**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра ТВ**

**отчЁт**

**по лабораторной работе № 4**

**по дисциплине «Интеллектуальный анализ данных»**

**Тема: Сегментация движущихся объектов на основе оценки поля векторов движения**

|  |  |  |
| --- | --- | --- |
| Студенты гр. 9105 |  | Шаривзянов Д. Р. |
|  |  | Чугунов Р. |
| Преподаватель |  | Поздеев А. А. |

Санкт-Петербург

2024

СЕГМЕНТАЦИЯ ДВИЖУЩИХСЯ ОБЪЕКТОВ НА ОСНОВЕ ОЦЕНКИ ПОЛЯ ВЕКТОРОВ ДВИЖЕНИЯ

**Цель работы:** моделирование алгоритма сегментации объектов на основе векторов движения.

Код программы:

#ifndef MOTIONVECTOR\_H

#define MOTIONVECTOR\_H

#include <vector>

#include <opencv2/opencv.hpp>

#include "logger.hpp"

using namespace cv;

using namespace std;

// Structure to store motion vectors

struct MotionVector {

    Point2f start;

    Point2f end;

    float magnitude;

};

class Image {

public:

    string output\_path = "output/Lab 3 Movement Vector/";

    Mat bgr1;

    Mat bgr2;

    Mat gray1;

    Mat gray2;

    int blockSize;

    Mat motionVectorImageOrig;

    Mat motionVectorImageFiltered;

    vector<vector<MotionVector>> motionVectorsOrig;

    vector<vector<MotionVector>> motionVectorsFiltered;

    vector<vector<int>> labels;

    Mat clusterVisualization;

};

/\*\*

 \* @brief Вычисление векторов движения между двумя кадрами с помощью MAD

 \*

 \* @param[in] prevFrame предыдущий кадр

 \* @param[in] currFrame текущий кадр

 \* @param[out] motionVectorsGrid двумерный вектор, содержащий векторы

 \*            движения для каждого блока

 \* @param[in] blockSize размер блока

 \*/

void computeMotionVectors(const Mat& prevFrame, const Mat& currFrame, vector<vector<MotionVector>>& motionVectorsGrid, int blockSize) {

    logger.info("  motion vectors using MAD as the cost function");

    // Проверяем размеры входных кадров

    if (prevFrame.size() != currFrame.size()) {

        throw invalid\_argument("Frames must have the same size.");

    }

    // Размеры сетки блоков

    int rows = prevFrame.rows / blockSize;

    int cols = prevFrame.cols / blockSize;

    // Инициализируем двумерный вектор сеткой блоков

    motionVectorsGrid.resize(rows, vector<MotionVector>(cols));

    // Многопоточная обработка

    cv::parallel\_for\_(cv::Range(0, rows), [&](const cv::Range& range) {

        for (int i = range.start; i < range.end; i++) {

            for (int j = 0; j < cols; j++) {

                int x = j \* blockSize;

                int y = i \* blockSize;

                // Координаты блока

                Rect blockROI(x, y, blockSize, blockSize);

                // Текущий блок из текущего кадра

                Mat currBlock = currFrame(blockROI);

                // Радиус поиска

                int searchRadius = blockSize;

                // Координаты области поиска

                int searchX = max(0, x - searchRadius);

                int searchY = max(0, y - searchRadius);

                int searchWidth = min(blockSize + 2 \* searchRadius, currFrame.cols - searchX);

                int searchHeight = min(blockSize + 2 \* searchRadius, currFrame.rows - searchY);

                // Область поиска

                Rect searchROI(searchX, searchY, searchWidth, searchHeight);

                Mat searchArea = prevFrame(searchROI);

                // Инициализируем лучшее смещение

                double minMAD = DBL\_MAX;

                Point2f bestOffset(0, 0);

                // Проходим по всем возможным блоках в области поиска

                for (int dy = 0; dy <= searchArea.rows - blockSize; dy++) {

                    for (int dx = 0; dx <= searchArea.cols - blockSize; dx++) {

                        // Координаты кандидата

                        Rect candidateROI(dx, dy, blockSize, blockSize);

                        // Блок-кандидат

                        Mat candidateBlock = searchArea(candidateROI);

                        // Вычисляем разницу между блоками

                        Mat diff;

                        absdiff(currBlock, candidateBlock, diff);

                        // Вычисляем MAD

                        double mad = sum(diff)[0] / (blockSize \* blockSize);

                        // Если MAD меньше, чем текущий минимум, то

                        // обновляем минимум и лучшее смещение

                        if (mad < minMAD) {

                            minMAD = mad;

                            bestOffset = Point2f(dx + searchX - x, dy + searchY - y);

                        }

                    }

                }

                // Вычисляем вектор движения

                Point2f startPoint(x + blockSize / 2.0, y + blockSize / 2.0);

                Point2f endPoint = startPoint + bestOffset;

                float magnitude = sqrt(bestOffset.x \* bestOffset.x + bestOffset.y \* bestOffset.y);

                // Сохраняем вектор движения в сетку

                motionVectorsGrid[i][j] = {startPoint, endPoint, magnitude};

            }

        }

    });

}

/\*\*

 \* @brief Отрисовка векторов движения поверх изображения

 \*

 \* Функция drawMotionVectors() отображает векторы движения на изображении.

 \* Она копирует исходное изображение в выходное и рисует векторы движения

 \* в виде стрелок, используя масштабирование для более компактного отображения.

 \*

 \* @param[in] src Исходное изображение

 \* @param[out] dst Изображение с отрисованными векторами движения

 \* @param[in] motionVectorsGrid Сетка векторов движения

 \*/

void drawMotionVectors(const Mat& src, Mat& dst, const vector<vector<MotionVector>>& motionVectorsGrid) {

    // Копируем исходное изображение в выходное

    src.copyTo(dst);

    // Коэффициент масштаба для сокращения длины векторов

    float scale = 3.0f;

    // Толщина стрелок

    int thickness = 1;

    // Цвет стрелок

    Scalar color(0, 255, 0); // Зелёный цвет

    // Проходим по всей сетке векторов движения

    for (const auto& row : motionVectorsGrid) {

        for (const auto& mv : row) {

            // Вычисляем масштабированный конец вектора

            Point2f scaledEnd = mv.start + scale \* (mv.end - mv.start);

            // Рисуем стрелку от конца вектора к началу

            arrowedLine(dst, scaledEnd, mv.start, color, thickness, LINE\_AA);

        }

    }

    // logger.info("Motion vectors drawn with scaling.");

}

/\*\*

 \* @brief Recursive median filter for motion vectors

 \*

 \* Функция recursiveMedianFilter() реализует рекурсивный медианный фильтр

 \* для векторов движения. Она проходит по всем векторным элементам

 \* сетки, собирает соседние векторы в окне kernelSize x kernelSize, находит

 \* векторную медиану и обновляет вектор движения в результирующем

 \* массиве.

 \*

 \* @param[in] srcVectors Исходная сетка векторов движения

 \* @param[out] dstVectors Результирующая сетка векторов движения

 \* @param[in] kernelSize Размер окна (kernel) для фильтрации (по умолчанию 3)

 \*/

void recursiveMedianFilter(const vector<vector<MotionVector>>& srcVectors, vector<vector<MotionVector>>& dstVectors, int kernelSize = 3) {

    // logger.info("   recursive median filter      kernel size: " + to\_string(kernelSize));

    int rows = srcVectors.size();

    int cols = srcVectors[0].size();

    // инициализация результирующего вектора

    dstVectors = srcVectors;

    // проходим по всем векторным элементам сетки

    for (int i = 0; i < rows; ++i) {

        for (int j = 0; j < cols; ++j) {

            const MotionVector& current = srcVectors[i][j];

            // собираем соседние векторы в окне kernelSize x kernelSize

            vector<Point2f> neighbors;

            for (int di = -kernelSize / 2; di <= kernelSize / 2; ++di) {

                for (int dj = -kernelSize / 2; dj <= kernelSize / 2; ++dj) {

                    int ni = i + di;

                    int nj = j + dj;

                    // проверяем, что сосед находится в пределах сетки

                    if (ni >= 0 && ni < rows && nj >= 0 && nj < cols) {

                        const MotionVector& neighbor = srcVectors[ni][nj];

                        // добавляем ненулевые векторы

                        if (neighbor.magnitude > 0) {

                            neighbors.push\_back(neighbor.end - neighbor.start);

                        }

                    }

                }

            }

            // если есть соседи, находим векторную медиану

            if (!neighbors.empty()) {

                Point2f medianVector = current.end - current.start;

                float minSumDistance = FLT\_MAX;

                for (const auto& candidate : neighbors) {

                    float sumDistance = 0;

                    for (const auto& neighbor : neighbors) {

                        sumDistance += norm(candidate - neighbor); // Норма L2

                    }

                    // обновляем медиану, если сумма расстояний меньше

                    if (sumDistance < minSumDistance) {

                        minSumDistance = sumDistance;

                        medianVector = candidate;

                    }

                }

                // обновляем вектор движения в результирующем массиве

                dstVectors[i][j].end = dstVectors[i][j].start + medianVector;

                dstVectors[i][j].magnitude = norm(medianVector); // Пересчитываем величину

            }

        }

    }

    // logger.info("Recursive median filter completed.");

}

// Функция для кластеризации векторов движения для сегментации

/\*\*

 \* @brief Кластеризация векторов движения на основе угловой схожести

 \*

 \* Функция выполняет кластеризацию векторов движения, применяя пороговое

 \* значение угла для определения сонаправленности векторов. Каждый блок

 \* изображения получает метку кластера, если он имеет ненулевой вектор

 \* движения и сонаправлен с другими блоками.

 \*

 \* @param motionVectorsGrid Сетка векторов движения для каждого блока

 \* @param labels Двумерный вектор для сохранения меток кластеров

 \* @param angleThreshold Пороговое значение угла (в градусах) для

 \*                       определения сонаправленности векторов

 \*/

void clusterMotionVectors(const vector<vector<MotionVector>>& motionVectorsGrid, vector<vector<int>>& labels, float angleThreshold = 30.0f) {

    // logger.info("Начало кластеризации векторов движения...");

    int rows = motionVectorsGrid.size();

    int cols = motionVectorsGrid[0].size();

    // Инициализация меток, -1 означает отсутствие метки

    labels = vector<vector<int>>(rows, vector<int>(cols, -1));

    int currentLabel = 0; // Начальная метка

    // Лямбда-функция - это анонимная функция, которая может быть объявлена

    // где угодно и может использовать любые локальные переменные, доступные

    // в месте объявления. Она необходима здесь, чтобы не создавать

    // глобальную функцию, которая будет доступна только в этом месте.

    // Лямбда-функция areVectorsAligned будет использоваться только в этом

    // месте, поэтому нет смысла объявлять отдельную глобальную функцию.

    // Лямбда-функция для проверки сонаправленности двух векторов. Она

    // вычисляет угол между двумя векторами, используя скалярное произведение,

    // и возвращает true, если угол меньше порогового значения (angleThreshold),

    // иначе возвращает false.

    auto areVectorsAligned = [&](const MotionVector& a, const MotionVector& b) -> bool {

        Point2f vecA = a.end - a.start;

        Point2f vecB = b.end - b.start;

        float dotProduct = vecA.x \* vecB.x + vecA.y \* vecB.y; // Скалярное произведение

        float magnitudeA = norm(vecA);

        float magnitudeB = norm(vecB);

        if (magnitudeA == 0 || magnitudeB == 0) return false;

        // Угол между векторами

        float cosTheta = dotProduct / (magnitudeA \* magnitudeB);

        float angle = acos(cosTheta) \* 180.0 / CV\_PI; // Угол в градусах

        return angle <= angleThreshold; // Возвращаем true, если угол <= порога

    };

    // Обход всех блоков изображения

    for (int i = 0; i < rows; ++i) {

        for (int j = 0; j < cols; ++j) {

            const MotionVector& mv = motionVectorsGrid[i][j];

            // Пропускаем блоки с нулевым вектором

            if (mv.magnitude == 0) continue;

            // Если у блока еще нет метки, создаем новую

            if (labels[i][j] == -1) {

                labels[i][j] = currentLabel;

                // logger.info("Присвоение новой метки: " + to\_string(currentLabel) + " блоку (" + to\_string(i) + ", " + to\_string(j) + ")");

                // Очередь для обработки соседей, которая позволяет

                // обрабатывать все соседние блоки, которые принадлежат

                // одному кластеру, в рамках одной операции.

                //

                // Начнем с блока, который имеет ненулевой вектор движения,

                // и присвоим ему метку currentLabel.

                // Затем обрабатываем всех его соседей, которые:

                // 1. имеют ненулевой вектор движения;

                // 2. еще не размечены;

                // 3. сонаправлены с вектором движения блока.

                //

                // Если найден сосед, который удовлетворяет условиям,

                // то присваиваем ему метку currentLabel и добавляем

                // его в очередь. Это позволяет обрабатывать

                // все соседние блоки, которые принадлежат одному

                // кластеру, в рамках одной операции.

                //

                // Алгоритм работает до тех пор, пока очередь не

                // будет пуста, что означает, что все блоки,

                // которые принадлежат одному кластеру,

                // будут обработаны.

                queue<pair<int, int>> toProcess;

                toProcess.push({i, j});

                while (!toProcess.empty()) {

                    auto [ci, cj] = toProcess.front();

                    toProcess.pop();

                    // logger.info("Обработка блока (" + to\_string(ci) + ", " + to\_string(cj) + ")");

                    // Проверяем 8 соседей

                    for (int di = -1; di <= 1; ++di) {

                        for (int dj = -1; dj <= 1; ++dj) {

                            if (di == 0 && dj == 0) continue; // Пропускаем сам блок

                            int ni = ci + di;

                            int nj = cj + dj;

                            // logger.info("Проверка соседа (" + to\_string(ni) + ", " + to\_string(nj) + ")");

                            // Проверяем границы

                            if (ni >= 0 && ni < rows && nj >= 0 && nj < cols) {

                                const MotionVector& neighbor = motionVectorsGrid[ni][nj];

                                // Проверяем условия: ненулевой вектор, еще не размечен, сонаправлен

                                if (neighbor.magnitude > 0 && labels[ni][nj] == -1 && areVectorsAligned(motionVectorsGrid[ci][cj], neighbor)) {

                                    // logger.info("Присвоение метки: " + to\_string(currentLabel) + " соседу (" + to\_string(ni) + ", " + to\_string(nj) + ")");

                                    labels[ni][nj] = currentLabel;

                                    toProcess.push({ni, nj}); // Добавляем соседа в очередь

                                }

                            }

                        }

                    }

                }

                // Увеличиваем номер текущей метки

                currentLabel++;

            }

        }

    }

    // logger.info("Кластеризация завершена. Всего кластеров: " + to\_string(currentLabel));

}

/\*\*

 \* @brief Визуализация контуров кластеров на изображении

 \*

 \* Функция visualizeClusterContours() отображает контуры кластеров на изображении,

 \* выделяя только внешние стороны границ кластеров. Она копирует исходное изображение

 \* в выходное и рисует линии для каждого кластера, используя случайные цвета.

 \*

 \* @param[in] src Исходное изображение

 \* @param[out] dst Изображение с отрисованными контурами кластеров

 \* @param[in] labels Сетка меток кластеров для каждого блока

 \* @param[in] blockSize Размер блока

 \*/

void visualizeClusterContours(const Mat& src, Mat& dst, const vector<vector<int>>& labels, int blockSize) {

    // Копируем исходное изображение в выходное

    src.copyTo(dst);

    // Карта для хранения случайных цветов кластеров

    map<int, Scalar> clusterColors;

    RNG rng(12345); // Генератор случайных чисел для устойчивого выбора цветов

    int rows = labels.size(); // Количество строк в сетке меток

    int cols = labels[0].size(); // Количество столбцов в сетке меток

    // Назначаем случайный цвет для каждого кластера

    for (int i = 0; i < rows; ++i) {

        for (int j = 0; j < cols; ++j) {

            int label = labels[i][j]; // Метка текущего блока

            if (label != -1 && clusterColors.find(label) == clusterColors.end()) {

                // Если метка не -1 и цвет еще не назначен, назначаем случайный цвет

                clusterColors[label] = Scalar(rng.uniform(0, 256), rng.uniform(0, 256), rng.uniform(0, 256));

            }

        }

    }

    // Обход всех блоков для нахождения внешних границ кластеров

    for (int i = 0; i < rows; ++i) {

        for (int j = 0; j < cols; ++j) {

            int label = labels[i][j];

            if (label == -1) continue; // Пропускаем блоки, не принадлежащие кластеру

            Point topLeft(j \* blockSize, i \* blockSize); // Верхний левый угол блока

            Point bottomRight((j + 1) \* blockSize - 1, (i + 1) \* blockSize - 1); // Нижний правый угол блока

            // Проверяем внешние стороны блока

            if (i == 0 || labels[i - 1][j] != label) {

                // Верхняя сторона

                line(dst, topLeft, Point(bottomRight.x, topLeft.y), clusterColors[label], 2);

            }

            if (i == rows - 1 || labels[i + 1][j] != label) {

                // Нижняя сторона

                line(dst, Point(topLeft.x, bottomRight.y), bottomRight, clusterColors[label], 2);

            }

            if (j == 0 || labels[i][j - 1] != label) {

                // Левая сторона

                line(dst, topLeft, Point(topLeft.x, bottomRight.y), clusterColors[label], 2);

            }

            if (j == cols - 1 || labels[i][j + 1] != label) {

                // Правая сторона

                line(dst, Point(bottomRight.x, topLeft.y), bottomRight, clusterColors[label], 2);

            }

        }

    }

}

// Main lab function to implement the full workflow

void lab3\_MotionVector(const Mat& img\_bgr1, const Mat& img\_bgr2) {

    logger.info("Lab 3: Motion Vectors");

    Image img;

    // создаём папку для выходных изображений

    if (\_access(img.output\_path.c\_str(), 0) != 0) {

        if (\_mkdir(img.output\_path.c\_str()) == -1) {

            logger.error("Failed to create directory: {}", img.output\_path);

            return;

        }

        logger.info("Created directory: {}", img.output\_path);

    }

    int scaleFactor = 2;

    resize(img\_bgr1, img.bgr1, Size(img\_bgr1.cols / scaleFactor, img\_bgr1.rows / scaleFactor));

    resize(img\_bgr2, img.bgr2, Size(img\_bgr1.cols / scaleFactor, img\_bgr1.rows / scaleFactor));

    imshow("img bgr 1", img.bgr1);

    imshow("img bgr 2", img.bgr2);

    cvtColor(img.bgr1, img.gray1, COLOR\_BGR2GRAY);

    cvtColor(img.bgr2, img.gray2, COLOR\_BGR2GRAY);

    // Compute motion vectors

    img.blockSize = 16;

    computeMotionVectors(img.gray1, img.gray2, img.motionVectorsOrig, img.blockSize);

    // Display original vectors

    drawMotionVectors(img.bgr2, img.motionVectorImageOrig, img.motionVectorsOrig);

    imshow("Original Motion Vectors", img.motionVectorImageOrig);

    imwrite(img.output\_path + "Original Motion Vectors.png", img.motionVectorImageOrig);

    // Filter motion vectors

    recursiveMedianFilter(img.motionVectorsOrig, img.motionVectorsFiltered, 3);

    // Display filtered vectors

    drawMotionVectors(img.bgr2, img.motionVectorImageFiltered, img.motionVectorsFiltered);

    imshow("Filtered Motion Vectors", img.motionVectorImageFiltered);

    imwrite(img.output\_path + "Filtered Motion Vectors.png", img.motionVectorImageFiltered);

    // Create segmentation mask

    Mat segmentationMask = Mat::zeros(img.gray1.size(), CV\_8UC1);

    clusterMotionVectors(img.motionVectorsFiltered, img.labels, 90.0F);

    visualizeClusterContours(img.bgr2, img.clusterVisualization, img.labels, img.blockSize);

    imshow("Segmentation", img.clusterVisualization);

    imwrite(img.output\_path + "Segmentation.png", img.clusterVisualization);

    waitKey(0);

}

#endif // MOTIONVECTOR\_H

1. Входные данные

Исходными данными являются два кадра видеопоследовательности со статичным фоном, на которых присутствуют различные перемещающиеся объекты.



Рисунок 1 – Кадр видеопоследовательности 1.



Рисунок 2 – Кадр видеопоследовательности 2.

1. Поиск векторов движения

За основу алгоритма был выбран корреляционный метод с использованием в качестве целевой функции MAD с полным перебором поиска.



Рисунок 3 – найденные вектора движения.

1. Фильтрация векторов движения

Применение рекурсивной векторной медианной фильтрации.



Рисунок 4 – фильтрованные вектора движения.

1. Кластеризация блоков с векторами движения

Далее реализуем алгоритм кластеризации блоков с векторами движения.

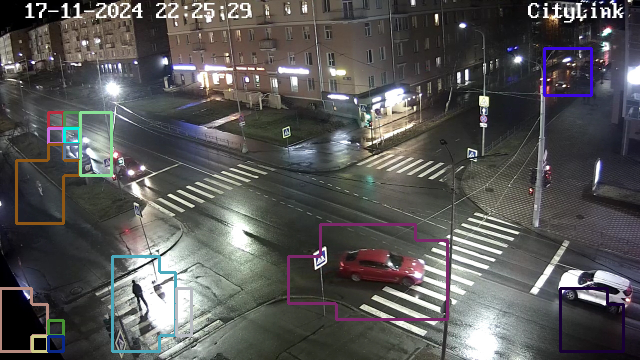


Рисунок 5 – кластеризованные блоки с векторами движения.

**Вывод:** в ходе работы был реализован алгоритм сегментации объектов на основе векторов движения. В качестве входных данных были выбраны два кадра с камеры дорожного наблюдения. В ходе алгоритма были реализованы корреляционный метод с использованием в качестве целевой функции MAD с полным перебором поиска, рекурсивная векторная медианная фильтрация и алгоритм кластеризации блоков с векторами движения.