Министерство образования Российской Федерации

МОСКОВСКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ им. Н.Э. БАУМАНА

Факультет: Информатика и системы управления
Кафедра: Информационная безопасность (ИУ8)

АЛГОРИТМЫ И СТРУКТУРЫ ДАННЫХ

Домашняя работа на тему:

«Реализация алгоритма для поиска похожих изображений»

Вариант 24

Преподаватель:
Чесноков В.О.

Ключарёв П. Г.

Студент:
Шапран А.В.

Группа:
ИУ8-54

Москва 2018

Содержание

ТЕОРЕТИЧЕСКАЯ ЧАСТЬ

Введение

С каждым годом объём обрабатываемой информации неуклонно растёт, что требует формализации и последующей алгоритмизации процессов, ранее выполнявшихся вручную. Одним из ключевых понятий в автоматической обработке информации является поиск объектов определённого класса, в частности, поиск изображений в больших базах данных. Кроме того, необходимо, чтобы такой поиск осуществлялся быстро и качественно.

По мере расширения доступа к электронным архивам изображений и видео возрастает значимость поиска изображения по его содержанию. Это объясняется тем, что значительная часть информации, поступающей из окружающего мира, воспринимается нами именно в зрительной форме.

Обзор существующих алгоритмов поиска изображений

Существующие методы поиска изображений в базах данных можно разделить на два вида:

1) поиск по текстовому описанию (Description-Based Image Retrieval);

2) поиск по визуальному содержанию (Content-Based Image Retrieval).

Поиск изображений на основе содержания имеет наибольшую популярность и является относительно новой прикладной областью, в которой применяются методы компьютерного зрения.

Характеристики сходства изображений можно разделить на три основные группы:

1) цветовое сходство;
2) текстурное сходство;
3) сходство формы.

Первой коммерческой системой поиска изображений является QBIC компании IBM, 1995 год. Поиск изображения в QBIC осуществляется

преимущественно по цветовым характеристикам. Такой метод не отвечает за смысловое содержание изображения.

В 1999 году Дэвидом Лоу (David G. Lowe) был предложен алгоритм SIFT (Scale-invariant feature transform). Данный алгоритм производит сравнение изображений по улучшенной мозаичной технике и опирается на их смысловые составляющие. Также он позволяет сравнивать изображения, подвергнутые таким трансформациям как зашумление, изменение масштаба, смещение объекта на сцене, повороты камеры или объекта.

В 2006 году H. Bay, T. Tuytelaars и L. Van Gool занимались развитием алгоритма SIFT и предложили его улучшенную версию SURF (Speeded Up Robust Features). Этот метод инвариантен к масштабированию и вращению, как алгоритм SIFT, но имеет значительно большую скорость вычислений.

В этом же году Edward Rosten выдвинул алгоритм FAST (the Features from Accelerated Segment Test), который имел великолепное улучшение в скорости, но давал неточные результаты и не был инвариантным к поворотам изображения. Однако этот алгоритм был одним из первых эвристических методов поиска особых точек изображения и использовался как часть более сложных алгоритмов.

Развитием алгоритма FAST и быстрого эвристического дескриптора BRIEF (Binary Robust Independent Elementary Features), предназначенного для характеристики окрестности каждой особой точки изображения, стал алгоритм ORB (Oriented FAST and Rotated BRIEF), предложенный Ethan Rublee, Vincent Rabaud, Kurt Konolige и Gary Bradski в 2011 году. Алгоритм ORB оказался достаточно быстрым и эффективным, так как получил улучшения не только в скорости, но и в стойкости к повернутым, зашумленным и отмасштабированным изображениям.

Основываясь на проведенном обзоре существующих алгоритмов, для реализации алгоритма поиска похожих изображений был выбран алгоритм ORB.

Алгоритм ORB

Работу алгоритма ORB можно представить в виде блок-схемы, как показано на рисунке 1.

Рисунок 1. Блок-схема работы алгоритма ORB

Согласно приведенной блок-схеме работы алгоритма ORB проводятся следующие этапы:

1) Поиск положения особых точек на исходном изображении и на нескольких изображениях из пирамиды уменьшенных изображений, используя алгоритм FAST

Для принятия решения о том, считать заданную точку C особой или нет, в этом методе рассматривается яркость пикселей на окружности с центром в точке C и радиусом 3, при этом периферийные пиксели нумеруются по часовой стрелке от 1 до 16. Таким образом, если среди этих 16 пикселей есть последовательность из n подряд идущих пикселей, удовлетворяющих уравнению (1), то можно рассматривать точку C как кандидат на особую.

$$|I_x - I_C| > t \quad (1)$$

Здесь t – некоторый заранее заданный фиксированный порог по яркости, I_x – яркость каждого пикселя из последовательности подряд идущих пикселей размера n, I_C – яркость пикселя в точке C.

Для достижения быстрых вычислений, как правило, n выбирается равным 12 или 9. Это зависит от конкретных требований. В алгоритме ORB n берется равным 9.

Для того, чтобы ускорить процесс, сначала проверяются только четыре пикселя под номерами: 1, 5, 9, 13. Если среди них есть 3 пикселя светлее или темнее, то выполняется полная проверка по 16 точкам, иначе – точка сразу помечается как не особая. Это сильно сокращает время работы, для принятия решения в среднем достаточно опросить всего лишь около 4 точек окружности.

На рисунке 2 представлен пример проверки точки C на принадлежность к классу особых точек.

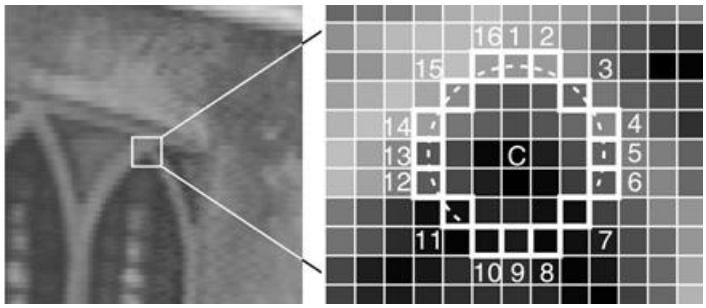

Рисунок 2. Пример проверки точки C алгоритмом FAST

2) Выбор N лучших точек, используя меру Харриса

Для обнаруженных точек вычисляется мера Харриса, кандидаты с низким значением меры Харриса отбрасываются.

То есть из точек, которые являются кандидатами на особые точки, выбираются те, у которых значение меры Харриса достаточно велико, и эти точки обозначаются как особые.

Для точки (x_0, y_0) рассматривается следующая функция:

$$S(u,v) = \sum_{x,y \in S} w(x,y)\big(Z(x+u, y+v) - Z(x,y)\big)^2 \qquad (2)$$

Здесь Z – функция интенсивности по координатным осям, S – некоторая окрестность точки (x_0, y_0), а $w(x,y)$ – весовая функция. В качестве весовой функции можно использовать гауссову функцию:

$$w(x,y) = \frac{1}{2\pi\sigma^2} e^{-\frac{(x-x_0)^2+(y-y_0)^2}{2\sigma^2}} \qquad (3)$$

Здесь σ – это параметр, который определяет уровень размытия.

Раскладывая $Z(x+u, y+v)$ в ряд Тейлора до членов первого порядка, получим:

$$S(u,v) = (u,v) \times \begin{pmatrix} \sum_{x,y \in S} \boldsymbol{w}(x,y)\big(Z_x(x,y)\big)^2 & \sum_{x,y \in S} \boldsymbol{w}(x,y) Z_x(x,y) Z_y(x,y) \\ \sum_{x,y \in S} \boldsymbol{w}(x,y) Z_x(x,y) Z_y(x,y) & \sum_{x,y \in S} \boldsymbol{w}(x,y)\left(Z_y(x,y)\right)^2 \end{pmatrix} \times \binom{u}{v}$$

$$S(u,v) = (u,v) \times \boldsymbol{A} \times \binom{u}{v} \qquad (4)$$

Где Z_x, Z_y – частные производные по x и по y функции интенсивности.

Анализируя собственные значения матрицы A, делается вывод относительно того, является ли точка (x_0, y_0) особой точкой или нет. Но так как вычислять собственные числа затратно, то мерой Харриса, показывающей, является ли точка особой, служит:

$$\mu(x_0, y_0) = detA - k(traceA)^2 \qquad (5)$$

Здесь k – эмпирическая константа, $k = 0{,}04 \div 0{,}06$;

$$detA = \sum_{x,y \in S} \boldsymbol{w}(x,y)\left(Z_x(x,y)\right)^2 * \sum_{x,y \in S} \boldsymbol{w}(x,y)\left(Z_y(x,y)\right)^2 - \left(\sum_{x,y \in S} \boldsymbol{w}(x,y) Z_x(x,y) Z_y(x,y)\right)^2$$

$$traceA = \sum_{x,y \in S} \boldsymbol{w}(x,y)\left(Z_x(x,y)\right)^2 + \sum_{x,y \in S} \boldsymbol{w}(x,y)\left(Z_y(x,y)\right)^2$$

Особыми считаются точки, мера Харриса которых больше, чем мера Харриса всех точек некоторой окрестности S_1, а значение меры Харриса превышает среднее значение по изображению.

3) Вычисление угла ориентации особой точки

После того, как особые точки найдены, вычисляются их дескрипторы, т.е. наборы признаков, характеризующие окрестность каждой особой точки.

BRIEF – быстрый эвристический дескриптор, строится из 256 бинарных сравнений между яркостями пикселей на размытом изображении. Бинарный тест между точками x и y определяется так:

$$\tau(p,u,v) = \begin{cases} 1, \text{если } p(u) < p(v); \\ 0, \text{если } p(u) \geq p(v). \end{cases} \quad (6)$$

Здесь $p(u)$ – интенсивность пикселя в точке u.

Координаты точек u и v определяются путем выбора произвольных значений x и y из диапазона $(0, \frac{1}{25}S^2)$, где S – заранее заданный размер патча (окрестности) каждой особой точки.

Полученный набор бинарных дескрипторов оказывается устойчивым к сменам освещения, перспективному искажению, быстро вычисляется и сравнивается, но очень неустойчив к вращению в плоскости изображения.

Алгоритм ORB может решить эту проблему. Вычисляется угол ориентации особой точки. Для этого, сначала вычисляются моменты яркости для окрестности особой точки:

$$m_{pq} = \sum_{x,y} x^p y^q I(x,y) \quad (7)$$

Здесь x, y – координаты пикселей, расположенных на окружности некоторого радиуса r, I – яркость.

Затем вычисляется центр тяжести - «центроид ориентации»:

$$C = (\frac{m_{10}}{m_{00}}, \frac{m_{01}}{m_{00}}) \quad (8)$$

Таким образом, остается определить угол ориентации особой точки:

$$\theta = arctan\left(\frac{m_{01}}{m_{10}}\right) = \arctan(\frac{\sum_{x,y} yI(x,y)}{\sum_{x,y} xI(x,y)}) \quad (9)$$

4) Поворот последовательности точек для бинарных сравнений в дескрипторе BRIEF

Имея угол ориентации особой точки, последовательность точек для бинарных сравнений в дескрипторе BRIEF поворачивается в соответствие с этим углом, как например, на рисунке 3.

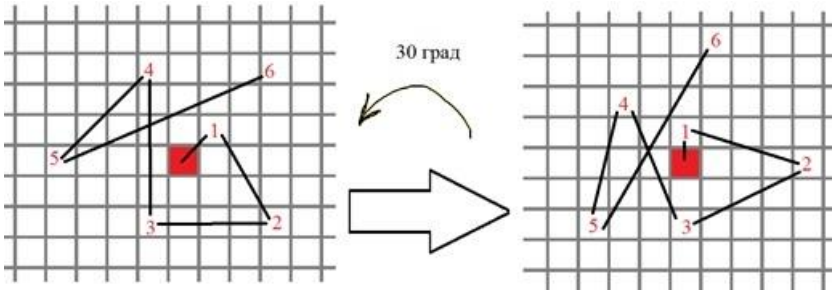

Рисунок 3. Пример поворота на 30 градусов против часовой стрелки точки из последовательности точек для бинарных сравнений в дескрипторе BRIEF

Новые положения для точек бинарных тестов вычисляются следующим образом:

$$\begin{pmatrix} x_i' \\ y_i' \end{pmatrix} = R(\theta) \begin{pmatrix} x_i \\ y_i \end{pmatrix} \quad (10)$$

Здесь $R(\theta) = \begin{pmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{pmatrix}$ – матрица поворота.

5) Вычисление двоичного дескриптора BRIEF. Поиск низкокорреляционных пиксельных блоков в жадном алгоритме

По полученным точкам вычисляется бинарный дескриптор BRIEF.

При повороте всех особых точек до нулевого угла, случайный выбор бинарных сравнений в дескрипторе перестаёт быть случайным. Необходимо произвести поиск низкокорреляционных пиксельных блоков в жадном алгоритме, который может выглядеть следующим образом:

1. Вычисление результата всех тестов для всех особых точек;
2. Упорядочивание всего множества тестов по их дистанции от среднего 0.5;
3. Создание списка R, который будет содержать отобранные «хорошие» тесты;
4. Добавление в R первого теста из отсортированного множества;
5. Сравнение следующего теста со всеми тестами в R. Если корреляция больше пороговой, то отбрасывается новый тест, иначе – добавляется;
6. Повтор п. 5 пока не наберётся нужное число тестов.

Пример результата проведения данного алгоритма представлен на рисунке 4.

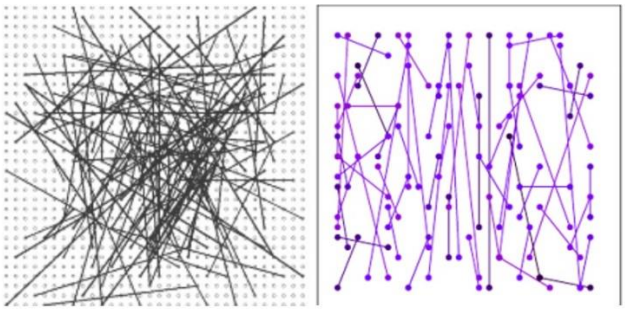

Рисунок 4. Пример поиска низкокорреляционных пиксельных блоков в жадном алгоритме

ИНСТРУКЦИЯ ПО ИСПОЛЬЗОВАНИЮ ПРОГРАММЫ

Программа была написана в интегрированной среде разработки программного обеспечения Microsoft Visual Studio на языке C++ с использованием сторонней библиотеки компьютерного зрения с открытым исходным кодом OpenCV. Для упрощения работы чтения изображений, то есть получения данных об изображении (значений RGB для каждого пикселя и тому подобное), было принято решение работать только с изображениями формата bmp (поддерживаются bmp файлы версий 1 – 5) либо производить конвертацию из другого формата в bmp.

Для более корректной работы программы с помощью подключенной библиотеки OpenCV изображения приводятся к одному масштабу 320x240 (нормировка изображений).

Компиляция программы:

Самый простой способ скомпилировать программу – это скачать исходники с github и запустить решение в Visual Studio.

ОПИСАНИЕ ЛОГИКИ ПРОГРАММЫ

Логика программы, реализующей алгоритм поиска похожих изображений ORB, в данном случае построена следующим образом:

1) Выбор режима работы программы

Работа программы разделена на несколько режимов, зависящих от входных данных (количества аргументов командной строки и команд, записанных в текстовых файлах). Всего есть три режима: режим построения дескрипторов для каждого изображения в директории, режим поиска похожего изображения в директории с уже обработанными изображениями и режим тестирования алгоритма. Режим тестирования включает в себя как построение дескрипторов, так и поиск похожего изображения в директории, а также вычисление результатов тестирования в виде ответа «$True$», если тест прошел успешно, и «$False$», в противном случае, который записывается в отдельный файл.

Пункты 2 – 4 выполняются в каждом режиме.

2) Чтение информации из изображения

Для реализации чтения информации из изображения используются следующие структуры для хранения данных изображения и класс для считывания этих данных:

Листинг 1. Структуры данных изображения

```
1. // CIEXYZTRIPLE значения
2. typedef int FXPT2DOT30;
3.
4. typedef struct
5. {
6.         FXPT2DOT30 ciexyzX;
7.         FXPT2DOT30 ciexyzY;
8.         FXPT2DOT30 ciexyzZ;
9. } CIEXYZ;
10.
11. typedef struct
12. {
```

```
13.       CIEXYZ  ciexyzRed;
14.       CIEXYZ  ciexyzGreen;
15.       CIEXYZ  ciexyzBlue;
16. } CIEXYZTRIPLE;
17.
18. // Заголовок файла растрового изображения
19. typedef struct
20. {
21.       unsigned short bfType;
22.       unsigned int   bfSize;
23.       unsigned short bfReserved1;
24.       unsigned short bfReserved2;
25.       unsigned int   bfOffBits;
26. } BITMAPFILEHEADER;
27.
28. // Информационный заголовок
29. typedef struct
30. {
31.       unsigned int   biSize;
32.       unsigned int   biWidth;
33.       unsigned int   biHeight;
34.       unsigned short biPlanes;
35.       unsigned short biBitCount;
36.       unsigned int   biCompression;
37.       unsigned int   biSizeImage;
38.       unsigned int   biXPelsPerMeter;
39.       unsigned int   biYPelsPerMeter;
40.       unsigned int   biClrUsed;
41.       unsigned int   biClrImportant;
42.       unsigned int   biRedMask;
43.       unsigned int   biGreenMask;
44.       unsigned int   biBlueMask;
45.       unsigned int   biAlphaMask;
46.       unsigned int   biCSType;
47.       CIEXYZTRIPLE   biEndpoints;
48.       unsigned int   biGammaRed;
49.       unsigned int   biGammaGreen;
50.       unsigned int   biGammaBlue;
51.       unsigned int   biIntent;
52.       unsigned int   biProfileData;
53.       unsigned int   biProfileSize;
54.       unsigned int   biReserved;
55. } BITMAPINFOHEADER;
56.
57. // Данные изображения RGB, значения яркости (интенсивности) для пикселя, его
58. //координаты и маркер, показывающий является ли пиксель особой точкой
59. typedef struct
60. {
61.       unsigned char  rgbBlue;
62.       unsigned char  rgbGreen;
```

```
63.       unsigned char  rgbRed;
64.       unsigned char  rgbReserved;
65.       unsigned int  intensity;
66.       unsigned int x;
67.       unsigned int y;
68.       bool special;
69. } PIXEL;
```

Листинг 2. Класс считывания данных изображения

```
1.  // Класс - изображение формата bmp с поддержкой 5 версий bmp и с 16-, 24-, 32-
2.  //битами на пиксель
3.  class BMP
4.  {
5.  public:
6.        BITMAPFILEHEADER file_header;
7.        BITMAPINFOHEADER file_info_header;
8.        std::vector<std::vector<PIXEL>> pixels; //  Вектор данных для каждого
    пикселя
9.        BMP() {}
10.       BMP(std::string file_name);
11.       ~BMP();
12.       void get_file_header(); //  Заполнение структуры BITMAPFILEHEADER
13.       void get_file_info_header(); // Заполнение структуры BITMAPINFOHEADER
14.       void get_pixels_info(); // Получение данных RGB, интенсивности ...
    каждого пикселя
15.       void print_pixels_info(); // Печать значений для каждого пикселя
    изображения (не используется в реализации алгоритма)
16.       int image_read(std::string img_path); // Функция получения данных
    изображения
17. private:
18.       const unsigned int height = { 240 };
19.       const unsigned int width = { 320 };
20.       std::ifstream file_stream;
21.       template <typename Type>
22.       void read(std::ifstream & fp, Type & result, std::size_t size) // Функция
    чтения блока байтов
23.       {
24.             fp.read(reinterpret_cast<char *>(&result), size);
25.       }
26.       unsigned char bit_extract(const unsigned int byte, const unsigned int
    mask); // Функция извлечения байтов
27.       void image_resize(std::string img_path); // Функция изменения размеров
    изображения
28.       void image_convert_to_bmp(std::string & img_path); // Функция конвертации
    в bmp
29. };
```

Сначала создается объект класса BMP затем вызывается функция получения данных изображения $image\_read(std::string\ img\_path)$, в которой происходит изменение размеров изображения, преобразования его в черно-белое (grayscale) и его конвертация при необходимости, вызов функций для заполнения описанных структур $get\_file\_header()$ и $get\_file\_info\_header()$ и функции получения данных о каждом пикселе изображения $get\_pixels\_info()$.

3) Поиск особых точек на изображении

Поиск особых точек производится в несколько этапов: обнаружение особых точек на изображении с помощью алгоритма FAST, отбор угловых точек с помощью детектора углов Харриса и определения локальных максимумов (отбор производится до тех пор, пока число отобранных точек не окажется меньше заданного параметра). Поиск особых точек реализован в классе $SpecialPoints$.

Листинг 3. Класс, реализующий поиск особых точек

```
1. class SpecialPoints
2. {
3. public:
4.       std::vector<std::pair<PIXEL, double>> special_points; // Вектор особых точек, хранящий пары - данные пикселя, значение меры Харриса
5.       SpecialPoints(std::vector<std::vector<PIXEL>> pixels);
6.       ~SpecialPoints();
7.       void fast(); // Поиск особых точек алгоритмом FAST
8.       void print_special_points(); // Вывод данных особых точек (в реализации алгоритма не используется)
9.       void get_processed_image(std::string file_name); // Сохранение обработанного изображения с нанесенными на него особыми точками
10. private:
11.      const unsigned int height = { 240 };
12.      const unsigned int width = { 320 };
13.      std::vector<std::vector<PIXEL>> pixels; //  Вектор данных для каждого пикселя
14.      const int fast_threshold = { 20 }; // Заранее заданный фиксированный порог по яркости
15.      const unsigned int radius = { 3 }; // Радиус окружности с центром в рассматриваемой точке (пикселе)
```

```
16.       const float harris_k = { 0.04f }; // Эмпирическая константа, использующаяся при вычислении меры Харриса
17.       unsigned int harris_threshold = { 4096 }; // Краевой порог
18.       const unsigned int max_count = { 2048 }; // Максимальное число особых точек
19.       const float two_pi = {6.28318531};
20.       const unsigned int sigma = {2}; // Значение сигмы в функции Гаусса (чем меньше сигма, тем боле крутой является функция Гаусса [функция Гаусса будет смотреть на небольшую окрестность изображения])
21.       std::deque<std::pair<unsigned int, unsigned int>> circle; // очередь - окружность, если используется для алгоритма fast, то очередь включает 1-16 пиксели, после 16 идет дополнительно 1-8 пиксели
22.       void make_circle(const unsigned int & y, const unsigned int & x, const unsigned int & radius, bool fast_alg); // Заполняет очередь координатами пикселей, лежащих на окружности с центром в (x,y) и радиусом radius
23.       void harris_selection(); // Выбор N особых точек используя меру Харриса и определения локальных максимумов
24.       double w(const unsigned int & x_0, const unsigned int & y_0, const unsigned int & x,const unsigned int & y); // Весовая функция - Функция Гаусса, обеспечивающая сглаживание
25.       bool harris_response(const int & start_x, const int & start_y, const std::vector<std::pair<PIXEL, double>>::iterator & it, std::vector<int> & indexes, double & res); // Вычисление меры Харриса для сдвинутого окна
26. };
```

Сначала создается объект класса $SpecialPoints$ затем вызывается функция поиска особых точек $fast()$, в которой ищутся особые точки так, как это описано в теоретической части и выполняется вызов функции $harris\_selection()$, которая в зависимости от результата работы функции $harris\_response(start\_x, start\_y, it, indexes, res)$, которая считает меру Харриса для сдвинутого окна относительно текущей особой точки и возвращает булевское значение, равное $true$, если мера Харриса больше порога, и $false$ в противном случае, удаляет точку из особых либо оставляет ее. Так же в этой функции происходит отбор локальных максимумов, особые точки с малыми значениями меры Харриса в окрестности некоторой другой особой точки (центральной) удаляются.

4) Вычисление двоичных дескрипторов для каждой особой точки изображения

Вычисление двоичных дескрипторов реализовано в классе $Descriptors$ с помощью алгоритма BRIEF, как это указано в теоретической части.

Листинг 4. Класс, реализующий построение дескрипторов

```
1.  class Descriptors
2.  {
3.  public:
4.        Descriptors(std::vector<std::pair<PIXEL, double>> special_points,
    std::vector<std::vector<PIXEL>> pixels);
5.        ~Descriptors();
6.        void brief(); // Вычисление дискрипторов для каждой особой точки
7.        void save_descriptors(std::string file_name, std::string image_name);
    // Сохранение данных изображения с вычисленными дискрипторами для каждой особой
    точки в файле file_name
8.        std::vector<std::pair<std::pair<PIXEL, double>, std::vector<bool>>>
    descriptors; // Вектор дискрипторов для каждой особой точки
9.  private:
10.       const unsigned int height = { 240 };
11.       const unsigned int width = { 320 };
12.       std::vector<std::pair<PIXEL, double>> special_points; // Вектор уже
    вычесленных особых точек
13.       std::vector<std::vector<PIXEL>> pixels; //  Вектор данных для каждого
    пикселя
14.       int size = { 52 }; // Размер патча
15.       const unsigned int nd_dimensional = { 256 }; // Число сравнений в патче
    (размерность двоичных дескрипторов)
16.       const unsigned int radius = { 3 }; // Радиус окружности с центром в
    рассматриваемой точке (пикселе)
17.       std::deque<std::pair<unsigned int, unsigned int>> circle; // очередь -
    окружность, 1-16 пиксели
18.       double get_tetta(const unsigned int & x, const unsigned int & y); //
    Вычисление угла ориентации особой точки с координатами (x,y)
19.       void make_circle(const unsigned int & y, const unsigned int & x); //
    Заполняет очередь координатами пткселей, лежащих на окружности
20.       std::pair<unsigned int, unsigned int>
    get_blurred_pixel_intensity_values(const int & x_1, const int & y_1, const int
    & x_2, const int & y_2); // Получение значений интенсивностей для двух пикселей
    на размытом изображении
21. };
```

Сначала создается объект класса $Descriptors$ затем вызывается функция построения дескрипторов $brief()$, в которой в соответствии с углом ориентации особой точки, который вычисляется в функции $get\_tetta(x, y)$, поворачиваются выбранные по некоторому закону координаты точек из патча

(окрестности особой точки). Значения яркостей в выбранных точках пересчитываются для размытого изображения с помощью функции $get\_blurred\_pixel\_intensity\_values(x\_1, y\_1, x\_2, y\_2)$ и затем сравниваются для составления дескрипторов, как описано в теоретической части.

Далее, если выбран режим построения дескрипторов, то полученные на данном этапе дескрипторы сохраняются в выходном файле с помощью функции $save\_descriptors(file\_name, image\_name)$. Если выбран режим поиска похожих изображений, то выполняются 1 – 4 этапы для изображения, для которого выполняется поиск похожих изображений, а затем выполняется сравнение дескрипторов изображений из директории с дескрипторами данного изображения.

5) Сравнение дескрипторов

Сравнение дескрипторов реализовано в классе $Comparator$ и представляет собой сравнение каждого дескриптора для первого изображения с каждым дескриптором для второго изображения. В данном случае выбрана не самая эффективная реализация сравнения дескрипторов изображений, однако для дескрипторов BRIEF существует не так много методов их сравнения. Одним из возможных улучшений является использование k-мерных деревьев. Кроме того, даже в библиотеке OpenCV сравнение дескрипторов реализовано похожим методом, который называют Brute-Force matcher. Есть также два варианта работы этого метода, первый реализован так, как и описано выше, а во втором дескрипторы считаются соседними, если для некоторого дескриптора D_i из первого изображения найден ближайший дескриптор D_j из второго изображения и наоборот, если для дескриптора D_j из второго изображения найден ближайший дескриптор D_i из первого изображения (две итерации сравнений дескрипторов).

Листинг 5. Класс, реализующий сравнение дескрипторов

```
1. // Данные для каждого изображения
```

```
typedef struct
{
	std::string name;
	std::string descriptors_count;
	std::vector<std::string> descriptors;
	void clear()
	{
		name.clear();
		descriptors_count.clear();
		descriptors.clear();
	}
} IMG;

class Comparator
{
public:
	Comparator(std::vector<std::pair<std::pair<PIXEL, double>, std::vector<bool>>> descriptors);
	~Comparator();
	void parser(std::string file_name); // Парсер для файла, содержащего данные (путь к изображению, количество дескрипторов, дескрипторы) для каждого изображения
	void hamming_comparison(); // Сравнение дескрипторов на основе вычисления расстояния Хэмминга
private:
	std::string path_out; // Путь, куда будет сохранен файл с результатами поиска похожих изображений
	std::vector<IMG> imgs; // Результат парсинга
	std::vector<std::pair<std::pair<PIXEL, double>, std::vector<bool>>> descriptors; // Вектор дискрипторов для изображения, для которого будет производиться поиск
	const unsigned int hamming_threshold = { 75 }; // Порог для расстояния Хэмминга между дескрипторами
	std::vector<unsigned int> hamming_distances; // Расстояния Хэмминга между дескрипторами, которые удовлетворяют условию
};
```

Сначала создается объект класса $Comparator$ затем вызывается функция $parser(file\_name)$, которая извлекает данные (путь к изображению, количество дескрипторов и сами дескрипторы) об обработанных изображениях и сохраняет их в векторе $imgs$, хранящий IMG – структуры. Затем вызывается функция $hamming\_comparison()$, в которой происходит сравнение дескрипторов изображений. Считается количество дескрипторов, которые имеют минимальное расстояние, меньшее порогового. Отношение

количества таких соседних дескрипторов и количества дескрипторов в меньшем из двух множеств дескрипторов изображений дает оценку схожести.

ОПИСАНИЕ ФОРМАТА ВХОДНЫХ И ВЫХОДНЫХ ДАННЫХ

1) Описание каждого изображения в папке с помощью дескрипторов особых точек

Формат входных данных:

Во входном файле задаются две команды (каждая с новой строки) – путь к директории с изображениями и путь к директории, в которой будут сохранены обработанные изображения.

Формат выходных данных:

Программа выводит в выходной файл названия изображений, количество дескрипторов особых точек для каждого изображения и сами дескрипторы. Также во второй указанной во входном файле директории создаютя копии обработанных изображений с нанесенными на них особыми точками, которые выделяются красным цветом.

2) Поиск похожих изображений

Формат входных данных:

Во входном файле задаются три команды, каждая с новой строки. Первая команда содержит путь к изображению, для которого будет производиться поиск похожих изображений. Вторая команда содержит путь к полученному выходному файлу, содержащему дескрипторы. Третья команда содержит путь к директории, в которой будет сохранено обработанное изображение.

Формат выходных данных:

Программа выводит в выходной файл с новой строки пути к найденным изображениям в данной директории.

Пример:

Входные данные в файле **input1**
C:\Users\name\photos C:\User\name\processed_photos
Выходные данные в файле **output1**

{
"ImageName": " C:\Users\name\photos \car.bmp",
"Count": "2",
"Descriptors": [
"110000000…10101101001101100110010 0110",
"110001010…10101001011101100110010 0110",
]
}

Входные данные в файле **input2**
C:\Users\name\photos\car2.bmp C:\output1.out C:\User\name\processed_photos
Выходные данные в файле **output2**
C:\Users\name\photos \car.bmp

ОПИСАНИЕ ТЕСТОВ

Тесты представляют из себя проверку работы алгоритма на различных входах. В данном случае необходимо проверить, что алгоритм верно ищет среди изображений те, которые являются похожими на входное изображение. При этом подразумевается похожесть по форме объектов на изображениях.

Тестирование включает в себя как построение дескрипторов, так и поиск похожего изображения в директории, а также вычисление результатов тестирования в виде ответа «True», если тест прошел успешно, и «False», в противном случае, который записывается в отдельный файл. Таким образом формат входных данных не изменяется, однако в данном случае при запуске программы в качестве аргументов (флагов) задаются пути к обоим входным файлам input1.in и input2.in. Формат выходных данных также не изменяется, добавляется только еще один выходной файл result.r, в котором как раз и выводятся результаты тестирования.

Были проведены следующие тесты:

1) Аддитивный шум

Данные для тестирования (рисунок 5):

Рисунок 5. Тестовые изображения, проверка на аддитивный шум

Исходные данные (рисунок 6):

Рисунок 6. Исходное изображение, для которого проводится тест

При проведении данного теста было обнаружено, что алгоритм не является инвариантным к аддитивному шуму, несмотря на то, что расположение особых точек на обработанных изображениях (первом из тестовых и исходном) сильно не отличалось. В выходном файле получено значение FALSE. При изменении порогового расстояния между дескрипторами в большую сторону наблюдается резкое увеличение ошибок.

2) Изменение освещения

Данные для тестирования (рисунок 7):

Рисунок 7. Тестовые изображения, проверка на изменение освещения

Исходные данные (рисунок 8):

Рисунок 8. Исходное изображение, для которого проводится тест

При проведении данного теста было обнаружено, что алгоритм правильно работает на данном наборе изображений, то есть является инвариантным к смене освещения. В выходном файле получено значение TRUE.

Изображение	Осветленные очки	Здание
Процент похожести с исходным изображением	82.012	11.2603
Значение порогового расстояния	45	

3) Повороты

Данные для тестирования (рисунок 9):

Рисунок 9. Тестовые изображения, проверка инвариантности к поворотам

Исходные данные (рисунок 10):

Рисунок 10. Исходное изображение, для которого проводится тест

При проведении данного теста было обнаружено, что

Изображение	Повернутое здание	Машина
Процент похожести с исходным изображением	62.2333	19.8347
Значение порогового расстояния	85	

4) Простой поиск похожих изображений на определенное изображение

Данные для тестирования (рисунок 11):

Рисунок 10. Тестовые изображения

Исходные данные (рисунок 11):

Рисунок 11. Исходное изображение, для которого проводится тест

Результаты теста следующие (рисунок 12):

Рисунок 12. Результаты теста

АНАЛИЗ ОСНОВНЫХ АЛГОРИТМОВ ПРОГРАММЫ

Код основных алгоритмов представлен в Приложении 1. Так же полный код программы доступен на github по ссылке https://github.com/alexshapran-as/Mi_ORB.

Основными алгоритмами в данной реализации являются: алгоритм поиска особых точек FAST, детектор углов Харриса с поиском локальных максимумов, построение дескрипторов алгоритмом BRIEF и сравнение дескрипторов.

1) Алгоритм FAST

В данном случае был выбран радиус равный 3 пикселям. Размер изображения приводится к 320x240 пикселям, данные размеры заданы, но могут быть изменены, поэтому размеры изображения будем считать входными параметрами MxN. Соответственно надо проверить 16 пикселей на окружности. Однако используется метод предварительной проверки четырех пикселей под номерами: 1, 5, 9, 13, чтобы понять нужна ли проверка всех 16 пикселей и ускорить алгоритм. В случае необходимости проверки 16 пикселей создается очередь, содержащая в необходимом порядке координаты 24-ех пикселей окружности, первые 16 пикселей обходят всю окружность и начиная сначала добавляются еще 8. Таким образом имеем очередь с 24 элементами, проходя по которым сравниваем яркости соответствующих пикселей с яркостью центрального пикселя.

Отрывок кода из программы, реализующий поиск особых точек алгоритмом FAST:

Листинг 6. Код реализации алгоритма FAST

```
1. for (unsigned int y = { radius }; y < height - radius; ++y)
2. {
3.       for (unsigned int x = { radius }; x < width - radius; ++x)
4.       {
5.             // Предварительная проверка четырех пикселей под номерами : 1, 5,
   9, 13
```

```
6.              unsigned int count = { 0 }; // Количество пикселей светлее или
    темнее центрального
7.              int intensity_difference = { 0 }; // Разница яркостей
8.              intensity_difference = int(pixels[y][x].intensity) - int(pixels[y
    + radius][x].intensity);
9.              if (intensity_difference > fast_threshold || intensity_difference
    < -fast_threshold)
10.                     count++;
11.             intensity_difference = int(pixels[y][x].intensity) -
    int(pixels[y][x + radius].intensity);
12.             if (intensity_difference > fast_threshold || intensity_difference
    < -fast_threshold)
13.                     count++;
14.             intensity_difference = int(pixels[y][x].intensity) - int(pixels[y
    - radius][x].intensity);
15.             if (intensity_difference > fast_threshold || intensity_difference
    < -fast_threshold)
16.                     count++;
17.             intensity_difference = int(pixels[y][x].intensity) -
    int(pixels[y][x - radius].intensity);
18.             if (intensity_difference > fast_threshold || intensity_difference
    < -fast_threshold)
19.                     count++;
20.             if (count >= 3) // Проверка всех 16 пикселей
21.             {
22.                     make_circle(y, x, radius, true);
23.                     unsigned int length = { 0 }; // Длина подряд идущих пикселей
    из окружности, удовлетворяющих условию |Ix - Ic| > t
24.                     for (std::deque<std::pair<unsigned int, unsigned
    int>>::iterator it = circle.begin(); it != circle.end(); it++)
25.                     {
26.                             intensity_difference = int(pixels[y][x].intensity) -
    int(pixels[it->first][it->second].intensity);
27.                             if (intensity_difference > fast_threshold ||
    intensity_difference < -fast_threshold)
28.                                     length++;
29.                             else
30.                                     length = { 0 };
31.                             if (length == 9) // Если есть 9 подряд идущих
    пикселей, удовлетворяющих условию |Ix - Ic| > t
32.                             {
33.                                     special_points.push_back(std::pair<PIXEL,
    double>(pixels[y][x], 0.0));
34.                                     pixels[y][x].special = { true };
35.                                     break;
36.                             }
37.                     }
38.             }
39.     }
40. }
```

В лучшем случае для каждого пикселя будет проведено 4 сравнения (для алгоритма это не лучший случай, но для оценки сложности будем считать, что это так). Всего пикселей, для которых проводятся сравнения, будет $(M - 2 * radius) * (N - 2 * radius)$. Поэтому сложность в лучшем случае составит $O\big(4 * (M - 2 * radius) * (N - 2 * radius)\big)$ или $O(M * N)$.

В худшем случае для каждого пикселя будет проведено 24 сравнения. Тогда сложность в худшем случае составит $O\big(24 * (M - 2 * radius) * (N - 2 * radius)\big)$ или $O(M * N)$, с большей константой, чем в лучшем случае.

В алгоритме используется очередь, заполняющаяся координатами пикселей на окружности, и вектор особых точек, таким образом сложность по памяти составит примерно $O(k)$, где k – число особых точек.

2) Детектор углов Харриса с поиском локальных максимумов

Время работы зависит от количества найденных особых точек и максимально возможного количества особых точек. Пусть k_i – число особых точек на изображении, оставшихся после i-ой итерации, причем k_0 – число особых точек, найденных алгоритмом FAST, m – максимальное число особых точек, r_i – число не угловых точек, найденных на i-ой итерации, l_i – число точек с меньшим значением меры Харриса, чем у локальных максимумов, найденных на i-ой итерации.

От каждой особой точки откладываются окрестности в 8 направлениях – окна размером 3x3, для которых вычисляется мера Харриса. Если мера Харриса оказывается меньше пороговой, то точка отбрасывается.

После отработки алгоритма FAST в функции $fast()$ вызывается функция $harris\_selection()$, в которой выполняется отбор угловых особых точек, причем есть еще несколько функций, которые реализуют вычисление меры Харриса для сдвинутого окна

($harris\_response(off\_x, off\_y, it, indexes, res)$), вычисление весовой функции сдвинутого окна ($w(x\_0, y\_0, x, y)$).

Отрывок кода из программы, реализующий детектор углов Харриса:

Листинг 7. Код реализации детектора углов Харриса

```
1.  void SpecialPoints::harris_selection()
2.  {
3.        for (; special_points.size() > max_count; harris_threshold *= 2)
4.        {
5.              std::vector<int> indexes; // Индексы точек, которые удаляются из
    особых
6.              for (std::vector<std::pair<PIXEL, double>>::iterator it =
    special_points.begin(); it != special_points.end(); ++it)
7.              {
8.                    int off_x = { 1 }; // Смещение координаты x - начальной для
    смещенного окна
9.                    int off_y = { 1 }; // Смещение координаты y - начальной для
    смещенного окна
10.                   double res = { 0.0 }; // Суммарное значение мер Харриса для
    окрестностей текущей особой точки
11.                   while (off_x != -1 && off_y != -2)
12.                   {
13.                         if (off_y == -2)
14.                         {
15.                               off_x--;
16.                               off_y = { 1 };
17.                         }
18.                         if (harris_response(off_x, off_y, it, indexes, res)
    == false) // Получение результата вычисления меры Харриса для смещенного окна
19.                         {
20.                               break;
21.                         }
22.                         off_y = off_y - 1;
23.                         if (off_x == 0 && off_y == 0) off_y = { -1 };
24.                   }
25.                   if (res != 0.0)
26.                   {
27.                         it->second = res;
28.                   }
29.             }
30.             int i = { 0 };
31.             for (const auto & elem : indexes) // Удаление не угловых особых
    точек
32.             {
33.                   pixels[special_points[(elem -
    i)].first.y][special_points[(elem - i)].first.x].special = { false };
```

```
34.                     special_points.erase(special_points.begin() + (elem - i));
35.                     ++i;
36.             }
37.             i = { 0 };
38.             indexes.clear();
39.             // Поиск локальных максимумов
40.             for (std::vector<std::pair<PIXEL, double>>::iterator it =
    special_points.begin(); it != special_points.end() - 1; ++it)
41.             {
42.                     for (unsigned int r = { 1 }; r <= radius; ++r) //
    рассматриваем окрестность особой точки, начиная от радиуса равного 1 до 3
43.                     {
44.                             make_circle(it->first.y, it->first.x, r, false);
45.                             for (const auto & coordinate : circle)
46.                             {
47.                                     if (coordinate.first <= 0 || coordinate.second
    <= 0 || coordinate.first >= height || coordinate.second >= width)
48.                                     {
49.                                             continue;
50.                                     }
51.                                     if
    (pixels[coordinate.first][coordinate.second].special == true) // Если на
    окружности оказалась особая точка
52.                                     {
53.                                             std::vector<std::pair<PIXEL,
    double>>::iterator it_neighbor = it; // Начиная от текущей особой точки ищем
    ту, которая оказалась на окружности
54.                                             for (; it_neighbor->first.y !=
    coordinate.first && it_neighbor->first.x != coordinate.second;)
55.                                             {
56.                                                     if (coordinate.first > it-
    >first.y) // Если точка на окружности выше центральной точки, то идем вверх
57.                                                             ++it_neighbor;
58.                                                     else if (coordinate.first < it-
    >first.y) // Иначе идем вниз
59.                                                             --it_neighbor;
60.                                             }
61.                                             if (it->second <= it_neighbor->second)
    // Если мера Харриса для текущей (центральной) особой точки больше, чем у той,
    что на окружности, то удаляем вторую
62.                                             {
63.                                                     if (std::find(indexes.begin(),
    indexes.end(), it - special_points.begin()) == indexes.end())
64.
        indexes.push_back(std::distance(special_points.begin(), it));
65.                                             }
66.                                             else // Иначе удаляем текущую
67.                                             {
68.                                                     if (std::find(indexes.begin(),
    indexes.end(), it_neighbor - special_points.begin()) == indexes.end())
```

```
69.
        indexes.push_back(std::distance(special_points.begin(), it_neighbor));
70.                                     }
71.                                 }
72.                             }
73.                         }
74.             }
75.             // Так как выбор индексов шел не снизу вверх, а в зависимости от
    расположения особой точки на окружности,
76.             // надо предварительно отсортировать вектор индексов удаляемых
    точек
77.             std::sort(indexes.begin(), indexes.end());
78.             for (const auto & elem : indexes)
79.             {
80.                     special_points.erase(special_points.begin() + (elem - i));
81.                     ++i;
82.             }
83.             i = { 0 };
84.             indexes.clear();
85.       }
86.       harris_threshold = { 4096 };
87. }
88.
89. double SpecialPoints::w(const unsigned int & x_0, const unsigned int & y_0,
    const unsigned int & x, const unsigned int & y)
90. {
91.       return (double)exp(-((double)((x - x_0) * (x - x_0) + (y - y_0) * (y -
    y_0))/(2.0 * sigma * sigma))) / (two_pi * sigma * sigma);
92. }
93.
94. bool SpecialPoints::harris_response(const int & start_x, const int & start_y,
    const std::vector<std::pair<PIXEL, double>>::iterator & it, std::vector<int> &
    indexes, double & res)
95. {
96.       double a = { 0.0 };
97.       double b = { 0.0 };
98.       double c = { 0.0 };
99.       if (int(it->first.y) + start_y <= 0 || int(it->first.y) + start_y >=
    special_points.size() || int(it->first.x) + start_x <= 0 || int(it->first.x) +
    start_x >= special_points.size())
100.                  return false;
101.            for (unsigned int y = it->first.y + start_y; y < it->first.y +
    start_y + 3; ++y)
102.            {
103.                    for (unsigned int x = it->first.x + start_x; x < it->first.x
    + start_x + 3; ++x)
104.                    {
105.                            double w_ = w(it->first.x, it->first.y, x, y);
106.                            int I_diff_x = pixels[y][x].intensity - pixels[y][x -
    1].intensity;
```

```
107.                        int I_diff_y = pixels[y][x].intensity - pixels[y -
    1][x].intensity;
108.                        a += w_ * I_diff_x * I_diff_x;
109.                        b += w_ * I_diff_y * I_diff_y;
110.                        c += w_ * I_diff_x * I_diff_y;
111.                  }
112.            }
113.            double R = (((a * b) - (c * c)) - harris_k * (a + b) * (a + b));
114.            if (R < harris_threshold)
115.            {
116.                  res = { 0.0 };
117.                  indexes.push_back(std::distance(special_points.begin(),
    it));
118.                  return false;
119.            }
120.            res += R;
121.            return true;
122.      }
```

Таким образом, если было найдено k_0 особых точек, то в лучшем случае, когда число оставшихся после удаления не угловых особых точек меньше максимального m (предполагается, что на изображении имеются не угловые точки), сложность алгоритма составит примерно $O(k_0 * r_0 * (k_0 - r_0) * l_0^2 \log_2 l_0)$. Оценка примерная, так как сложно оценить сколько потребуется операций в некоторых частях кода, как например поиск особых точек в окрестности других особых точек для вычисления локальных максимумов.

В худшем случае потребуется несколько раз выполнять внешний цикл до тех пор, пока число особых точек не станет меньше максимального m. Пусть q – это число итераций внешнего цикла, тогда сложность составит примерно $O(\sum_{i=0}^{q-1}(k_i * r_i * (k_i - r_i) * l_q^2 \log_2 l_q))$, причем $k_{q-1} < m$ и другие $k_i > m$ $(i = 0..q-2)$.

Сложность по памяти составит в лучшем случае $O(r_0 + l_0)$ и в худшем случае $O(\sum_{i=0}^{q-1}(r_i + l_i))$.

3) BRIEF

Здесь проводится 256 сравнений в окрестности каждой особой точки. Окрестностью является окно SxS, а точки из этой окрестности выбираются случайным образом из интервала $(0, \frac{1}{25}S^2)$, затем они поворачиваются на угол ориентации особой точки.

Отрывок кода из программы, реализующий построение дескрипторов BRIEF:

Листинг 8. Код реализации построения дескрипторов BRIEF

```
1.  void Descriptors::brief()
2.  {
3.        std::vector<bool> zero_vec;
4.        zero_vec.assign(nd_dimensional, 0);
5.        for (std::vector<std::pair<PIXEL, double>>::iterator it =
    special_points.begin(); it != special_points.end(); ++it)
6.        {
7.              double tetta = get_tetta(it->first.x, it->first.y);
8.              std::vector<bool> t; // Тест на патче размера (size x size)
9.              for (unsigned int i = { 0 }; i < nd_dimensional; ++i) //
    Проведение 256 сравнений и получение дескриптора для патча
10.             {
11.                   size = { 10 };
12.                   while (it->first.x + (size * size / 25) >= width || it-
    >first.y + (size * size / 25) >= height || it->first.x - (size * size / 25) < 0
    || it->first.y - (size * size / 25) < 0) // Если патч большой
13.                   {
14.                         size /= 2;
15.                   }
16.                   int x_1, y_1, x_2, y_2; // Координаты двух пикселей, которые
    будут участвовать в сравнении
17.                   int x_1_rotated, y_1_rotated, x_2_rotated, y_2_rotated; //
    Повернутые координаты
18.                   do
19.                   {
20.                         //
    https://www.researchgate.net/publication/51798992_BRIEF_Computing_a_local_binar
    y_descriptor_very_fast
21.                         x_1 = (0 + rand() % (size * size / 25));
22.                         y_1 = (0 + rand() % (size * size / 25));
23.                         x_2 = (0 + rand() % (size * size / 25));
24.                         y_2 = (0 + rand() % (size * size / 25));
25.                         x_1_rotated = it->first.x + (int)round((x_1 *
    cos(tetta)) - (y_1 * sin(tetta)));
26.                         y_1_rotated = it->first.y + (int)round((x_1 *
    sin(tetta)) + (y_1 * cos(tetta)));
```

```
27.                         x_2_rotated = it->first.x + (int)round((x_2 *
    cos(tetta)) - (y_2 * sin(tetta)));
28.                         y_2_rotated = it->first.y + (int)round((x_2 *
    sin(tetta)) + (y_2 * cos(tetta)));
29.                     } while (x_1_rotated < 0 || x_2_rotated < 0 || y_1_rotated <
    0 || y_2_rotated < 0 ||
30.                                     x_1_rotated >= width || x_2_rotated >= width
    || y_1_rotated >= height || y_2_rotated >= height);
31.                     std::pair<unsigned int, unsigned int>
    blurred_pixel_intensity_values =
    get_blurred_pixel_intensity_values(x_1_rotated, y_1_rotated, x_2_rotated,
    y_2_rotated);
32.                     if (blurred_pixel_intensity_values.first <
    blurred_pixel_intensity_values.second)
33.                         t.push_back(1);
34.                     else
35.                         t.push_back(0);
36.             }
37.             if (t != zero_vec)
38.             {
39.                     descriptors.push_back(std::pair<std::pair<PIXEL, double>,
    std::vector<bool>>(*it, t));
40.             }
41.       }
42. }
43.
44. double Descriptors::get_tetta(const unsigned int & x, const unsigned int & y)
45. {
46.       int m_0_1 = { 0 };
47.       int m_1_0 = { 0 };
48.       make_circle(y, x);
49.       for (const auto & coordinate : circle)
50.       {
51.             m_0_1 += coordinate.first *
    pixels[coordinate.first][coordinate.second].intensity; // sum(y*I(x,y))
52.             m_1_0 += coordinate.second *
    pixels[coordinate.first][coordinate.second].intensity; // sum(x*I(x,y))
53.       }
54.       return atan((double)m_0_1 / m_1_0);
55. }
56.
57. void Descriptors::make_circle(const unsigned int & y, const unsigned int & x)
58. {
59.       circle.clear();
60.       circle.push_back(std::pair<unsigned int, unsigned int>(y + radius, x));
61.       for (unsigned int i = { 0 }; i < 2 * radius + 1; ++i)
62.       {
63.             unsigned int j = { 0 };
64.             if (i == 1 || i == 5)
65.                   j = { 1 };
```

```
66.             else if (i == 2 || i == 3 || i == 4)
67.                   j = { 2 };
68.             circle.push_front(std::pair<unsigned int, unsigned int>(y - radius
    + i, x + 1 + j));
69.             circle.push_back(std::pair<unsigned int, unsigned int>(y - radius
    + i, x - 1 - j));
70.       }
71.       circle.push_front(std::pair<unsigned int, unsigned int>(y + radius, x));
72. }
73.
74. std::pair<unsigned int, unsigned int>
    Descriptors::get_blurred_pixel_intensity_values(const int & x_1, const int &
    y_1, const int & x_2, const int & y_2)
75. {
76.       unsigned int new_intensity_1 = { 0 };
77.       unsigned int new_intensity_2 = { 0 };
78.       unsigned int count = { 0 };
79.       for (int i = { -2 }; i != 3; ++i)
80.       {
81.             for (int j = { -2 }; j != 3; ++j)
82.             {
83.                   if (x_1 + i >= 0 && x_2 + i >= 0 && x_1 + i < width && x_2 +
    i < width && y_1 + j >= 0 && y_2 + j >= 0 && y_1 + j < height && y_2 + j <
    height)
84.                   {
85.                         new_intensity_1 += pixels[y_1 + j][x_1 +
    i].intensity;
86.                         new_intensity_2 += pixels[y_2 + j][x_2 +
    i].intensity;
87.                         count++;
88.                   }
89.             }
90.       }
91.       new_intensity_1 = new_intensity_1 / count;
92.       new_intensity_2 = new_intensity_2 / count;
93.       if (new_intensity_1 == 0)
94.             new_intensity_1 = pixels[y_1][x_1].intensity;
95.       if (new_intensity_2 == 0)
96.             new_intensity_2 = pixels[y_2][x_2].intensity;
97.       return std::pair<unsigned int, unsigned int>(new_intensity_1,
    new_intensity_2);
98. }
```

Пусть на данном этапе есть w особых точек, отобранных алгоритмом FAST, детектором углов Харриса и определением локальных максимумов. Таким образом, алгоритм работает за $O(w)$ и имеет достаточно большую

константу, которая зависит от длины дескрипторов, радиуса окружности для определения угла ориентации каждой особой точки и числа пикселей, используемых для получения значения яркости некоторого пикселя после размытия.

Сложность по памяти также $O(w)$.

4) Сравнение дескрипторов

Сравнение дескрипторов реализовано первым вариантом метода Brute-Force matcher. Каждый дескриптор из множества дескрипторов для первого изображения сравнивается с каждым дескриптором из множества дескрипторов для второго изображения. По отношению количества найденных соседних дескрипторов к мощностям обоих множеств дескрипторов делается вывод о похожести изображений. В общем случае задано, что при $\geq 20\%$ схожести по отношению количества найденных соседних дескрипторов к мощностям обоих множеств дескрипторов и при $\geq 50\%$ схожести по сумме этих отношений изображения являются похожими, но этот параметр изменялся для различных тестов, как и пороговое расстояние между дескрипторами.

Отрывок кода из программы, реализующий сравнение дескрипторов:

Листинг 9. Код реализации сравнения дескрипторов

```
1. void Comparator::hamming_comparison()
2. {
3.       for (const auto & img : imgs) // Перебор данных по каждому изображению из пропарсенного файла (по каждому изображению из папки)
4.       {
5.             for (const auto & desc1 : descriptors) // Перебор дескрипторов для входного изображения, для которого ищутся похожие изображения
6.             {
7.                   unsigned int hamming_distance = { 0 }; // Промежуточное расстояние Хэмминга между двумя дескрипторами
8.                   unsigned int min_hamming_distance = { 256 }; // Минимальное расстояние Хэмминга среди промежуточных
9.                   for (const auto & desc2 : img.descriptors) // Перебор дескрипторов для очередного изображения из пропарсенного файла
```

```
10.                     {
11.                             for (unsigned int i = { 0 }; i < 256; ++i)
12.                             {
13.                                     if (hamming_distance > hamming_threshold) //
    Исключение из рассмотрения расстояния Хэмминга, которое уже больше чем порог
14.                                             break;
15.                                     hamming_distance = hamming_distance +
    ((desc2[i] - '0') ^ (int)desc1.second[i]);
16.                             }
17.                             if (hamming_distance < min_hamming_distance)
18.                                     min_hamming_distance = hamming_distance;
19.                             hamming_distance = { 0 };
20.                     }
21.                     if (min_hamming_distance <= hamming_threshold)
22.                     {
23.                             hamming_distances.push_back(min_hamming_distance);
24.                     }
25.             }
26.             if (((float)hamming_distances.size() / descriptors.size()) >= 0.2 &&
    ((float)hamming_distances.size() / std::stoi(img.descriptors_count)) >= 0.2 &&
    ((float)hamming_distances.size() / descriptors.size()) +
    ((float)hamming_distances.size() / std::stoi(img.descriptors_count)) >= 0.5)
27.             {
28.                     std::ofstream fout(path_out + "output2.out",
    std::ios_base::app);
29.                     if (!fout.is_open())
30.                     {
31.                             return;
32.                     }
33.                     fout << img.name << std::endl;
34.                     fout.close();
35.             }
36.             hamming_distances.clear();
37.     }
38. }
```

Пусть число изображений, среди которых происходит поиск похожих равно k, число дескрипторов для i-ого изображения, среди которых происходит поиск похожих, равно A_i, а число дескрипторов для исходного изображения равно B. Тогда сложность данного алгоритма составит $O(\sum_{i=1}^{k} A_i * B)$.

Сложность по памяти $O(\sum_{i=1}^{k} p_i)$, где p_i – число соседних дескрипторов для i-ой итерации сравнения двух изображений.

ВЫБОР СТРУКТУР ДАННЫХ

В программе использовались такие стандартные структуры данных, как $std::dequeuer <>$ и $std::vector <>$. Их выбор обосновывается простотой заполнения, быстрым доступом к элементам.

ПРИЛОЖЕНИЕ 1

Листинг 10. Алгоритм FAST, поиск меры Харриса с определением локальных максимумов

```
1. void SpecialPoints::make_circle(const unsigned int & y, const unsigned int & x,
   const unsigned int & radius, bool fast_alg)
2. {
3.       circle.clear();
4.       circle.push_back(std::pair<unsigned int, unsigned int>(y + radius, x));
5.       for (unsigned int i = { 0 }; i < 2 * radius + 1; ++i)
6.       {
7.             unsigned int j = { 0 };
8.             if (i == 1 || i == 5)
9.                   j = { 1 };
10.            else if (i == 2 || i == 3 || i == 4)
11.                  j = { 2 };
12.            circle.push_front(std::pair<unsigned int, unsigned int>(y - radius
   + i, x + 1 + j));
13.            circle.push_back(std::pair<unsigned int, unsigned int>(y - radius
   + i, x - 1 - j));
14.      }
15.      circle.push_front(std::pair<unsigned int, unsigned int>(y + radius, x));
16.      if (fast_alg)
17.      {
18.            for (unsigned int i = { 0 }; i < 2 * (radius + 1); ++i) /*8*/
19.            {
20.                  circle.push_back(circle[i]);
21.            }
22.      }
23. }
24.
25. void SpecialPoints::harris_selection()
26. {
27.      for (; special_points.size() > max_count; harris_threshold *= 2)
28.      {
29.            std::vector<int> indexes; // Индексы точек, которые удаляются из
   особых
30.            for (std::vector<std::pair<PIXEL, double>>::iterator it =
   special_points.begin(); it != special_points.end(); ++it)
31.            {
```

```
32.                     int off_x = { 1 }; // Смещение координаты x - начальной для
    смещенного окна
33.                     int off_y = { 1 }; // Смещение координаты y - начальной для
    смещенного окна
34.                     double res = { 0.0 }; // Суммарное значение мер Харриса для
    окрестностей текущей особой точки
35.                     while (off_x != -1 && off_y != -2)
36.                     {
37.                             if (off_y == -2)
38.                             {
39.                                     off_x--;
40.                                     off_y = { 1 };
41.                             }
42.                             if (harris_response(off_x, off_y, it, indexes, res)
    == false) // Получение результата вычисления меры Харриса для смещенного окна
43.                             {
44.                                     break;
45.                             }
46.                             off_y = off_y - 1;
47.                             if (off_x == 0 && off_y == 0) off_y = { -1 };
48.                     }
49.                     if (res != 0.0)
50.                     {
51.                             it->second = res;
52.                     }
53.             }
54.             int i = { 0 };
55.             for (const auto & elem : indexes) // Удаление не угловых особых
    точек
56.             {
57.                     pixels[special_points[(elem -
    i)].first.y][special_points[(elem - i)].first.x].special = { false };
58.                     special_points.erase(special_points.begin() + (elem - i));
59.                     ++i;
60.             }
61.             i = { 0 };
62.             indexes.clear();
63.             // Поиск локальных максимумов
64.             for (std::vector<std::pair<PIXEL, double>>::iterator it =
    special_points.begin(); it != special_points.end() - 1; ++it)
65.             {
66.                     for (unsigned int r = { 1 }; r <= radius; ++r) //
    рассматриваем окрестность особой точки, начиная от радиуса равного 1 до 3
67.                     {
68.                             make_circle(it->first.y, it->first.x, r, false);
69.                             for (const auto & coordinate : circle)
70.                             {
71.                                     if (coordinate.first <= 0 || coordinate.second
    <= 0 || coordinate.first >= height || coordinate.second >= width)
72.                                     {
```

```
73.                                     continue;
74.                                 }
75.                                 if
    (pixels[coordinate.first][coordinate.second].special == true) // Если на
    окружности оказалась особая точка
76.                                 {
77.                                     std::vector<std::pair<PIXEL,
    double>>::iterator it_neighbor = it; // Начиная от текущей особой точки ищем
    ту, которая оказалась на окружности
78.                                     for (; it_neighbor->first.y !=
    coordinate.first && it_neighbor->first.x != coordinate.second;)
79.                                     {
80.                                         if (coordinate.first > it-
    >first.y) // Если точка на окружности выше центральной точки, то идем вверх
81.                                             ++it_neighbor;
82.                                         else if (coordinate.first < it-
    >first.y) // Иначе идем вниз
83.                                             --it_neighbor;
84.                                     }
85.                                     if (it->second <= it_neighbor->second)
    // Если мера Харриса для текущей (центральной) особой точки больше, чем у той,
    что на окружности, то удаляем вторую
86.                                     {
87.                                         if (std::find(indexes.begin(),
    indexes.end(), it - special_points.begin()) == indexes.end())
88.
            indexes.push_back(std::distance(special_points.begin(), it));
89.                                     }
90.                                     else // Иначе удаляем текущую
91.                                     {
92.                                         if (std::find(indexes.begin(),
    indexes.end(), it_neighbor - special_points.begin()) == indexes.end())
93.
            indexes.push_back(std::distance(special_points.begin(), it_neighbor));
94.                                     }
95.                                 }
96.                             }
97.                         }
98.                 }
99.                 // Так как выбор индексов шел не снизу вверх, а в завсисимости от
    расположения особой точки на окружности,
100.                // надо предварительно отсортировать вектор индексов
    удаляемых точек
101.                std::sort(indexes.begin(), indexes.end());
102.                for (const auto & elem : indexes)
103.                {
104.                    special_points.erase(special_points.begin() + (elem -
    i));
105.                    ++i;
106.                }
```

```
107.                    i = { 0 };
108.                    indexes.clear();
109.            }
110.            harris_threshold = { 4096 };
111.    }
112.
113.    double SpecialPoints::w(const unsigned int & x_0, const unsigned int &
    y_0, const unsigned int & x, const unsigned int & y)
114.    {
115.            return (double)exp(-((double)((x - x_0) * (x - x_0) + (y - y_0) *
    (y - y_0))/(2.0 * sigma * sigma))) / (two_pi * sigma * sigma);
116.    }
117.
118.    bool SpecialPoints::harris_response(const int & start_x, const int &
    start_y, const std::vector<std::pair<PIXEL, double>>::iterator & it,
    std::vector<int> & indexes, double & res)
119.    {
120.            double a = { 0.0 };
121.            double b = { 0.0 };
122.            double c = { 0.0 };
123.            if (int(it->first.y) + start_y <= 0 || int(it->first.y) + start_y
    >= special_points.size() || int(it->first.x) + start_x <= 0 || int(it->first.x)
    + start_x >= special_points.size())
124.                    return false;
125.            for (unsigned int y = it->first.y + start_y; y < it->first.y +
    start_y + 3; ++y)
126.            {
127.                    for (unsigned int x = it->first.x + start_x; x < it->first.x
    + start_x + 3; ++x)
128.                    {
129.                            double w_ = w(it->first.x, it->first.y, x, y);
130.                            int I_diff_x = pixels[y][x].intensity - pixels[y][x -
    1].intensity;
131.                            int I_diff_y = pixels[y][x].intensity - pixels[y -
    1][x].intensity;
132.                            a += w_ * I_diff_x * I_diff_x;
133.                            b += w_ * I_diff_y * I_diff_y;
134.                            c += w_ * I_diff_x * I_diff_y;
135.                    }
136.            }
137.            double R = (((a * b) - (c * c)) - harris_k * (a + b) * (a + b));
138.            if (R < harris_threshold)
139.            {
140.                    res = { 0.0 };
141.                    indexes.push_back(std::distance(special_points.begin(),
    it));
142.                    return false;
143.            }
144.            res += R;
145.            return true;
```

```
146.      }
147.
148.      void SpecialPoints::fast()
149.      {
150.            for (unsigned int y = { radius }; y < height - radius; ++y)
151.            {
152.                  for (unsigned int x = { radius }; x < width - radius; ++x)
153.                  {
154.                        // Предварительная проверка четырех пикселей под
    номерами : 1, 5, 9, 13
155.                        unsigned int count = { 0 }; // Количество пикселей
    светлее или темнее центрального
156.                        int intensity_difference = { 0 }; // Разница яркостей
157.                        intensity_difference = int(pixels[y][x].intensity) -
    int(pixels[y + radius][x].intensity);
158.                        if (intensity_difference > fast_threshold ||
    intensity_difference < -fast_threshold)
159.                              count++;
160.                        intensity_difference = int(pixels[y][x].intensity) -
    int(pixels[y][x + radius].intensity);
161.                        if (intensity_difference > fast_threshold ||
    intensity_difference < -fast_threshold)
162.                              count++;
163.                        intensity_difference = int(pixels[y][x].intensity) -
    int(pixels[y - radius][x].intensity);
164.                        if (intensity_difference > fast_threshold ||
    intensity_difference < -fast_threshold)
165.                              count++;
166.                        intensity_difference = int(pixels[y][x].intensity) -
    int(pixels[y][x - radius].intensity);
167.                        if (intensity_difference > fast_threshold ||
    intensity_difference < -fast_threshold)
168.                              count++;
169.                        if (count >= 3) // Проверка всех 16 пикселей
170.                        {
171.                              make_circle(y, x, radius, true);
172.                              unsigned int length = { 0 }; // Длина подряд
    идущих пикселей из окружности, удовлетворяющих условию |Ix - Ic| > t
173.                              for (std::deque<std::pair<unsigned int,
    unsigned int>>::iterator it = circle.begin(); it != circle.end(); it++)
174.                              {
175.                                    intensity_difference =
    int(pixels[y][x].intensity) - int(pixels[it->first][it->second].intensity);
176.                                    if (intensity_difference >
    fast_threshold || intensity_difference < -fast_threshold)
177.                                          length++;
178.                                    else
179.                                          length = { 0 };
180.                                    if (length == 9) // Если есть 9 подряд
    идущих пикселей, удовлетворяющих условию |Ix - Ic| > t
```

```
                                        {
special_points.push_back(std::pair<PIXEL, double>(pixels[y][x], 0.0));
                                                pixels[y][x].special = { true };
                                                break;
                                        }
                                }
                        }
                }
                // Отбор N особых точек с помощью вычисления меры Харриса
                harris_selection();
        }
```

Листинг 11. Алгоритм BRIEF

```
void Descriptors::brief()
{
        std::vector<bool> zero_vec;
        zero_vec.assign(nd_dimensional, 0);
        for (std::vector<std::pair<PIXEL, double>>::iterator it =
special_points.begin(); it != special_points.end(); ++it)
        {
                double tetta = get_tetta(it->first.x, it->first.y);
                std::vector<bool> t; // Тест на патче размера (size x size)
                for (unsigned int i = { 0 }; i < nd_dimensional; ++i) //
Проведение 256 сравнений и получение дескриптора для патча
                {
                        size = { 10 };
                        while (it->first.x + (size * size / 25) >= width || it-
>first.y + (size * size / 25) >= height || it->first.x - (size * size / 25) < 0
|| it->first.y - (size * size / 25) < 0) // Если патч большой
                        {
                                size /= 2;
                        }
                        int x_1, y_1, x_2, y_2; // Координаты двух пикселей, которые
будут участвовать в сравнении
                        int x_1_rotated, y_1_rotated, x_2_rotated, y_2_rotated; //
Повернутые координаты
                        do
                        {
                                //
https://www.researchgate.net/publication/51798992_BRIEF_Computing_a_local_binar
y_descriptor_very_fast
                                x_1 = (0 + rand() % (size * size / 25));
                                y_1 = (0 + rand() % (size * size / 25));
                                x_2 = (0 + rand() % (size * size / 25));
                                y_2 = (0 + rand() % (size * size / 25));
```

```
25.                         x_1_rotated = it->first.x + (int)round((x_1 *
   cos(tetta)) - (y_1 * sin(tetta)));
26.                         y_1_rotated = it->first.y + (int)round((x_1 *
   sin(tetta)) + (y_1 * cos(tetta)));
27.                         x_2_rotated = it->first.x + (int)round((x_2 *
   cos(tetta)) - (y_2 * sin(tetta)));
28.                         y_2_rotated = it->first.y + (int)round((x_2 *
   sin(tetta)) + (y_2 * cos(tetta)));
29.                   } while (x_1_rotated < 0 || x_2_rotated < 0 || y_1_rotated <
   0 || y_2_rotated < 0 ||
30.                                   x_1_rotated >= width || x_2_rotated >= width
   || y_1_rotated >= height || y_2_rotated >= height);
31.                   std::pair<unsigned int, unsigned int>
   blurred_pixel_intensity_values =
   get_blurred_pixel_intensity_values(x_1_rotated, y_1_rotated, x_2_rotated,
   y_2_rotated);
32.                   if (blurred_pixel_intensity_values.first <
   blurred_pixel_intensity_values.second)
33.                         t.push_back(1);
34.                   else
35.                         t.push_back(0);
36.             }
37.             if (t != zero_vec)
38.             {
39.                   descriptors.push_back(std::pair<std::pair<PIXEL, double>,
   std::vector<bool>>(*it, t));
40.             }
41.       }
42. }
43.
44. double Descriptors::get_tetta(const unsigned int & x, const unsigned int & y)
45. {
46.       int m_0_1 = { 0 };
47.       int m_1_0 = { 0 };
48.       make_circle(y, x);
49.       for (const auto & coordinate : circle)
50.       {
51.             m_0_1 += coordinate.first *
   pixels[coordinate.first][coordinate.second].intensity; // sum(y*I(x,y))
52.             m_1_0 += coordinate.second *
   pixels[coordinate.first][coordinate.second].intensity; // sum(x*I(x,y))
53.       }
54.       return atan((double)m_0_1 / m_1_0);
55. }
56.
57. void Descriptors::make_circle(const unsigned int & y, const unsigned int & x)
58. {
59.       circle.clear();
60.       circle.push_back(std::pair<unsigned int, unsigned int>(y + radius, x));
61.       for (unsigned int i = { 0 }; i < 2 * radius + 1; ++i)
```

```
62.       {
63.             unsigned int j = { 0 };
64.             if (i == 1 || i == 5)
65.                   j = { 1 };
66.             else if (i == 2 || i == 3 || i == 4)
67.                   j = { 2 };
68.             circle.push_front(std::pair<unsigned int, unsigned int>(y - radius
    + i, x + 1 + j));
69.             circle.push_back(std::pair<unsigned int, unsigned int>(y - radius
    + i, x - 1 - j));
70.       }
71.       circle.push_front(std::pair<unsigned int, unsigned int>(y + radius, x));
72. }
73.
74. std::pair<unsigned int, unsigned int>
    Descriptors::get_blurred_pixel_intensity_values(const int & x_1, const int &
    y_1, const int & x_2, const int & y_2)
75. {
76.       unsigned int new_intensity_1 = { 0 };
77.       unsigned int new_intensity_2 = { 0 };
78.       unsigned int count = { 0 };
79.       for (int i = { -2 }; i != 3; ++i)
80.       {
81.             for (int j = { -2 }; j != 3; ++j)
82.             {
83.                   if (x_1 + i >= 0 && x_2 + i >= 0 && x_1 + i < width && x_2 +
    i < width && y_1 + j >= 0 && y_2 + j >= 0 && y_1 + j < height && y_2 + j <
    height)
84.                   {
85.                         new_intensity_1 += (unsigned int)(0.299*(unsigned
    int)pixels[y_1 + j][x_1 + i].rgbRed + 0.587*(unsigned int)pixels[y_1 + j][x_1 +
    i].rgbGreen + 0.114*(unsigned int)pixels[y_1 + j][x_1 + i].rgbBlue);
86.                         new_intensity_2 += (unsigned int)(0.299*(unsigned
    int)pixels[y_2 + j][x_2 + i].rgbRed + 0.587*(unsigned int)pixels[y_2 + j][x_2 +
    i].rgbGreen + 0.114*(unsigned int)pixels[y_2 + j][x_2 + i].rgbBlue);
87.                         count++;
88.                   }
89.             }
90.       }
91.       new_intensity_1 = new_intensity_1 / count;
92.       new_intensity_2 = new_intensity_2 / count;
93.       if (new_intensity_1 == 0)
94.             new_intensity_1 = pixels[y_1][x_1].intensity;
95.       if (new_intensity_2 == 0)
96.             new_intensity_2 = pixels[y_2][x_2].intensity;
97.       return std::pair<unsigned int, unsigned int>(new_intensity_1,
    new_intensity_2);
98. }
```

Листинг 12. Сравнение дескрипторов

```
1. void Comparator::hamming_comparison()
2. {
3.       for (const auto & img : imgs) // Перебор данных по каждому изображению из пропарсенного файла (по каждому изображению из папки)
4.       {
5.             for (const auto & desc1 : descriptors) // Перебор дескрипторов для входного изображения, для которого ищутся похожие изображения
6.             {
7.                   unsigned int hamming_distance = { 0 }; // Промежуточное расстояние Хэмминга между двумя дескрипторами
8.                   unsigned int min_hamming_distance = { 256 }; // Минимальное расстояние Хэмминга среди промежуточных
9.                   for (const auto & desc2 : img.descriptors) // Перебор дескрипторов для очередного изображения из пропарсенного файла
10.                  {
11.                        for (unsigned int i = { 0 }; i < 256; ++i)
12.                        {
13.                              if (hamming_distance > hamming_threshold) // Исключение из рассмотрения расстояния Хэмминга, которое уже больше чем порог
14.                                    break;
15.                              hamming_distance = hamming_distance + ((desc2[i] - '0') ^ (int)desc1.second[i]);
16.                        }
17.                        if (hamming_distance < min_hamming_distance)
18.                              min_hamming_distance = hamming_distance;
19.                        hamming_distance = { 0 };
20.                  }
21.                  if (min_hamming_distance <= hamming_threshold)
22.                  {
23.                        hamming_distances.push_back(min_hamming_distance);
24.                  }
25.            }
26.            if (((float)hamming_distances.size() / descriptors.size()) >= 0.3 || ((float)hamming_distances.size() / std::stoi(img.descriptors_count)) >= 0.3)
27.            {
28.                  std::ofstream fout(path_out + "output2.out", std::ios_base::app);
29.                  if (!fout.is_open())
30.                  {
31.                        return;
32.                  }
33.                  fout << img.name << std::endl;
34.                  fout.close();
35.            }
36.            hamming_distances.clear();
37.      }
38. }
```