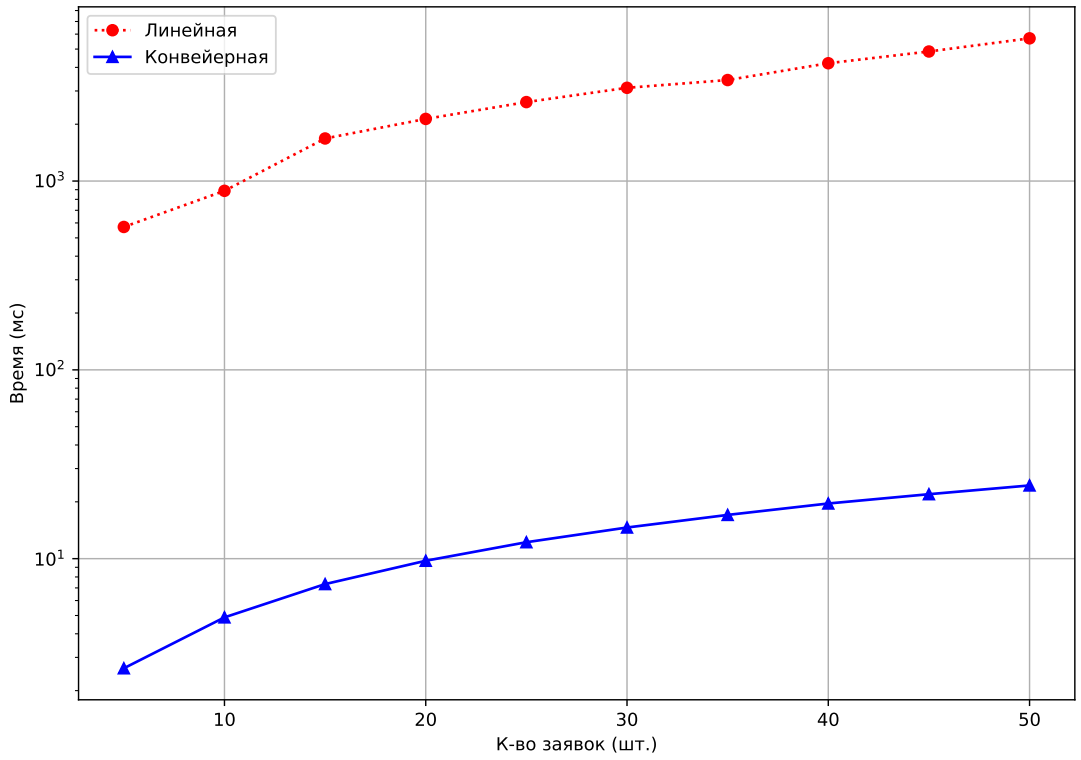

Рисунок 4.2 – Сравнения времени работы алгоритмов линейной и конвейерной обработки датасетов файлов MIDI при варьировании числа заявок

Таблица 4.1 – Результаты измерения времени работы реализуемых алгоритмов при варьировании числа заявок

К-во файлов (шт.)	Время (мс)	
	Линейный	Конвейерный
1	280633.58	2755.60
2	559750.83	5334.26
3	839651.58	8110.10
4	1117778.79	10803.40
5	1395546.68	13297.58
6	1674180.20	16128.27
7	1953854.11	18811.78
8	2234085.88	21565.09
9	2511400.88	24301.44
10	2791956.47	27051.17

На рисунке 4.3 приведены результаты измерения времени работы алгоритмов линейной и конвейерной обработки датасетов файлов MIDI при варьировании числа файлов в датасете; таблица 4.2 содержит данные, по которой был построен данный график.

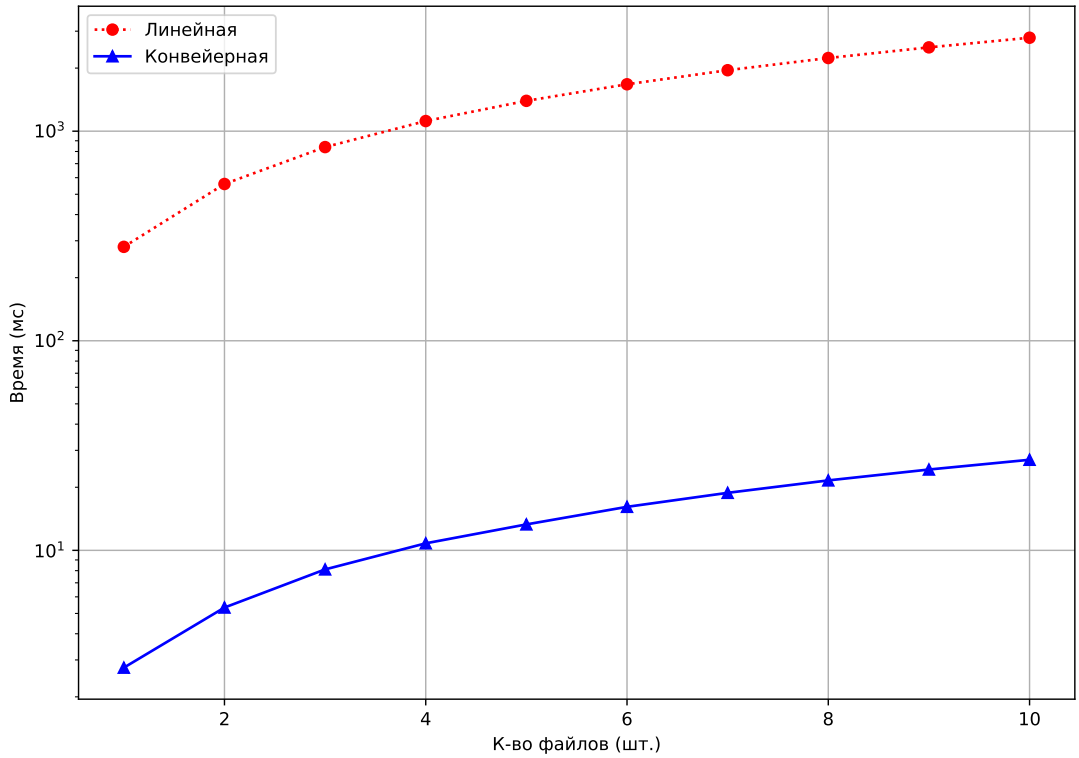

Рисунок 4.3 – Сравнения времени работы алгоритмов линейной и конвейерной обработки датасетов файлов MIDI при варьировании числа файлов в датасете

Таблица 4.2 – Результаты измерения времени работы реализуемых алгоритмов при варьировании числа заявок

К-во заявок (шт.)	Время (мс)	
	Линейный	Конвейерный
5	571257.46	2626.42
10	887551.33	4886.41
15	1680618.36	7333.44
20	2133323.53	9746.04
25	2619305.55	12219.58
30	3117679.17	14619.90
35	3427449.75	17047.36
40	4206998.96	19596.30
45	4854466.86	21948.68
50	5699781.42	24419.38

4.4 Вывод

В результате исследования реализуемых алгоритмов по времени выполнения можно сделать следующие выводы:

1) при варьировании числа заявок линейный алгоритм обработки датасета, состоящем из двух файлов MIDI, выполнялся дольше конвейерного в среднем в 214.5 раз;

2) при варьировании числа файлов линейный алгоритм обработки датасета при 10 заявках выполнялся дольше конвейерного в среднем в 103.7 раз.

ЗАКЛЮЧЕНИЕ

В результате выполнения лабораторной работы по исследованию алгоритмов сортировок решены следующие задачи:

1) описана организация конвейерной обработки данных;

2) описаны алгоритмы обработки данных, которые использованы в текущей лабораторной работе;

3) определены средства программной реализации;

4) реализована программа, выполняющая конвейерную обработку с количеством лент не менее трех в однопоточной и многопоточной реализациях;

5) проведен сравнительный анализ процессорного времени выполнения реализованных алгоритмов:

   — при варьировании числа заявок линейный алгоритм обработки датасета, состоящем из двух файлов MIDI, выполнялся дольше конвейерного в среднем в 214.5 раз;

   — при варьировании числа файлов линейный алгоритм обработки датасета при 10 заявках выполнялся дольше конвейерного в среднем в 103.7 раз.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. C++ language. — [Электронный ресурс]. — Режим доступа: `https://en.cppreference.com/w/cpp/language` (дата обращения: 21.12.2023).