

# Описание программной части робота-художника

Латыпов Владимир Витальевич

27 июля 2021 г.

## Содержание

1	Формулировка задачи	4
2	Описание программы в общих чертах	4
2.1	Представление «решения» — набора мазков . . . . .	4
2.2	Задание функции ошибки . . . . .	5
2.3	Растеризация мазков . . . . .	6
2.4	Учёт цвета при оптимизации . . . . .	10
2.4.1	Знание цвета при рендеринге необходимо . . . . .	10
2.4.2	Определение цвета по положению при подсчёте ФО	10
2.4.3	Фиксированный цвет в зонах . . . . .	11
2.5	Разделение картины на зоны . . . . .	11
2.5.1	Обоснование эффективности . . . . .	11
2.5.2	Прямоугольные зоны . . . . .	12
2.5.3	Зоны произвольной формы . . . . .	13
2.6	Сортировка мазков перед выпуском . . . . .	13
2.6.1	Сортировка по территориальному признаку . . . .	14
2.6.2	Оптимальная расстановка цветов . . . . .	15
3	Технические аспекты	15
3.1	Основное . . . . .	15
3.2	Библиотеки . . . . .	15
3.2.1	OpenCV . . . . .	15
3.2.2	Pythonic . . . . .	15
3.2.3	lunasvg . . . . .	16
3.2.4	PowerfulGA . . . . .	16
4	Алгоритмы оптимизации	17
4.1	Общий принцип ГА . . . . .	17
4.2	Термины . . . . .	17
4.3	Примерная последовательность действий ГА . . . . .	18
4.4	Конкретная реализация ГА и авторские модификации . .	19
4.4.1	hazing_percent: скорость сходимости . . . . .	19
5	Наблюдения	20
5.1	Неравенство зон . . . . .	20

<b>6</b>	<b>Дальнейшее развитие</b>	<b>22</b>
6.1	Внедрить быстрый пересчёт функции ошибки . . . . .	22
6.2	Разделение мазков по слоям . . . . .	22
6.3	Добавить возможность использования локальных методов оптимизации . . . . .	23
6.4	Организовать систему тестирования различных алгоритмов на различных функциях . . . . .	23
6.5	Контроль уровня разнообразия особей в ГА . . . . .	24
6.5.1	Задание метрики . . . . .	25
6.6	Улучшить алгоритм поиска цветов и разделения на зоны	26
6.7	Перенести графические вычисления на видеокарту . . . . .	26

## 1. Формулировка задачи

Для того, чтобы робот-художник нарисовал что-либо, ему нужно предоставить данные в определённом формате, а именно — не набор пикселей, как требуется для показа на мониторе, а набор «мазков»: это связано с конструкцией самого робота. Мазки решено было представлять в виде кривых Безье второго порядка (то есть квадратичных), к которым добавлены параметры «толщина» и «цвет». (сама кривая задаётся тремя точками на плоскости)

Но на вход подаются рисунки не в векторном, а в растровом формате. Найти такую комбинацию мазков, которая бы лучше всего соответствовала картине/изображению — задача нетривиальная, имеющая множество решений.

Конечно же, я выбрал решать задачу самостоятельно, а не использовать готовые библиотеки.

## 2. Описание программы в общих чертах

Таким образом, решено было использовать эвристические алгоритмы оптимизации: Генетический алгоритм и Симуляция отжига. Подробное описание алгоритмов, моей их реализации и улучшений представлено в секции 4.

### 2.1. Представление «решения» — набора мазков

Программа работает с последовательностями мазков, находя лучшую из них. Однако, чтобы алгоритм оптимизации работал с ними, наборы мазков должны быть представлены в виде векторов в  $\mathbb{R}^n$ .

Каждый мазок является кривой Безье второго порядка и задаётся с помощью семи параметров.

$$\begin{cases} x(t) = p_{1x} + (1-t)^2 \cdot (p_0 - p_1)_x + t^2 \cdot (p_2 - p_1)_x \\ y(t) = p_{1y} + (1-t)^2 \cdot (p_0 - p_1)_y + t^2 \cdot (p_2 - p_1)_y \end{cases} \quad (1)$$

Формат данных в «геноме» таков:

, где  $p_{nx} \& p_{ny}, n \in \{0, 1, 2\}$  — координаты направляющей точки под номером n (из всего 3 у каждого мазка), а width — толщина мазка.

Здесь нет параметра «цвет». Причина объяснена в разделе: 2.4.

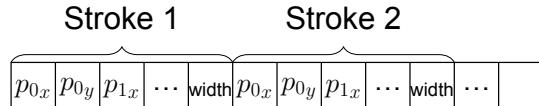


Рис. 1: Схема хранения генома

## 2.2. Задание функции ошибки

Чтобы решить задачу алгоритмом оптимизации, нужна некая метрика — функция, которая будет определять степень «неподходящести» данного ей решения. Именно она будет передаваться алгоритму оптимизации. В нашем случае вычисление функции ошибки (далее — ФО) включает в себя растеризацию мазков (отображение их на изображении) и вычисления, производящие сравнения полученного результата с желаемым. ФО необходимо задать таким образом, чтобы она отражала качество полученной комбинации мазков, причём в любой точке направление её уменьшения соответствовало направлению улучшения результата. За основу была выбрана MSE (Mean Square Error):

$$MSE = \frac{1}{width \cdot height} \cdot \sum_{y=0}^{height} \sum_{x=0}^{width} \sum_{c \in \{r,g,b\}} \left( \overrightarrow{\text{rendered}_{x,y,c}} - \overrightarrow{\text{original}_{x,y,c}} \right)^2 \quad (2)$$

MSE — универсальная мерилика для схожести изображений, она повсеместно используется при работе с ними. Но в нашем случае, о чём свидетельствует практика, целесообразно добавить в ФО компоненту, «наказывающую» за наложение мазков друг на друга, а также за пустые (ничем не закрашенные) места.

Первое улучшение очевидно — при прочих равных лучше ситуация, при которой та же картина достигнута с меньшим использованием краски (а если не вводить эту компоненту, наложено в найденном решении может быть сразу много ( $> 2$ ) мазков в одной точке). Если нет разницы, зачем переплачивать?

С теоретической точки зрения может быть непонятна надбавка за пустоты: ведь если место пустое, оно и так не даёт оптимальное MSE. Однако на практике пустоты недостаточно быстро и полно покрываются (особенно — на ускоренном режиме) без этой надбавки.

Таким образом, функция ошибки сделана так, чтобы максимально стимулировать правильно распределение мазков.

### 2.3. Растеризация мазков

Имея мазок, заданный в виде трёх точек на плоскости, толщины и цвета, нужно уметь его отобразить на «холсте», то есть в виде набора пикселей. Это нужно, чтобы подсчитать функцию ошибки для заданного набора мазков, причём так, чтобы результат максимально соответствовал мазку, рисуемому роботом. В качестве достаточно точной модели описания такого мазка возьмём круглую кисть, перемещающуюся по заданной траектории. Есть много способов произвести растеризацию. Нужно выбрать тот, который будет производительным и в то же время максимально близким к реальному мазку.

Самый простой — для некоторого количества точек на кривой Безье (с достаточно маленьким шагом, примерно один пиксель) проводим вертикальную линию: вверх на `width` и вниз — тоже. Это даёт высокую производительность и сносно выглядит на участках, близких к горизонтальным, но результат, полученный таким способом, очень далёк от реальности на вертикальных участках:

Есть разные способы избавиться от этих недостатков, сохраняя максимальную производительность. Например:

- Совмещать горизонтальные и вертикальные полосы
- Проводить полосы перпендикулярно направлению кривой в данной точке

В каждом из них будут наблюдаться пустые места, полости, что недопустимо.

Ультимативным же способом является подражание реальной жизни: «проведение» круглой «кистью» по экрану. То есть берутся точки на кривой на небольшом расстоянии друг от друга, далее из каждой рисуется круг радиусом `width`. Однако в таком случае каждая точка, попадающая в мазок, обрабатывается много раз (для близких кругов), что значительно замедляет рендеринг. Если же увеличить шаг, этой проблемы можно частично избежать, но мазок стал бы неровным. При маленьком шаге это выглядит так:

В будущем планируется улучшить алгоритм для ускорения растеризации при почти том же качестве. Рассматриваются варианты:

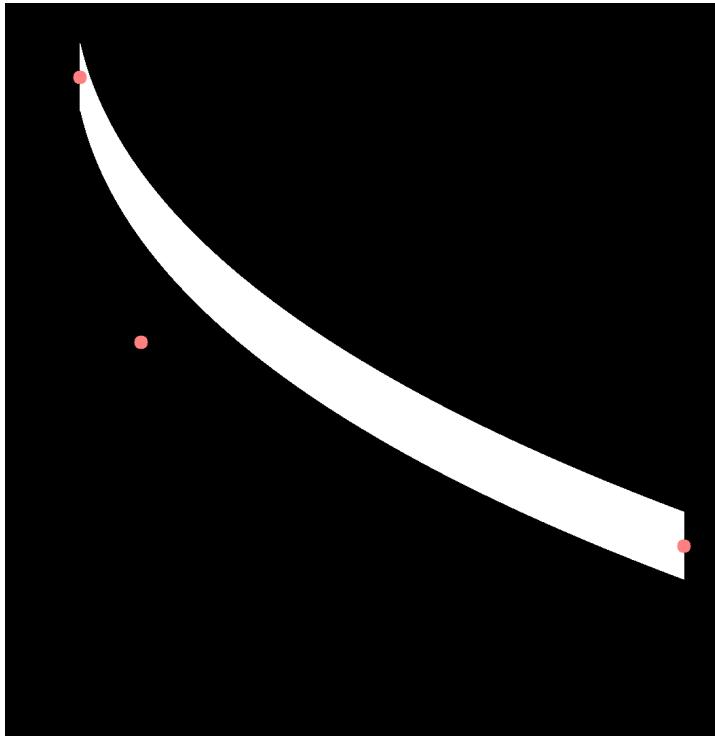


Рис. 2: (Красным обозначены точки, задающие кривую)

- Заменить круглую кисть на также гладкую, но с более медленным закруглением с дальней от вектора кривой в данной точке стороны, поворачивая кисть соответствующим образом: Такое изменение поможет уменьшить артефакты при увеличении шага между точками на кривой, то есть позволит сделать шаг больше, ускорив процесс.
- Автоматически разбивать мазок на «полигоны». Для этого нужно пройтись по кривой и с некоторым шагом (уже побольше, чем раньше), отметить для каждой рассматриваемой точки на прямой, содержащей её и перпендикулярной текущему направлению, точки в обе стороны от неё на расстоянии  $width$ . Каждая из них добавляется в соответствующий список в порядке обхода списка. Потом полигоны, полученные из соседних точек на кривой и соответствующим им вынесенным точкам, заливаются нужным цветом. На концах же мазка рендерятся круги.

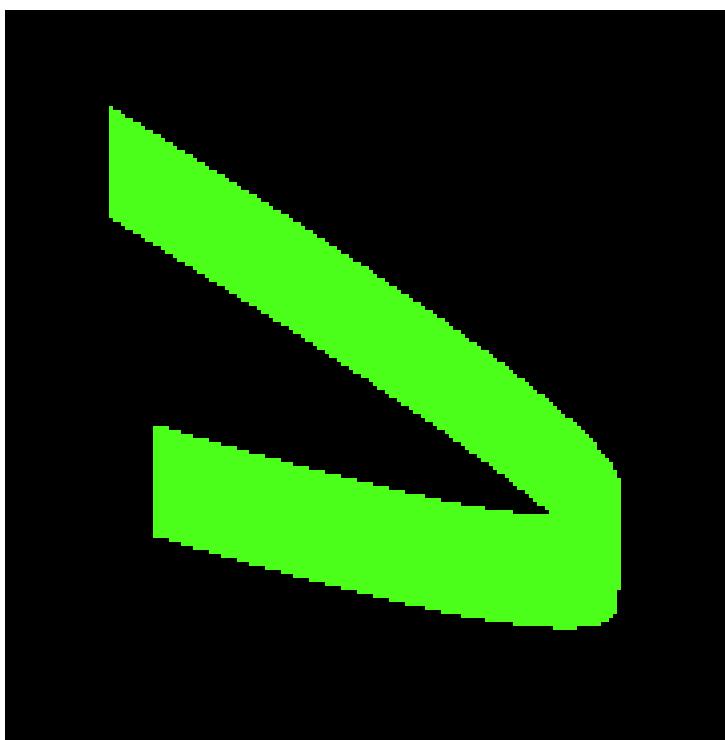
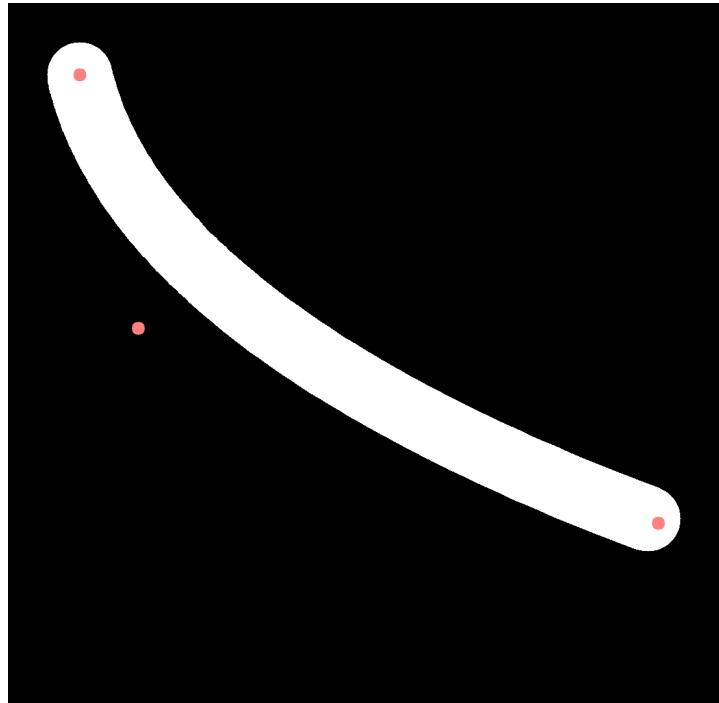


Рис. 3: Иногда такой способ добавляет свой шарм



При использовании этого метода никакая существенная часть пикселей мазка не обрабатывается много раз, что говорит о высокой эффективности алгоритма. Поэтому я собираюсь внедрить такой метод в ближайшее время.

Что же касается совместимости с видеокартой, для круга и модифицированной кисти понятно, как определить *bouding-box*, и понятно, как по координатам пикселя быстро определить, принадлежит ли он этому примитиву. А рендеринг полигонов производится аппаратно.

Подробнее про внедрение видеокарты можно почитать в разделе 6.7.

#### 2.4. Проведение операций ГА

Задачу можно представить чисто математически — как оптимизацию функции  $\mathbb{R}^n \leftarrow \mathbb{R}$ . Несмотря на то, что в пределе такой подход приведёт к некоторому результату,

- Это займёт очень много времени

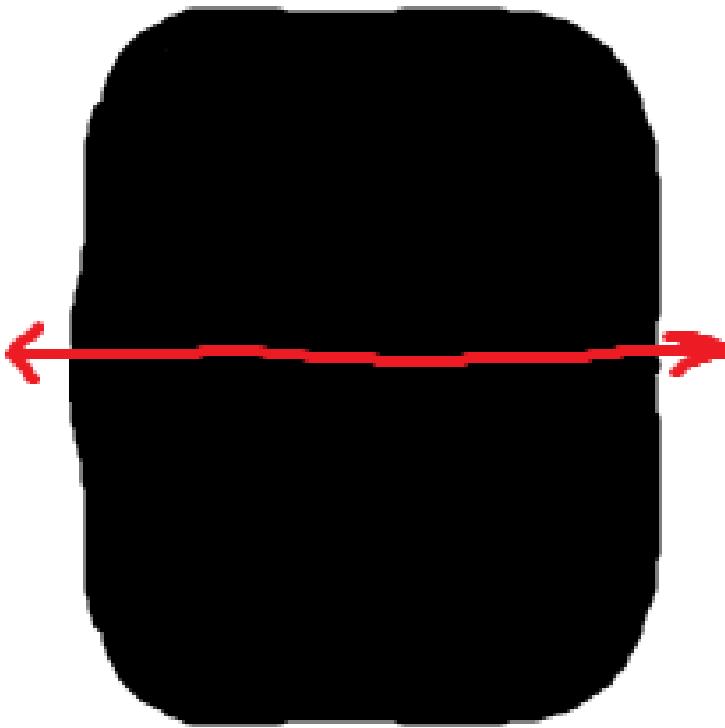


Рис. 4: (красным обозначено направление вектора кривой)

- Результат не будет соответствовать некоторым критериям из реальной жизни

Операции, по умолчанию производящиеся в ГА, несколько изменены и адаптированы для конкретной задачи.

Важная проблема решения «в лоб» (которую нужно устраниить именно так) — то, что, если разрешить в качестве области поиска по каждой координате указать весь диапазон изображения по этой координате, типичный мазок будет растянут на всю картину. Но, во-первых, настоящие мазки — совсем не такие — они существенно меньше изображения. Во-вторых, понятно, что алгоритму будет несравненно более сложно найти нужную комбинацию мазков. Например, потому, что она, скорее всего, будет предполагать малый размер мазков, который, конечно, возможен при такой постановке задачи, однако алгоритму потребуется

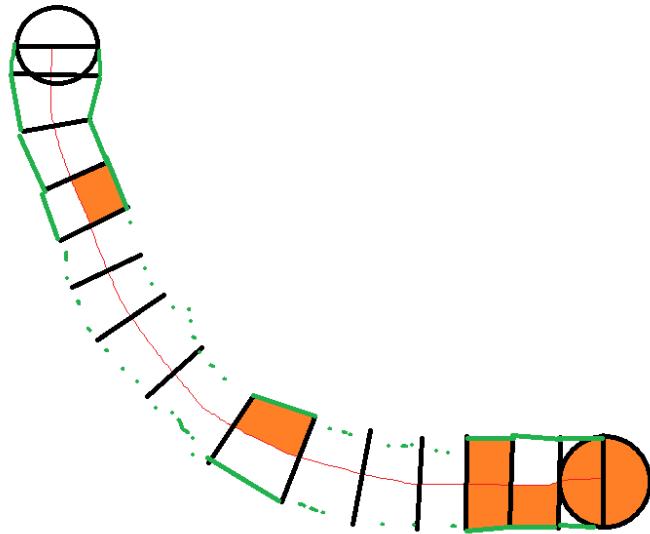


Рис. 5: Схема полигональной разбивки мазка

огромное количество ресурсов, чтобы достичь его.

Поэтому рассмотрение координат разных точек как совершенно независимых параметров — плохая идея. Я решил составить набор ограничений на разные параметры мазков:

- Минимальные и максимальные высота и ширина bounding\_box-а мазка
- Минимальные и максимальные толщина и длина мазка
- Всписываемость в изображение
- Искривлённость

Для каждого параметра необходимо уметь:

- Определять, правда ли, что заданный мазок подходит по ограничения
- «Подправлять» мазок так, чтобы значения параметров пришли в норму
- Случайно генерировать набор мазков, чтобы все соответствовали критериям

Насчёт второго пункта важно отметить: мутации точек по амплитуде — в среднем заметно меньше размера самих мазков, поэтому от методов корректировки не нужно очень высокое качество (например, какие-то гарантии сохранения формы или центра или чего-то ещё): неидеальности корректировки — тоже часть мутаций — небольших случайных изменений.

В случае с координатами по каждой отражаются от соответствующей границы изображения, но со случайнym коэффициентом  $k \in [0; 0.5]$ . То есть новое место находится на перпендикуляре к краю изображению, проходящем через изначальную точку, и удалено от границы на  $k \cdot d_0$ , где  $d_0$  — расстояние от границы до изначальной точки. Это делается, чтобы не наблюдалось скопление мазков по краям. Затем (для подстраховки) по каждой координате каждой точки просто делается обрезка по диапазону  $\{0, \max_{coord}\}$ .

В случае с размерами `bounding_box`-а точки, если что-то не так для какой-то из координат, происходит `scaling` точек мазка по этой координате так, чтобы он соответствовал ближайшему разрешённому значению.

Тощина просто обрезается по разрешённым границам.

Длина просто считается, и, если вдруг она слишком велика или мала (хотя это маловероятно, если подходит под `bounding box`, а если и так и происходит, то превышение или преуменьшение всё равно незначительное) Длина кривой определяется классическим образом, по этой формуле:

$$L = \int_a^b \sqrt{x'^2(t) + y'^2(t)} dt \quad (3)$$

Применение этого к кривым Безье можно найти здесь: <https://members.loria.fr/samuel.hornus/quadratic-arc-length.html>

Пройдёмся по операциям ГА:

#### 2.4.1. Первоначальная генерация популяции

Как было сказано выше, здесь не просто генерируются координаты задающих мазок точек по отдельности, а сначала случайно выбираются точки в прямоугольнике с размером чуть больше, чем разрешённый размер `bounding_box`-а, происходит смещение в случайное место картинки, а в конце — используется корректирующая функция.

### 2.4.2. Мутация и корректировка

Здесь сначала делается небольшое случайное изменение координат, а потом — корректировка по указанным правилам.

### 2.4.3. Скрешивание

У новой особи просто-напросто берётся часть мазков у одного родителя, а часть — у другого.

Практика показала, что проведение этой модернизации существенно повысило качество «продукта».

## 2.5. Учёт цвета при оптимизации

### 2.5.1. Знание цвета при рендеринге необходимо

Цвет является обязательным параметром мазка, без которого непонятно, как его растеризовать. Но значит ли это, что цвет обязательно должен быть одним из параметров алгоритма оптимизации? Конечно же, нет.

### 2.5.2. Определение цвета по положению при подсчёте ФО

Во-первых, цвет может быть легко определён, зная положение. Самый простой способ — найти среднее арифметическое цветов пикселей. Мы автоматически получим минимальное MSE, так как для каждой из цветовых компонент соответствующая часть MSE — сумма квадратичных функций (для каждого пикселя), причём «с ветвями вверх» (так как на бесконечностях она уходит в  $+\infty$ ), которая имеет одну точку с нулевой производной — как раз в среднем арифметическом:

$$MSE_c(value_c) = \sum_{pixel \in pixels} (value_c - pixel_c)^2 \Rightarrow \quad (4)$$

$$MSE'_c(value_c) = 2 \cdot \left( \|pixels\| \cdot value_c - \sum_{pixel \in pixels} pixel_c \right)$$

, где  $pixel_c$  — значение канала  $c$  пикселя  $pixel$ .

То есть у MSE достигается производная ноль в этой точке:

$$MSE'_c(value_c) = 0 \iff value_c = \frac{\sum_{pixel \in pixels} pixel_c}{\|pixels\|} = \overline{pixel} \quad (5)$$

Следовательно, больше нигде ноль не достигается, так как функция квадратичная. Более того, это минимум, так как функция — «ветви вверх».

### 2.5.3. Фиксированный цвет в зонах

Во-вторых, при делении на зоны все пиксели, относящиеся к данной зоне, имеют строго заданный цвет. Подробнее об этом читать в секции 2.5.

## 2.6. Разделение картины на зоны

### 2.6.1. Обоснование эффективности

Известно, что время оптимизации до заданного качества очень сильно увеличивается при увеличении количества параметров. Причём существенно более быстро, чем линейно. Следовательно, можно получить выгоду, разделяя эти параметры каким-то образом и оптимизируя группы по отдельности.

$$F(\|parameters\|) \gg n \times F\left(\frac{\|parameters\|}{n}\right) \quad (6)$$

Вопрос в том, в каких случаях это делать можно (то есть в каких случаях качество оптимизации заметно не уменьшится при разбиении), а в каких — нет.

Надо понимать, что нельзя независимо оптимизировать близкие мазки: только их комбинация позволит понять, хорошо ли предпринятое изменение для каждого и для всех в целом. Находить положение одного, не зная положение другого, почти бесполезно. Однако, если мазки находятся далеко друг от друга, можно безболезненно произвести деление.

Пояснение: Применительно к данной задаче супераддитивность времени выполнения алгоритма по отношению к размерности входного пространства для функций в целом можно описать так: если мутация затрагивает большое количество мазков из разных сторон изображения, «картина» для функции ошибки очень зашумлена: если ФО увеличилась, определить, следствием какой именно части мутаций было то

или иное изменение ФО, можно только очень «некачественно» (то есть с малой вероятностью). А следовательно, мутации будут хаотично приниматься и наслаиваться, но это не обеспечит устойчивого движение к оптимуму через комбинирование правильных мутаций. Причём выбор маленького количества трансформаций за мутацию — не выход: нужно именно не такое малое количество изменений за один раз, причём таких, чтобы они были рядом, то есть сильно влияли друг на друга. А совсем малое количество изменений за мутацию — плохо, так как эти переменные сильно зависимы друг от друга, оптимизация, например, сначала по одной, потом по другой и т.д. приведёт не к нахождению глобального оптимума, а к застреванию в локальном. Чтобы выйти из него, нужно изменение сразу большого количества переменных. Конечно, ГА — это не Hill climbing algorithm, он не «отрезает» сразу любое изменение, не приведшее к результату, но вышеупомянутые механизмы всё равно привозят к уменьшению производительности при слишком маленьком количестве изменений за одну мутацию.

Поэтому (так как важно немалое количество изменений, сконцентрированных в одном месте, чтобы избежать попадания в локальный оптимум, — с одной стороны и отсутствие сбивания рыночных целевофункциональных сигналов другими частями картины), важно как-либо разделять изображения на зоны: функция ошибки сепарабельна, но только до какого-то размера зоны, при малом количестве мазков, близких друг другу, это перестаёт работать.

### 2.6.2. Прямоугольные зоны

Проще всего в реализации — делить изображение на прямоугольные зоны. Каждая рассматривается как отдельное изображение, для него находятся мазки, которые потом со сдвигом добавляются «в общий котёл».

Встаёт очевидный вопрос: как избежать заметности границ этих зон на финальном изображении?

Первое средство — расчерчивать эти границы наложенными друг на друга. То, какую часть перекрытие должно составлять от всей зоны, можно настроить эмпирически, запуская программу с разными значениями этого параметра и сравнивая плотность в местах стыка и на остальной картине. Подобранное значение почти универсально, то есть может быть применено и для обработки других картин.

Второе средство — после получения результатов для зон «разблокировать положение» мазков, лежащих рядом со стыками, запустив оп-

тимизацию для них уже на основе данных целой картины. Конечно, и здесь имеет смысл отдельно оптимизировать «крести», образующиеся при стыке четырёх зон. Также можно в качестве ограничений использовать не весь *box* картины, а зоны этих крестов. Более того, если окажется, что плотности в местах стыка будет не хватать, можно добавить некоторое количество новых мазков (т.н. kleящего вещества).

Последние улучшения не были внедрены, так как ↓↓↓

### 2.6.3. Зоны произвольной формы

Понятно, что лучшее качество даёт деление на зоны, связанное со структурой картинки. То есть цвет должен не сильно меняться в пределах зоны. Вопрос в том, как изображение на такие зоны поделить.

Такие программы, как Adobe Illustrator, умеют эту выполнять эту операцию (правда, с некоторыми проблемами, о них — позже).

## 2.7. Регуляция кривизны мазков

Если не регулировать кривизну мазков дополнительно, можно заметить, что

### 2.8. Сортировка мазков перед выпуском

В случае растеризации набора мазков на компьютере время выполнения операции почти совсем не зависит от порядка следования мазков (если не считать незаметное влияние кэширования в процессоре), так как закрашивание пикселя происходит через *random access*. Однако в реальности это далеко не так: правильный порядок мазков (как по координатам, так и по цветам) может сильно уменьшить время отрисовки, которое весьма велико, и поэтому нельзя забывать про задачу правильного расположения мазков.

#### 2.8.1. Сортировка по территориальному признаку

В первом приближении количество цветов очень небольшое (скорее всего, не больше десяти), а, чтобы сменить цвет, нужно вести кисть к банке с водой, смывать краску предыдущего цвета, брать новый, а потом опять нести кисть через весь стол. Поэтому разумно сначала разделить мазки на группы по цветам, а в каждой из них располагать их так, чтобы минимизировать пройденное кистью расстояние при последовательной отрисовке мазков этого цвета.

По сути это модификация задачи Коммивояжёра, где в роли городов выступают мазки. Разница в том, что мазки гораздо более некорректно считать материальными точками: их размер — порядка расстояний между ними. То есть «вход» в мазок находится с одной стороны, а «выход» — с другой, причём для каждого мазка появляется два варианта его проведения — с одной стороны и с другой.

Конечно, можно оформить нахождение лучшей последовательности как оптимизационную задачу, где параметрами будут:

- Некая перестановка идентификаторов всех мазков
- Вектор бинарных величин для каждого мазка (с какой стороны начинать его рисование)

Но это будет неоправданно ресурсозатратно, так как от решения этой задачи зависит только скорость рисования, но не качество конечного продукта.

В качестве простого и быстрого алгоритма напрашивается обход «змейкой»: сортировать мазки сначала по одной координате, затем — по другой (в качестве якорной точки разумно использовать точку  $\vec{r} = bezierCurve(t = 0.5)$ ). Однако на практике первая координата примерно никогда не совпадает у нескольких мазков, поэтому сортировка по сути происходит только по ней, что приводит к постоянному метанию с одной стороны холста на другую.

Потому разумнее разделить изображение на прямоугольные зоны такого размера, чтобы количество зон было порядка количества мазков в зоне, тогда делений по одной координате:  $\approx \sqrt[4]{\|strokes\|}$ . Далее можно произвольно выбрать порядок мазков внутри каждой зоны: будет гарантироваться, что между каждыми мазками будет пройдено расстояние  $\leq \sqrt{\frac{image\_height^2}{n\_separators} + \frac{image\_width^2}{n\_separators}}$ .

Как вариант — запустить алгоритм оптимизации внутри каждой зоны: здесь, как проверено на похожих данных в репозитории PowerfulGA возможно почти моментально получить хороший, скорее всего — идеальный — результат.

Сейчас используется разбиение на зоны с дальнейшей сортировкой по одной из координат.

Такой способ даёт удовлетворительное качество, однако рассматривается использование алгоритма Кристофидеса — возможно, он даёт более хорошее качество (гарантирует длину пути не более, чем в 1.5 раз больше минимального).

## 2.8.2. Оптимальная расстановка цветов

### 3. Технические аспекты

#### 3.1. Основное

Программа написана на языке программирования C++ (так как требовалась максимальная скорость), сборка осуществляется с помощью CMake. Проект можно скомпилировать под Windows (компилятор MSVC) и под Linux (тестировалось на g++-10).

Код хранится в [github](#)-репозитории (кликально).

#### 3.2. Библиотеки

##### 3.2.1. OpenCV

Для работы с изображениями используется OpenCV, но не модуль машинного обучения, а лишь примитивные операции с изображениями: чтение из популярных форматов, сохранение в них, хранение и копирование матрицы пикселей и т.д.

##### 3.2.2. Pythonic

Для работы проекта также необходима библиотека «`pythonic`»: она написана мной, подключается также через CMake. Она отвечает за базовые функции и структуры данных. Я использую её во всех более или менее крупных проектах на C++. В ней на данный момент есть:

- Простые вспомогательные функции для работы со строками, контейнерами, форматированного вывода
- Вызов питоновской библиотеки `matplotlib` для построения графиков
- Базовые алгоритмы наподобие бинарного поиска и дерева отрезков
- Функционал для работы со временем, в том числе — анализатор последовательных запусков процесса
- Платформонезависимая работа с кодировками и файловыми системами
- Примитивы для вычислительной геометрии

- Функции для работы со статистикой
- Многомерный шаблонный массив с количеством измерений, изменяемом в run-time
- Сглаживание функций и построение примерной функции распределения в пространстве с заданной размерностью по набору `sampler`-ов с помощью гауссовых ворот
- Функционал для работы с многопоточностью, в том числе — `thread pool`, умеющий снимать нагрузку с ожидающих потоков с помощью `std::condition_variable`.

### 3.2.3. `lunasvg`

Для работы с SVG используется библиотека `lunasvg`.

Р. С. У этой библиотеки отличный автор, он изучает проекты, в которых библиотека используется, и пишет рекомендации о best practice её использования.

### 3.2.4. `PowerfulGA`

Функционал по методам оптимизации реализован мной и вынесен в отдельную репозиторию: `click` (там не только Генетический алгоритм, как можно было подумать из названия, но и симуляция отжига, градиентный спуск, метод Ньютона; планируется добавить много других алгоритмов) Более подробное описание в секции → 4

## 4. Алгоритмы оптимизации

### 4.1. Общий принцип ГА

Идея работы генетического алгоритма заимствована у природы: также, как в ходе эволюции, происходит появление оптимального организма для заданных условий — в ходе работы алгоритма ищется набор параметров, при котором фитнес-функция максимальна.

В природе тот, кто лучше приспособлен к окружающей среде, в большей степени получает доступ к размножению, в результате чего новые особи получают признаки от лучших родителей. Из-за того, что геном наследуется от обоих родителей, получившиеся комбинации могут повторно комбинировать в себе черты, что даёт большое преимущество

перед бесполым размножением (аналогом которого являются такие методы оптимизации, как градиентный спуск или отжиг). Нужно пояснить, что половое размножение не обязательно предполагает наличие нескольких полов. Особи в ГА — аналоги гермафродитов из живой природы: в отличие от людей, каждая особь может скрещиваться с каждой.

Это позволяет именно лучшим чертам, по каким-либо причинам появившимся у особей, переходить в следующее поколение.

За их появление отвечают мутации — небольшие случайные изменения в геноме.

Из принципа работы можно понять, что алгоритм эвристический: сложно доказать его сходимость или что-либо гарантировать с вероятностью 100%. Зато исследования показывают, что именно этот алгоритм даёт лучшие результаты для самых сложных функций. В разделе 4.4.1, какие меры предпринимаются, чтобы не дать алгоритму попасть в локальный минимум, не добравшись до глобального.

## 4.2. Термины

Набор параметров представляется в виде «генома» — некой структуры данных, содержащей информацию об этом наборе. Особь — в контексте алгоритма будет использоваться в качестве синонима к геному.

Геном состоит из генов — каждый из них содержит информацию о каком-либо признаке (в случае природы) или параметре (в случае ГА).

В каждый момент времени алгоритм работает с популяцией — набором геномов. Это аналог популяции в природе.

Мутация — как и в реальной жизни — небольшое случайное изменение генома без строго определённого направления.

## 4.3. Примерная последовательность действий ГА

В общих чертах работа ГА выглядит так:

Инициализация: Сгенерировать случайную популяцию, каждый ген каждого генома — в заданных пределах.

Затем — повторять, пока не закончится заданное количество итераций или не будет достигнуто требуемое значение фитнес-функции:

1. Посчитать фитнес-функции для каждой из особей. Для большинства задач этот шаг занимает бо́льшую часть времени исполнения, поэтому нужно оптимизировать именно его, в частности — параллелизовать, запуская независимые вычисления на нескольких потоках.

2. Каким-либо образом отобрать особи на скрещивание
3. Произвести скрещивание, получив «отпрысков» — часть нового поколения
4. Сформировать новое поколение, используя, возможно, в разных пропорциях, различные источники геномов, а именно:
  - Отпрысков, полученных на предыдущем шаге в результате скрещивания
  - Лучшие особи из прошлой популяции
  - Случайные особи — чтобы не дать алгоритму сойтись раньше времени, попав в локальный минимум
  - Возможно, результаты скрещивания одновременно более двух особей.
5. Произвести мутации в некоторых особях этого поколения (лучшие из мутаций внедряются в популяцию на следующих итерациях)
6. Опционально — «обрезать» (то есть насильственно подправить) те мутации, которые привели к выходу каких-либо параметров или их комбинаций за пределы допустимого.
7. Если алгоритм подходит к концу, добавить лучший геном из предыдущего поколения (чтобы он не подвергся мутации)

#### 4.4. Конкретная реализация ГА и авторские модификации

Учитывая тот факт, что в большинстве задач, решаемых мною, бо́льшая часть вычислительного времени ( $\gg 95\%$ ) используется для подсчёта функции ошибки, а не для манипуляций с геномами (это подтверждается результатами профайлинга), задача состоит в том, чтобы минимизировать количество подсчётов функции ошибки, пусть и ценою более долгой работы с геномами.

Первое изменение — отказ от дискретного кодирования геномов.

Для алгоритма по каждой переменной задан её диапазон.

Традиционный подход — разделить диапазон на  $2^N$  частей и кодировать номер части в геноме как битовую последовательность из  $N$  бит. Для подсчёта функции ошибки этот номер перекодируется назад

в соответствующую точку непрерывной величины. Мутацией в данном случае является изменение случайного количества каких-то битов этого номера.

А скрещивание обычно происходит путём случайного выбора значений каждого из двоичных разрядов в родительских родительских геномах.

Предварительное тестирование показало бо́льшую эффективность хранения самого числа (без кодирования) по сравнению с традиционным подходом (хранение битовой последовательности), однако планируется провести более тщательное тестирование (см. 6.4)

#### 4.4.1. `hazing_percent`: скорость сходимости

Соотношение элит баланс между скоростью сходимости и избежанием застревания в локальном минимуме. Нужно исследовать всю область поиска, для этого нужно не давать сразу огромный бонус при размножении и переходе в другое поколение за некоторое преимущество. Так получится обеспечить развитие нескольких «очагов», внутри которых и будет происходить «шлифование» «идей искать в этой области».

Алгоритм реализован мной на языке C++, он хранится в GitHub репозитории <https://github.com/donRumata03/PowerfulGA>.

## 5. Наблюдения

### 5.1. Неравенство зон

Когда я заметил, что зоны, на которые Adobe Illustrator делит изображение, могут очень сильно отличаться в размере (отношение площадей может достигать 1000 раз), мне захотелось измерить это неравенство численно, чтобы при разработке нового алгоритма измерять его качество в том числе по этому параметру.

Существует большое количество метрик, я решил выбрать основные из них:

- Индекс Джини (вместе с кумулятивным графиком распределения дохода (также известен как кривая Лоренца))
- Процент «дохода» 1% самых богатых от общего «дохода» (В случае зон вместо дохода используется занимаемая площадь).

- Процент самых богатых, имеющих в сумме 50 % от общего дохода.

Индекс Джини рассчитывается как отношение площади между кривой Лоренца и «линией равенства» к площади под линией равенства. Иными словами,  $G = \frac{A}{A+B}$  на этой схеме:

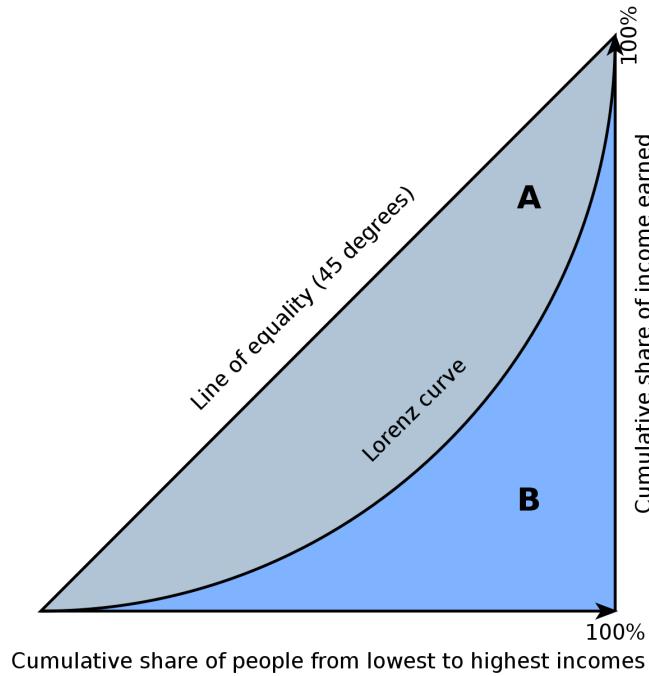


Рис. 6: Типичная кривая Лоренца

Альтернативный способ посчитать коэффициент, использующийся в реализации:

$$G = \frac{\sum_{i=1}^n \sum_{j=i+1}^n |y_i - y_j|}{n \cdot \sum_{i=1}^n y_i} \quad (7)$$

Чем индекс выше, тем большее неравенство наблюдается в стране. Более того, использование именно этой метрики позволяет комплексно оценить неравенство между анализируемыми объектами — в отличие от рассмотрения процентов дохода заданного квантиля.

Результаты оказались впечатляющими:

- $Gini\_index \approx 76\%$
- 1% крупнейших зон покрывают ≈ 18% изображения

- 6.25% зон покрывают половину изображения

Так выглядит кумулятивный график распределения площади:

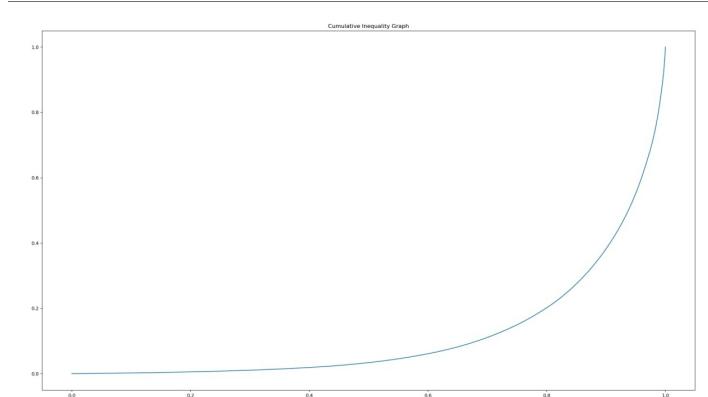


Рис. 7: Кривая лоренца для зон

Нетрудно заметить, что ни в одной стране мира нет такого неравенства, как среди зон:

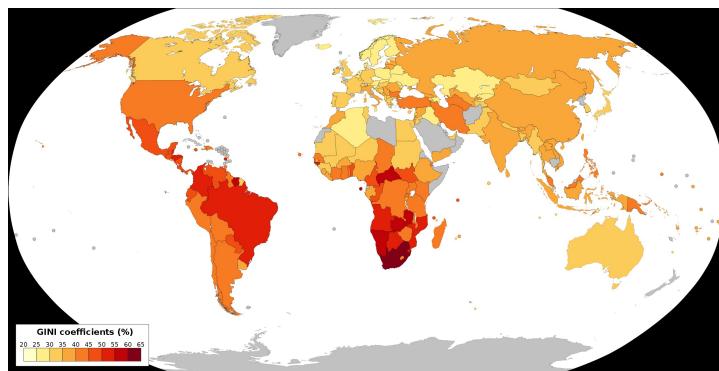


Рис. 8: Индекс Джинни по странам мира

Даже в ЮАР индекс Джинни составляет 57.8%.

## 6. Дальнейшее развитие

Несмотря на то, что программа уже работоспособна, есть ещё много идей и планов по её усовершенствованию:

## 6.1. Внедрить быстрый пересчёт функции ошибки

Это улучшение давно напрашивается, но оно несколько теряет в эффективности из-за того, что в одной мутации в среднем изменяется не так мало мазков (однако это количество убывает со временем). В настоящий момент ведётся работа над внедрением.

## 6.2. Разделение мазков по слоям

Нетрудно заметить, что при рисовании картин художники сначала проходятся по холсту черновыми мазками большого размера, а затем — прорабатывают детали. Таких уровней детализации зачастую бывает немало.

Пример того, как художник (<https://www.youtube.com/watch?v=VaXHtai2alU>) рисует картину по слоям:



Рис. 9: Фон | Рельеф фона | Детализация заднего плана | Основные объекты

Поэтому стоит попробовать сначала заполнять картинку толстыми, грубыми мазками (то есть просто с большей шириной, а в реальной жизни это будет отражаться в большем размере кисти и в более сильном нажатии).

## 6.3. Добавить возможность использования локальных методов оптимизации

Такие методы, как градиентный спуск и метод Ньютона позволяют достичь гораздо большей скорости сходимости (в случае метода Ньютона — сходимость квадратичная), но требуют умения посчитать градиент функции ошибки в любой точке, а также вектор вторых производных по каждому из аргументов.

Сами алгоритмы реализованы и находятся в этой папке. Предусмотрена опция подсчёта первой и вторых производных через подстановку близких значений параметров:

$$f'(x_0) \approx \frac{f(x_0 + \Delta x) - f(x_0)}{\Delta x} \quad (8)$$

Однако в случае с мазками при маленьких изменениях параметров функция ошибки остаётся неизменной, так как это приводит к тому же набору закрашенных пикселей. Соответственно, нужно либо радикально увеличивать разрешение изображения, либо использовать аналитические методы. То есть нужно математически посчитать изменение функции ошибки при бесконечно малом изменении из параметров функции.

#### 6.4. Организовать систему тестирования различных алгоритмов на различных функциях

Звучит как нечто весьма простое, но реальность сложнее, чем кажется. Напрашивающийся вариант — дать каждому алгоритму заданное количество вычислений функции ошибки и сравнить, какой результат они получат.

Однако функция ошибки нелинейная, поэтому сложно будет понять, насколько сильному различию в качестве алгоритма соответствует полученная численная разница в результатах.

Целесообразно сравнивать количество итераций, требующееся алгоритмам для получения заданного результата. Но и тут не всё так просто: нельзя просто запустить алгоритмы на неограниченное количество итераций и ждать достижения нужного значения функции, так как во многих из них (как минимум — в моей модификации ГА) то, как будет проведена каждая отдельная итерация, сильно зависит от процента выполнения на момент её прохождения: происходит планирование, использующее информацию о максимальном количестве итераций.

Поэтому нет никакого другого выхода, кроме того, чтобы запускать этот алгоритм с разным количеством итераций и смотреть, когда он в среднем будет доходить до заданного порога. Это необходимо автоматизировать.

В идеальном случае для поиска порога можно было бы использовать бинарный поиск, но в реальности (с поправкой на шум) имеет смысл использовать эвристическую модификацию н-арного поиска (объяснить!). Для полной оценки планируется построить график достаточного количества итераций от требуемого значения функции в интересующей нас зоне.

Это нужно проделать для нескольких сложных функций (например, из списка в Википедии [https://en.wikipedia.org/wiki/Test\\_functions\\_for\\_optimization](https://en.wikipedia.org/wiki/Test_functions_for_optimization)) и на основе этого сделать вывод об общем качестве работы.

Умение хорошо оценивать работу алгоритма «в полной комплектации» даёт возможность оптимизировать гиперпараметры. То есть мы запускаем мета-алгоритм, параметры которого — это гиперпараметры основного алгоритма, а функция ошибки — качество его работы. В случае с ГА, например, гиперпараметрами могут быть:  $hazing_{percent}$ ,  $elite/hyperelite$ ,  $fit_{pow}$ ,  $epoch_{pow}$ ,

#### 6.5. Контроль уровня разнообразия особей в ГА

Известно (4.4.1), что в ГА важен баланс между скоростью сходимости и разнообразием в популяции. Второе важно, чтобы преждевременно не попасть в локальный оптимум, а получше «исследовать» всю часть пространства, отведённую для поиска.

Есть много механизмов для изменения баланса (они также описаны в 4.4.1). Однако сейчас эти механизмы действуют вслепую — по заранее заготовленному плану, не зависящему от текущей ситуации в популяции. Нужно перейти от задания действий перед запуском программы к заданию требуемых результатов. А именно — нужно:

1. Задать метрику разнообразия, устойчивую к помехам и выдающую данные в человекочитаемом формате (чтобы задавать её зависимость).
  2. Научиться использовать регуляторные средства ( 4.4.1) для перевода популяции к заданному значению метрики.
  3. На основе экспериментов определить, какой должна быть зависимость требуемого значения метрики от процента выполнения.

### 6.5.1. Задание метрики

Важно, что она должна быть сравнима при подсчёте для разных функций и областей, желательно — находясь всё время в том же интервале, например,  $\in [0; 1]$ .

В простом случае можно считать метрику как отношение  $n$ -мерного «объёма» выпуклой оболочки к  $n$ -мерному «объёму» всего пространства поиска. (алгоритм нахождения MBO в  $n$ -мерном пространстве: [https://neerc.ifmo.ru/wiki/index.php?title=%D0%92%D1%8B%D0%BF%D1%83%D0%BA%D0%BB%D0%B0%D1%8F\\_%D0%BE%D0%B1%D0%BE%D0%BB%D0%BE%D1%87%D0%BA%D0%BB%D0%BC%D0%BD%D0%BE%D1%8F](https://neerc.ifmo.ru/wiki/index.php?title=%D0%92%D1%8B%D0%BF%D1%83%D0%BA%D0%BB%D0%B0%D1%8F_%D0%BE%D0%B1%D0%BE%D0%BB%D0%BE%D1%87%D0%BA%D0%BB%D0%BC%D0%BD%D0%BE%D1%8F)

D0%BO\_%D0%B2\_n-%D0%BC%D0%B5%D1%80%D0%BD%D0%BE%D0%BC\_%D0%BF%D1%80%D0%BE%D1%81%D1%82%D1%80%D0%B0%D0%BD%D1%81%D1%82%D0%B2%D0%B5) Проблемой этой метрики может стать то, что в пространствах с высокой и низкой размерностью при таком же количестве точек могут быть несопоставимые результаты. Однако понятно, что может быть несколько точек «по краям», а все остальные — сконцентрированы на одном пятачке. Метрика покажет, что точки достаточно диверсифицированы, что окажется неверным.

Варианты исправить этот недочёт такие: либо как-то отбросить сколько-то процентов самых «дальних» точек ( $\square\square$  «выбросы») или пытаться выделять «очаги» — скопления точек, либо производить те же операции учитывая все точки, но «нечётко»: например, вместо того, чтобы не учитывать точку, учитывать её с маленьkim весом.

Проще всего — удалять каждый раз точку, максимально удалённую от «центра масс» (постоянно поддерживая актуальное его местоположение), убрав таким образом  $\approx 10\%$  точек, а потом найти отношение объёмов.

Другой напрашивающийся подход — сначала посчитать в сетке точек (например,  $\approx 10$  по каждому измерению) значение плотности распределения геномов вокруг этой точки, потом для каждого генома определить, в зоне с какой примерной плотностью он находится и, например, просуммировать эти значения, потом сравнить результат со значением для идеально равномерного распределения и для всех геномов, находящихся в одной точке и откалибровать, переведя в шкалу от 0 до 1.

Однако для пространств с высокой размерностью высокой размерностью (то есть с большим количеством параметров) такое произвести не получится, так как количество узлов для достаточной детализации становится слишком большим:

$$N_{nodes} = segments\_per\_axis^{dimensions} \quad (9)$$

То есть для 10-и делений по оси уже для 9-и параметров оказывается совершенно неприемлемое число операций.

Но в таком случае можно считать плотность не в узлах, которых становится больше, чем точек.

## 6.6. Улучшить алгоритм поиска цветов и разделения на зоны

Сейчас для разделения изображения на зоны используется Adobe Illustrator. По заданному количеству цветов (и, следовательно, уровню

детализации) он разделяет изображение на зоны, присваивая каждой какой-то из цветов палитры так, чтобы он хорошо . Сама палитра тоже формируется в ходе работы алгоритма.

Скорее всего, для этого используется один из популярных алгоритмов, описанных здесь, или некая проприетарная их вариация. Зоны, на которые происходит деление, описываются частями плоскости, ограниченными кривыми безье — «*path*» формате *svg*. Несмотря на то что формально алгоритм выполняет свою работу, большое количество зон имеет очень продолжительную форму, а также наблюдается неимоверный разброс в размерах между разными зонами (см. 5.1): всё это уменьшает эффективность процесса.

## 6.7. Перенести графические вычисления на видеокарту

Также напрашивается улучшение. Это может существенно ускорить работу алгоритма, особенно — генетического (так как при нём можно распараллелить вычисление для целой популяции), причём только в случае, если не используется быстрый пересчёт функции ошибки или мутирует очень много мазков одновременно. Как бы то ни было, когда-нибудь стоит добавить эту возможность. Тестовый проект с использованием OpenCL я уже написал.

Изначальная картинка будет переноситься в видеопамять один раз — в начале работы программы. Выделить память для матрицы можно также один раз, а потом каждый раз её очищать (этую операцию можно производить параллельно).

Если использовать видеокарту для распараллеливания вычислений, встаёт вопрос, на каком именно уровне производить разделение. Варианты такие:

1. Каждое изображение — на своём потоке. Такой способ подходит только для ГА в случае огромного размера поколения (так как потоков у видеокарты порядка нескольких тысяч). Для отжига — никогда.
2. Каждый мазок — на своём потоке. Тут возникают проблемы с синхронизацией, так как порядок наложения важен: как минимум не должны появляться в хаотическом порядке пиксели из мазков разного цвета. Даже если происходит смешение цветов при наложении, синхронизация важна. Это можно сделать через дополнительную структуру данных в виде прямоугольной матрицы, в которой для каждого пикселя будет записываться список цветов

с приоритетами (индексами мазков, а значит, и числами, определяющими порядок слоёв). Потом уже независимо для каждого пикселя будет происходить обработка смешения или наложения с замещением «попавших» на него цветов. Всё упирается в умение синхронизировать потоки. Тут нужно либо симуляцию мьютекса для каждого пикселя (то есть матрицу булевых значений «занят-не занят»), либо умение распределить мазки по потокам так, чтобы в каждый момент времени не было никаких двух с накладывающимися bounding-box'ами.

В первом случае либо нужно покупать видеокарту с поддержкой атомарных значений, либо придётся вставлять дополнительные проверки, чтобы два потока не прочитали почти одновременно состояние «не занят» и не зашли туда до того, как какой-либо из них успеет записать состояние «закрыто».

Во втором случае уменьшится количество потоков, одновременно работающих над одной «картиной».

3. Проводить распараллеливание на графическом примитиве низшего уровня, использующемся в данном алгоритме (см. 2.3). Например, полигоны или круги. Видеокарта умеет эффективно отрисовывать такие примитивы. Однако количество точек в примитивах обычно невелико: существенно меньше, чем ядер в видеокарте.

Причём всегда можно комбинировать разделения на разных уровнях.

Когда будет произведена растеризация, на той же видеокарте посчитается функция ошибки: в этом случае легко сделать это независимо для каждого пикселя. Единственное — нужно помнить, что для добавления наказания за наложения и пустоты в функцию ошибки надо составлять дополнительные матрицы, в которых это будет указано.

Адаптация алгоритмов под видеокарту описана здесь: 2.3.