Санкт-Петербургский государственный университет
Факультет прикладной математики – процессов управления
Кафедра Моделирования экономических систем

Щербаков Глеб Александрович

Магистерская диссертация

Разработка общего алгоритма резервного копирования для мобильных устройств с использованием внешнего диска

Направление 01.04.02 «Прикладная математика и информатика»
Основная образовательная программа ВМ.5754
«Математические методы цифровизации экономики»

Научный руководитель:
доктор физ.-мат. наук, профессор,
кафедра моделирования экономических систем
Смирнов Николай Васильевич

Санкт-Петербург
2022 г.

Содержание

Введение

Резервное копирование – важная задача, поскольку отказы оборудования и программного обеспечения, а также человеческий фактор могут привести к потери важной информации. Ещё более важны резервные копии для ноутбуков и мобильных устройств поскольку они достаточно часто становятся объектами краж. В современном обществе телефон сопровождает человека большую часть его дня, не удивительно что в нем начинает хранится большое количество личной информации. Поэтому очень важно сохранить эти данные.

Несмотря на то, что современные технологии позволяют сохранять резервную копию в облачных сервисах, вероятность утечки данных хранимых на портативных накопителях намного меньше.

В рамках исследовательской работы изучим различные методы резервного копирования, а также алгоритм позволяющий определять сходства файлов. Изучив различные подходы, мы реализуем свой программный комплекс для создания резервной копии данных с мобильного устройства.

На текущий момент более 70% мобильных телефонов работают под управлением операционной системы `Android` [1]. Поэтому мобильное приложение разработано под данную операционную систему.

Обзор литературы

Не существует универсального подхода к созданию копий данных с устройства. При выборе типа резервного копирования необходимо отталкиваться от множества факторов: скорость работы, объем занимаемого места, нагрузка на оборудование. Обзор типов резервного копирования содержится в работах [2]–[4].

Особой важностью для нас обладают алгоритмы позволяющие определять сходство файлов. Одним из распространённых подходов к поиску похожих документов можно считать алгоритм нечёткого хэширования, описанный в текстах [5]–[6]. Изначально подход использовался для поиска спама [7]–[8], однако позднее был применён и для обнаружения вредоносных программ и похожих дел в судебной экспертизе.

При поиске одинаковых изображений важно выделить ключевые элементы, которые определяют уникальность каждого изображения. Алгоритм SSIM описан в тексте [9], помимо него мы будет использовать алгоритмы, реализованные в библиотеке OpenCV. Почерпнуть информацию о них можно в следующих работах: FAST [10], ORB [11], BRISK [12].

Глава 1. Инструментарий

В данной главе содержатся сведения, необходимые для реализации программного комплекса, реализующего наш алгоритм; они относятся как к CS, так и к чисто программированию и математике.

1.1 Типы резервного копирования

Рассмотрим основные типы резервного копирования [2]–[4].

Определение 1. *Полная копия всего набора данных. Хотя полные резервные копии, возможно, обеспечивают лучшую защиту, большинство организаций не используют их ежедневно, потому что они отнимают много времени и часто требуют большой емкости диска или ленты.*

Определение 2. *Инкрементальные резервные копии – выполняется резервное копирование только тех данных, которые изменились с момента предыдущего резервного копирования.*

Определение 3. *Дифференциальное резервное копирование – выполняется резервное копирование только тех данных, которые изменились с момента полного бекапа.*

Определение 4. *Snapshots and replication – snapshots фиксируют изменения в наборах данных и хранят эту информацию в виде набора указателей на основе справочной и полной копии данных. Моментальные снимки могут создаваться часто, что позволяет пользователям возвращаться к определенной дате для восстановлении данных. Snapshots являются внутренними процессами, что означает, что они хранятся в той же системе, что и исходные данные. Чтобы использовать snapshots в качестве резервной копии, необходимо провести репликацию.*

Определение 5. *Continuous data protection (CDP) – метод полагается на полное резервное копирование, как основу своей работы. CDP фиксирует изменения по мере их появления. Основным недостатком CDP является большая вычислительная мощность при обновлении большого объёма данных.*

Определение 6. *Синтетическое резервное копирование – метод, создаёт полную копию данных не путём чтения исходных данных, а путем использования существующих инкрементальных и дифференциальных резервных копий и более старой резервной копии.*

В качестве метода резервного копирования был выбран инкрементальный метод. Continious data protection не подходит т.к. у нас нет возможности фиксировать каждое изменение из-за доступности подключения диска к нескольким устройствам, а также отсутствия микропроцессора в нем. По этой же причине нам не подходит синтетическое резервное копирование.

1.2 Context Triggered Piecewise Hash

Одним из самых эффективный способов определения схожести файлов не использующих сжатия является алгоритм фази хэширования.

Context Triggered Piecewise Hash также известный как алгоритм нечеткого хеширования основан на использовании скользящего хеша.

Алгоритм вычисления хэша делит файл на блоки. Размер блоков не является постоянным, но средний размер блока подлежит настройке. Затем каждый блок преобразуется в два шестнадцатеричных числа, первое из которых представляет собой содержимое, а второй его размер. Наконец, хеш записывается как множитель, за которым следует каждый блок.

Благодаря своей особенности формирования хэща, он хорошо подходит нам для выявления изменённых файлов.

1.3 Кусочное хеширование

Первоначально разработанный Николасом Харбором для dcfldd [5], кусочное хеширование использует произвольный алгоритм хеширования для создания множества контрольных сумм для файла вместо одной. Вместо того, чтобы генерировать один хэш для всего файла, хеш создается для многих дискретных сегментов фиксированного размера файла. Например, один хеш создается для первых 512 байтов ввода, другой хеш – для следующих 512 байтов и так далее. Изначально этот метод был разработан для уменьшения ошибок во время криминалистической визуализации. Если произошла

ошибка, только один из кусочных хэшей станет недействительным. Оставшаяся часть кусочных хэшей и, таким образом, целостность оставшейся части данных по-прежнему будут гарантированы.

Кусочное хеширование может использовать либо криптографические алгоритмы хеширования, такие как MD5, либо более традиционные алгоритмы хеширования, такие как хэш Фаулера-Нолла-Vo (FNV).

1.4 Алгоритм скользящего хэширования

Алгоритм скользящего хеширования создаёт псевдослучайное значение только на основе текущего контекста ввода. Скользящий хеш работает, поддерживая состояние, основанное исключительно на последних нескольких байтах ввода. Каждый байт добавляется к состоянию по мере его обработки и удаляется из состояния после обработки установленного количества других байтов.

Предполагая, что у нас есть ввод из n символов, мы говорим, что i-ый байт ввода представлен как b_i. Таким образов слово ввода состоит из $b_i, i = \overline{1,n}$. В любой позиции p во входных данных состояние скользящего хэша будет зависеть только от последних s байтов файла. Таким образом, значение скользящего хэша r может быть выражено как функция последних нескольких байтов, как показано в следующей формуле.

$r_p = F(b_p, b_{p-1}, ..., b_{p-s})$

Скользящая хэш функция построена так, чтобы можно было устранить влияние одного из байтов. Таким образом, для данного r_p можно вычислить r_{p+1} хеш, удалив влияние b_{p-s}, представленного как функция $X(b_{p_s})$ и добавив влияние b_{p+1}, представленного как функция $Y(b_{p+1})$. Т.е.

$r_{p+1} = r_p - X(b_{p-s}) + Y(b_{p+s})\ r_{p+1} = F(b_{p+1}, ..., b_{p-s+1})$

1.5 Объеденный хеш алгоритм

В то время как текущие программы кусочного хеширования, такие как dcfldd, используют фиксированные смещения для определения того, когда запускать и останавливать традиционный алгоритм хеширования, алгоритм CTPH использует скользящий хеш. Когда на выходе скользящего хеша созда-

ется конкретный результат или значение триггера, запускается традиционный хеш. То есть при обработке входного файла начинают вычислять традиционный хеш для файла. Одновременно необходимо также вычислить скользящий хеш для файла. Когда скользящий хэш создает значение триггера, значение традиционного хеша записывается в сигнатуре CTPH, а традиционный хеш сбрасывается.

Следовательно, каждое записанное значение в сигнатуре CTPH зависит только от части входных данных, и изменения во входных данных приведут только к локализованным изменениям в сигнатуре CTPH. Например, при изменении байта ввода не более двух, а во многих случаях только одно из традиционных значений хеш-функции будет изменено(большая часть подписи CTPH останется прежней). Поскольку большая часть подписи остается прежней, файлы с изменениями все еще могут быть связаны с подписями CTPH известных файлов.

1.6 Алгоритм спама

Алгоритм спама использует хэши FNV для традиционных хэшей, которые производят 32-битный вывод для любого ввода . В спам-рассылке д-р Триджелл дополнительно уменьшил хеш-значение FNV, записав только кодировку base64 шести младших битов (LS6B) каждого значения хеш-функции.

Перед обработкой входного файла мы должны выбрать значение триггера для скользящего хеша. Значением триггера называется размер блока. Две константы, минимальный размер блока, b_{min}, и длина спама, S , используются для установки начального размера блока для ввода n байтов с помощью уравнения:

$b_{init} = bmin2^{log_2(n/Sb_{min})}$

Таким образом, размер блока, вычисленный до считывания ввода, называется начальным размером блока, b_{init} .

После обработки каждого байта ввода обновляется скользящий хэш и результат сравнивается с размером блока. Если скользящий хеш дает значение, которое по модулю размера блока равно размеру блока минус один, скользящая контрольная сумма достигла значения триггера. В это время за-

кодированное в base64 значение LS6B традиционного хеша добавляется к первой части окончательной подписи. Точно так же, когда скользящая контрольная сумма дает значение, равное по модулю удвоенного размера блока удвоенному размеру блока минус один, значение LS6B традиционного хэша в кодировке base64 добавляется ко второй части хэша спам-суммы.

После обработки каждого байта ввода проверяется окончательная подпись. Если первой части подписи недостаточно после обработки всего ввода, размер блока уменьшается вдвое, и ввод обрабатывается снова.

Окончательная подпись спама состоит из размера блока, двух наборов LS6B и имени входного файла в кавычках. Первый набор LS6B вычисляется с размером блока b, а остальные $2b$.

1.7 Сравнение подписей спама

Две подписи спама можно сравнить, чтобы определить, являются ли файлы, из которых они были получены, гомологичными. При обследовании рассматривается размер блока, удаляются любые последовательности, а затем вычисляется взвешенное расстояние редактирования между ними, как определено ниже. Расстояние редактирования масштабируется для получения оценки соответствия или консервативной взвешенной меры упорядоченных гомологичных последовательностей, обнаруженных в обоих файлах.

Поскольку триггеры для традиционного хэша основаны на входном файле и размере блока, сравнивать можно только подписи с идентичным размером блока. Алгоритм спама генерирует подписи для каждого ввода на основе размеров блока b и $2b$, поэтому можно сравнить две подписи, если размеры блока, указанные в подписях, находятся в пределах степени двойки. Например, с двумя подписями, первая с размером блока b_x и вторая с b_y , первая подпись имеет значения CTPH для размеров блока b_x и $2b_x$, вторая – для y и $2b_y$. Мы можем сравнить эти две подписи, если $b_x = b_y$, $2b_x = b_y$ или $b_x = 2b_y$.

После определения размеров блоков все повторяющиеся последовательности удаляются, поскольку не передают много информации о содержимом файла.

Вычисляем взвешенное расстояние редактирования между двумя хешами с помощью динамического программирования.

Метрика расстояния e между строками s_1 и s_2 определяется как «минимальное количество точечных мутаций, необходимых для преобразования s_1 в s_2», где точечная мутация означает изменение, вставку или удаление буквы [7]. Алгоритм спама использует взвешенную версию формулы расстояния редактирования, первоначально разработанную для программы чтения новостей USENET trn [8]. В этой версии каждая вставка i или удаление d оценивается в единицу, каждое изменение c оценивается в три, а каждая замена w (то есть правильные символы, но в обратном порядке) имеет вес в пять. Взвешенное расстояние редактирования обозначается как $e(s_1, s_2)$ для строк s_1 и s_2 длиной l_1 и l_2.

$e(s_1, s_2) = i + d + 3c + 5w$

$c + w \leq min(l_1, l_2)$

$i + d = |l_1 - l_2|$

Оценка соответствия – мера того, сколько битов этих двух подписей идентичны и находятся в одном порядке. Чем выше оценка соответствия, тем более вероятно, что подписи были получены от общего предка и тем более вероятно, что исходные файлы были получены от общего предка.

$M = 100 - (\frac{100Se(s_1,s_2)}{64(l_1+l_2)})$

Поскольку алгоритм спама полностью детерминирован, идентичные файлы будут производить одинаковый хэш. Поскольку хэши совпадают то расстояние редактирования будет равно нулю. Поэтому по итоговой метрике мы получаем полное совпадение.

1.8 Пример использования алгоритма сравнения

В качестве демонстрационного файла взят список покупок List.docx (рис. 1). Пользователь решает выделить интересующие его продукты в списке и переименовать файл в List2.docx (рис. 2). После подключения диска к мобильному устройству, алгоритм обнаруживает, что файл List.docx был не только изменён, но и переименован.

Полка для овощей и фруктов:
Картофель
Морковь
Лук
Чеснок
Петрушка
Укроп
Яблоки/бананы
Лимон

Полка для молочных продуктов:
Масло сливочное
Кефир
Молоко детское (7 упаковок)
Сметана
Творог
Сыр

Полка для консервов и закаток:
Горчица
Малиновое варенье
Томатная паста
Рыбные консервы
Консервированный горошек
Консервированная кукуруза
Детское питание 7 банок
Сгущенка
Мед

Рис. 1: Список продуктов

Полка для овощей и фруктов:
Картофель
Морковь
Лук
Чеснок
Петрушка
Укроп
Яблоки/бананы
Лимон

Полка для молочных продуктов:
Масло сливочное
Кефир
Молоко детское (7 упаковок)
Сметана
Творог
Сыр

Полка для консервов и закаток:
Горчица
Малиновое варенье
Томатная паста
Рыбные консервы
Консервированный горошек
Консервированная кукуруза
Детское питание 7 банок
Сгущенка
Мед

Рис. 2: Измененный список продуктов

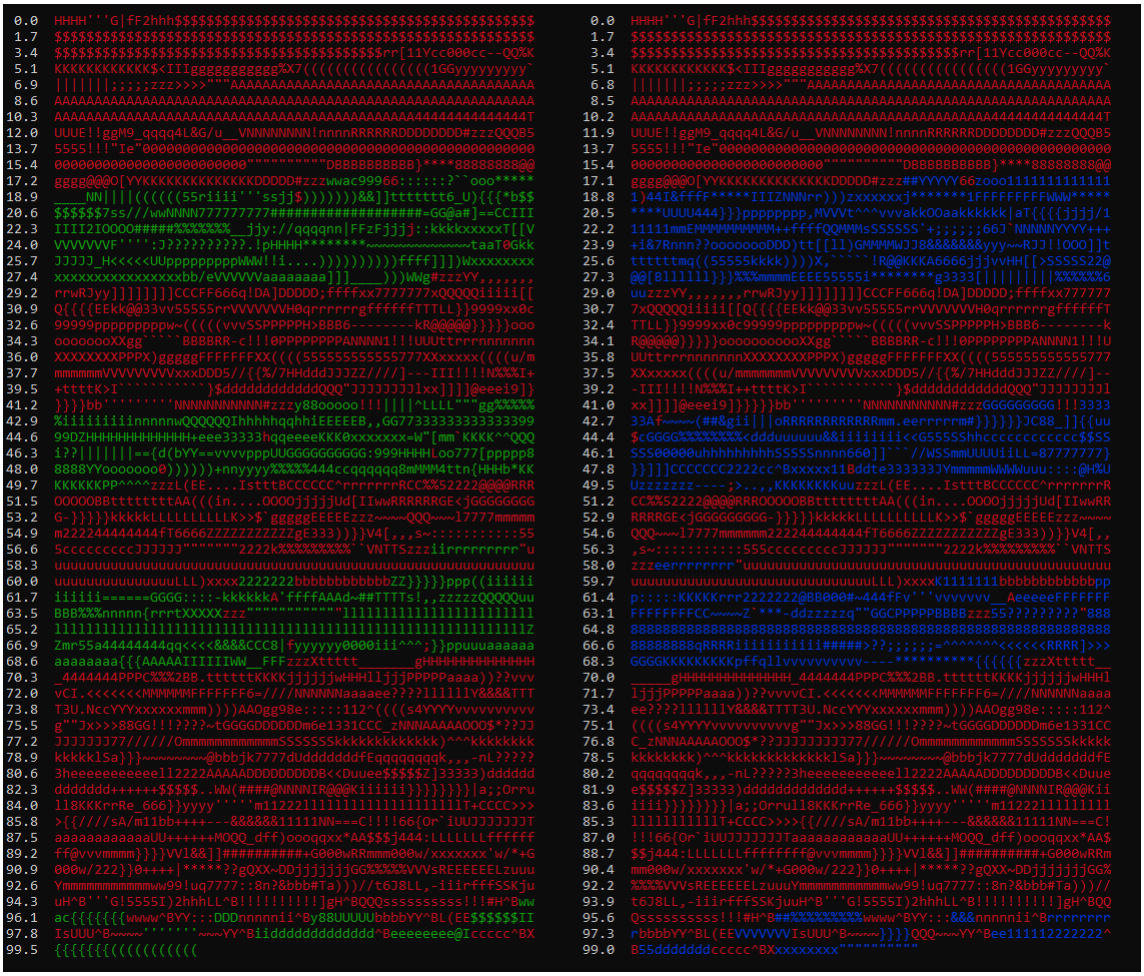

Рис. 3: Хеши файлов сравнения

На рис. 3 представлена визуализация сравнения хэшей, рассмотренных ранее файлов. Коэффициент схожести 0.705, что является достаточным для предположения о том, что List2.docx и List.docx являются одним файлом. Предоставляем пользователю информацию о том, что файл List.docx был переименован.

1.9 Алгоритм сравнения изображений

Алгоритм определения схожести изображений разделим на два шага. Первым делом, если два изображения имеют одинаковый размер, то используем алгоритм SSIM для проверки изменения формата файла. Если изображение было изменено более значительно: поворот, сжатие, блюр, то следует выделить особо важные точки(фичи), по которым мы сможем сделать предпо-

ложение о схожести двух файлов. В качестве меры для определения близости двух векторов фич мы будем использовать алгоритм расстояния Хэмминга. Таким образом мы применим к множеству изображений детекторы фич (на вбор пользователю доступны FAST, ORB, BRISK), для каждого нового множества мы рассчитаем расстояние между каждым изображением и отсортируем по убыванию, усредним результаты векторов схожести и отобразим отсортированный по убывают список наиболее схожих файлов, предоставив пользователю выбор.

1.10 SSIM

Индекс структурного сходства является методом измерения схожести между двумя изображениями путём полного сопоставления [9].

Алгоритм определяет меру схожесть между двумя яркостными изображения $M \times N$, $I_1(i,j)$, $I_2(i,j)$ как мультипликативная комбинацию яркости $I(i,j)$, контрастного сходства $c(i,j)$ и структурного подобия $s(i,j)$.

Пусть W_{ij} обозначает область размера $k \times k$, охватывающую индексы $i,\ldots,i+k-1 \times j,\ldots,j+k-1$ и пусть $w(m,n)$ функция веса каждого индекса (m,n) этой области.

Локальные статистики, вычисляемые для сравниваемых изображений.
$\mu_1(i,j) = \sum_{m,n\in W_{i,j}} w(m,n)I_1(m,n)$
$\mu_2(i,j) = \sum_{m,n\in W_{i,j}} w(m,n)I_2(m,n)$
$\sigma_1^2(i,j) = \sum_{m,n\in W_{i,j}} w(m,n)I_1^2(m,n) - \mu_1^2(i,j)$
$\sigma_2^2(i,j) = \sum_{m,n\in W_{i,j}} w(m,n)I_2^2(m,n) - \mu_2^2(i,j)$
$\sigma_{12}(i,j) = \sum_{m,n\in W_{i,j}} w(m,n)I_1(m,n)I_2(m,n) - \mu_1(i,j)\mu_2(i,j)$
Тогда яркость, контрастность и структурное подобие определяются как:
$l(i,j) = \frac{2\mu_1(i,j)\mu_2(i,j)+C_1}{\mu_1^2(i,j)+\mu_2^2(i,j)+C_1}$
$c(i,j) = \frac{2\sigma_1(i,j)\sigma_2(i,j)+C_2}{\sigma_1^2(i,j)+\sigma_2^2(i,j)+C_2}$
$s(i,j) = \frac{\sigma_{12}(i,j)+C_3}{\sigma_1(i,j)\sigma_2(i,j)+C_3}$
Локальный показатель качества определяется как
$Q(i,j) = l(i,j)c(i,j)s(i,j)$
$SSIM(I_1,I_2) = \frac{1}{MN}\sum_{i=1}^{M}\sum_{j=1}^{N} Q(i,j)$
Если $SSIM(I_1,I_2) = 1$, то изображения считаются одинаковыми. Чем

более приближено значение к единице, тем более схожи изображения. Важно отметить, что для алгоритма необходимо, что размер сравниваемых объектов совпадал.

1.11 Fast Feature Detection

Используется для обнаружения углов. Впервые представлен в 2006 году [10]. Преимуществом можно назвать высокую скорость вычислений, которую можно ещё увеличить с помощью приёмов машинного обучения.

Суть алгоритма:

- Выбираем пиксель p на исследуемой картинке, задаем его интенсивность I_p.
- Выбираем подходящее пороговое значение t.
- Задаем шестнадцатипиксельный круг с центом в исследуемом.
- Пиксель является углом, если существует набор n смежных пикселей, яркость которых выше $I_p + t$ или ниже $I_p - t$.
- Для ускорения алгоритма сравниваем яркость пикселей $1, 5, 9, 13$ с I_p.
- Если по меньшей мере 3 из 4 значений яркости не выше или ниже $I_p + t$, то p не является углом; выбираем другой пиксель для проверки. Иначе проверяем для текущего пикселя все другие.
- Повторить для каждого пикселя.

Алгоритм не без ограничений: при $n < 12$ работает плохо, в зависимости от очерёдности проверки пикселей меняется скорость работы.

Как использовать ML для улучшения алгоритма?

- Задать обучающее множество.
- Для каждого элемента-изображения использовать FAST.
- Для найденных пикселей запомнить находящиеся вокруг них 16 и записать их все в вектор признаков P.

- Разделить P на подмножества P_d, P_s, P_b в зависимости от их относительной яркости.
- Применяем дерево решений с переменной K_p, символизирующей, является ли текущий пиксель искомым.
- Полученное дерево используем на других изображениях.

Ограничение алгоритма – плохо работает на сгенерированных компьютером изображениях. Решение – размытие по Гауссу.

1.12 Oriented FAST and Rotated BRIEF

Сокращенно ORB. Разработан в 2011 году как открытая альтернатива схожим проприетарным алгоритмам [11]. FAST (детектор) и BRIEF (дескриптор) оба были улучшены в результате работы над ORB.

Про FAST речь шла выше, однако, в данном подходе добавляется рассмотрение ориентации (пространственность). Используется пирамида уменьшенных изображений, получаемых из исходного путём понижения разрешения. На каждом уровне находят ключевые точки и их ориентацию.

Поговорим теперь про BRIEF. Данный алгоритм берёт все найденные FASTом ключевые точки и переводит их в единый бинарный вектор. Используется для этого сравнение яркостей пикселей на размытом по Гауссу изображении:

$$\tau(P, x, y) = \begin{cases} 1 : p(x) < p(y) \\ 0 : p(x) \geq p(y). \end{cases}$$

Проблема в том, что при повороте изображения сильно падает производительность BRIEFa, чтобы справиться с этим, используется rBRIEF – Rotation-aware BRIEF, использующий ориентацию окружения ключевой точки.

Далее, в ORBe вычисляется т.н. «центроид ориентации», пары пикселей для сравнения поворачиваются и уже для них проводится сравнение.

1.13 Расстояние Хэмминга

Для вычисления расстояния между векторами будем использовать формулу Хэмминга:

$$d_H(X_i, X_j) = \sum_{s=1}^{p} sign|x_i^s - x_j^s|,$$

где X_i, X_j – битовые векторы.

1.14 Binary Robust Invariant Scalable Keypoints

Сокращенно BRISK [12]. Идея похожа на ORB, исполнение несколько отличается.

В BRISK слои пространства масштабирования состоят из n октав c_i и n внутренних октав d_i для $i = \{0, 1, ..., n-1\}$ и $n = 4$. Октавы формируются путём постепенной половинной выборки исходного изображения. Каждая внутренняя октава d_i расположена между слоями c_i и c_{i+1}. Первая внутренняя октава d_0 получается путём дискретизации исходного изображения c_0 в 1,5 раза, а остальные внутренние октавные слои получаются последовательной половинной выборкой. Следовательно, если t обозначает масштаб, то $t(c_i) = 2^i$ и $t(d_i) = 2^i \times 1.5$.

Ключевая концепция дескриптора BRISK – особый шаблон, используемый для выборки ключевой точки. Данный шаблон определяет N точек, равномерно распределенных по окружностям, концентричным по отношению к ключевой точке.

Чтобы избежать эффектов алиасинга при дискретизации интенсивности изображения точки p_i в шаблоне мы применяем сглаживание по Гауссу со стандартным отклонением σ_i пропорциональным расстоянию между точками соответствующем окружности. Позиционируя и масштабируя шаблон соответственно для конкретной ключевой точки k на изображении, рассмотрим одну из $N(N-1)/2$ пар точек дискретизации (p_i, p_j). Сглаженные значения интенсивности в этих точках равны $I(p_i, \sigma_i)$ и $I(p_j, \sigma_j)$ соответственно и используются для оценки локального градиента $g(p_i, p_j)$ по формуле:

$$g(p_i, p_j) = \frac{(p_j - p_i)I(p_j, \sigma_j) - I(p_i, \sigma_i)}{||p_j - p_i||^2},$$

где множество всех пар точек выборки

$$A = \{(p_i, p_j) \in R^2 \times R^2 | i < N \; and \; j < i \; and \; i, j \in N\}$$

Выделим два подмножества: множество пар с коротким и длинным расстоянием

$$S = \{(p_i, p_j) \in A \mid ||p_j - p_i|| < \delta_{max}\} \subseteq A,$$

$$L = \{(p_i, p_j) \in A \mid ||p_j - p_i|| < \delta_{min}\} \subseteq A.$$

Установим пороговое расстояние равным $\sigma_max = 9.75t$ и $\sigma_min = 13.67t$, где t есть маштаб k. Оценим общее направление характеристическое направление шаблон ключевой точки k следующем образом:

$$g = \begin{pmatrix} g_x \\ g_y \end{pmatrix} = \sum_{p_i, p_j \in L} g(p_i, p_j).$$

Пары с дальним расстояния используются для вычисления потому, что локальные градиенты аннигилируют друг друга и таким образом не нужны при определении глобального градиента.

Для формирования дескриптора, нормализованного по вращению и масштабу применяется шаблон выборки, повёрнутый на $\alpha = arctran2(g_y, g_x)$ в окрестности точки k. Битовый вектор дескриптор d_k получается путем выполнения всех сравнений интенсивности на коротком расстоянии пар точке $(p_i^\alpha, p_j^\alpha) in S$, таким образом каждый бит b соответствует:

$$b = \begin{cases} 1 : I(p_j^\alpha, \sigma_j) > I(p_i^\alpha, \sigma_i) \\ 0 : otherwise \end{cases} \quad \forall (p_i^\alpha, (p_j^\alpha) \in S.$$

BRISK использует детерминированный шаблон выборки, что приводит к единой точке выборки. Следовательно адаптированное сглаживание по Гауссу

не будет случайно искажать информационное содержание сравнения яркости на размытие двух близких точке выборки при сравнении.

BRISK использует значительно меньше точек выборки, чем попарные сравнения, ограничивая сложность поиска значений интенсивности.

Глава 2. Основной алгоритм

2.1 Алгоритм классификации файлов

В данной главе представлен алгоритм резервного копирования (рис. 4).

- При первом запуске бэкапа все файлы с телефона копируются на диск.
- При повторном использовании сравниваем файлы из полного бекапа и файлы, которые необходимо отслеживать и квалифицируем по следующим типам:
  - Если файл существует в отслеживаемой папке и присутствует в папке бэкапа, а также дата модификации файлов совпадают, то файлы одинаковые.
  - Если файл существует в отслеживаемой папке и присутствует в папке бэкапа, и дата модификации файла в телефоне новее, чем на диске, то файл обновлён.
  - Если файл существует в телефоне и не присутствует на диске, то файл спорный.
  - Если файл существует в папке бэкапа, но не присутствует в телефоне, то файл спорный.
  - Если для спорного файла на телефоне найден похожий спорный файл в папке бэкапа, то файл обновлен.
  - Если для спорного файла на диске не найден похожий спорный файл на телефоне, то файл удалён.
  - Если для спорного файла на телефоне не найден похожий спорный файл на диске, то файл новый.

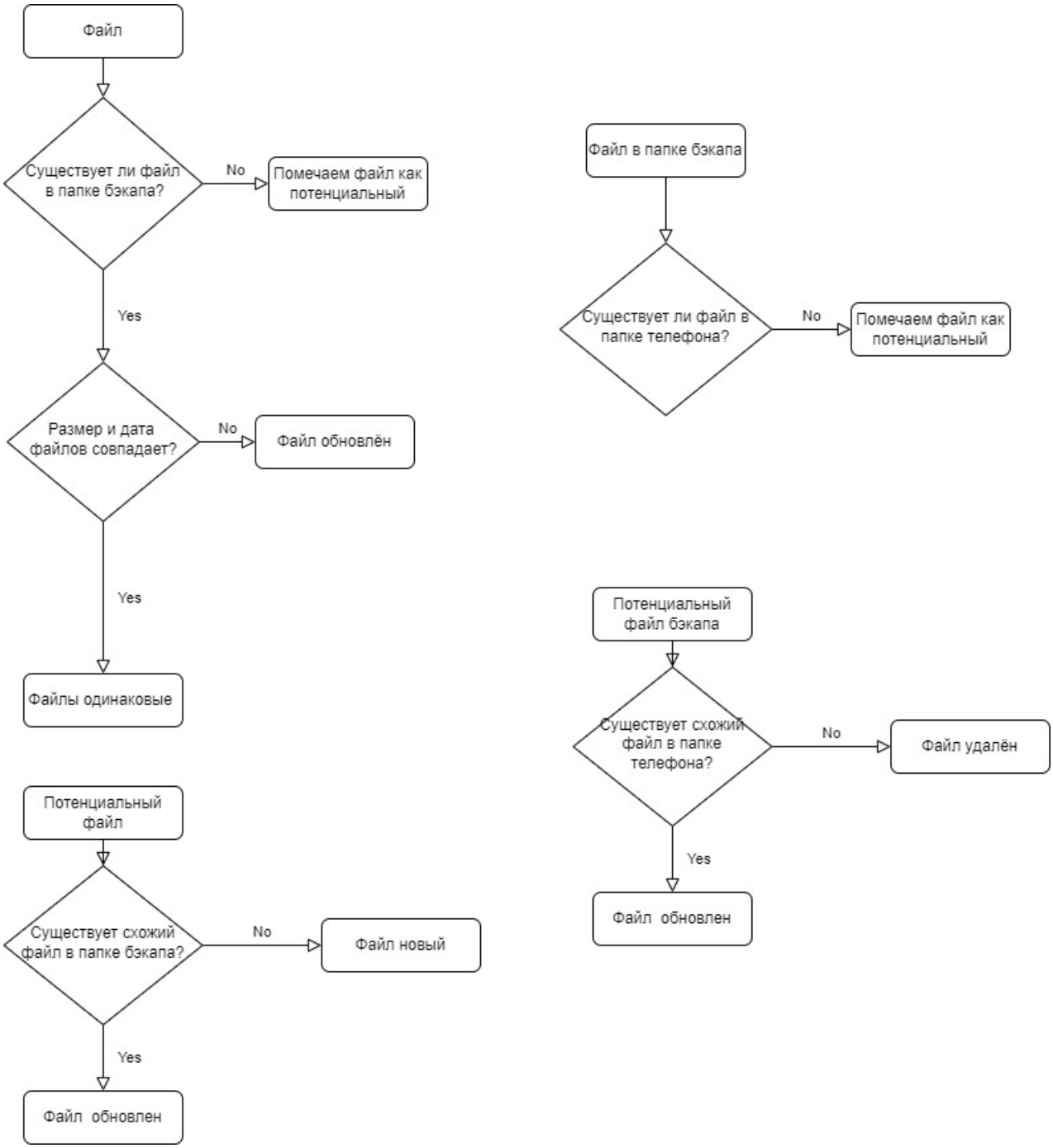

Рис. 4: Алгоритм классификации файлов

2.2 Обнаружение изменённых файлов

Поскольку мы не можем непрерывно отслеживать файлы на диске, а в частности переименование файла – диск не обладает микропроцессором, а также может быть подключен к устройству не через наш программный

комплекс, то необходимо использовать алгоритм позволяющие определять схожесть файлов программно.

Если файл был только переименован и не были изменены данные внутри файла, то для обнаружения этого фактора необходимо, чтобы у старого файла и нового совпадали наборы битов.

Если же файл был не только переименован, но и было изменено внутреннее содержимое файла, то алгоритмы позволяющие определить сходство можно разделить на два типа:

- Алгоритмы для файлов форматов, которые не используют сжатие данных такие как txt, word.
- Алгоритмы для файлов форматов, которые используют сжатие такие как jpg, webP.

Принципиальное отличие в том, что при изменения файла, использующего сжатие, вся последовательность бит может быть изменена.

Глава 3. Тестирование итогового комплекса

Поскольку изображения, полученные в результате редактирования пользователями отличаются от изображений которые может сгенерировать машина.

3.1 Обзор пользовательского интерфейса мобильного приложения

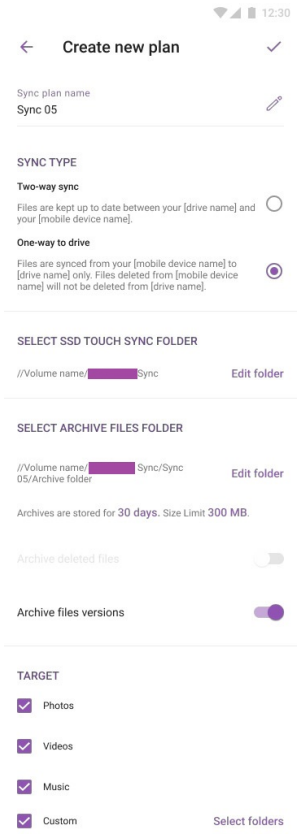

Рис. 5: Экран создания/редактирования плана

На экране создания плана (рис. 5) представляется возможность выбрать имя плана создания резервной копии. Доступен один из двух вариантов синхронизации:

- Two-way sync – файлы из телефона и жесткого диска синхронизируются. Файлы удаленные с телефона будут удалены и из жесткого диска.
- One-way sync – файлы из телефона синхронизируются с жестким диском. Файлы удалённые из телефона не удаляются с жесткого диска.

Также предоставляется возможность выбрать путь хранения резервной копии, архивировать удалённые файлы, выбор базовых типов файлов для архивации разбитых по категориям: фотографии, видео, музыка, и кастомный

тип. Последний позволяет выбрать конкретные папки для создания резервной копии.

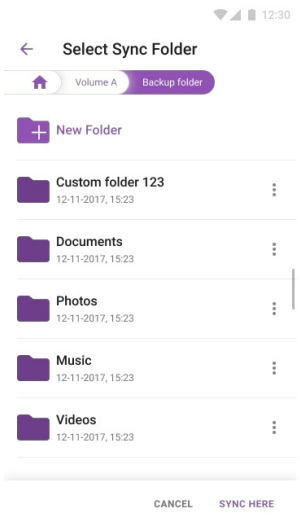

Рис. 6: Экран выбора папки для синхронизации

На экране выбора папки (рис. 6) пользователю предоставляется выбрать желаемые папки для синхронизации.

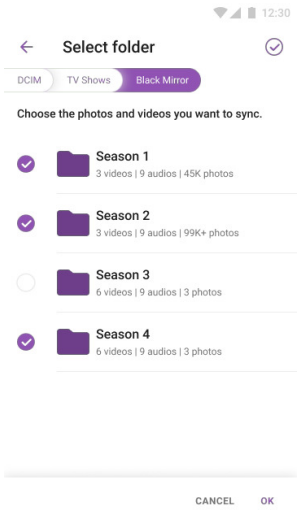

Рис. 7: Экран выбора папок для синхронизации

Аналогично на экране выбора папок для бэкапа (рис. 7) можно отметить желаемые папки для отслеживания.

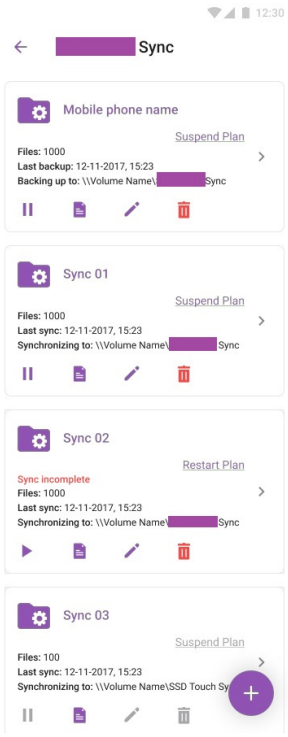

Рис. 8: Экран с планами синхронизаций

На экране статуса синхронизации (рис. 8) отображается список всех доступных планов. Возможно: запустить план, посмотреть отчёт предыдущей синхронизации, изменить параметры плана, и удалить план. Также пользователь может наблюдать общее количество файлов, последнее время синхронизации и путь к папке с файлами резервной копии.

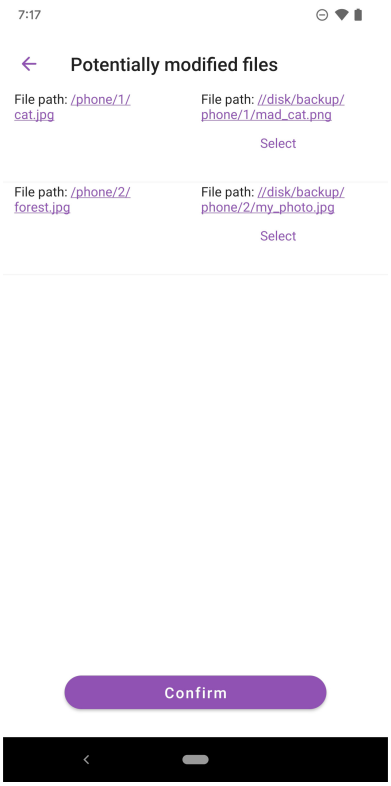

Рис. 9: Экран сравнения файлов

На экране изменённых файлов (рис. 9) каждому потенциально изменённому файлу соотносится наиболее схожий файл.

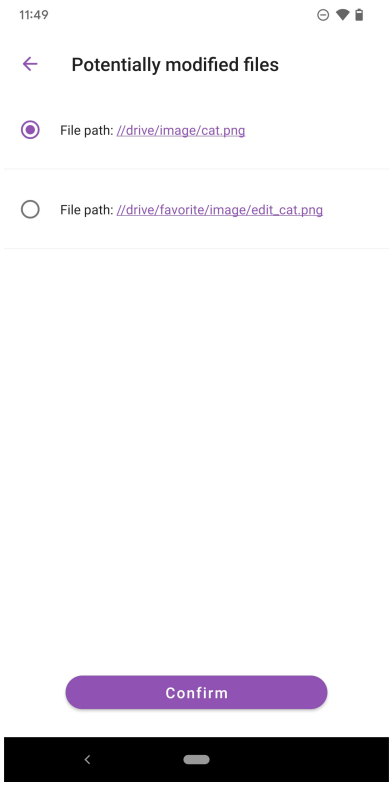

Рис. 10: Экран наиболее схожих файлов

На экране подробностей об изменённом файле (рис. 10) отображается список схожих файлов упорядоченных по близости схожести.

3.2 Основные выводы

Таким образом, разработанное нами приложение реализует поставленную изначально задачу разработки приложения резервного копирования. Кроме того, проведена оптимизация процесса создания резервной копии путём выбора оптимального варианта её реализации, файлы просматриваются и отбираются для копирования выборочно, с учётом их свойств на основе хорошо отработанных алгоритмов. Интерфейс приложения прост и понятен в использовании, что позволяет говорить о широких возможностях его применения самыми разными группами пользователей.

Несомненно, в перспективе стоит задуматься и о дальнейшей оптимизации приложения, например, с использованием данных о пользовании, их накоплением, систематизацией и анализом.

Заключение

Основные результаты работы:

- Разработан алгоритм классификации файлов для бэкапа.
- Создан программный комплекс для определения изменённых текстовых файлов и изображений.
- Алгоритм интегрирован в мобильное `Android`-приложение с количеством скачиваний более 10 тысяч.
- Получены положительные отзывы от пользователей.
- По результатам работы сделан доклад на LI Международной конференции "Процессы управления и устойчивость"(CPS'20) и опубликована статья [13].

Список литературы

[1] GlobalStats [Электронный ресурс]: URL:https://gs.statcounter.com/os-market-share/mobile/worldwide (дата обращения: 20.02.22).

[2] Techtarget [Электронный ресурс]: URL:https://www.techtarget.com/searchdatabackup/feature/Full-incremental-or-differential-How-to-choose-the-correct-backup-type (дата обращения: 20.02.22).

[3] Pair [Электронный ресурс]: URL:https://www.pair.com/support/kb/snapshots-and-backups-what-is-the-difference/ (дата обращения: 20.02.22).

[4] Cloudian [Электронный ресурс]: URL:https://cloudian.com/guides/data-protection/continuous-data-protection/ (дата обращения: 20.02.22).

[5] Nicholas Harbour, Defence Coumputer Forensics Lab, 2002.

[6] Jesse D. Kornblum. Identifying almost identical files using context triggered piecewise hashing // Digital Investigaton. 2006. P.91-97.

[7] Llyod Allison. Dynamic programming algorithm (DPA) for edit-distance // Monash University. 1997.

[8] Andrew Tridgell, Spamsum README, 2002.

[9] Zhou Wang, A. C. Bovik, H. R. Sheikh and E. P. Simoncelli. Image quality assessment: from error visibility to structural similarity // IEEE Transactions on Image Processing, vol. 13, no. 4, pp. 600-612, April 2004, doi: 10.1109/TIP.2003.819861.

[10] E. Rosten, R. Porter and T. Drummond. Faster and Better: A Machine Learning Approach to Corner Detection // IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 32, no. 1, pp. 105-119, Jan. 2010, doi: 10.1109/TPAMI.2008.275.

[11] E. Rublee, V. Rabaud, K. Konolige and G. Bradski. ORB: An efficient alternative to SIFT or SURF // 2011 International Conference on Computer Vision, 2011, pp. 2564-2571, doi: 10.1109/ICCV.2011.6126544.

[12] S. Leutenegger, M. Chli and R. Y. Siegwart. BRISK: Binary Robust invariant scalable keypoints // 2011 International Conference on Computer Vision, 2011, pp. 2548-2555, doi: 10.1109/ICCV.2011.6126542.

[13] Щербаков Г. А, Сачков А. В. Разработка мобильного приложения, учитывающего предпочтения пользователя // Процессы управления и устойчивость. 2020. Т. 7. № 1. С. 304–310.

Программный код приложения

Программа написана на языке Kotlin. Ниже представлены ключевые классы приложения.

UniformFuzzyHash реализует алгоритм fuzzy hashing
Входные данные – набор бит.
Выходные данные – хэш.

Листинг 1: UniformFuzzyHash

```
import java.util.*

class UniformFuzzyHash private constructor() {
    enum class SimilarityTypes(
    name: String
    ) {
        SIMILARITY("Similarity"), REVERSE_SIMILARITY(
        "Reverse"
        ),
        MAXIMUM("Maximum"), MINIMUM("Minimum"), ARITHMETIC_MEAN(
        "ArithMean"
        ),
        GEOMETRIC_MEAN(
        "GeomMean"
        );

        companion object {
            fun names(): List<String> {
                val similarityTypes = values()
                val similarityTypesNames: MutableList<String> =
                ArrayList(similarityTypes.size)
                for (similarityType in similarityTypes) {
                    similarityTypesNames.add(
                    similarityType.name
                    )
                }
                return similarityTypesNames
            }
        }
    }

    var factor = 0
    private set
```

```
34          private var dataSize = 0
35          private var blocks: MutableList<UniformFuzzyHashBlock>? =
36          null
37          private var blocksSet: Set<UniformFuzzyHashBlock>? =
38          null
39
40          constructor(
41          data: ByteArray?,
42          factor: Int
43          ) : this() {
44              if (data == null) {
45                  throw NullPointerException("Data is null.")
46              }
47              computeUniformFuzzyHash(data, factor)
48          }
49
50          private fun computeUniformFuzzyHash(
51          data: ByteArray,
52          factor: Int
53          ) {
54
55              // Factor check.
56              checkFactor(factor)
57
58              // Attributes assignment.
59              this.factor = factor
60              dataSize = data.size
61              blocks = LinkedList()
62
63              // Size in bytes of the rolling window.
64              // Size in bytes of factor + 5.
65              val windowSize = sizeInBytes(factor) + 5
66
67              // Window size shifter.
68              // Used to extract old data from the window.
69              // (2 ^ (8 * windowSize)) % factor.
70              val windowSizeShifter =
71              shiftBytesMod(windowSize, factor)
72
73              // Window hash match value to produce a block.
74              // Any number between 0 and factor - 1 should be valid.
75              val windowHashMatchValue = factor - 1
76
77              // Rolling window hash.
78              var windowHash: Long = 0
79
```

```
80              // Block hash.
81              var blockHash: Long = 0
82
83              // Block starting byte position (0 based).
84              var blockStartingBytePosition = 0
85
86              // Hash computation.
87              for (i in data.indices) {
88
89                  // Unsigned datum.
90                  val datum = ubyte(data[i])
91
92                  // Window hash shift, new datum addition and old datum extraction.
93                  if (i < windowSize) {
94                      windowHash =
95                      ((windowHash shl java.lang.Byte.SIZE) + datum) % factor
96                  } else {
97                      val oldDatum =
98                      ubyte(data[i - windowSize])
99                      windowHash = (
100                     (windowHash shl java.lang.Byte.SIZE) + datum -
101                     oldDatum * windowSizeShifter
102                     ) % factor
103
104                     // Due to the subtraction, the modulo result might be negative.
105                     if (windowHash < 0) {
106                         windowHash += factor.toLong()
107                     }
108                 }
109
110                 // Block hash shift and new datum addition.
111                 blockHash =
112                 ((blockHash shl java.lang.Byte.SIZE) + datum) %
113                 BLOCK_HASH_MODULO
114
115                 // Possible window hash match (block production).
116                 // Match is only checked if the initial window
117                 has already been computed.
118                 // Last data byte always produces a block.
119                 if (windowHash == windowHashMatchValue.toLong() &&
120                 i >= windowSize - 1 ||
121                 i == data.size - 1
122                 ) {
123
124                     // New block addition.
125                     (blocks as LinkedList<UniformFuzzyHashBlock>).add(
```

```
126                     UniformFuzzyHashBlock(
127                     blockHash.toInt(),
128                     blockStartingBytePosition,
129                     i
130                     )
131                     )
132
133                     // Block hash reset.
134                     blockHash = 0
135
136                     // Next block starting byte position.
137                     blockStartingBytePosition = i + 1
138                 }
139             }
140         }
141
142         override fun toString(): String {
143
144             // String builder.
145             // Initial capacity enough to build the full hash string.
146             val strB = StringBuilder(
147             ToStringUtils.FACTOR_WITH_SEP_MAX_CHARS +
148             ToStringUtils.BLOCK_WITH_SEP_MAX_CHARS *
149             blocks!!.size
150             )
151
152             // Factor.
153             strB.append(factor)
154             strB.append(ToStringUtils.FACTOR_SEPARATOR)
155
156             // Blocks.
157             var i = 0
158             for (block in blocks!!) {
159                 if (i++ != 0) {
160                     strB.append(ToStringUtils.BLOCKS_SEPARATOR)
161                 }
162                 block.toString(strB)
163             }
164             return strB.toString()
165         }
166
167         fun similarity(
168         other: UniformFuzzyHash?
169         ): Double {
170
171             // Parameters check.
```

```
172             if (other == null) {
173                 throw NullPointerException("The Uniform Fuzzy Hash is null.")
174             }
175             if (other === this) {
176                 return 1.0
177             }
178             require(other.factor == factor) {
179                 "The Uniform Fuzzy Hashes factors are different."
180             }
181             if (blocks!!.size == 0 || other.blocks!!.size == 0) {
182                 return 0.0
183             }
184 
185             // Sum of the sizes in bytes of the blocks of this Uniform Fuzzy Hash
186             // which are also in the
187             // introduced one.
188             var sizeSum = 0
189 
190             // Check which blocks of this Uniform Fuzzy
191             // Hash are in the set of blocks of the other
192             // Uniform Fuzzy Hash.
193             other.accessBlocksSet()
194             for (block in blocks!!) {
195                 if (other.blocksSet!!.contains(block)) {
196 
197                     // Add their size to the sum of sizes.
198                     sizeSum += block.blockSize
199                 }
200             }
201 
202             // Similarity computation.
203             return sizeSum.toDouble() / dataSize
204         }
205 
206         fun similarity(
207         other: UniformFuzzyHash?,
208         similarityType: SimilarityTypes?
209         ): Double {
210             if (other == null) {
211                 throw NullPointerException("The Uniform Fuzzy Hash is null.")
212             }
213             return when (similarityType) {
214                 SimilarityTypes.SIMILARITY -> this.similarity(
215                 other
216                 )
217                 SimilarityTypes.REVERSE_SIMILARITY -> other.similarity(
```

```
                this
                )
                else -> {
                    val similarity =
                    this.similarity(other)
                    val reverse =
                    other.similarity(this)
                    when (similarityType) {
                        SimilarityTypes.MAXIMUM -> if (similarity >= reverse) {
                            similarity
                        } else {
                            reverse
                        }
                        SimilarityTypes.MINIMUM -> if (similarity >= reverse) {
                            reverse
                        } else {
                            similarity
                        }
                        SimilarityTypes.ARITHMETIC_MEAN ->
                        (similarity + reverse) / 2
                        SimilarityTypes.GEOMETRIC_MEAN -> Math.sqrt(
                        similarity * reverse
                        )
                        else -> similarity
                    }
                }
            }
        }

        override fun equals(
        obj: Any?
        ): Boolean {
            if (obj == null) {
                return false
            }
            if (this === obj) {
                return true
            }
            if (obj is UniformFuzzyHash) {
                val other = obj
                if (factor != other.factor) {
                    return false
                }
                if (dataSize != other.dataSize) {
                    return false
                }
```

```
264                 if (blocks!!.size != other.blocks!!.size) {
265                     return false
266                 }
267                 val thisBlocksIterator: Iterator<UniformFuzzyHashBlock> =
268                 blocks!!.iterator()
269                 val otherBlocksIterator: Iterator<UniformFuzzyHashBlock> =
270                 other.blocks!!.iterator()
271                 while (thisBlocksIterator.hasNext()) {
272                     if (thisBlocksIterator.next() != otherBlocksIterator.next()) {
273                         return false
274                     }
275                 }
276                 return true
277             }
278             return false
279         }
280 
281         override fun hashCode(): Int {
282             val prime = 31
283             var result = 1
284             result = prime * result + factor
285             result = prime * result + dataSize
286             result = prime * result + blocks!!.size
287             return result
288         }
289 
290         fun getBlocks(): List<UniformFuzzyHashBlock> {
291             return Collections.unmodifiableList(blocks)
292         }
293 
294         protected fun accessBlocksSet(): Set<UniformFuzzyHashBlock> {
295             if (blocksSet == null) {
296                 blocksSet = HashSet(blocks)
297             }
298             return blocksSet!!
299         }
300 
301         companion object {
302             protected const val BLOCK_HASH_MODULO =
303             Int.MAX_VALUE
304 
305             fun checkFactor(
306             factor: Int
307             ) {
308                 require(factor > 2) { "Factor must be greater than 2." }
309                 require(factor % 2 != 0) { "Factor must be odd." }
```

```
310             }
311
312             private fun sizeInBytes(
313             number: Int
314             ): Int {
315                 return (
316                 Integer.SIZE - Integer.numberOfLeadingZeros(
317                 number
318                 ) - 1
319                 ) / java.lang.Byte.SIZE + 1
320             }
321
322             private fun shiftBytesMod(
323             bytesShift: Int,
324             modulo: Int
325             ): Int {
326                 var ret: Long = 1
327                 for (i in 0 until bytesShift) {
328                     ret =
329                     (ret shl java.lang.Byte.SIZE) % modulo
330                 }
331                 return ret.toInt()
332             }
333
334             private fun ubyte(
335             b: Byte
336             ): Int {
337                 return if (b >= 0) {
338                     b.toInt()
339                 } else {
340                     b.toInt() - 2 * Byte.MIN_VALUE
341                 }
342             }
343         }
344     }
```

Листинг 2: FuzzySimularitySource

```
1   Класс FuzzySimularitySource
2   \\На вход принимается список документов с телефона
3   и файлы с жёсткого диска.
4   \\На выходе каждому файлу соотносится
5   вектор индексов позиции файлов упорядоченных по убыванию схожести.
6   import android.content.Context
7   import androidx.core.net.toUri
```

```
8   import com.seagate.sdk.io.domain.IIO
9   import com.seagate.tote.home.filelist.domain.entity.FileMetaDataEntity
10  import io.reactivex.rxjava3.core.Single
11  import io.reactivex.rxjava3.kotlin.toObservable
12  import io.reactivex.rxjava3.kotlin.zipWith
13  import toothpick.InjectConstructor
14  import java.io.ByteArrayOutputStream
15  import java.io.InputStream
16
17  @InjectConstructor
18  class FuzzySimilaritySource(
19  private val context: Context,
20  private val io: IIO
21  ) {
22
23      fun compareTwoFiles(
24      list: List<FileMetaDataEntity>,
25      otherList: List<FileMetaDataEntity>
26      ): Single<List<List<Int>>> {
27          return getFuzzyHash(list).zipWith(
28          getFuzzyHash(
29          otherList
30          )
31          )
32          .map { (hashList, otherHashList) ->
33              hashList.map {
34                  getNearest(it, otherHashList)
35              }
36          }
37      }
38
39      private fun getNearest(
40      hash: UniformFuzzyHash,
41      hashList: List<UniformFuzzyHash>
42      ) =
43      hashList.mapIndexed { index, otherHash ->
44          index to otherHash.similarity(
45          hash,
46          UniformFuzzyHash.SimilarityTypes.SIMILARITY
47          )
48      }
49      .sortedBy { it.second }
50      .map { it.first }
51
52      private fun getFuzzyHash(
53      fileList: List<FileMetaDataEntity>
```

```
54          ): Single<List<UniformFuzzyHash>> =
55          fileList.toObservable()
56          .concatMapSingle { io.getFileUri(it.path) }
57          .map {
58              context.contentResolver.openInputStream(
59              it.toUri()
60              )!!
61          }
62          .map { getFuzzyHash(it) }
63          .toList()
64
65          private fun getFuzzyHash(
66          inputStream: InputStream
67          ): UniformFuzzyHash {
68              val os =
69              ByteArrayOutputStream(inputStream.available())
70              val buffer = ByteArray(4096)
71              var bytesRead: Int
72              while (inputStream.read(buffer)
73              .also { bytesRead = it } != -1
74              ) {
75                  os.write(buffer, 0, bytesRead)
76              }
77              inputStream.close()
78              return UniformFuzzyHash(
79              os.toByteArray(),
80              7
81              )
82          }
83      }
```

Класс ImageSimularitySource

На вход принимается список изображений с телефона и файлы с жёсткого диска, а также тип детектора для определения ключевых точек.

На выходе каждому файлу соотносится вектор индексов позиции файлов упорядоченных по убыванию схожести.

Листинг 3: ImageSimularitySource

```
1   import android.content.Context
2   import androidx.core.net.toUri
3   import com.seagate.sdk.io.domain.IIO
4   import com.seagate.tote.home.filelist.domain.entity.FileMetaDataEntity
5   import io.reactivex.rxjava3.core.Single
```

```
6   import io.reactivex.rxjava3.kotlin.toObservable
7   import io.reactivex.rxjava3.kotlin.zipWith
8   import org.opencv.android.OpenCVLoader
9   import org.opencv.core.CvType
10  import org.opencv.core.Mat
11  import org.opencv.core.MatOfDMatch
12  import org.opencv.core.MatOfKeyPoint
13  import org.opencv.features2d.*
14  import org.opencv.imgcodecs.Imgcodecs
15  import toothpick.InjectConstructor
16  import java.io.ByteArrayOutputStream
17  import java.io.InputStream
18
19  @InjectConstructor
20  class ImageSimilaritySource(
21  private val context: Context,
22  private val io: IIO
23  ) {
24
25      val detectorList by lazy {
26          listOf(
27          ORB.create(),
28          FastFeatureDetector.create(),
29          BRISK.create()
30          )
31      }
32
33      private val matcher by lazy {
34          DescriptorMatcher
35          .create(DescriptorMatcher.BRUTEFORCE_HAMMING)
36      }
37
38      private val minDistance = 10
39
40      fun compareFiles(
41      list: List<FileMetaDataEntity>,
42      otherList: List<FileMetaDataEntity>,
43      detector: Feature2D = ORB.create()
44      ): Single<List<List<Int>>> {
45          OpenCVLoader.initDebug()
46          return getDescriptors(
47          list,
48          detector
49          ).zipWith(
50          getDescriptors(
51          otherList,
```

```
52              detector
53              )
54              )
55              .map {
56                  getMatches(
57                  it.first,
58                  it.second
59                  )
60              }
61          }
62
63          private fun getMatches(
64          descriptors: List<Mat>,
65          otherDescriptors: List<Mat>
66          ): List<List<Int>> {
67              return descriptors.map { descriptor ->
68                  val vectorMatches = otherDescriptors
69                  .map { otherDesriptor ->
70                      val matches = MatOfDMatch()
71                      matcher.match(
72                      descriptor,
73                      otherDesriptor,
74                      matches
75                      )
76                      matches.toList()
77                      .filter { it.distance < minDistance }.size
78                  }
79                  vectorMatches
80              }
81          }
82
83          private fun getDescriptors(
84          list: List<FileMetaDataEntity>,
85          detector: Feature2D
86          ): Single<List<Mat>> =
87          list.toObservable()
88          .concatMapSingle { loadResource(it) }
89          .map { image ->
90              getDescriptors(
91              image,
92              detector
93              )
94          }
95          .toList()
96
97          private fun getDescriptors(
```

```
98          image: Mat,
99          detector: Feature2D
100         ): Mat {
101             val descriptors = Mat()
102             val keypoints = MatOfKeyPoint()
103             detector.detect(image, keypoints)
104             detector.compute(
105             image,
106             keypoints,
107             descriptors
108             )
109             return descriptors
110         }
111
112         private fun loadResource
113         (fileMetaDataEntity: FileMetaDataEntity): Single<Mat> =
114         io.getFileUri(fileMetaDataEntity.path)
115         .map {
116             context.contentResolver.openInputStream(
117             it.toUri()
118             )!!
119         }
120         .map { loadResource(it) }
121
122         private fun loadResource(
123         inputStream: InputStream,
124         flags: Int = -1
125         ): Mat {
126             val os =
127             ByteArrayOutputStream(inputStream.available())
128             val buffer = ByteArray(4096)
129             var bytesRead: Int
130             while (inputStream.read(buffer)
131             .also { bytesRead = it } != -1
132             ) {
133                 os.write(buffer, 0, bytesRead)
134             }
135             inputStream.close()
136             val encoded =
137             Mat(1, os.size(), CvType.CV_8U)
138             encoded.put(0, 0, os.toByteArray())
139             os.close()
140             val decoded =
141             Imgcodecs.imdecode(encoded, flags)
142             encoded.release()
143             return decoded
```

```
144             }
145         }
```

Классы ORB, FAST, BREAK реализованы в библиотеке OpenCV.