Московский авиационный институт
(национальный исследовательский
университет)

Факультет прикладной математики и физики

Кафедра вычислительной математики
и программирования

Курсовой проект по курсу "Дискретный анализ"

Студент: Д. С. Левштанов
Руководитель: А. А. Журавлев
Группа: 80-208Б
Дата:
Оценка:
Подпись:

Москва, 2017

Курсовой проект "Аудио поиск"

Задача:

Необходимо реализовать поиск совпадений по аудиофайлам.

Запуск и параметры:

./prog index –input <input file> –output <index file>
./prog search –index <index file> –input <input file> –output <output file>

Входные данные:

Входные файлы содержат в себе имена файлов с аудио записями по одному файлу в строке.

Результат работы программы:

Результатом ответа на каждый запрос является строка с названием файла, с которым произошло совпадение, либо строка "! NOT FOUND" , если найти совпадение не удалось.

1 Описание

В данном курсовом проекте, на языке c++ была реализована задача "Аудио поиска". То есть требовалось составить базу на основе некоторой выборки аудиофайлов, а затем производить среди них поиск файла, который больше остальных совпадает с образцом. Для того, чтобы выполнить эту задачу, необходимо изучить базовые понятия теории звука.

Звук – это вибрации, которые распространяются в твёрдых, жидких и газообразных средах и могут расшифровываться ушами. Эти вибрации могут быть смоделированы с помощью синусоидальной формы.

Чистые тона и реальные звуки

Чистый тон – это звук, который можно представить в виде синусоиды. Синусоидальный сигнал характеризуется частотой – количество тактов в секунду (измеряется в Герцах) и амплитудой – размер каждого такта. Человек может слышать чистые тона от 20 Гц до 20 000 Гц и этот диапазон уменьшается с возрастом. Восприятие громкости зависит от частоты чистого тона. Например, чистый тон с амплитудой 10 и частотой 30 Гц будет тише, чем чистый тон с амплитудой 10 и частотой 1000 Гц.

Но чистые тона не существуют в обычной природе, каждый звук в мире - это сумма нескольких чистых тонов при различных амплитудах. Реальный звук может состоять из тысячи чистых тонов.

Спектограмма

В песня обычно присутствует несколько инструментов, а иногда и несколько певцов. Все эти инструменты создают комбинацию синусоид кратных частот и в целом представляет собой еще большее сочетание синусоид.

Музыку возможно наглядно представить на спектрограмме. Обычно это диаграмма с тремя измерениями, где на горизонтальной оси у нас есть время, на вертикальной оси - частота чистого тона, и третье измерение описывается цветом и представляет собой амплитуду частот в определенное время.

Sampling

Аналоговые сигналы - это непрерывные сигналы, что означает, что если взять одну секунду аналогового сигнала, то можно разделить эту секунду на сколько угодно частей. В цифровом мире мы не можем позволить себе хранить бесконечное количество информации. Мы должны иметь минимальную единицу, например, 1 миллисекунда. В течение этой

единицы времени звук не может меняться, так что этот блок должен быть достаточно коротким, чтобы цифровая песня звучала как аналоговая, но в тоже время и достаточно большим, чтобы ограничить пространство, необходимое для хранения музыки.

Стандартная единица времени в цифровой музыке составляет 44 100 единиц в секунду. Именно это число используется потому, что по теореме Найквиста-Шеннона, для того, чтобы сигнал был восстановлен однозначно и без потерь, необходимо использовать частоту в 2 раза больше максимальной, а человек способен слышать звуки от 20гц до 20кГц. Это значит, что нужно использовать частоту не менее 40000Гц, но в 80-х годах музыкальные корпорации (например, Sony) выбрали 44100 Гц, так как это было больше 40000 Гц и совместимо со стандартами видео NTSC и PAL.

Квантование

Мы узнали, как оцифровать частоты музыки, но что делать с громкостью? Громкость является относительной мерой. Для одинаковой громкости в сигнале, если увеличить громкость колонок звук будет выше. Громкость измеряет разброс между самым низким и самым высоким уровнем звука в песне.

Квантование - это разбиение диапазона значений непрерывной или дискретной величины на конечное число интервалов. В музыке стандартно уровень квантования кодируется на 16 бит, что означает 65536 уровней.

От цифрового звука к частотам

Мы знаем, как перейти от аналогового звука в цифровой. Но как получить частоты из цифрового сигнала? Эта важно, так как аудио поиск работает только с частотами. Для аналоговых (и, следовательно, непрерывных) сигналов, происходит преобразование, называемое "непрерывные преобразование Фурье"(НПФ). Это преобразование превращает функцию времени в функцию частоты. Для дискретного же сигнала существует "дискретное преобразование Фурье"(ДПФ).

Примечание: ДПФ применимо только для одного канала, поэтому, если мы имеем стереосигнал, нам нужно превратить его в моно.

Дискретное преобразование Фурье

ДПФ применяется для дискретных сигналов и дает дискретный спектр (частоты в сигнале). Вот формула для преобразования цифрового сигнала в частоту

$$X(n) = \sum_{k=0}^{N-1} x[k] e^{-j(2\pi kn/N)}$$

В этой формуле:

- N – это размер окна: количество образцов, из которых состоит сигнал.
- X(n) – n-й "контейнер" частот.
- x(k) – k-й семпл звукового сигнала.

Здесь используется "контейнер" частот, а не частота, потому что ДПФ дает дискретный спектр и "контейнер" частот является наименьшей единицей частоты, которую ДПФ может вычислить. Размер "контейнера" называется частотным разрешением и равен частоте дискретизации сигнала, деленной на размер окна.

Оконные функции

Если мы хотим получить частоту каждых 0.1 секундных частей в односекундной музыке, то надо применить преобразование Фурье сначала к первой 0.1 секундной части, потом ко второй и тд. Это делается с помощью применения к сигналу оконной функции. Но эта функция также производит спектральные утечки – появление новых частот, ранее не существовавших внутри звукового сигнала. Получаются они из-за того, что сильные частоты влияют на более слабые соседние.

Мы не можем полностью избежать спектральных утечек, но мы можем контролировать, как утечка будет себя вести при правильном выборе оконной функции. Существуют такие оконные функции, как: прямоугольное окно, окно Блэкмана, окно Хемминга и др.

Прямоугольное окно, самое простое в использовании, так как нужно просто разделить сигнал на маленькие части. Оно хорошо применимо для синусоид сопоставимой силы, но это плохой выбор для синусоид различных амплитуд (что и имеет место в песне).

Окно Блэкмана лучше, для того, чтобы избежать случая, когда утечка спектра сильных частот скрывает слабые частоты. Но, оно плохо работает с шумом, и скрывает даже больше частот чем прямоугольное окно. Это проблема для аудио поиска, так как в поисковых образцах могут содержаться посторонние шумы.

Окно Хэмминга находится между этими двумя крайностями и является лучшим выбором для алгоритма аудио поиска.

Быстрое преобразование Фурье

В дискретном преобразование Фурье для вычисления одного "контейнера" нам потребуется N сложений и N умножений (где N это размер окна), что дает сложность $O(n^2)$, что очень много. Но к счастью существует реализация ДПФ, которая работает за $O(nlogn)$ и называется быстрое преобразование Фурье(БПФ). Это важное ускорение, так как никто не хочет долго ждать, чтобы найти песню.

Самая простая реализация БПФ - это алгоритм, который работает по принципу разделяй и властвуй. Идея заключается в том, что вместо прямого вычисления преобразования Фурье на N-образец окна, алгоритм:

- делит окно-образец на 2 N/2-образца окна.
- вычисляет (рекурсивно) БПФ для 2 N/2-образцов окна.
- эффективно вычисляет БПФ для N-окон из 2-х предыдущих БПФ.

Downsampling

Можно заметить, что увеличение размера окна улучшает частотное разрешение, но оно также увеличивает время вычислений. Но есть трюк, который помогает сохранить частотное разрешение и уменьшить размер окна в то же время, это называется downsampling. Если взять, например, песню с 44100Гц, и понизить частоту до 11025Гц (44100/4), то мы получим то же частотное разрешение, применив БПФ к 44100Гц песне с 4096 окном или применив БПФ к 11025Гц песне с 1024 окном. Единственное различие заключается в том, что измененной песне, будут только частоты от 0 до 5000Гц. На самом деле большинство не слышат большой разницы между музыкой в 11025Гц и музыкой в 44100Гц.

Понижение сделать очень просто, достаточно взять группу, например, из 4 семплов и превратить их в один, взяв среднее значение. Единственный нюанс состоит в том, что перед понижением сигнала нужно отфильтровать высокие частоты в звуке, чтобы избежать наложения спектров.

В итоге все основные частоты сохраняются в песне, и при этом улучшается скорость работы алгоритма.

Реализация

При реализации программы мы используем библиотеку для работы с аудио – mpg123. При ее инициализации задаем частоту равной 41000Гц, использование одного канала и кодировку float. Это было проделывается для того, чтобы все песни считывались в одинаковом формате.

Построчно читая названия песен из файла, мы открываем их и, считывая данные по частям, фильтруем высокие частоты и делаем понижение частоты, беря среднее значение от каждых четырех семплов.

Только после этого мы запускаем быстрое преобразование Фурье. Один из неприятных побочных эффектов БПФ заключается в том, что, проведя анализ, мы теряем информацию о времени. Мы можем видеть звуковые частоты и их амплитуды, но вот где именно в произведении эти частоты встречаются, не знаем. Нам нужно как-то узнать точные значения времени, когда появляется каждая из частот. Именно поэтому мы будем пользоваться чем-то вроде скользящего окна, и подвергать трансформации лишь ту часть сигнала, которая в это окно попадает.

Как только у нас будут сведения о частотных характеристиках сигнала, можно приступать к формированию цифровой сигнатуры музыкального произведения. Это – самая важная часть всего процесса распознавания музыки. Главная сложность здесь – выбрать из огромного количества частот именно те, которые важнее всего. Обычно мы обращаем внимание на частоты с максимальными амплитудами. Однако, в одной песне диапазон сильных частот может варьироваться очень сильно. Поэтому, вместо того, чтобы сразу проанализировать весь частотный диапазон, мы можем выбрать несколько более мелких интервалов и найти в них частоты с самыми высокими уровнями. Эти сведения и формируют сигнатуру для конкретного анализируемого блока данных, а она, в свою очередь, является частью сигнатуры всей песни.

Для упрощения поиска музыкальных композиций их сигнатуры используются как ключи в хэш-таблице. Ключам соответствуют значения времени, когда набор частот, для которых найдена сигнатура, появился в произведении, и идентификатор самого произведения (название песни).

После обработки таким способом всю библиотеку музыкальных записей, мы получаем базу данных с полными сигнатурами каждого произведения.

Для того, чтобы распознать песню, еще нужно прогнать через вышеописанный процесс вычисления сигнатур. Затем запустить поиск вычисленных хэш-тегов в базе данных.

Но у многих фрагментов различных произведений хэш-тэги совпадают. Всякий раз, когда удаётся обнаружить совпадающий хэш-тег, число возможных совпадений уменьшается, но весьма вероятно, что только лишь эти сведения не позволят нам настолько сузить диапазон поиска, чтобы остановиться на единственной правильной песне. Поэтому в алгоритме распознавания музыкальных произведений нам нужно проверять ещё отметки времени.

Фрагмент песни, который мы ищем, может быть из любого её места,

поэтому мы просто не в состоянии напрямую сравнивать относительное время внутри записанного фрагмента с тем, что есть в базе данных.

Однако если найдено несколько совпадений, можно проанализировать относительный тайминг совпадений, и, таким образом, повысить достоверность поиска.

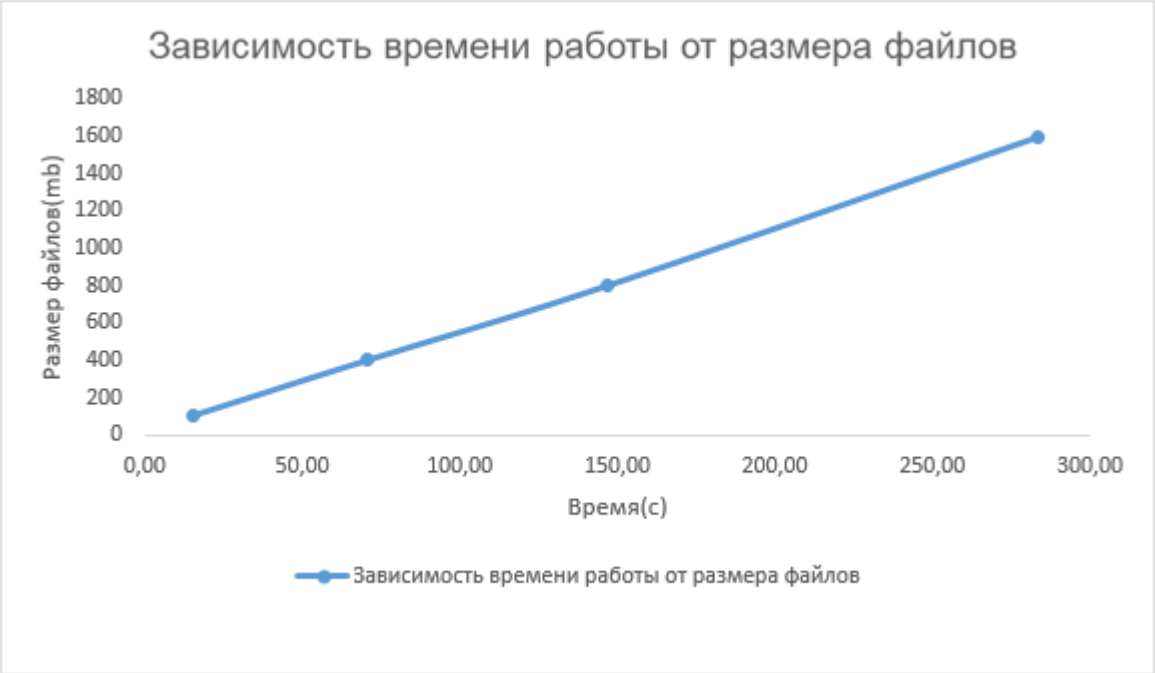
И маловероятно, что каждый обработанный фрагмент песни полностью совпадёт с аналогичным фрагментом из базы данных. Поэтому, вместо того, чтобы исключать из списка совпадений всё, кроме единственной верной композиции, в конце поиска мы вычисляем процент совпавших семплов и та песня, у которой процент наибольший и будет является ответом. Если же процент совпадения меньше 10%, то песня считается ненайденной.

2 Тест производительности

Для теста производительности я скомпилировал программу с ключом оптимизации -o2, а время работы программы замерял с помощью утилиты time.

Для начала я измерил время создания базы для поиска с разным размером входных файлов. Я сделал 4 замеряя для музыкальных файлов общим размером примерно 100мб, 400мб, 800мб и 1600мб. Вот результаты:

- 100mb – 0m15.153s
- 400mb – 1m10.778s
- 800mb – 2m26.658s
- 1600mb – 4m43.212s

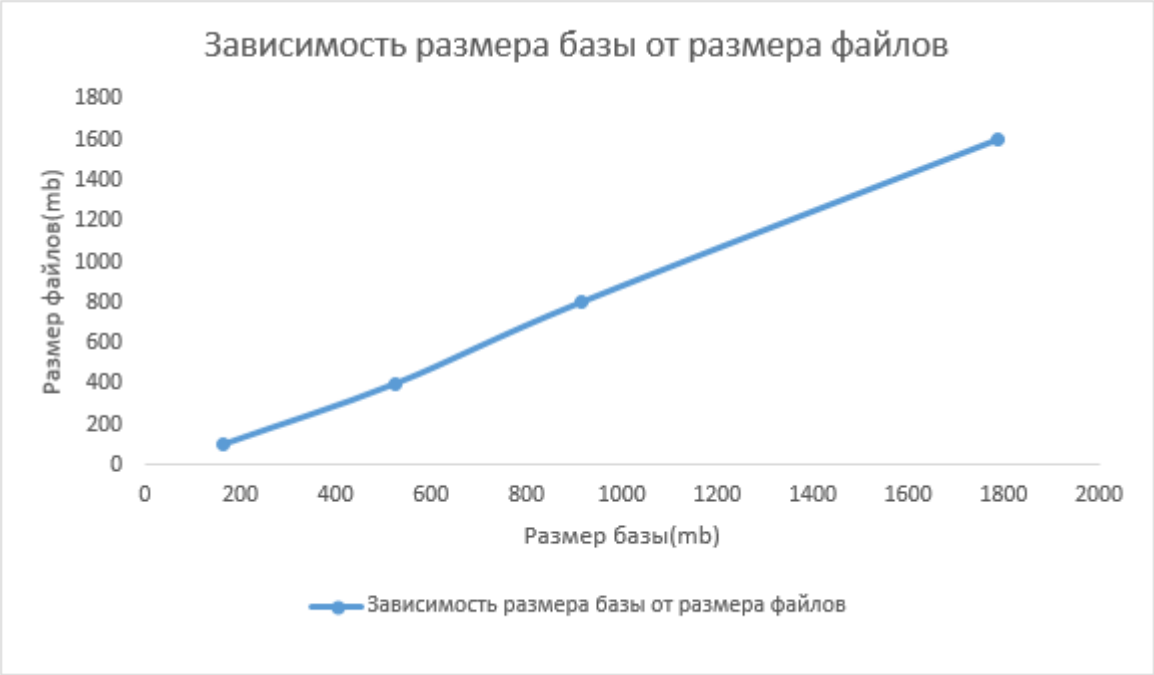


Отсюда видно, что сложность создания базы чуть больше чем линейная.

График размеров полученных баз при этих же входных данных:

- 100mb – 162mb
- 400mb – 525mb

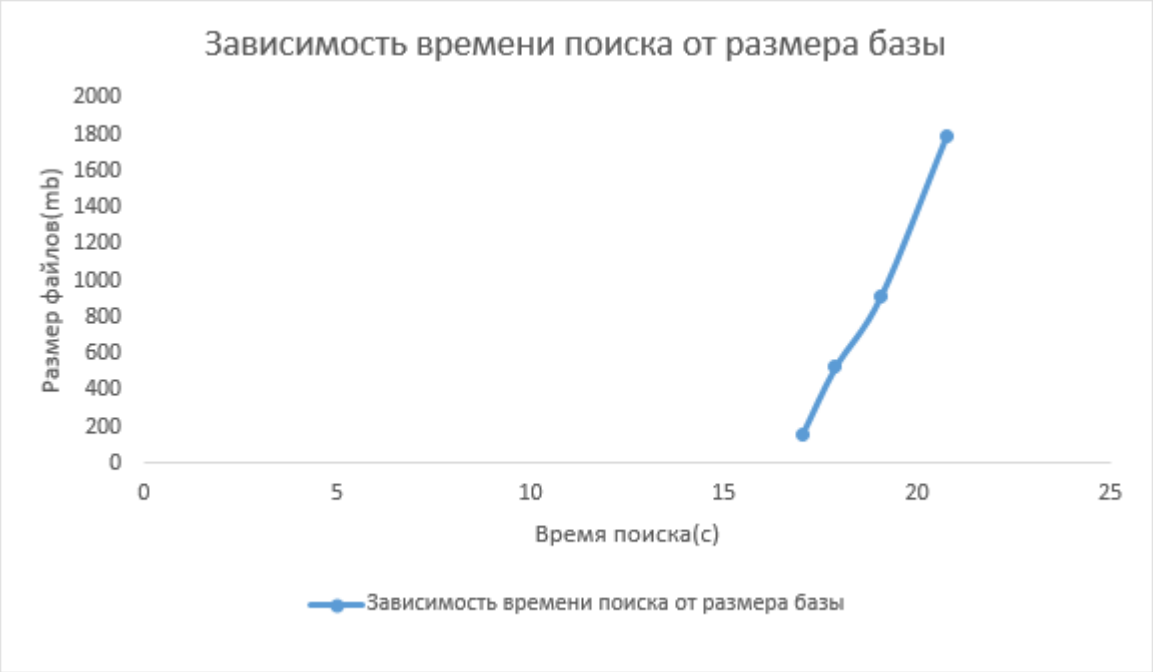
- 800mb – 915mb
- 1600mb – 1789mb



Зависимость тоже близка к линейной.

В итоге я измерил время работы программы пытаясь найти 100мб образцов в базах разных размеров (которые были получены ранее).

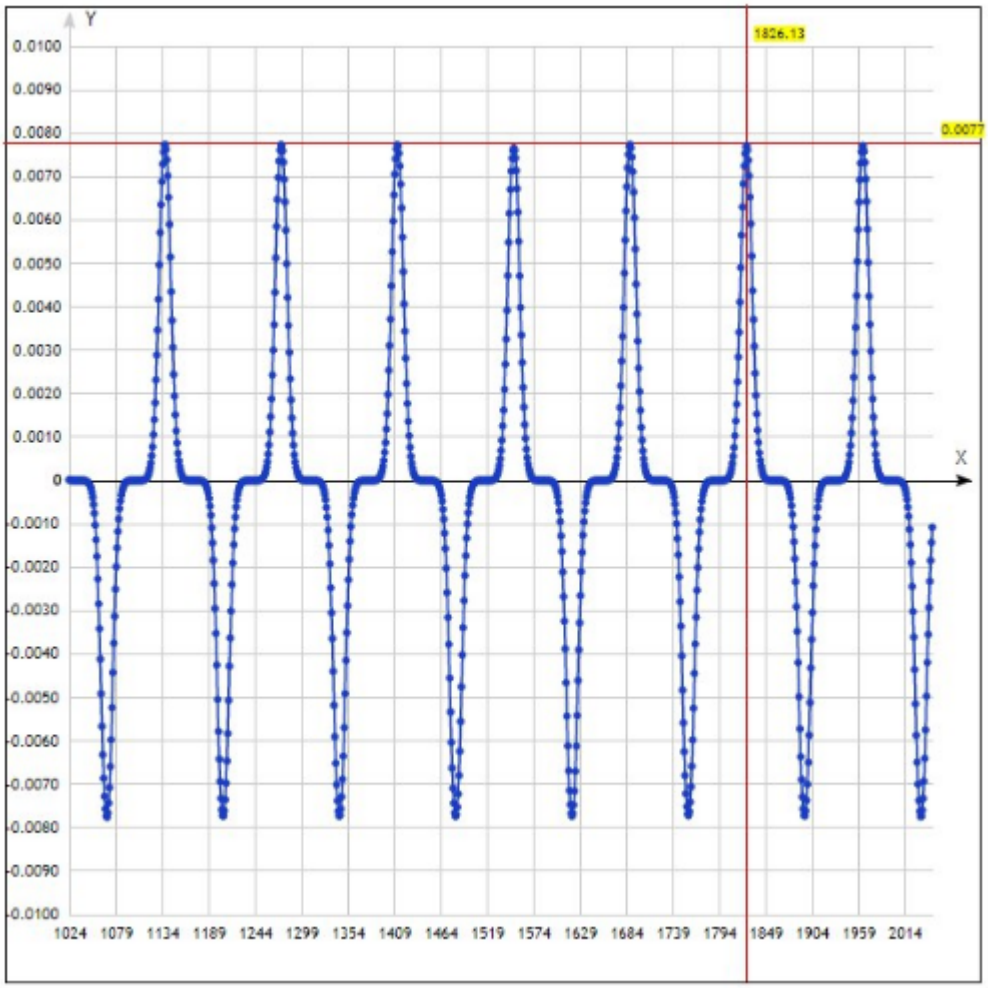
- 162mb – 0m17.022s
- 525mb – 0m17.868s
- 915mb – 0m19.057s
- 1789mb – 0m20.725s



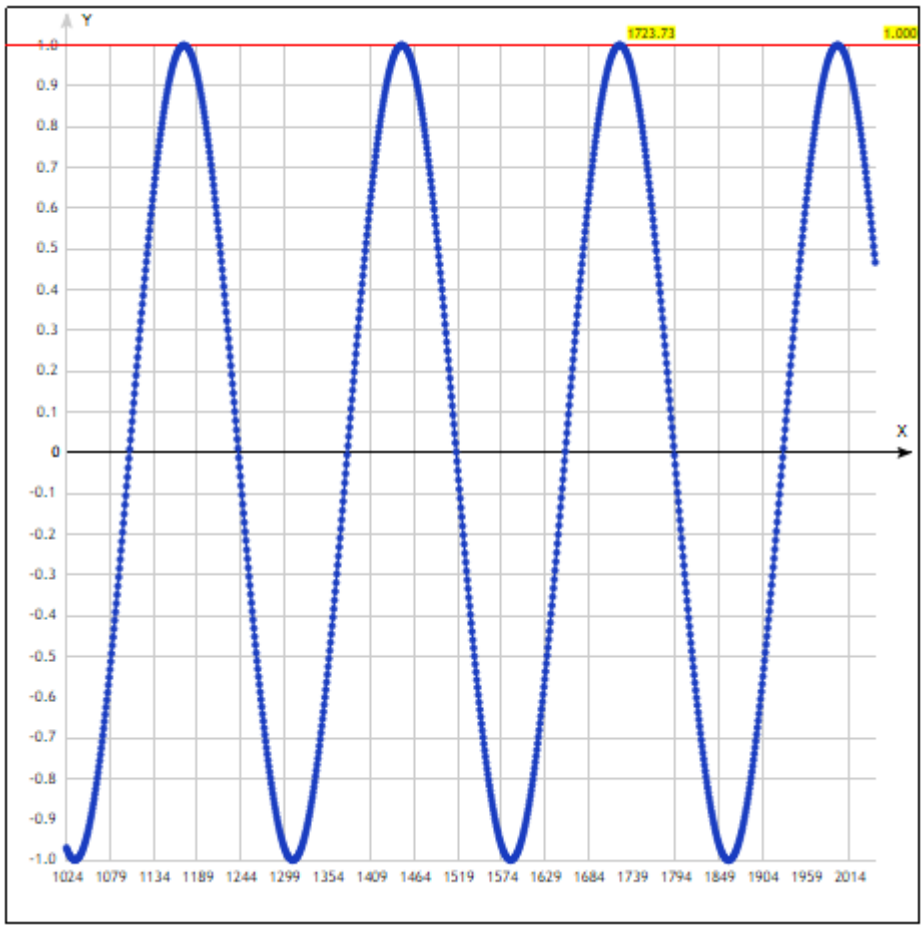
Отсюда можно увидеть, что основное время тратиться на процесс создания "отпечатка" песни, чем на сам поиск. И поэтому если заранее создать большую базу "отпечатков" , то поиск можно осуществлять достаточно быстро.

3 Дневник отладки

Изначально я считывал данные как float, но клал их в массив с double, и у меня выходило что два семпла float лежали в одном элементе массива double. Из-за этого во многих песнях процент совпадения был маленьких. Так же это можно увидеть на графике для частоты 40Гц: Он выглядит неестественно, так как звук одной частоты должен представлять синусоиду.



Но после замены всех переменных с double на float, процент совпадения в песне значительно увеличивается и график приходит в норму.



4 Исходный код

```
1  //TAudioSearch.h
2  #pragma once
3  #define _USE_MATH_DEFINES
4  #include <iostream>
5  #include <string.h>
6  #include <vector>
7  #include <complex>
8  #include <cmath>
9  #include <fstream>
10 #include <unordered_map>
11 #include <map>
12 #include <list>
13 #include <valarray>
14 #include "mpg123.h"
15
16
17 enum TErrorCode {
18     AS_OK,
19     AS_CANT_OPEN_FILE,
20     AS_FILE_CORRUPT,
21     AS_INITIALIZE_ERROR,
22     AS_ALLOCATE_ERROR,
23     AS_NOT_FOUND
24 };
25
26 typedef std::complex<float> Complex;
27 typedef std::valarray<Complex> CArray;
28 const int32_t FUZ_FACTOR = 2;
29 const int32_t UPPER_LIMIT = 300;
30 const int32_t LOWER_LIMIT = 40;
31 const int32_t RANGE[] = { LOWER_LIMIT, 80, 120, 180, UPPER_LIMIT + 1
       };
32
33 struct DataPoint {
34     int32_t songId;
35     int32_t time;
36 };
37
38 class TAudioSearch {
39     static std::unordered_map<int64_t, std::list<DataPoint>> hashMap;
40     static std::vector<std::string> songs;
41 public:
42     int32_t CreateMusicBase(mpg123_handle* mh, std::string input, std::
          string output);
43     int32_t FindSongInBase(mpg123_handle* mh, std::string base, std::
          string intput, std::string output);
44
```

```
45 private:
46     std::string songName;
47     bool searchMode;
48     int32_t songId;
49     size_t songLen;
50     std::unordered_map<int32_t, std::map<int32_t, int32_t>> matchMap;
51 
52     int32_t DecodeAudioFile(mpg123_handle* mh);
53     int32_t MakeSpectrum(std::vector<float>& data);
54     int32_t DetermineKeyPoints(std::vector<CArray>& result);
55 
56     int32_t Serialization(std::string output);
57     int32_t Deserialization(std::string input);
58 
59     void Fft(CArray& x);
60     int64_t Hash(int64_t p1, int64_t p2, int64_t p3, int64_t p4);
61     size_t GetIndex(int32_t freq);
62 };
```

```
1  //TAudioSearch.cpp
2  #include "TAudioSearch.h"
3 
4  std::unordered_map<int64_t, std::list<DataPoint>> TAudioSearch::
       hashMap;
5  std::vector<std::string> TAudioSearch::songs;
6 
7  int32_t TAudioSearch::CreateMusicBase(mpg123_handle* mh, std::string
       input, std::string output) {
8      searchMode = false;
9      std::ifstream fin(input);
10     if (!fin)
11         return AS_CANT_OPEN_FILE;
12     while (std::getline(fin, songName)) {
13         songId = songs.size();
14         songs.push_back(songName);
15         if (DecodeAudioFile(mh) != AS_OK) {
16             std::cerr << "Error in file " << songName << std::endl;
17             songs.pop_back();
18         }
19     }
20     fin.close();
21     if (hashMap.empty())
22         std::cerr << "Base empty" << std::endl;
23     return Serialization(output);
24 }
25 
26 int32_t TAudioSearch::FindSongInBase(mpg123_handle* mh, std::string
       base, std::string input, std::string output) {
27     searchMode = true;
```

```
28      int32_t stat = Deserialization(base);
29      if (stat != AS_OK)
30          return stat;
31      std::ifstream fin(input);
32      std::ofstream fout(output);
33      while (std::getline(fin, songName)) {
34          if (DecodeAudioFile(mh) != AS_OK) {
35              std::cerr << "Error in file '" << songName << "'" << std::
                    endl;
36              continue;
37          }
38          int32_t bestCount = 0;
39          int32_t bestSong = -1;
40          for (int32_t id = songs.size() - 1; id >= 0; id--) {
41              auto tmpMap = matchMap.find(id);
42              int32_t bestCountForSong = 0;
43              if (tmpMap == matchMap.end())
44                  continue;
45              for (auto entry : tmpMap->second) {
46                  if (entry.second > bestCountForSong)
47                      bestCountForSong = entry.second;
48              }
49
50              if (bestCountForSong > bestCount) {
51                  bestCount = bestCountForSong;
52                  bestSong = id;
53              }
54          }
55          if (bestSong == -1 || (bestCount / (float)songLen) * 100 < 10)
56              fout << "! NOT FOUND " << std::endl;
57          else
58              fout << songs[bestSong] << std::endl;
59      }
60      fout.close();
61      fin.close();
62      return AS_OK;
63  }
64
65  int32_t TAudioSearch::Serialization(std::string output) {
66      std::ofstream fout(output);
67      if (!fout)
68          return AS_CANT_OPEN_FILE;
69      if (songs.empty()) {
70          std::cerr << "Base empty" << std::endl;
71          fout << 0;
72      }
73      else {
74          fout << songs.size() << ' ';
75          for (std::string i : songs)
```

```
76              fout << i << std::endl;
77          fout << hashMap.size() << ' ';
78          for (auto i : hashMap) {
79              fout << i.first << ' ';
80              fout << i.second.size() << ' ';
81              for (auto list : i.second) {
82                  fout << list.songId << ' ' << list.time << ' ';
83              }
84          }
85      }
86      fout.close();
87      return AS_OK;
88  }
89
90  int32_t TAudioSearch::Deserialization(std::string input) {
91      std::ifstream fin(input);
92      if (!fin)
93          return AS_CANT_OPEN_FILE;
94      std::string str;
95      size_t size, sizeList;
96      int64_t hash;
97      int32_t songId, time;
98      std::list<DataPoint> buflist;
99      fin >> size;
100     if (!size)
101         std::cerr << "Base empty" << std::endl;
102     else {
103         songs.resize(size);
104         fin.get();
105         for (size_t i = 0; i < size; i++)
106             std::getline(fin, songs[i]);
107         fin >> size;
108         hashMap.reserve(size);
109         for (size_t i = 0; i < size; i++) {
110             fin >> hash;
111             fin >> sizeList;
112             for (size_t j = 0; j < sizeList; j++) {
113                 fin >> songId >> time;
114                 buflist.push_back({ songId, time });
115             }
116             hashMap.insert(std::make_pair(hash, buflist));
117             buflist.clear();
118         }
119     }
120     fin.close();
121     return AS_OK;
122 }
123
124 void TAudioSearch::Fft(CArray& x) {
```

```
125     size_t N = x.size(), k = N, n;
126     float thetaT = (float)M_PI / N;
127     Complex phiT = Complex(cos(thetaT), -sin(thetaT)), T;
128     while (k > 1) {
129         n = k;
130         k >>= 1;
131         phiT = phiT * phiT;
132         T = 1.0L;
133         for (size_t l = 0; l < k; l++) {
134             for (size_t a = l; a < N; a += n) {
135                 size_t b = a + k;
136                 Complex t = x[a] - x[b];
137                 x[a] += x[b];
138                 x[b] = t * T;
139             }
140             T *= phiT;
141         }
142     }
143     size_t m = (size_t)log2(N);
144     for (size_t a = 0; a < N; a++) {
145         size_t b = a;
146         b = (((b & 0xaaaaaaaa) >> 1) | ((b & 0x55555555) << 1));
147         b = (((b & 0xcccccccc) >> 2) | ((b & 0x33333333) << 2));
148         b = (((b & 0xf0f0f0f0) >> 4) | ((b & 0x0f0f0f0f) << 4));
149         b = (((b & 0xff00ff00) >> 8) | ((b & 0x00ff00ff) << 8));
150         b = ((b >> 16) | (b << 16)) >> (32 - m);
151         if (b > a)
152         {
153             Complex t = x[a];
154             x[a] = x[b];
155             x[b] = t;
156         }
157     }
158 }
159 
160 int32_t TAudioSearch::DecodeAudioFile(mpg123_handle* mh) {
161     size_t done;
162     int channels, encoding;
163     long rate;
164     size_t bufferSize = mpg123_outblock(mh);
165     unsigned char* ucBuffer = new unsigned char[bufferSize];
166     if (mpg123_open(mh, songName.c_str()) != MPG123_OK)
167         return AS_CANT_OPEN_FILE;
168     mpg123_getformat(mh, &rate, &channels, &encoding);
169     std::vector<float> data;
170     size_t doubleSize = sizeof(float);
171     float RC = 1.0f / (2000 * 2 * (float)M_PI);
172     float dt = 1.0f / 44100;
173     float alpha = dt / (RC + dt);
```

```
174     for (int totalBtyes = 0; mpg123_read(mh, ucBuffer, bufferSize, &
            done) == MPG123_OK; ) {
175         float* dBuffer = reinterpret_cast<float*>(ucBuffer);
176         size_t dataSize = bufferSize / doubleSize;
177         for (size_t i = 1; i < dataSize; i++)
178             dBuffer[i] = dBuffer[i - 1] + (alpha*(dBuffer[i] - dBuffer[i
                    - 1]));
179         size_t i = 0;
180         for (; i + 4 <= dataSize; i += 4)
181             data.push_back((dBuffer[i] + dBuffer[i + 1] + dBuffer[i + 2]
                    + dBuffer[i + 3]) / 4);
182         if (i > dataSize) {
183             i -= 4;
184             while (i < dataSize) {
185                 data.push_back(dBuffer[i]);
186                 i++;
187             }
188         }
189         totalBtyes += done;
190     }
191
192     delete[] ucBuffer;
193     mpg123_close(mh);
194     return MakeSpectrum(data);
195 }
196
197 int32_t TAudioSearch::MakeSpectrum(std::vector<float>& data) {
198     const size_t ONE_CHANK_SIZE = 1024;
199     size_t totalSize = data.size();
200     size_t amountPossible = totalSize / ONE_CHANK_SIZE;
201
202     std::vector<CArray> result(amountPossible, CArray(ONE_CHANK_SIZE));
203
204     for (size_t times = 0; times < amountPossible; times++) {
205         for (size_t i = 0; i < ONE_CHANK_SIZE; i++)
206             result[times][i] = data[(times * ONE_CHANK_SIZE) + i];
207         Fft(result[times]);
208         break;
209     }
210     data.clear();
211     return DetermineKeyPoints(result);
212 }
213
214 size_t TAudioSearch::GetIndex(int32_t freq) {
215     size_t i = 0;
216     while (RANGE[i] < freq) i++;
217     return i;
218 }
219
```

```
220 int64_t TAudioSearch::Hash(int64_t p1, int64_t p2, int64_t p3, int64_t
        p4) {
221     return (p4 - (p4 % FUZ_FACTOR)) * 100000000
222         + (p3 - (p3 % FUZ_FACTOR)) * 100000
223         + (p2 - (p2 % FUZ_FACTOR)) * 100
224         + (p1 - (p1 % FUZ_FACTOR));
225 }
226
227 int32_t TAudioSearch::DetermineKeyPoints(std::vector<CArray>& result)
        {
228     songLen = result.size();
229     std::vector<std::vector<float>> highscores(songLen, std::vector<
            float>(5, 0));
230     std::vector<std::vector<int64_t>> points(songLen, std::vector<
            int64_t>(5, 0));
231     matchMap.clear();
232     for (size_t t = 0; t < songLen; t++) {
233         for (int freq = LOWER_LIMIT; freq < UPPER_LIMIT - 1; freq++) {
234             float mag = log(std::abs(result[t][freq]) + 1);
235             size_t index = GetIndex(freq);
236
237             if (mag > highscores[t][index]) {
238                 highscores[t][index] = mag;
239                 points[t][index] = freq;
240             }
241         }
242
243         int64_t h = Hash(points[t][0], points[t][1], points[t][2],
                points[t][3]);
244
245         if (searchMode) {
246             auto listPoints = hashMap.find(h);
247             if (listPoints != hashMap.end()) {
248                 for (DataPoint& dP : listPoints->second) {
249                     int32_t offset = abs(dP.time - (int32_t)t);
250                     auto it2 = matchMap.find(dP.songId);
251                     if (it2 == matchMap.end()) {
252                         std::map<int32_t, int32_t> tmpMap;
253                         tmpMap.insert(std::make_pair(offset, 1));
254                         matchMap.insert(std::make_pair(dP.songId, tmpMap)
                                ;
255                     }
256                     else {
257                         auto tmp = it2->second.find(offset);
258                         if (tmp == it2->second.end()) {
259                             it2->second.insert(std::make_pair(offset, 1))
                                    ;
260                         }
261                         else {
```

```
                        int32_t count = tmp->second + 1;
                        it2->second.erase(offset);
                        it2->second.insert(std::make_pair(offset,
                            count));
                    }
                }
            }
        }
        else {
            std::list<DataPoint> listPoints;
            auto it = hashMap.find(h);
            if (it == hashMap.end()) {
                listPoints.push_back({ songId, (int32_t)t });
                hashMap.insert(std::make_pair(h, listPoints));
            }
            else
                it->second.push_back({ songId, (int32_t)t });
        }
    }
    return AS_OK;
}
```

```
//main.cpp
#include "TAudioSearch.h"

void PrintUsage(const char* cProgName) {
    std::cerr << "USAGE:" << cProgName << " index --input <input file>
        --output <index file>" << std::endl;
    std::cerr << cProgName << " search --index <index file> --input <
        input file> --output <output file>" << std::endl;
}

void PrintError(int32_t errorCode) {
    switch (errorCode) {
    case AS_OK:
        break;
    case AS_CANT_OPEN_FILE:
        std::cerr << "Could not open file" << std::endl;
        break;
    case AS_FILE_CORRUPT:
        std::cerr << "File corrupt" << std::endl;
        break;
    case AS_INITIALIZE_ERROR:
        std::cerr << "Resampler has not been properly initialized" <<
            std::endl;
        break;
    case AS_ALLOCATE_ERROR:
        std::cerr << "Could not allocate the frame" << std::endl;
```

```
24          break;
25      case AS_NOT_FOUND:
26          std::cerr << "Song not found" << std::endl;
27          break;
28      default:
29          std::cerr << "Unknown error" << std::endl;
30          break;
31      }
32  }
33
34  int main(int argc, char *argv[]) {
35      bool searchMode = false;
36      if (argc == 1) {
37          PrintUsage(argv[0]);
38          return 1;
39      }
40      else if (!strcmp(argv[1], "index")) {
41          if (argc < 6 || strcmp(argv[2], "--input") || strcmp(argv[4], "
                --output")) {
42              std::cerr << "Wrong parameters for 'index'" << std::endl;
43              PrintUsage(argv[0]);
44              return 1;
45          }
46      }
47      else if (!strcmp(argv[1], "search")) {
48          if (argc < 8 || strcmp(argv[2], "--index") || strcmp(argv[4], "
                --input") || strcmp(argv[6], "--output")) {
49              std::cerr << "Wrong parameters for 'search'" << std::endl;
50              PrintUsage(argv[0]);
51              return 1;
52          }
53          searchMode = true;
54      }
55      else {
56          std::cerr << "Wrong mode" << std::endl;
57          PrintUsage(argv[0]);
58          return 1;
59      }
60      TAudioSearch as;
61      mpg123_handle* mh = nullptr;
62      int32_t err = mpg123_init();
63      if (err != MPG123_OK || (mh = mpg123_new(nullptr, &err)) == nullptr
            ){
64          PrintError(AS_INITIALIZE_ERROR);
65          return AS_INITIALIZE_ERROR;
66      }
67      long flags = MPG123_MONO_MIX | MPG123_QUIET | MPG123_FORCE_FLOAT;
68      if (mpg123_param(mh, MPG123_FLAGS, flags, 0) != MPG123_OK) {
69          PrintError(AS_INITIALIZE_ERROR);
```

```
70          return AS_INITIALIZE_ERROR;
71      }
72      mpg123_format_none(mh);
73      if (mpg123_format(mh, 44100, MPG123_MONO, MPG123_ENC_FLOAT) !=
            MPG123_OK) {
74          PrintError(AS_INITIALIZE_ERROR);
75          return AS_INITIALIZE_ERROR;
76      }
77      PrintError(searchMode ? as.FindSongInBase(mh, argv[3], argv[5],
            argv[7]) : as.CreateMusicBase(mh, argv[3], argv[5]));
78      mpg123_delete(mh);
79      mpg123_exit();
80      return 0;
81  }
```

```
1   #makefile
2   TARGET = DA_CP
3   CC=g++
4   FLAGS = -std=c++17 -Wall -pedantic -o2
5
6   OBJ = main.o TAudioSearch.o
7
8   .PHONY: all clean
9
10  all: $(TARGET)
11
12  clean:
13    rm -rf $(TARGET) *.o
14
15  TAudioSearch.o: TAudioSearch.cpp TAudioSearch.h
16    $(CC) $(FLAGS) -c $< -o $ -I./includemain.o: main.cpp
          TAudioSearch.h(CC)(FLAGS) -c < −o -I./include
17
18  $(TARGET): $(OBJ)
19    $(CC) $(FLAGS) $(OBJ) -o $ -L./lib -lmpg123
```

5 Выводы

Выполнив курсовую работу, я понял то, как устроен звук в цифровом понимании, из чего он состоит, на что влияет каждый из параметров. Также я изучил 2 библиотеки по работе с аудио. Первая была ffmpeg – это целый набор свободных библиотек, которые позволяют записывать, конвертировать и передавать цифровые аудио- и видеозаписи в различных форматах. Но она оказалась слишком громоздкой и ее функционал был слишком избыточен для моей задачи. Поэтому я использовал mpg123 – это библиотека, которая имеет меньший функционал чем ffmpeg, но она полностью удовлетворяет все требования моей задачи.

Так же я понял, как примерно работает популярное приложение shazam, которое помогает распознавать музыку. Принцип его работы не так сложен, как это кажется на первый взгляд, но в тоже время существует множество нюансов, которые следует учитывать.

Вообще музыка играет огромную роль в жизни человека, и с помощью алгоритмов можно сделать не только жизнь людей чуточку проще, распознавая музыку, которую они где-то услышали, но также можно исследовать произведения на плагиат, искать исполнителей, которые вдохновляли некоторых первопроходцев в каком-либо жанре музыки и тд.

Список литературы

[1] *Статья: "How does Shazam work"*
URL: `http://coding-geek.com/how-shazam-works/`

[2] *Статья: "Shazam: алгоритмы распознавания музыки, сигнатуры, обработка данных"*
URL: `https://habrahabr.ru/company/wunderfund/blog/275043/`