Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Московский государственный технический университет
имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Теоретическая информатика и компьютерные технологии»

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

К КУРСОВОЙ РАБОТЕ

НА ТЕМУ:

Обнаружение текста на изображениях

Студент ИУ9-52 _____ А.С. Шишигина
(Подпись, дата)

Руководитель курсовой работы _____ И.Э. Вишняков
(Подпись, дата)

Консультант _____ О.В. Криндач
(Подпись, дата)

2018 г.

Содержание

ВВЕДЕНИЕ

Обнаружение текста на произвольных естественных изображениях в отличие от его локализации в документах и других источниках текстовой информации имеет ряд дополнительных сложностей. К таковым можно отнести сложность фона, вариативность масштаба и направления, плохое освещение и различное качество изображений. Результаты обнаружения текста и его дальнейшего распознавания применяются для оцифровки документов, индексирования и извлечения информации из графиков и чертежей, индексирования изображений в поисковых системах, для автоматического построение аннотаций и мгновенного перевода, а также для преобразования текста в речь и многое другое.

Целью работы является разработка и реализация алгоритма детектирования и локализации текста на естественных изображениях.

Для достижения поставленной цели в рамках курсовой работы были поставлены следующие *задачи*:

- провести обзор существующих методов
- на основе исследований выбрать алгоритм для реализации
- выделить основные этапы работы алгоритма
- реализовать каждый этап, рассмотреть варианты модификации
- протестировать на независимых данных

I. Обзор существующих методов обнаружения текста на произвольных изображениях

Решение поставленной задачи обнаружения текста на естественных изображениях сильно варьируется в зависимости от конкретной области применения и необходимых требований от работы алгоритма. В настоящее время имеет место целый ряд подходов. Среди них выделяют:

- методы, основанные на анализе текстур;

- методы, основанные на анализе компонент связности;

- гибридные методы;

- методы, использующие глубокое обучение.

1.1 *М*етоды, основанные на анализе текстур

Главной идеей, использующейся в данных методах является то, что текст имеет отличительную текстуру [1, 2]. Таким образом, для выделения текстовых и нетекстовых областей при анализе используются текстурные признаки, например, локальная интенсивность, вейвлет-коэффициенты, результаты применения фильтров. Это позволяет алгоритмам неплохо отрабатывать при сложном фоне. Однако в общем случае алгоритмы имеет высокую вычислительную сложность, так как требуют обработки нескольких масштабов и многочисленное применение свертки. К тому же, данные методы работают хорошо по большей части только при горизонтально направленном тексте и чувствительны к изменению масштаба и поворотам.

1.2 *М*етоды, основанные на компонентах связности

В первую очередь на основе некоторых локальных признаков (однородность цвета или принадлежности границе) выделяются компоненты связности, которые

в свою очередь являются кандидатами в текстовые символы. Затем производится фильтрация с использованием различных эвристик или с помощью обученного классификатора, после чего оставшиеся символы объединяются в текст. Данный подход обычно более вычислительно эффективен, так как объем обрабатываемой информации в виде компонент соответственно мал. Также в связи с реализуемой инвариантностью к вращению и изменению масштаба и фона алгоритмы, использующие данный подход, занимают значительную часть среди уже предложенных вариантов решения задачи обнаружения текста.

Основополагающими алгоритмами, использующими этот подход, являются SWT [3] (*Stroke Width Transform* — преобразование ширины штриха) и MSER [4] (*Maximally Stable Extremal Regions* - максимально стабильные экстремальные области).

Основная идея первого алгоритма следующая: единственное, что отличает текстовые компоненты от нетекстовых, — относительно постоянная ширина штриха. Сходство значений данного признака вместе с пространственной близостью являются показателем принадлежности одной компоненте.

В MSER компонентами считаются элементы с экстремальным свойством функции интенсивности. Для каждого значения интенсивности обработанное изображение бинаризуется по соответствующему порогу. Области белого цвета считаются областями экстремума. После анализа количества итераций, при котором область экстремума оставалась неизменной (стабильной), и определении порогового значения, неподходящие области отсеиваются. При этом оставшиеся являются максимально стабильными экстремальными областями и считаются кандидатами в текстовые символы. В алгоритме предложенном *Neumann* производится последующая фильтрация обученным классификатором.

У обоих алгоритмов, как и у многих других методов, использующих компоненты связности, недостатками является использование множества

эвристик, регулирование параметров вручную, а также способность обнаруживать только горизонтальный текст.

Следует упомянуть, что существует множество разработанных алгоритмов, в какой-то мере решающих определенные проблемы данного подхода. Одним из таких является метод, предложенный *Yao* [5] в 2012 году. Основой работы алгоритма выделения компонент является построение SWT, но дальнейший анализ производится с помощью обученного классификатора, что позволяет не использовать некоторые эвристики. При правильно обученной модели можно достичь инвариантности к вращению и иметь возможность обнаруживать текст на разных языках.

1.3 *Г*ибридные методы

Гибридные методы используют одновременно преимущества двух выше указанных подходов. Например, в алгоритме, предложенном *Liu* [6], краевые пиксели всех возможных текстовых областей извлекаются с использованием сложной стратегии обнаружения границ. Затем проверяются значения градиентных и геометрический признаков с целью отбора потенциальных текстовых областей. И только после этого, производится текстурный анализ для лучших результатов.

С отличие от этого, гибридный метод, изложенный в работе *Pan* [7], использует анализ текстур на начальном этапе. С помощью классификатора, обученного на текстурных признаках (*histogram of oriented gradients, HOG* — гистограмма направленных градиентов) строятся карты вероятности, и уже на их основе выделяются компоненты связности.

1.4 *М*етоды, использующие глубокое обучение

Глубокое обучение позволяет избавиться от недостатков традиционных способов выделения компонент связности, а именно выбора признаков вручную и использования эвристик. В связи с объяснимым ростом популярности нейронных сетей последние исследования для решения задачи обнаружения текста основаны на данном методе [8, 9]. Основным преимуществом является то, что нейросети способны выделять признаки самостоятельно, используя неразмеченные данные, при этом выдавая качественные результаты. Также, как говорилось ранее, в большей части следующим этапом после обнаружения текстовых областей является распознавание текстовых символов. Глубокое обучение позволяет объединить данные задачи в *end-to-end* распознавание текста, выдающее на выходе сразу распознанный текст.

II. Разработка алгоритма обнаружения текста на изображениях

2.1 *О*писание алгоритма и выделение основных этапов работы

После изучения возможных вариантов обнаружения текста на произвольных изображениях за основу реализуемого алгоритма было решено взять метод, предложенный в работе *Yao*. Для этапа выделения компонент связности используется метод SWT, выбор обоснован его относительной эффективностью и результативностью. К тому же, он способен определять компоненты связности, напрямую используя карту границ. Однако в отличие от оригинального варианта, разработанный алгоритм инвариантен к вращению и изменению масштаба, а также способен детектировать текст на разных языках и в разном направлении. Этапы работы приведены в следующей схеме (Рис. 1).

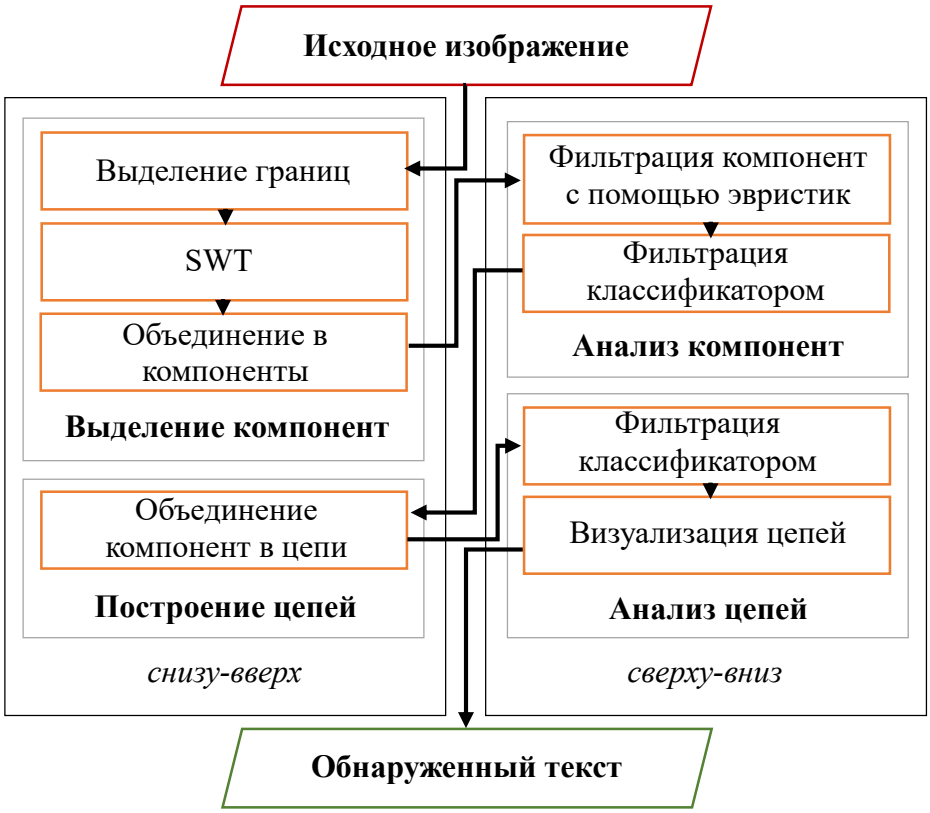


Рис. 1 - *Общая схема работы алгоритма*

В процессе обработки изображения для получения результата чередуются две разные модели:

- *снизу-вверх* группирование (сперва пиксели образуют связные компоненты, затем компоненты объединяются в цепи)

- *сверху-вниз* сокращение (нетекстовые компоненты и цепи обнаруживаются и устраняются)

2.2 Выделение компонент связности

Рассмотрим подробно работу алгоритма построения SWT-изображения для дальнейшего объединения пикселей в компоненты связности. Преобразование ширины штриха — локальный оператор, который вычисляет для каждого пикселя ширину штриха, частью которого он (пиксель) является с наибольшей вероятностью. Штрих можно описать как непрерывную часть изображения, образующую полосу почти постоянной ширины (Рис. 2).

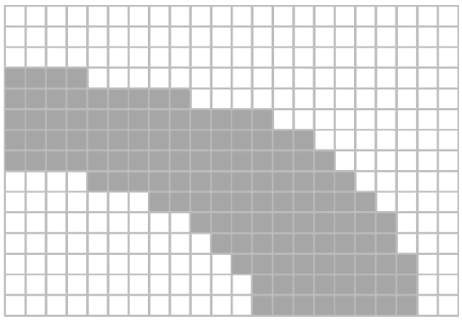
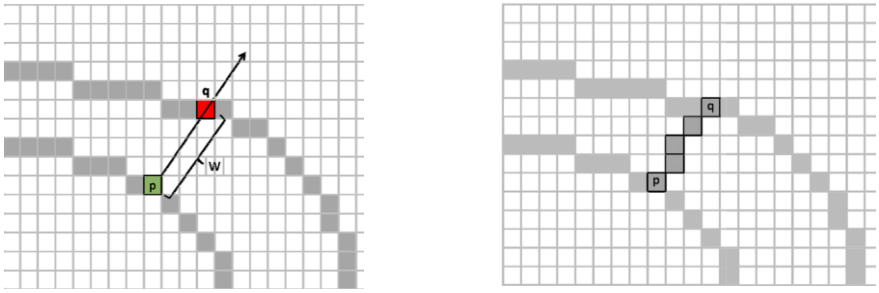


Рис. 2 - *Представление стандартного штриха в SWT-изображении*

Исходное значение каждого элемента будущей SWT-матрицы устанавливается равным -1. В первую очередь, для устранения шума и уменьшения вычислительных затрат исходное изображение сглаживается и преобразуется в оттенки серого, а затем применяется фильтра Гаусса [10] для размытия изображения. Далее для обработанного изображения применяется алгоритм Кэнни обнаружения границ [11]. Используя оператор Шарра [12], вычисляем направления градиента для всего изображения, после чего рассматриваем направление градиента d_p каждого граничного пикселя p.

Если пиксель принадлежит границе штриха, то d_p должен быть точно перпендикулярен направлению штриха. Продолжаем луч $r = p + n * d_p$, $n > 0$ до тех пор, пока не найдем другой граничный пиксель q. Затем рассматриваем направление градиента d_q для пикселя q. Если d_q строго противоположно d_p ($d_q = - d_p \pm \frac{\pi}{6}$), то каждый элемент выходного изображения SWT, соответствующий пикселю вдоль сегмента $[p, q]$, принимает значение минимума из ширины штриха $w = ||p — q||$ и его текущего значения. Иначе, если подходящий пиксель q не нашелся или если d_q не противоположно направлено с d_p, луч отбрасывается.



(а) (б)

Рис. 3 - *Построение SWT-изображения*

(а) p — пиксель на границе штриха, q – соответствующий пиксель на другой стороне штриха

(б) каждый пиксель вдоль луча принимает значение минимума из его текущего значения и найденной шириной штриха

Как показано на Рис. 4, значения SWT для более сложных случаев, как углы, после первого прохода, описанного выше, не будут соответствовать реальной ширине штриха. Таким образом, мы совершаем второй обход по каждому сформированному лучу и считаем медиану m значений в SWT-матрице всех его пикселей. Затем каждый пиксель луча со значением SWT выше принимает значение m.

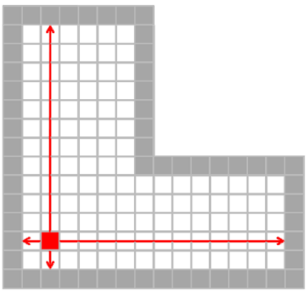
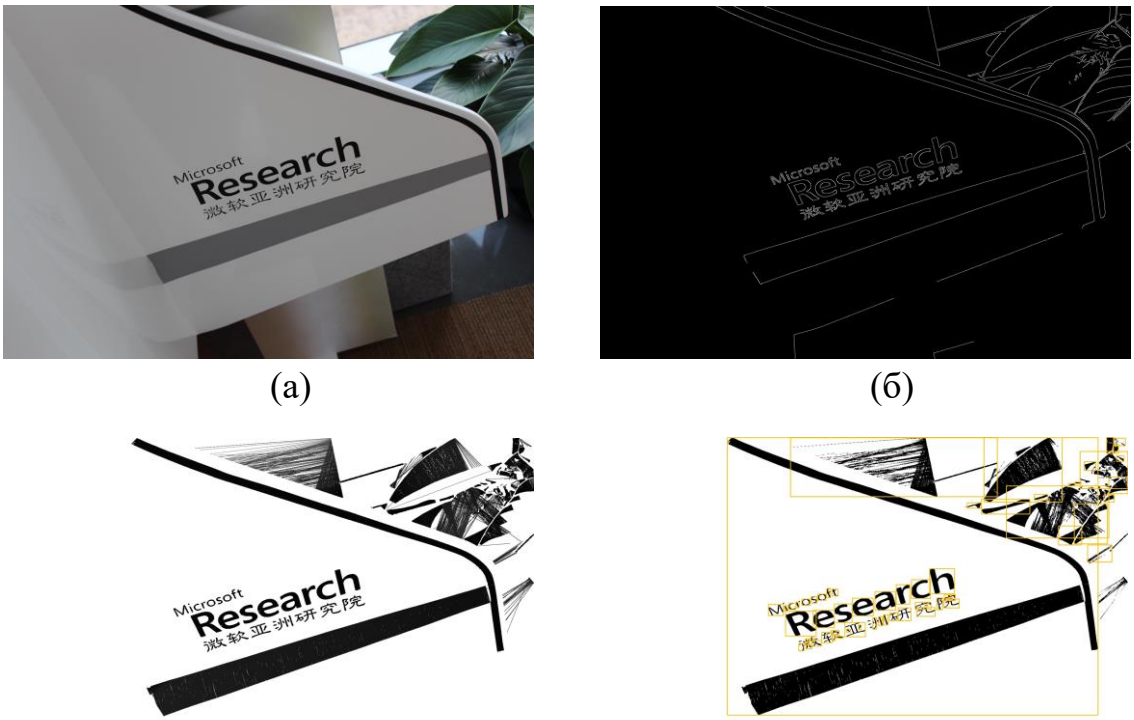


Рис. 4 - *Неопределенность на пересечении нескольких штрихов*

На следующем шаге формируются компоненты связности. Соседние пиксели, отношение ширины штриха которых не превышает 3, группируются вместе при помощи правила ассоциативности.



(а) (б)

(в) (г)

Рис. 5 - *Результаты первого этапа алгоритма*

(а) исходное изображение

(б) выделение границ Кэнни

(в) SWT-изображение

(г) выделенные компоненты связности

2.3 Анализ компонент и построение обученного классификатора

На данном этапе происходит получение признаков компонент связности и их классификация на текстовые и нетекстовые компоненты с помощью двухэтапной фильтрации.

Для компоненты связности *c* с соответствующими значимыми *q* пикселями вычисляются ограничивающая рамка *bb(c)* (ее ширина и высота определяются как *w(c)* и *h(c)*, соответственно), соотношение сторон *AR(c)*, коэффициент заполнения *OR(c)*, плотность *D(c)*, среднее значение ширины штриха *mean(c),* среднеквадратическое отклонение ширины штриха *SD(c),* а также коэффициент вариации ширины штриха *WV(c).* При этом данные свойства очень быстро вычисляются. В таблице 1 представлены формулы их вычисления.

Таблица 1 - Формулы для вычисления признаков компонент

Свойство	**Определение**
соотношение сторон	$AR(c) = min\{\frac{w(c)}{h(c)}, \frac{h(c)}{w(c)}\}$
среднее значение ширины штриха	$mean(c) = \frac{1}{q}\sum_{i=1}^{q} SWT_i$
среднеквадратическое отклонение ширины штриха	$SD(c) = \sqrt{\frac{1}{q}\sum_{i=1}^{q}(SWT_i - mean(c))^2}$
плотность	$D(c) = \frac{q}{\pi(w(c) + h(c))}$
коэффициент заполнения	$OR(c) = \frac{q}{w(c)h(c)}$
коэффициент вариации ширины штриха	$WV(c) = \frac{SD(c)}{mean(c)}$

В первую очередь, исключим компоненты, которые, вероятнее всего, образованы шумом и одиночными линиями. Такие компоненты можно выявить по соотношению сторон ограничивающей рамки и коэффициента заполнения (значение хотя бы одного из этих свойств, меньше порога $t = 0.1$). Кроме того, исключаются компоненты маленького размера, которые будут текстовыми с меньшей вероятностью. Это позволяет не только отбросить очевидные нетекстовые компоненты, но и уменьшить вычислительную сложность алгоритма. В работе в качестве порога было выбрано значение $q_{min} = 40$.

Второй этап представляет собой работу классификатора, обученного выявлять нетекстовые компоненты, которые трудно удалить предварительным фильтром. Для обучения использовался набор признаков на уровне компонент, фиксирующих различия геометрических и текстурных признаков между текстовыми и нетекстовыми компонентами. Критериями для выделения признаков являются инвариантность к изменению масштаба, инвариантность к вращению и низкая вычислительная стоимость. Для полного построения дескриптора к набору выше указанных признаков также добавляется ориентация и центр объекта, входящего в компоненту.

В оригинальной статье для выявления центра и ориентации был использован метод *CamShift* [13], который обычно используется для детектирования конкретных объектов на изображении. Однако при реализации данного способа некоторые текстовые символы имели нулевые значения соответствующих признаков, что квалифицировалось как пропуски в обучающей выборке. В этой связи мною был выбран другой подход.

Преобразование Хафа [14] позволяет находить на изображении простые формы, в том числе линии. В основе его работы лежит утверждение о том, что любая точка бинаризованного изображения может быть частью некоторого набора возможный линий. При этом для описания прямых используется пространство параметров, с помощью которых можно задать прямую на плоскости в полярных координатах. Далее для каждой точки этого пространства

суммируется количество голосов, отданных за нее. Точки локальных максимумов соответствующей аккумуляторной функции будут совпадать с параметрами линий, присутствующих на изображении.

Вместе с тем имеется возможность регулировать минимальное расстояние между точками и достаточное количество голосов для детектирования прямой. Таким образом, мы можем находить направление строки текста или отдельно стоящей буквы, рассматривая границы штриха как направляющие линии. Однако во втором случае при символах, не включающих в себя прямые линии (например, «о»), результат не будет соответствовать направлению соседствующих компонент (символов в строке). Поэтому был выбран первый вариант (Рис. 6).

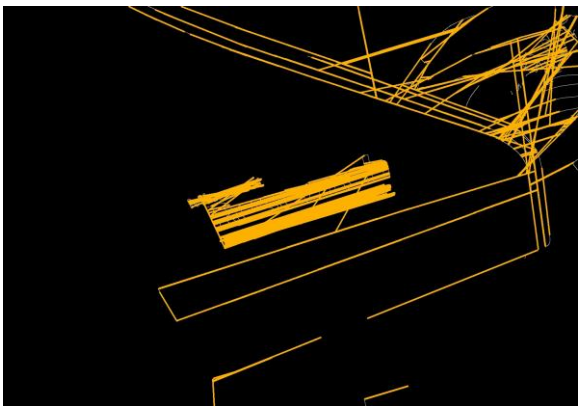


Рис. 6 - *Применение преобразования Хафа*

Центром объекта назначается точка, имеющая в качестве *(x, y)* среднее значение соответствующих координат входящих в компоненту значимых точек.

2.4 Построение текстовых цепей

На данном этапе кандидаты объединяются в цепи. Также этот шаг включает в себя первоначальную фильтрацию, так как символьные кандидаты, которые не могут быть объединены в цепи, считаются шумами или помехами фона и удаляются.

В первую очередь, компоненты объединяются в пары. Введем понятие *характеристического масштаба* S (1), равного сумме ширины и длины компоненты. Если два кандидата имеют похожую ширину штриха (2), схожие размеры (3) и достаточно близко расположены друг к другу (4), то они считаются парой. Расстояние вычисляется как евклидово расстояние между центрами компонент. Заметим, что символьный кандидат может принадлежать нескольким парам.

$$S(c) = w(c) + h(c) \quad (1)$$

$$\frac{mean(c_i)}{mean(c_j)} < 2 \quad (2)$$

$$\frac{S(c_i)}{S(c_j)} < 2.5 \quad (3)$$

$$dist(c_i, c_j) < 2 * max(w(c_i), w(c_j)) \quad (4)$$

Каждая пара представляет собой цепочку. Для каждой пары цепей, имеющих по меньшей мере один общий кандидат и близкое направление (модуль разницы меньше $\frac{\pi}{6}$), проверяется выполнение условия схожести. Процесс продолжается до тех пор, пока никакие цепи не могут больше быть объединены. Символьные кандидаты, которые не входят ни в одну цепочку, отбрасываются.

2.5 Анализ цепей и фильтрация

Кандидаты в цепи, сформированные на предыдущем этапе, могут содержать ложные срабатывания, которые представляют собой случайные комбинации фоновых помех (таких как листья деревьев и трава) и повторяющиеся узоры (например, кирпичи или окна). Для детектирования соответствующих нетекстовых цепей были введены следующие признаки: количество символьных кандидатов (обусловлено тем, что ложноположительные результаты обычно имеют очень мало (случайные помехи) или слишком много (повторяющиеся

шаблоны) кандидатов), средняя вероятность, среднее значение направления, вариативность размера, вариативность расстояния, среднее отношение осей, средняя плотность, количество уникальных цветов части изображения, соответствующей цепи. Все необходимые формулы для их вычисления представлены в следующей таблице:

Таблица 2 - Формулы для вычисления признаков цепей

Свойство	**Определение**
количество символьных кандидатов	$count(ch)$
средняя вероятность	$MP(ch) = \frac{1}{count(ch)} \sum_{i=1}^{count(ch)} pred_i$
среднее значение направления	$MD(ch) = \frac{1}{count(ch)} \sum_{i=1}^{count(ch)} dir_i$
средний размер	$MS(ch) = \frac{1}{count(ch)} \sum_{i=1}^{count(ch)} S_i$
среднеквадратическое0 отклонение размера	$SSD(ch) = \sqrt{\frac{1}{q} \sum_{i=1}^{count(ch)} (S_i - MS(ch))^2}$
коэффициент вариации размера	$SV(ch) = \frac{SD(\mathrm{ch})}{MS(ch)}$
расстояние компоненты	$Dist_i = \min(dist_{j \neq i}^{count(ch)}(c_i,\ c_j)$, где $dist(c_i,\ c_j)$ вычисляется как евклидово расстояние между центрами компонент
среднее расстояние	$MDist(ch) = \frac{1}{count(ch)} \sum_{i=1}^{count(ch)} Dist_i$
среднеквадратическое отклонение расстояния	$DSD(ch) = \sqrt{\frac{1}{q} \sum_{i=1}^{count(ch)} (Dist - MDist(\mathrm{ch}))^2}$

коэффициент вариации расстояния	$DV(ch) = \frac{DSD(ch)}{MDist(ch)}$
среднее отношение осей	$MAR(ch) = \frac{1}{count(ch)} \sum_{i=1}^{count(ch)} AR_i$
средняя плотность	$MD(ch) = \frac{1}{count(ch)} \sum_{i=1}^{count(ch)} D_i$

«Выжившие» цепи выводятся системой как обнаруженный текст. Для каждого финального текста вычисляется наименьшая ограничивающая рамка, содержащая все символы. На рисунке изображен пример работы на изображении, содержащим направленный текст на английском и китайском языках (Рис. 7).

Рис. 7 - *Финальный результат*

III. Реализация алгоритма обнаружения текста

3.1 Обзор выбранных инструментов разработки

Для реализации предложенного алгоритма было решено использовать такие языки программирования, как *C++* и *Python* (версии 2.7). Для обработки изображений и применения стандартных алгоритмов была выбрана библиотека компьютерного зрения *OpenCV* (версии 3.3.0) [15]. Сохранение промежуточных данных с целью передачи их между модулями, реализованных в том числе на разных языках, производится с помощью протокола сериализации *protobuf* (версии 3.6.1) [16]. Выбор обоснован эффективностью и простотой работы с ним, а также тем, что описанная структура данных может компилироваться в класс на необходимом языке программирования.

Этап классификации реализован с помощью библиотеки машинного обучения *CatBoost* [17]. Также для визуализации процесса обучения, проверки этапов работы и составления обучающей выборки использовались такие дополнительные библиотеки языка Python, как *plot* [18], *sklearn* [19] и *pandas* [20], соответственно. Кроме того, для эффективного анализа изображений, а именно, обхода всех пикселей на этапе выделения компонент связности были интегрированы библиотеки *graph* и *unordered_map* собрания библиотек *boost* [21] языка C++.

Также для оптимизации работы алгоритма был использован модуль OpenCV, предназначенный для реализации некоторых методов на GPU с помощью технологии CUDA. Для совместимости версий необходимо использовать следующие инструменты*: CUDA Toolkit 9.0, cuDNN 7.2.1, gcc* и *g++* версии 6.0.

Оценка времени работы проводилась при следующей конфигурации тестовой платформы: процессор Intel® Core™ i3-7100U CPU @ 2.40 GHz, ОЗУ

6.00 ГБ, мобильная видеокарта NVIDIA GeForce 940MX. Рассмотрим полученные данные на разных этапах.

3.2 Детали реализации

3.2.1 Выделение компонент связности

Построение SWT-изображения и компонент связности реализовано в файлах *StrokeWidthTransform.cpp/h* и *ConnectedComponents.cpp/h*, соответственно.

В процессе реализации было найдено несколько проблем. Во-первых, так как один проход описанного алгоритма при стандартном следовании направлению градиента гарантирует нахождение только темных штрихов на светлом фоне, а на естественных изображениях цвет текста варьируется, данный алгоритм необходимо прогонять дважды: для темных и светлых символов (Рис. 8). При этом для второго обхода необходимо изменить направление градиента в противоположную сторону (Приложение 1).

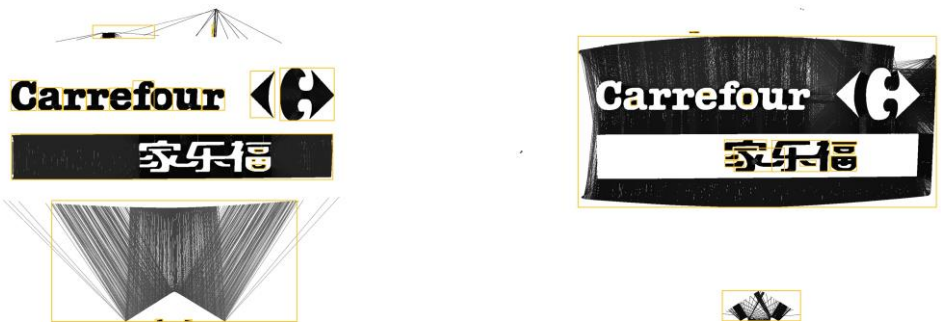


Рис. 8 - *Двойной проход для разных цветов текста*

Во-вторых, на тренировочных изображениях случалось, что границы имели просветы, то есть были незамкнутыми. В данной ситуации построенное SWT-изображение имеет погрешности, которые в последствие оказывают влияние на корректное выделение компонент (Рис. 9). Для улучшения результатов было

решено использовать математическую морфологию изображения, а именно размыкание, т.е. последовательное применение эрозии и наращивания. Далее пиксели, не принадлежащие пересечению исходного SWT-изображения и обработанного, удаляются из компоненты связности.

(а) (б) (в) (г)

Рис. 9 - *Применение размыкания для удаления погрешностей*

(а) неправильное выделение компоненты

(б) применение эрозии

(в) применение наращивания

(г) преобразованное выделение компоненты (красным цветом помечены пиксели исходного изображения, не входящие в преобразованное)

Процесс объединения пикселей в компоненты реализован с помощью построения графа, в котором символьные кандидаты соответствуют компонентам связности, связность вершин которых зависит от параметров, выделенных ранее (Приложение 2).

3.2.2 Анализ компонент и построение обученного классификатора

Получение признаков компоненты и сохранение их в нужном для обработки формате реализовано в файле *feature_extraction.py.* Работа классификатора, процесс его обучения отображены в файлах *clf.py* и *training_plot.ipynb*.

В качестве классификатора компонентного уровня было выбрано градиентное усиление на деревьях решений, реализованное с помощью библиотеки *CatBoost*. Преимуществом данного классификатора является

возможность работать с категориальными признаками напрямую. Такими признаками в данной задаче будут являться название изображения и принадлежность к темному или светлому тексту. Это позволит учитывать предсказания уже обработанных компонент с такими же значениями этих категориальных признаков, так как при подходящих значениях числовых они с большей вероятностью будут относиться к одному классу.

Обучающая выборка, включающая в себя текстовые и нетекстовые компоненты, была создана самостоятельно. Широко используемый в своей области набор данных ICDAR, предназначенный для задачи обнаружения текста, имеет два основных недостатка. Во-первых, большинство текстовых строк (или отдельных символов) представлены в горизонтальном виде. Во-вторых, весь текст на английском. В этой связи компоненты выборки были получены из изображений набора данных MSRA-TD500 [22], который был представлен в оригинальной статье. При выборе изображений учитывалась вариативность ориентации текста, языка и масштаба. Созданная обучающая выборка содержит 4654 объекта.

На промежуточном этапе реализации изначально в качестве модели классификации был выбран ансамбль, использующий вдобавок результаты нескольких дескрипторов изображения, описывающих ключевые точки. Такие признаки, как *SIFT (Scale-invariant feature transform)* [23], *SURF (Speeded-Up Robust Features)* [24] и *ORB (Oriented FAST and Rotated BRIEF)* [25] инвариантны к вращению и изменению масштаба. К тому же, они имеют реализацию в библиотеке OpenCV. Однако после оценки качества их работы было принято решение отказаться от данного подхода, так как использование этих признаков только ухудшало общее качество. Стоить заметить, что данные дескрипторы дают неплохое значение точности, однако значение общей метрики ухудшалось за счет неполноты полученных результатов.

3.2.3 Анализ цепей и фильтрация

Класс цепей текста соответствует файлу *components_chain.py.* Здесь представлены методы для построения связанных пар, а также объединения нескольких цепей в одну. Скрипт *text_localizing.py* реализует общую работу этой части алгоритма. В методе *extract_features* происходит вычисление значений определенных признаков для цепей (строк), а в методе *draw_bounding_box* реализуется отрисовка финальных ограничивающих рамок обнаруженного текста. Файл *write_chain_features.py* необходим для записи полученного дескриптора цепей изображения в требуемый формат, а также для построения обучающей выборки.

Обучающая выборка была построена на основе того же MSRA-TD500, размеченные данные которого включают в себя положение ограничивающей рамки текста и угол ее наклона.

3.3 Оптимизация

Для оптимизации работы программы, как говорилось ранее, был использован встроенный модуль библиотеки OpenCV, реализующий некоторые известные алгоритмы с использованием технологии CUDA. CUDA — это архитектура параллельных вычислений от NVIDIA, позволяющая существенно увеличить вычислительную производительность благодаря использованию GPU, т.е. графических процессоров. Следующие алгоритмы были реализованы с помощью данной технологии: вычисление градиента яркости изображения с помощью оператора Шарра, построение размыкания (эрозии и наращивания), преобразование Хафа и размытие фильтром Гаусса. Тем самым, при отработке данной части алгоритма без оптимизации за 7.392 секунд применение вычислений на графическом процессоре позволило сократить это время до 6.638 секунд.

Пример эквивалентных частей кода с реализацией на CPU и GPU:

```
//CPU
vector<Vec4i> lines;
HoughLinesP(edge, lines, 1, CV_PI / 180, 50, 10, 100);

//GPU
cuda::GpuMat linesMat;
Ptr<cuda::HoughSegmentDetector> hough = cuda::createHoughSegmentDetector(1.0f, (float)CV_PI / 180, 50, 10, 100);
hough->detect(cuda::GpuMat(edge), linesMat);
if (!linesMat.empty()) {
    lines.resize((unsigned long)linesMat.cols);
    Mat h_lines(1, linesMat.cols, CV_32SC4, &lines[0]);
    linesMat.download(h_lines);
}
```

Один из вариантов способа уменьшения времени работы также является распараллеливание двух проходов построения SWT-изображения. Это позволяет сократить время выполнения данной части алгоритма до двух раз.

3.4 Руководство администратора и пользователя

Чтобы использовать готовое приложение, необходимо получить версию исходного кода, размещенного в git-репозитории *https://github.com/laneesra/TextDetection*, в удобном формате. Также для работы необходимо наличие protobuf-компиляторов и библиотеки OpenCV версии 3.3.0 для обоих языков C++ и Python, CMake версии 3.10 и выше, а также других библиотек, упомянутых ранее. Далее необходимо собрать проект, выполнив следующую последовательность команд (для операционной системы Ubuntu):

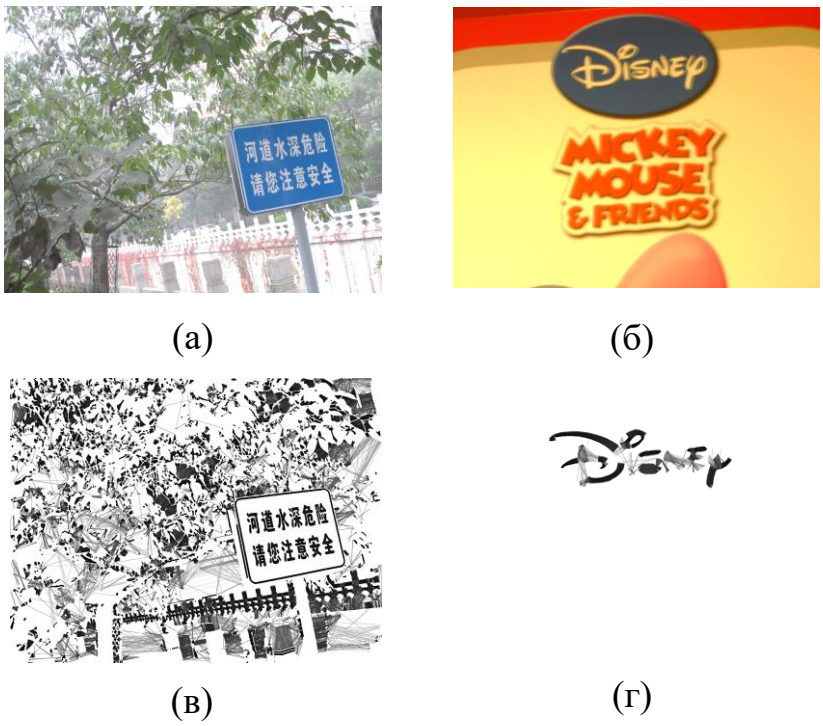
cd TextDetection
./build

Чтобы запустить приложение пользователю необходимо перейти в директорию проекта и запустить выполнение скрипта *./detect_text*. Затем следует ввести абсолютный путь до изображения, которое нужно обработать.

IV. Тестирование

Оценка результатов работы на произвольных изображениях проводилась на тестовом наборе данных знакомого MSRA-TD500, содержащим 200 изображений.

На начальном этапе используется детектор границ Кэнни, включающий в себя двойную пороговую фильтрацию. Однако исходные изображения имеют разное разрешение и качество, что при фиксированных значениях нижнего и верхнего порогов (80 и 160, соответственно) влечет за собой неправильную работу на изображениях плохого качества. Таким образом, детектор границ не находит ничего, что означает отсутствие компонент связности, а, значит, и текста.



(а) (б)

(в) (г)

Рис. 10 - *Результаты построения SWT-изображений*
(а) исходное изображение хорошего качества и (в) его SWT-изображение
(б) исходное изображение плохого качества и (г) его SWT-изображение

Следующий этап выделения компонент является наиболее весомым в плане затраченного времени. С учетом оптимизации время работы может

варьироваться от 2 до 15 и более секунд. Причиной тому является то, что при построении графа необходимо проанализировать значения всех пикселей, соответствующих штриху, а также значения их соседей.

Далее рассмотрим этап фильтрации компонент, в частности, результат работы классификатора. Для оценки качества работы классификатора обоих уровней использовались метрики: точность *(precision)* (5), полнота *(recall)* (6) и F-мера *(F-measure)* (7). Для их определения введем понятие матрицы ошибок (Таблица 3). Столбы соответствуют истинным меткам класса, а строки — результату работы классификатора.

Таблица 3 - Матрица ошибок

	y = 1	*y = 0*
pred = 1	True Positive (TP)	False Positive (FP)
pred = 0	False Negative (FN)	True Negative (TN)

Точность можно интерпретировать как долю компонент, названных классификатором текстовыми и при этом действительно являющимися ими. В свою очередь полнота характеризует способность модели обнаруживать данный класс в целом, т.е. долю компонент, классифицированных текстовыми, из общего количества реальных объектов этого класса. При этом решающей метрикой для отбора модели являлась F-мера, которая представляет собой гармоническое среднее между точностью и полнотой.

$$precision = \frac{TP}{TP+FP} \qquad (5) \qquad recall = \frac{TP}{TP+FN} \qquad (6)$$

$$F = \frac{2 * precision * recall}{precision + recall} \qquad (7)$$

Оценка обобщающей способности алгоритма и независимости созданной выборки производилась с помощью скользящего контроля (кросс-валидации). Данная процедура подразумевает разделение выборки *N* способами на две

непересекающиеся подвыборки (тренировочную и тестовую). Общая оценка качества формируется из среднего значения всех *N* разбиений.

На следующем графике показан процесс обучения классификатора компонент на обучающей выборке (Рис. 11). Ось ординат соответствует значению F-меры, ось абсцисс — количеству итераций. При этом модель была настроена с отслеживанием переобучения. Таким образом, среднее значение качества на обучающей части выборки примерно равно 0.95, а на тестовой — 0.90. Оптимальное количество итераций работы классификатора, при котором достигается максимальное значение F-меры, но не случается переобучение равно 737.

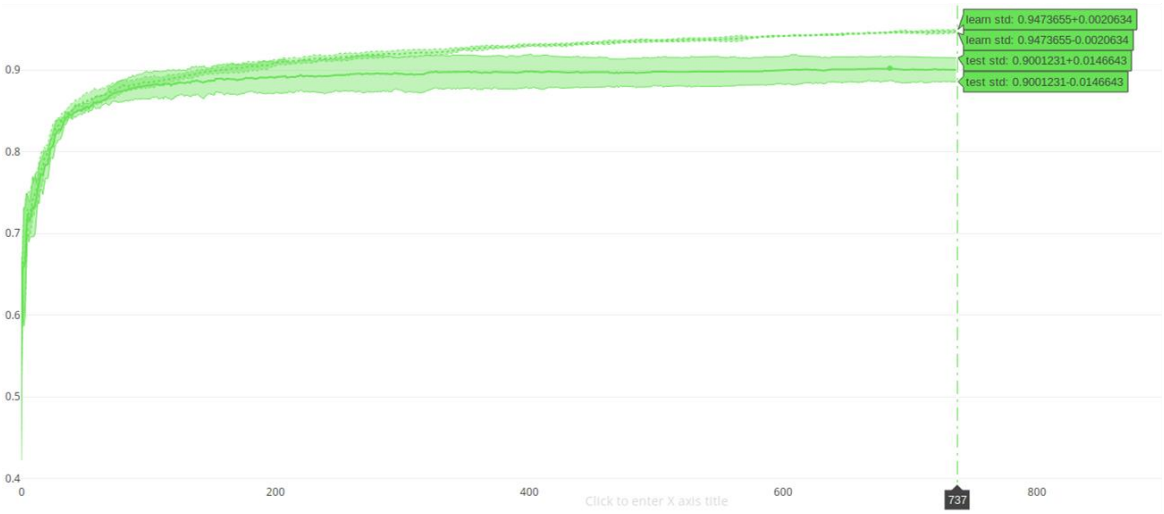


Рис. 11 - *Процесс обучения классификатора компонент*

Построенный нами классификатор в большей части отсекает нетекстовые компоненты, однако иногда случается, что некоторые истинные символы теряются, так как ложно распознаются как нетекстовые (Рис. 12).

(а) (б)

Рис. 12 - *Результат работы классификатора компонент*
(а) компоненты, полученные после построения SWT-изображения
(б) компоненты, распознанные классификатором как символьные кандидаты

В заключении, подведем итоги объединения и последующей классификации текстовых цепей. На следующем графике представлен процесс обучения классификатора уровня текстовых цепей (Рис. 13). Все настройки соответствуют построенному классификатору уровня компонент. При этом полученное среднее значение F-меры на тестовых данных соответствует 0.85, а на обучающих — 0.89. При этом количество итераций равно 75.

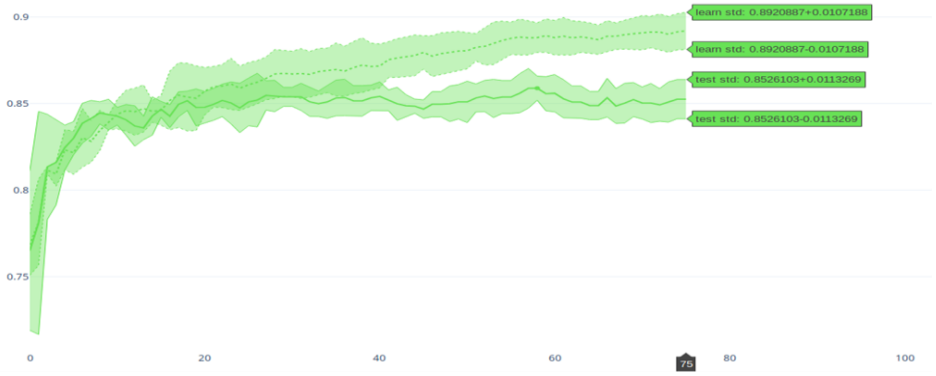


Рис. 13 - *Процесс обучения классификатора цепей*

При анализе полученных ограничивающий рамок текста, было замечено, что зачастую текст помечается целой областью, то есть не разделяется на отдельные строки и слова. Решением данной проблемы может быть другой способ построения цепей, например, с помощью минимальных остовных

деревьев графа. Как показано на Рис.14 г), построенный классификатор не всегда способен выявить все ложноположительные строки, соответствующие листьям деревьев и другим помехам.

Рис. 14 - *Результат работы классификатора цепей*

(а), (в) неотфильтрованные текстовые цепи и (б),(г) цепи, распознанные классификатором как текст

Весь этап работы классификатора и фильтрации занимает от 1 до 3 секунд. Таким образом, общее время выполнения составляет от 4 до 20 и более секунд.

Рис. 15 - *Финальный обнаруженный текст*

ЗАКЛЮЧЕНИЕ

В работе представлен один из вариантов решения задачи обнаружения текста на естественных изображениях. При этом реализованный алгоритм способен обнаруживать текст на различных языках, в разном направлении и масштабе.

Однако хорошие результаты соответствуют только изображениям хорошего качества. В качестве решения данной проблемы можно использовать глубокое обучение, а именно, самоорганизующиеся карты Кохонена [26], способные выделять необходимые текстурные признаки и обучаться, используя их, самостоятельно. Этот метод был выбран для дальнейшего изучения в данной области.

СПИСОК ИСПОЛЬЗОВАННОЙ ЛИТЕРАТУРЫ

1. Gllavata J., Ewerth R., Freisleben B. Text Detection in images based on unsupervised classification of high-frequency wavelet coefficients // In Proc. of CVPR, 2004.
2. Kim K.I., Jung K., Kim J.H. Texture-based approach for text detection n images using support vector machines and continuously adaptive mean shift algorithm. // IEEE Trans. PAMI, 25(12), 2003 – pp. 1631-1639.
3. Epshtein B., Ofek E., Wexter Y. Detecting text in natural scenes with stroke width transform. // In Proc. of CVPR, 2010.
4. Neumann L., Matas J. A method for text localization and recognition in real-world images. // In Proc. of ACCV, 2010.
5. Yao C., Bai X., Liu W., Ma Y., Tu Z. Detecting texts of arbitrary orientations in natural images. // In Proc. of CVPR, 2012.
6. Liu Y., Gotoand S., Ikenaga A contoyr-based robust algorithm for text detection in color images. // IEICE Transactions on Infromation and Systems, 89(3), 2006 – pp. 1221-1230.
7. Pan Y., Hou X., Liu C. A hybrid approach to detect and localize texts in natural scene images. // IEEE Trans. Image Processing, 20(3), 2011 – pp. 800-813.
8. Coates A., Carpenter B., Case C. Text Detection and Caracter Recognition in Scene Images with Unsupervised Feature Learning. // In IEEE Xplore, 2011.
9. Wang T., Wu D. J., Coates A., Ng. A.Y. End-to-end text recognition with convolutional neural networks. // In Proc. of ICPR, 2012.
10. Shapiro, L.G., Stockman G. C. Computer Vision. // Prentence Hall, 2001 - pp. 137-150.
11. Canny J.E. A computational approach to edge detection. // In IEEE Trans. PAMI, 1986.
12. Scharr H., Jahne B., Korkel S. Principles of filter design. // Handbook of Computer Vision and Applications. Academic Press, 1999.

13. Bradski, G.R. Real time face and object tracking as a component of a perceptual user interface. // Applications of Computer Vision, 1998. WACV '98. In Proc. of Fourth IEEE Workshop 19-21, Oct 1998 – pp. 214-219.

14. Duda R.O., Hart P.E. Use of the Hough Transformation to detect lines and curves in pictures. // Comm. ACM, Vol. 15, No.1, Jan 1972 – pp. 11-15.

15. https://docs.opencv.org/

16. https://developers.google.com/protocol-buffers/

17. https://catboost.ai/

18. https://matplotlib.org/users/pyplot_tutorial.html

19. https://scikit-learn.org/stable/

20. https://pandas.pydata.org/

21. https://www.boost.org/

22. http://www.iapr-tc11.org/mediawiki/index.php/MSRA_Text_Detection_500_Database (MSRA-TD500)

23. Lowe D.G., Object recognition from local scale-invariant features. // In Proc. of the International Conference on Computer Vision, 1999 – pp. 1150-1157.

24. Bay H., Ess A., Tuytelaars T. Van Gol L. SURF: Speeded Up Robust Features. // International Journal of Innovative Research in Computer and Communication Engineering Vol. 1, Issue 2, April 2013

25. Rublee, Ethan; Rabaud, Vincent; Konolige, Kurt; Bradski, Gary ORB: an efficient alternative to SIFT or SURF // IEEE International Conference on Computer Vision (ICCV), 2011.

26. Petrov N., Jordanov I.N. Unsupervised Texture Image Classification using Self-Organizing Maps, April 2012.

27. Zhu Y., Yao C., Bai X. Scene Text Detection and Recognition: Recent Advances and Future Trends // Frontiers of Computer Science, 10(1), June 2015.

Приложение 1.

Построение SWT-изображения.

```
void StrokeWidthTransform::buildSWT(bool dark_on_light) {
   float prec = .05;
   for (int row = 0; row < edge.rows; row++) {
      for (int col = 0; col < edge.cols; col++) {
         if (edge.at<uchar>(row, col) > 0) {
            Ray r;
            SWTPoint p(col, row);
            r.p = p;
            vector<SWTPoint> points;
            points.push_back(p);
            float curX = (float)col + 0.5f;
            float curY = (float)row + 0.5f;
            int curPixX = col;
            int curPixY = row;
            float G_x = gradientX.at<float>(row, col);
            float G_y = gradientY.at<float>(row, col);
            // normalize gradient
            float mag = sqrt((G_x * G_x) + (G_y * G_y));
            if (dark_on_light){
               G_x = -G_x / mag;
               G_y = -G_y / mag;
            } else {
               G_x = G_x / mag;
               G_y = G_y / mag;
            }
            while (true) {
               curX += G_x * prec;
               curY += G_y * prec;
               if ((int)(floor(curX)) != curPixX || (int)(floor(curY)) != curPixY) {
                  curPixX = (int)(floor(curX));
                  curPixY = (int)(floor(curY));
                  if (curPixX < 0 || (curPixX >= edge.cols) || curPixY < 0 || (curPixY >=
edge.rows)) {
                     break;
                  }
                  SWTPoint pnew(curPixX, curPixY);
                  points.push_back(pnew);
                  if (edge.at<uchar>(curPixY, curPixX) > 0) {
                     r.q = pnew;
                     float G_xt = gradientX.at<float>(curPixY, curPixX);
                     float G_yt = gradientY.at<float>(curPixY, curPixX);
                     mag = sqrt((G_xt * G_xt) + (G_yt * G_yt));
                     if (dark_on_light){
```

```
                    G_xt = -G_xt / mag;
                    G_yt = -G_yt / mag;
                } else {
                    G_xt = G_xt / mag;
                    G_yt = G_yt / mag;
                }
                if (acos(G_x * -G_xt + G_y * -G_yt) <  M_PI / 2.) {
                    float length = sqrt(((float)r.q.x - (float)r.p.x)*((float)r.q.x - (float)r.p.x)
+ ((float)r.q.y - (float)r.p.y)*((float)r.q.y - (float)r.p.y));
                    for (auto &point : points) {
                        int x = point.x;
                        int y = point.y;
                        if (SWTMatrix.at<float>(y, x) < 0) {
                            SWTMatrix.at<float>(y, x) = length;
                        } else {
                     SWTMatrix.at<float>(y, x) = min(length, SWTMatrix.at<float>(y, x));
                        }
                    }
                    r.points = points;
                    rays.push_back(r);
                }
                break;
            }
        }
    }
}

void StrokeWidthTransform::medianFilter() {
    for (auto &ray : rays) {
        for (auto &point : ray.points) {
            int x = point.x;
            int y = point.y;
            point.SWT = SWTMatrix.at<float>(y, x);
        }
        sort(ray.points.begin(), ray.points.end(), [](const SWTPoint &lhs, const SWTPoint
&rhs) -> bool {
                return lhs.SWT < rhs.SWT;
        });
        float median = ray.points[ray.points.size() / 2].SWT;
        for (auto &point : ray.points) {
            point.SWT = min(median, point.SWT);
        }
    }
}
```

Приложение 2.

Выделение компонент связности

```
void ConnectedComponents::findComponentsBoost() {
   boost::unordered_map<int, int> map;
   boost::unordered_map<int, SWTPoint> reverse_map;
   typedef boost::adjacency_list<boost::vecS, boost::vecS, boost::undirectedS> Graph;
   int num_vertices = 0;
   for (int row = 0; row < SWTMatrix.rows; row++) {
      for (int col = 0; col < SWTMatrix.cols; col++) {
         if (SWTMatrix.at<float>(row, col) > 0) {
            map[row * SWTMatrix.cols + col] = num_vertices;
            SWTPoint p(row, col, SWTMatrix.at<float>(row, col));
            reverse_map[num_vertices] = p;
            num_vertices++;
         }
      }
   }
   Graph g(num_vertices);
   for (int row = 0; row < SWTMatrix.rows; row++) {
      for (int col = 0; col < SWTMatrix.cols; col++) {
         float swt = SWTMatrix.at<float>(row, col);
         if (swt > 0) {
            int this_pixel = map[row * SWTMatrix.cols + col];
            if (col + 1 < SWTMatrix.cols) {
               float right = SWTMatrix.at<float>(row, col + 1);
               if (right > 0 && (swt / right <= 3.0 || right / swt <= 3.0))
                  add_edge(this_pixel, map.at(row * SWTMatrix.cols + col + 1), g);
            }
            if (col - 1 >= 0) {
               float left = SWTMatrix.at<float>(row , col - 1);
               if (left > 0 && (swt / left <= 3.0 || left / swt <= 3.0))
                  add_edge(this_pixel, map.at(row * SWTMatrix.cols + col - 1), g);
            }
            if (row + 1 < SWTMatrix.rows) {
               float upper = SWTMatrix.at<float>(row + 1, col);
               if (upper > 0 && (swt / upper <= 3.0 || upper / swt <= 3.0))
                  add_edge(this_pixel, map.at((row + 1) * SWTMatrix.cols + col), g);
            }
            if (row - 1 >= 0) {
               float down = SWTMatrix.at<float>(row - 1, col);
               if (down > 0 && (swt / down <= 3.0 || down / swt <= 3.0))
                  add_edge(this_pixel, map.at((row - 1) * SWTMatrix.cols + col), g);
            }
         }
      }
   }
   vector<int> c(num_vertices);
   int num_comp = boost::connected_components(g, &c[0]);
   for (int j = 0; j < num_comp; j++) {
```

```
        components.add_components();
    }
    for (int j = 0; j < num_vertices; j++) {
        Component* comp = components.mutable_components(c[j]);
        auto points = comp->mutable_points();
        auto point = points->Add();
        point->set_x(reverse_map[j].x);
        point->set_y(reverse_map[j].y);
        point->set_swt(reverse_map[j].SWT);
        comp->set_isdarkonlight(isDarkOnLight);
    }
}
```