Курсовой проект по курсу Дискретный Анализ. Аудиопоиск

Выполнил студент группы М8О-307Б-21 МАИ *Друхольский Александр.*

Условие

Задача:

> Реализуйте систему для поиска аудиозаписи по небольшому отрывку.
>
> $./prog\ index\ --input\ <input\ file>\ --output\ <index\ file>$

Ключ	Значение
$--input$	входной файл с именами файлов для индексации
$--output$	выходной файл с индексом

> $./prog\ search\ --index\ <index\ file>\ --input\ <inputfile>\ --output\ <outputfile>$

Ключ	Значение
$--index$	входной файл с индексом
$--input$	входной файл с запросами
$--output$	выходной файл с ответами на запросы

Все файлы будут даны в формате MP3 с частотой дискретизации 44100Гц. Входные файлы содержат в себе имена файлов с аудио записями по одному файлу в строке. Результатом ответа на каждый запрос является строка с названием файла, с которым произошло совпадение, либо строка "! NOT FOUND", если найти совпадение не удалось.

Метод решения

> **Индексация**. Для решения поставленной задачи нам требуется реализовать алгоритм БПФ - быстрое преобразование Фурье, которое позволяет получить частотный спектр сигнала. Для начала воспользуемся предложенной библиотекой, которая кодирует входящий сигнал и возвращает массив отсчётов. Эти значения мы и будем преобразовывать с помощью алгоритма Фурье. Для аудио мы получаем частотный спектр. Для каждого окна у нас теперь имеется набор значений, где смысл i-го числа в том, что частоты в окрестности i Гц имеют интенсивность равную модулю комплексного числа. Таким образом для каждого аудио мы можем сделать отпечаток со значениями после преобразования Фурье. Учитываем, что

алгоритм подразумевает, что мы восстанавливаем N/2 + i-е значение из i-го значения. Данное замечание нам понадобится при определении помехоустойчивости. Ширину окна примем за 4096, а шаг возьмём 2048 (такой точности достаточно для аудиопоиска, а шаг в 1024 довольно ощутимо увеличивает время работы).

Помехоустойчивость. Нам требуется, чтобы аудиопоиск был способен найти отрывок плохого качества (запись с телефона, эквалайзер, шумы и так далее). Очевидно, что алгоритмы точного сравнения и поиска подстроки в данном случае не будут эффективны, так как искажения аудио могут быть значительны, а разные источники искажения действуют на частоты неодинаково (колонки, эквалайзер). Можно заметить, что при искажениях частотная характеристика сохраняется, но из-за неравномерного воздействия нам потребуется разбить спектр на частоты. Я решил разбить спектр на 7 октав, максимальная частота при этом 2048, ведь, как сказано выше, частоты в полученной характеристике не превышают 2048 Гц. В каждом диапазоне сохраним частоту с максимальной интенсивностью. Для каждого окна отсчётов получим отпечаток, в котором вместо большого количества значений будет всего 7. При сравнении будем допускать некоторое количество несовпадений (количество следует подобрать для наиболее корректной работы программы).

Поиск. Как сказано выше, не имеет смысла использовать алгоритмы точного сравнения и поиска подстроки. Хорошим вариантов для данной задачи будет использование расстояния Левенштейна. Такой метод подойдёт нам, ведь очень велика вероятность, что точного вхождения при искажениях найти не удастся, но зато мы можем посчитать коэффициет совпадения, по которому и будем определять нашли мы вхождение или нет. Коэффициент будем определять как $coef = 1 - levlen(audio, sample)/(2 * n)$, где n - минимальное из длин аудио и отрывка (чаще всего именно длина отрывка). Таким образом получим число от 0 до 1. Логично, что чем ближе число к 1, тем меньше расстояние Левенштейна, следоватльно, тем более отрывки схожи. Расстояние Левентшейна можем считать, используя алгоритм Вагнера-Фишера за $O(n^2)$.

Проблема появляется в тот момент, когда требуется считать коэффициенты совпадения для отрывка и отрывков исходного аудио. Проблема в том, что непонятно, с какого отсчёта начинать сравнивать, ведь отрывок может быть дан из любой части песни. Изначально я реализовал наивный алгоритм, то есть просто "прикладываем"к аудиотреку, шагая по всем отсчётам. Даже при первом тестировании стало понятно, что подобный алгоритм работает слишком долго, а именно за $O(n * m * m)$, где n -длина отрывка, m -длина аудио. При попытках улучшить этот алгоритм увеличивая каким-либо образом шаг, мы, естественно, теряем точность проверки. Оказывается, мы можем идти не по всем отчётам. Будем итерироваться только по отсчётам, кратным длине отрывка, и на каждой итерации брать предыдущий коэффициент и новый. Если их сумма больше определенного значения, то мы начинаем проверять промежуток между этими отсчётами, сдвигаясь по пра-

вилам бинарного поиска в ту сторону, где был получен наибольший коэффициент совпадения. Таким образом можно достигнуть сложноти $O(n * m * log(m))$ для проверки вхождения одного отрывка в одну аудиозапись.

Описание программы

Индексация:

1. $vector < short > decoder()$ - кодировщик исходного сигнала.
2. $int\ bitReversalPermutation(int\ sz, inta)$ - функция, которая с помощью битовых операций вычисляет битовый реверс числа для поиска новой позиции.
3. $void\ fastFourier(vector < double > \&\ window, vector < complex < double >> \&\ res\_window, complex < double > z, int\ left, int\ right)$ - рекурсивная функция, производящая вычисления самой функции. На каждом углублении рекурсии мы передаем z*z, согласно формуле. left и right - левая и правая границы рассматриваемого куска.
4. $void\ calculateSpectrum(vector < short > sample)$ - функция, запускающая процесс подсчёта функции Фурье для каждого окна шириной 4096 с шагом 2048. Здесь же мы применяем к отсчётам окно Ханна, про которое написано выше.
5. *class AuPrint* - отпечаток аудиосигнала. Методы этого класса отвечают за сохранение и загрузку отпечатков.
6. $vec7\ splitFrequency(vector < complex < double >> cur_spectrum)$ - получение характеристики по диапазонам частот.

Поиск:

1. $double\ startSearch(AuPrint\ track, AuPrint\ sample)$ - Поиск вхождения, вычисление позиции для поиска коэффициента совпадения.
2. $double\ compare(int\ pos, AuPrint\ track, AuPrint\ sample)$ - алгоритм Вагнера-Фишера вычисления расстояния Левентшейна.
3. $int\ matchingVecs(vec7\ v1, vec7\ v2)$ - сравнение характеристик отсчёта с допускаемым числом несовпадений.

Дневник отладки

Далее представлены основные ошибки, с которыми я столкнулся при тестировании.

WA Ошибка, связанная с неправильным нахождением вхождений. Основная суть проблемы - выбор неправильных погрешностей при сравнении.

TL Слишком долгий поиск при наивном методе. Исправил, написав бинарный поиск.

Тест производительности

Для тестов я взял альбом одного исполнителя со средней длительностью треков примерно 2:20. Отрывки будем вырезать тоже из этих же данных.

Индексация:

Длительность индексации при примерно одинаковых длительностях треков будет зависеть линейно от их количества.

№	Количество треков	Время, с
1	1	9.12375
2	2	20.1224
3	4	39.5559
4	7	62.502
5	14	114.959

Тестирование показало, что сложность действительно зависит от M линейно, при постоянной ширине окна.

Поиск:

По вертикали - количество сэмплов, по горизонтали - количество треков в библиотеке. Сложность поиска одного отрывка в одном треке $O(n * m * log(m))$, но в реализации у бинарного поиска фиксированное количество итераций. Тогда для k отрывков и l количества треков в библиотеке сложность: $O(k * l * n * m)$. При приблизительно не меняющихся n и m исследуем сложность $O(k * l)$.

$k \backslash l$	1	2	4	7
1	1.86112	2.76519	4.38797	6.23403
2	3.56061	5.32654	8.38192	11.7971
4	6.35232	9.55279	15.1986	21.4949
7	9.53872	14.1553	22.5916	31.8727

По вертикали можно четко видеть линейную зависимость. По горизонтали она тоже есть, но она не наглядна, так как в замерах времени пришлось учитывать и время на кодировку, что портит вычисления.

Проверка на помехи: Исследуем несколько возможных источников помех:

Помеха	Средняя точность
Нет	0.9075
Диктофон телефона	0.383436
Эквалайзер	0.815
Ускорение x1.2	0.862069
Ускорение x1.4	0.733333
Замедление x0.8	0.77
Замедление x0.6	0.622189
Тональность +	0.40796
Тональность -	0.465174

Показатели эквалайзера выставлялись рандомно для частот. Тональность песни изменялась на 2 тона в обе стороны. Тесты показали, что эквалайзер и небольшое изменение темпа песни не сказываются негативно на аудиопоиске. Сильные изменения темпа уже превносят более весомые ошибки в вычисления. Изменение тональности же практически полностью портит алгоритм поиска (при повышении тональности более четкое совпадение нашло вообще с неверным треком). Диктофон телефона, который записал звук с плохих колонок тоже показал низкое качество точности, но, если быть честным, с задачей алгоритм почти во всех случаях (в том числе и с телефоном) справился.

Выводы

Я реализовал алгоритм аудиопоиска, используя быстрое преобразование Фурье и расстояние Левенштейна. Основная сложность реализации подобной системы в том, что коэффициенты, при которых мы считаем, что треки совпали, стоит сделать динамическими в зависимости от слышимого источника. Потому что происходит проблема с тем, что когда трек не должен быть найден вообще, его сопоставляет с каким либо файлом (например, у меня тесты из треков одного альбома и при искажениях треки могли определяться ложно). В целом, алгоритм работает и даже противостоит помехам. Если обернуть данный код в приложение, то получится технология, похожая на Shazam.