МИНОБРНАУКИ РОССИИ
Федеральное государственное бюджетное образовательное учреждение
высшего образования
**«САРАТОВСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ
ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ
ИМЕНИ Н. Г. ЧЕРНЫШЕВСКОГО»**
Кафедра математической кибернетики и компьютерных наук

РЕАЛИЗАЦИЯ И СРАВНЕНИЕ АЛГОРИТМОВ АВТОМАТИЧЕСКОЙ СБОРКИ ПАЗЛОВ

КУРСОВАЯ РАБОТА

студента 3 курса 351 группы
направления 09.03.04 — Программная инженерия
факультета КНиИТ
Синкевича Артема Александровича

Научный руководитель
доцент, к. ф.-м. н. _____ А. С. Иванова

Заведующий кафедрой
к. ф.-м. н., доцент _____ С. В. Миронов

Саратов 2022

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

На протяжении сотен лет людей развлекала задача сборки целостной картины из кусочков пазла. Можно предположить, что те же самые стратегии, которые сегодня используются людьми для решения этой головоломки, использовались для сборки тех первых пазлов, которые создал британский картограф Джон Спилсбери в 18-м веке, и полагались на форму и текстуру деталей головоломки. Проблемы сборки пазлов длительное время вызывают интеллектуальный интерес людей, а также являются жизненно важными задачами в таких областях, как археология [1]. Наиболее значительными материальными свидетельствами многочисленных обществ прошлого являются горшки, которые они оставили после себя, но иногда значимая историческая информация доступна только тогда, когда первоначальная форма реконструирована из разрозненных частей. Компьютерные программы решения пазлов помогают исследователям не только восстанавливать горшки из их фрагментов [2], но и реконструировать измельчённые документы или фотографии [3; 4]. Автоматическая сборка пазлов также может оказаться полезной в компьютерной криминалистике для работы с удалёнными или переставленными блочными данными изображения (например, формата JPEG), которые трудно распознать и систематизировать [5; 6]. Программы сборки пазлов также используются при создании и обработке изображений, позволяя плавно переставлять сцены с сохранением исходного содержания [7]. Чжао и др. [8] показали, что пазлы также связаны с восстановлением речи, в записи которой были переставлены части.

В литературе представлены разные варианты задачи сборки пазлов и алгоритмы её решения: использующие форму фрагментов, цветовые данные или и то, и другое. В данной работе, как и в недавних статьях по этой теме, рассматривается задача сборки прямоугольных изображений из квадратных деталей с цветовыми данными, для которых известна ориентация в пространстве.

Целью данной работы является изучение различных алгоритмов автоматической сборки пазлов и сравнение их друг с другом по критериям времени работы и точности сборки.

В результате написания работы должны быть решены следующие задачи:

— реализация различных методов сравнения деталей;
— изучение и реализация выбранных алгоритмов сборки пазлов;
— создание приложения с графическим интерфейсом для выполнения и визуализации результатов работы алгоритмов;
— изучение методов оценки результатов;
— сравнение реализованных алгоритмов, использующих выбранные методы сравнения деталей, по различным критериям.

1 Задача сборки пазлов и методы её решения

Вычислительная задача сборки пазла была впервые представлена почти шестьдесят лет назад в фундаментальной работе Фримана и Гарднера [9]. Как и в случае с материальной головоломкой, вычислительная задача состоит в том, чтобы соединить несколько более мелких частей пазла и сформировать полное изображение. В литературе представлено множество решений этой задачи. Вслед за Фриманом и др. [9] в нескольких ранних работах исследуются аспекты использования информации о форме кусочков пазла и сопоставления контуров для поиска подходящих деталей. Например, в [10] хорошо известная человеческая стратегия сборки сначала краёв изображения используется путём идентификации краевых частей с помощью информации о форме, а затем представления сборки границ пазла как задачи коммивояжёра. Для пазлов с квадратными частями в работе [11] предложено решение, использующее Марковские случайные поля, но соблюдение глобальных ограничений (например, что каждая деталь должна использоваться только один раз) оказывается трудным. Померанц и др. [12] продемонстрировали улучшенную производительность с помощью жадного алгоритма, который сегментирует частичное решение («выращивая» одну компоненту), а затем сдвигает собранные части в поисках лучшего соответствия. Сложные решения (во время создания компонент) неизбежно принимаются раньше, чем это абсолютно необходимо. Оба вышеупомянутых решения предполагают, что известны ориентация деталей и размеры собранного пазла.

Демейн и Демейн [13] показали, что при наличии неоднозначности в определении совместимости деталей пазла сборка головоломки представляет собой NP-трудную задачу. Следовательно, нельзя определить глобальную функцию, которую можно эффективно оптимизировать. Другими словами, существует слишком много способов, которыми можно собрать кусочки пазла, чтобы оценить их все. Вместо этого исследователи предложили множество стратегий сборки, в том числе жадный выбор деталей пазла, идентификацию и сборку краёв, а также Марковские случайные поля.

Косиба и др. [14] были первыми, кто использовал как форму фрагментов пазла, так и цветовую информацию изображения, так как их алгоритм пытается создавать решения, в которых соседние детали имеют схожие цвета за счёт вычисления цветовой совместимости вдоль совпадающего контура.

Чанг и др. последовали за ними, используя в качестве «штрафа» квадраты несоответствий цвета по границе, а позже в [11; 15] эта мера несовместимости (в цветовом пространстве LAB) утверждается как хороший выбор среди возможных вариантов. Другие работы, в которых рассматривается цвет деталей [16–20], используют небольшие вариации этой меры несходства. Из этих работ выделяется [16], где соприкасающиеся профили двух деталей пазла сопоставляются с динамическим искажением времени, [19], где используется закрашивание из статьи [21] для дорисовывания содержимого через границу фрагмента, и [12], где мера совместимости основана на предсказывании значений за границей детали.

В нескольких недавних работах [11; 12; 15; 16] исследовались пазлы с квадратными деталями. В них было несколько неявных предположений, во-первых, что размеры головоломки известны. В [11] опорные детали (с известным корректным расположением) необязательны, но желательны для получения качественных результатов, а в [15] требуется одна опорная деталь. В этих предыдущих работах предполагается, что ориентация каждого фрагмента пазла известна, и неизвестно только расположение каждой части в пазле. Следовательно, пара деталей головоломки может соединиться только четырьмя различными способами. Галлагер в статье [22] назвал это «типом 1» задачи сборки пазла с квадратными элементами, а также ввёл «тип 2» и «тип 3»: в задаче типа 2 неизвестны ни расположение, ни ориентация какой-либо детали, а в типе 3 известно положение, но неизвестна ориентация деталей.

В задаче типа 3 возможно 4^N решений, где N — количество деталей. Задача состоит в том, чтобы определить, какая из четырёх возможных ориентаций каждой детали является правильной. Хотя эта задача является наименее сложной в вычислительном отношении из трёх основных типов, она приводит к элегантному решению с применением Марковских случайных полей в работе [22].

В задаче типа 2 неизвестно ни расположение, ни ориентация какой-либо части. Это усложняет задачу несколькими путями. Во-первых, пара деталей может сочетаться в любой из 16 конфигураций (вторая деталь может быть выше, слева, справа или под первой деталью, а также она может иметь любую из четырёх ориентаций). Количество возможных решений по сравнению с типом 1 (для которого возможно $N!$ решений, то есть все перестановки дета-

лей) увеличивается в 4^N раз. Во-вторых, алгоритм сборки должен учитывать как вращение, так и перемещение деталей или компонент. В-третьих, размеры прямоугольного пазла менее полезны, потому что неизвестно, должен ли находиться собранный пазл в портретной или альбомной ориентации. В более поздних статьях, таких как [23] и [24] исследуются решения именно второго типа задачи, а первый рассматривается только как частный случай.

Также в [23] были введены типы 4, 5, 6, 7: в них пазл состоит из двух изображений с разных сторон, и алгоритму дополнительно требуется определить для каждой детали её лицевую сторону. В задаче типа 7 это единственное требование, а №4, 5, 6 соответствуют расширениям типов 2, 1, 3. Таким образом количество возможных решений увеличивается в 2^N раз, а количество возможных сочетаний двух деталей — до 64 в случае задачи типа 4.

Во всех решениях задачи автоматической сборки пазлов (по цветовым данным деталей) есть две важные части: мера совместимости частей пазла при их соединении в пару и стратегия сборки головоломки. Далее будут рассмотрены возможные решения каждой из частей задачи типа 1.

1.1 Методы сравнения деталей

В работе [11] представлены несколько мер сравнения деталей:

— На основе бустинг-классификатора: для каждой детали из отобранных для обучения пар выбирались граничные двухпиксельные полосы, вычислялась сумма квадратов разностей для соответствующих полос в изображениях, и на этих данных обучался бустинг-классификатор;

— На основе набора деталей: вычисляется как минимальное расстояние (сумма квадратов разностей цветов пикселей) между деталью, полученной вырезанием центра из соединения двух сравниваемых деталей, и всеми другими деталями из набора;

— На основе статистики изображений: вычисляется свёртка изображения детали, полученной вырезанием центра из соединения двух сравниваемых деталей, с помощью фильтров из статьи [25], и определяется совместимость деталей по степени отклика на границе деталей.

Выяснилось, что эти алгоритмы оказываются значительно менее точными, чем более простой метод «несходства», описанный далее.

1.1.1 Метод SSD

Мера совместимости SSD (Sum of Squared Differences, сумма квадратов разностей) основывается на предположении, что соседние детали пазла в исходном изображении, как правило, имеют схожие цвета вдоль их соприкасающихся краёв, то есть сумма (по всем соседним пикселям) квадратов разностей цветов (по всем цветовым каналам) должна быть минимальной. Пусть изображения деталей x_i, x_j представлены в цветовом пространстве L*a*b* матрицей $K \times K \times 3$, где K — ширина/высота детали (в пикселях), то их несовместимость $C_{LR}(x_i, x_j)$ при условии, что x_j находится справа от x_i равна

$$C_{LR}(x_i, x_j) = \sqrt{\sum_{k=1}^{K} \sum_{c=1}^{3} (x_i(k, K, c) - x_j(k, 1, c))^2}, \tag{1}$$

где $x_i(n, m, c)$ — значение цветового канала c пикселя в строке n и столбце m детали x_i. Аналогично определяется и случай, когда деталь x_i находится над x_j:

$$C_{UD}(x_i, x_j) = \sqrt{\sum_{k=1}^{K} \sum_{c=1}^{3} (x_i(K, k, c) - x_j(1, k, c))^2}, \tag{2}$$

При решении задачи типа 1 случаи, когда x_j находится слева или снизу от x_i рассматривать не требуется, так как они эквивалентны ситуациям, когда x_i находится справа или сверху от x_j.

По формулам 1, 2 можно видеть, что используется только последний столбец/строка детали x_i и первый столбец/строка детали x_j, и поэтому асимптотика времени вычисления для одной пары деталей — $O(K)$, для всех пар — $O(N^2K)$, где N — количество деталей.

В этом методе, как и в методе MGC, используется цветовое пространство LAB (L*a*b*) вместо распространённого RGB, так как первое, в отличие от второго, учитывает восприятие цвета человеком, и расстояние между цветами в нём (Евклидово между векторами компонентов) более равномерно [26].

В приложении А приведена реализация функции, вычисляющей по заданному изображению, его размерам и ширине/высоте детали в пикселях SSD-несовместимость в цветовом пространстве L*a*b* для каждой пары деталей. Так как результат для одной пары не зависит от результата для другой, то вычисления проводятся параллельно.

1.1.2 Метод MGC

В работе [22] была предложена мера MGC (Mahalanobis Gradient Compatibility, совместимость градиентов Махаланобиса), которая описывает локальные градиенты на границе деталей пазла и имеет два улучшения по сравнению со мерой совместимости SSD. Во-первых, «штрафуются» изменения градиентов интенсивности цветов, а не изменения самой интенсивности. Другими словами, если у детали пазла есть градиент вблизи края, то ожидается, что соседняя деталь продолжит градиент. Во-вторых, вместо того, чтобы равномерно штрафовать все отклонения от постоянного градиента (то есть с помощью евклидова расстояния), вычисляется ковариация между цветовыми каналами и после этого используется расстояние Махаланобиса. Таким образом, детали пазла смежны по методу MGC, если градиент на границе деталей близок к градиентам (внутри деталей) по обе стороны от границы.

Пусть изображения деталей x_i, x_j определены так же, как и в предыдущем разделе, и x_j находится справа от x_i, тогда при вычислении несовместимости $C_{LR}(x_i, x_j)$ сначала требуется найти распределение цветовых градиентов вблизи правого края детали x_i. Пусть G_L — массив градиентов с 3 столбцами (по одному для каждого цветового канала) и K строками (где K — размер детали в пикселях). G_L описывает изменение интенсивности вдоль правого края x_i (поскольку эта деталь находится слева) и вычисляется как

$$G_L(k, c) = x_i(k, K, c) - x_i(k, K-1, c) \tag{3}$$

Аналогично определяется G_R для левого края детали x_j:

$$G_R(k, c) = x_j(k, 1, c) - x_j(k, 2, c) \tag{4}$$

Среднее распределение этих градиентов по каждому цветовому каналу с правой стороны детали x_i находится как

$$\mu_L(c) = \frac{1}{K} \sum_{k=1}^{K} G_L(k, c) \tag{5}$$

И для левой стороны детали x_j:

$$\mu_R(c) = \frac{1}{K}\sum_{k=1}^{K} G_R(k,c) \tag{6}$$

Для каждого цветового канала μ_L представляет собой среднюю разницу между последними двумя столбцами x_i. Далее по G_L строится матрица ковариаций S_L (и S_R по G_R) размера 3×3, которая фиксирует взаимосвязь между градиентами цветовых каналов вблизи края детали. Для численной стабильности инвертирования матрицы к диагональным элементам добавляется $EPS = 10^{-6}$. Кроме того, вычисляется градиент из правой части x_i в левую часть x_j:

$$G_{LR}(k,c) = x_j(k,1,c) - x_i(k,K,c) \tag{7}$$

И градиент в противоположную сторону:

$$G_{RL}(k,c) = -G_{LR}(k,c) \tag{8}$$

Далее можно определить односторонние несовместимости деталей (использующие только градиенты внутри одной детали и градиент между деталями):

$$D_{LR}(x_i,x_j) = \sqrt{\sum_{k=1}^{K}(G_{LR}(k)-\mu_L)S_L^{-1}(G_{LR}(k)-\mu_L)^T} \tag{9}$$

$$D_{RL}(x_i,x_j) = \sqrt{\sum_{k=1}^{K}(G_{RL}(k)-\mu_R)S_R^{-1}(G_{RL}(k)-\mu_R)^T} \tag{10}$$

С помощью которых вычисляется симметричная мера несовместимости:

$$C_{LR}(x_i,x_j) = D_{LR}(x_i,x_j) + D_{RL}(x_i,x_j) \tag{11}$$

Аналогично составляются формулы для случая, когда x_j находится снизу от x_i.

Из приведённых выше формул следует, что используются только два последних столбца/строки детали x_i и два первых столбца/строки детали x_j, и поэтому асимптотика времени вычисления для одной пары деталей $-O(K)$,

для всех пар — $O(N^2K)$, то есть такая же, как и у метода SSD, но со значительно большим константным множителем из-за необходимости вычисления ковариационной матрицы и прочих значений.

В приложении Б приведена реализация функции, вычисляющей по заданному изображению, его размерам и ширине/высоте детали в пикселях MGC-несовместимость в цветовом пространстве L*a*b* для каждой пары деталей. Так как результат для одной пары не зависит от результата для другой, то вычисления проводятся параллельно. Также было оптимизировано вычисление матрицы ковариаций за счёт её симметричности, и обратной к ней матрицы с помощью упрощения явной формулы её нахождения.

1.2 Алгоритмы сборки

1.2.1 Генетический алгоритм

Большинство из упомянутых в разделе 1 алгоритмов являются жадными и, таким образом, подвержены большому риску схождения к локальным оптимумам. Несмотря на большой потенциал применения генетического алгоритма, успех предыдущих попыток был ограничен пазлами из 64 деталей [27], скорее всего, из-за большой сложности эволюционных вычислений в целом и трудности разработки эффективного оператора скрещивания для этой задачи в частности. Но в статье [28] был представлен генетический алгоритм, являющийся одним из лучших для решения поставленной задачи на данный момент.

Для генетического алгоритма требуется ввести несколько определений и уточнить их для данной задачи:

- Особь — одно из решений задачи.
- Популяция — набор особей, представляющих собой одно поколение [29].
- Поколение — один этап выполнения генетического алгоритма.
- Хромосома — информация, определяющая особь, в данной задаче — матрица $n \times m$ (n и m — количество строк и столбцов в изображении, разбитом на детали), состоящая из пар строк и столбцов, задающих детали пазла. Таким образом, хромосома однозначно определяет решение.
- Функция приспособленности (fitness function) — оценка решения задачи данной хромосомой. Для задачи сборки пазлов используется функция на основе одного из методов сравнения деталей. Пусть $C_d(x_i, x_j)$ — несов-

местимость деталей x_i и x_j при сопоставлении в направлении d, $x_{i,j}$ — матрица-хромосома размера $n \times m$, тогда неприспособленность особи с этой хромосомой равна

$$\sum_{i=1}^{n}\sum_{j=1}^{m-1} C_{LR}(x_{i,j}, x_{i,j+1}) + \sum_{i=1}^{n-1}\sum_{j=1}^{m} C_{UD}(x_{i,j}, x_{i+1,j}), \tag{12}$$

таким образом, вычисляется сумма несовместимостей по всем границам смежных деталей. Для ускорения вычисления функции приспособленности предварительно рассчитываются несовместимости всех пар деталей пазла.

Алгоритм состоит из нескольких этапов:

1. Создание начальной популяции;
2. Создание новых поколений, пока не выполнено условие остановки:
   а) Отбор лучших особей в новое поколение («элитарность»);
   б) Отбор пар особей из текущего поколения;
   в) Скрещивание этих пар и составление нового поколения.

Начальная популяция генерируется как необходимое количество случайных перестановок. В качестве критерия остановки выбрано достижение заданного количества поколений. При создании нового поколения в него сразу копируются $ELITISM\_COUNT = 4$ особей, определённых как лучшие с помощью функции приспособленности. Размер популяции и количество поколений в приложении, созданном для данной работы, выбирается пользователем перед запуском алгоритма.

Как и в статье [28], в этой работе используется «метод рулетки» для выбора родителей для скрещивания, но модифицированный из-за необходимости назначать меньшую вероятность выбора особям с большим значением функции неприспособленности, а также предотвращения схождения к одной хромосоме. Пусть P — размер популяции, f_i — значение функции неприспособленности для i-й особи, тогда F_i — её преобразованное значение и p_i — вероятность выбора i-й особи:

$$F_i = 0.95 + 0.9\frac{\min_{j=1,\dots,P} f_j - f_i}{\max_{j=1,\dots,P} f_j - \min_{j=1,\dots,P} f_j}, p_i = \frac{F_i}{\sum_{j=1}^{P} F_j} \tag{13}$$

При создании нового поколения выбирается столько же пар, сколько особей

в текущем поколении, за вычетом количества уже отобранных, при этом каждая особь в паре выбирается случайно среди всех в текущем поколении с помощью вероятностей p_i. Затем из каждой пары получается новая особь с помощью оператора скрещивания.

Наиболее сложной частью генетического алгоритма является оператор скрещивания (кроссовера). Простая реализация может создавать новую дочернюю хромосому случайным образом так, что каждый элемент результирующей матрицы будет соответствующим элементом первого или второго родителя. Этот подход обычно создаёт хромосомы с дубликатами и/или отсутствующими деталями пазла, что, конечно же, является неправильным решением задачи. Трудность, присущая оператору кроссовера, вполне могла сыграть решающую роль в задержке разработки решения, основанного на методах эволюционных вычислений.

Скрещивание применяется к двум хромосомам, выбранным из-за их высоких значений функции приспособленности, причём эта функция является мерой попарной совместимости всех соседних деталей пазла. В лучшем случае она вознаграждает за правильное размещение соседних деталей рядом друг с другом, но не имеет возможности определить правильное расположение детали. Поскольку первое поколение состоит из случайных хромосом, а каждое следующее постепенно улучшается, разумно предположить, что некоторые правильно собранные компоненты пазла появляются на протяжении поколений. Принимая во внимание неспособность функции приспособленности вознаградить правильную позицию, ожидается, что такие компоненты, скорее всего, появятся в неправильных местах. Оператор кроссовера должен передавать «хорошие черты» от родителей к ребёнку, тем самым создавая, возможно, лучшее решение. Обнаружение правильной компоненты не является тривиальным; её следует рассматривать как хорошую черту, которую нужно использовать и передавать следующим поколениям. Таким образом, оператор скрещивания должен допускать независимость от позиции, то есть возможность сдвигать целые правильно собранные компоненты в попытке правильно разместить их в дочерней хромосоме.

Также оператор скрещивания должен уметь обнаруживать правильно собранные компоненты, которые, возможно, находятся не на своём месте. Какая компонента должна быть передана потомству? Случайный подход может по-

казаться привлекательным, но он может оказаться непрактичным из-за огромного размера пространства решений задачи. Возможно применение эвристик для того, чтобы отличить правильные компоненты от неправильных.

Таким образом, хороший оператор скрещивания должен создавать корректные дочерние хромосомы, решать вопросы обнаружения предположительно правильно собранных компонент пазла у родителей и независимости положения этих компонент при передаче потомству.

В [28] предлагается оператор кроссовера, решающий описанные проблемы. Пусть имеются две родительские хромосомы, то есть два различных полных расположения всех деталей пазла, тогда оператор конструирует дочернюю хромосому, постепенно «наращивая» ядро, используя обоих родителей в качестве «консультантов». Оператор начинает с одной детали и постепенно присоединяет другие к доступным границам ядра. Новые фрагменты могут быть соединены только с уже установленными, поэтому возникающее изображение всегда будет непрерывным. Детали добавляются из множества доступных до тех пор, пока не останется ни одной свободной. Следовательно, каждая деталь будет появляться ровно один раз в результирующей хромосоме. Поскольку размер изображения известен заранее, оператор скрещивания может убедиться, что нет выхода за границы. Таким образом гарантируется получение корректного изображения.

Ключевым свойством метода «выращивания» ядра является то, что конечное местоположение каждой детали определяется только после того, как ядро достигнет своего окончательного размера и дочерняя хромосома будет завершена. До этого все фрагменты могут сдвигаться в зависимости от направления роста ядра. Первая деталь, например, может оказаться в нижнем левом углу изображения, если ядро будет расти только вверх и вправо. С другой стороны, та же самая деталь может в конечном итоге быть помещена в центр изображения, в его правый верхний угол или в любое другое место. Именно это важное свойство обеспечивает независимость компонент изображения от позиции. А именно, правильно собранная компонента пазла в родительской хромосоме, возможно, смещённая по отношению к её истинному местоположению, может быть скопирована во время скрещивания, сохраняя свою структуру, на правильное место в потомстве.

В описанном методе нужно уметь определять, какую деталь выбрать

из множества доступных, и где её разместить. Имея ядро (частичное изображение), можно определить все границы, у которых может быть размещена новая деталь. Граница детали обозначается парой (x_i, d), состоящей из детали и направления. Оператор скрещивания состоит из трёх этапов. Во-первых, проверяется, существует ли такая граница, с которой согласны оба родителя, то есть существует ли деталь x_j, которая находится в направлении d от x_i в обоих родителях. Если это так, то x_j добавляется в ядро в соответствующем месте. Если родители соглашаются о двух и более границах, то случайным образом выбирается одна из них. Очевидно, что если деталь уже используется, её нельзя повторно назначить, и поэтому она игнорируется, как будто родители не согласны с этой границей.

Если между родителями нет согласия ни по одной детали на какой-либо границе, начинается вторая фаза, использующая концепцию «лучших приятелей», впервые введённую в работе [12]. Две детали считаются лучшими приятелями, если каждая деталь считает другую наиболее совместимой с ней. Таким образом, x_i и x_j — лучшие приятели, если

$$\forall x_k : C_{d_1}(x_i, x_j) \leqslant C_{d_1}(x_i, x_k) \tag{14}$$

и

$$\forall x_l : C_{d_2}(x_i, x_j) \leqslant C_{d_2}(x_i, x_l), \tag{15}$$

где $C_d(x_i, x_j)$ — несовместимость деталей x_i и x_j при сопоставлении в направлении d, а d_1 и d_2 — противоположные направления (например, право и лево). На втором этапе оператор проверяет, содержит ли один из родителей деталь x_j в направлении d от x_i, которая также является лучшим приятелем x_i (учитывая направление). Если это так, то деталь выбирается и назначается. Как и прежде, если доступно несколько деталей-лучших приятелей, то одна из них выбирается случайно. Если найдена деталь, которая уже размещена, она игнорируется, и продолжается поиск других подходящих. Если детали с лучшим приятелем не существует, оператор переходит к заключительной третьей фазе, где выбирается случайная граница, и ей назначается наиболее совместимая доступная деталь. Также вводятся мутации: оператор с малой вероятностью во время первой и последней фазы помещает случайную доступную деталь вместо выбранной на этом этапе. В реализации были выбраны вероятности

$MUTATION\_RATE\_1 = 0.001$ и $MUTATION\_RATE\_3 = 0.005$ для первой и третьей фазы соответственно.

В приложении В приведена оптимизированная реализация генетического алгоритма, вызываемая с помощью функции `algorithm_step`, создающей очередное поколение для изображения по уже вычисленным несовместимостям деталей, «лучшим приятелям» и текущему поколению. Так как результат скрещивания двух хромосом не зависит от других, то вычисления производятся параллельно. Перед выполнением алгоритма для изображения должны быть найдены лучшие приятели с помощью функции `find_best_buddies`. Главная из приведённых функций — `chromosomes_crossover`, скрещивающая две хромосомы. Функция реализует алгоритм, описанный выше, но при этом сохраняются позиции, подходящие для каждой фазы и детали, которые возможно установить, а также позиции, не соответствующие фазам. На каждом шаге проверяются ещё не обработанные позиции и те, которые необходимо перепроверить из-за размещения деталей рядом с ними. Так как многие позиции не перепроверяются каждый раз, то этот подход позволяет в большинстве случаев уменьшить время работы функции.

1.2.2 Алгоритм циклических ограничений

В статье [24] представлен алгоритм, ключевая идея которого отличается от предыдущих стратегий и состоит в том, чтобы явно найти все «маленькие циклы» и сгруппировать их в «циклы из циклов» более высокого порядка, поэтапно увеличивая их размер. В этом методе циклы из деталей пазла, в частности циклы из 4 элементов, используются как способ обнаружения выбросов, тогда как предыдущие алгоритмы стараются избегать их, как, например, алгоритм из работы Галлагера [22], в котором конструируются свободные от циклов деревья из деталей пазла. Во время восходящей сборки некоторые из обнаруженных маленьких циклов могут оказаться ложными. Затем алгоритм переходит к этапу нисходящей сборки, в котором неиспользованные компоненты из циклов объединяются с доминирующими компонентами при отсутствии геометрического конфликта. В противном случае компоненты разбиваются на подциклы, и снова предпринимаются попытки слияния с меньшими циклами. Если циклы из 4 деталей всё ещё конфликтуют с доминирующими компонентами, они удаляются, так как считаются выбросами.

Цикл из деталей пар-кандидатов (то есть деталей, считающихся доста-

точно совместимыми) указывает на консенсус среди нескольких попарных совместимостей. Хотя легко найти 4 детали, которые соединяются в цепочку через три их границы с низкой ошибкой, маловероятно, что четвёртый край, завершающий цикл, также окажется случайно среди пар-кандидатов. На самом деле, чтобы построить маленький цикл, который не состоит из правильных сочетаний, по крайней мере две пары деталей в нём должны быть неправильными. Хотя некоторые маленькие циклы будут содержать неправильные пары, из которых, тем не менее, составляются циклы, вероятность этого уменьшается по мере сборки циклов всё более высокого порядка. Таким образом маленький цикл размерности 3, построенный из 4 маленьких циклов размерности 2, отражает консенсус среди многих пар деталей.

Термин «маленький цикл» используется, чтобы подчеркнуть, что описываемый метод фокусируется на максимально коротких циклах деталей на каждом этапе — циклах длины 4. Можно было бы рассматривать более длинные циклы, но в этом алгоритме используются только маленькие, потому что более длинные циклы с меньшей вероятностью будут полностью состоять из правильных пар деталей, и пространство возможных циклов увеличивается экспоненциально с длиной цикла. Хотя возможно перечислить все циклы длины 4 из пар-кандидатов в пазле, это намного труднее для более длинных циклов. Этот алгоритм постепенно переходит от неточных пар-кандидатов к значительно более точным циклам высших порядков, и крупные циклы уже обнаруживаются путём нахождения совместимых групп маленьких циклов.

Перед сборкой циклов нужно рассмотреть метрику попарной несовместимости деталей и стратегию поиска сочетающихся пар-кандидатов. Возможно использовать как метод SSD, так и MGC; необходимо вычислить несовместимости для всех пар деталей и всех ориентаций пар. Абсолютные расстояния между потенциальными подходящими деталями не сопоставимы (например, детали изображения неба всегда будут иметь меньшую несовместимость), поэтому вместо них используются коэффициенты несходства, как и в [22]. Для каждого края каждой детали все несовместимости делятся на наименьшую несовместимость с этой деталью в этом направлении. Сочетания деталей с коэффициентом менее $CANDIDATE\_MATCH\_RATIO = 1.15$ считаются парами-кандидатами для дальнейшего рассмотрения. Для эффективности вычислений максимальное количество пар-кандидатов, которые

может составить одна деталь в одном направлении, ограничивается константой $MAX\_CANDIDATES = 10$. Также для предотвращения создания большого количества маленьких циклов из одинаковых деталей (например, полностью белых или чёрных), которые определяются коэффициентом $CANDIDATE\_MATCH\_RATIO\_EQUAL = 1.0001$, также ограничивается их количество значением $MAX\_CANDIDATES\_EQUAL = 7$. Из пар-кандидатов, созданных в разных направлениях от деталей (например, влево и вправо), создаются такие пары, в которых и первая деталь считает вторую кандидатом, и наоборот, с помощью пересечения соответствующих множеств.

В алгоритме циклических ограничений для представления и маленьких циклов, и групп деталей, и итогового решения используются двумерные матрицы, состоящие из пар строк и столбцов, задающих детали пазла. Для таких матриц определяется совместимость: если матрицы U и V имеют как минимум две общие детали, то они выравниваются по одной из них. Если нет геометрического конфликта, такого как наложение разных деталей или повторение деталей вне области пересечения, то U и V геометрически согласованы, что обозначается как $U \sim V$. В противном случае геометрическое несоответствие матриц обозначается как $U \perp V$. Если $U \sim V$, то возможно объединение двух матриц $U \oplus V$. Если в матрицах меньше двух общих деталей, то считается, что они не связаны друг с другом, и они обозначаются как $U||V$.

Первый этап алгоритма заключается в нахождении маленьких циклов. Пары-кандидаты (матрицы размеров 1×2 и 2×1) обозначаются как маленькие циклы первого порядка. Также формируются маленькие циклы порядка 2 с размерами 2×2, каждый из четырёх пар-кандидатов. Как только все циклы определённого порядка обнаружены, из четвёрок циклов собираются циклы следующего порядка, если расположение деталей геометрически согласовано среди всех маленьких циклов более низкого порядка. Таким образом алгоритм итеративно строит маленькие циклы порядка i путем сборки из циклов порядка $i - 1$. Процедура продолжается до тех пор, пока не удастся создать ни один цикл некоторого порядка $M + 1$.

Описывая алгоритм более формально, можно определить множество деталей пазла $\Omega_1 = \{\omega_{1,1}, \omega_{1,2}, \ldots, \omega_{1,N}\}$, где N — количество всех деталей, а также множество пар-кандидатов как разреженную трёхмерную матрицу M_1

размера $N \times N \times 2$, где $M_1(x_i, x_j, d) = 1$ означает, что детали x_i и x_j считаются совместимыми в направлении d (горизонтальное или вертикальное). Также, как и создаются маленькие циклы второго порядка Ω_2 с помощью M_1 из четырёх деталей, возможно определить построение циклов $(i+1)$-го порядка из циклов i-го. Пусть $\Omega_i = \{\omega_{i,1}, \omega_{i,2}, \ldots, \omega_{i,N_i}\}$ — множество маленьких циклов порядка i, где N_i — их количество, тогда по ним вычисляется матрица совместимости M_i размера $N_i \times N_i \times 2$. Циклы $\omega_{i,x}$ и $\omega_{i,y}$ считаются совместимыми, если они геометрически согласованы ($\omega_{i,x} \sim \omega_{i,y}$), то есть корректно пересекаются в области размера $i \times (i-1)$ (горизонтально) или $(i-1) \times i$ (вертикально), и при объединении циклов вне области пересечения не возникает повторяющихся деталей. Из элементов Ω_i создаётся цикл порядка $i+1$ (матрица размера $(i+1) \times (i+1)$), если все смежные подциклы совместимы в соотвествии с M_i. Все найденные циклы составляют множество Ω_{i+1}. Если оно окажется пустым, то i — высший порядок маленьких циклов.

На втором этапе алгоритм работает с маленькими циклами по убыванию их порядка и последовательно объединяет всё меньшие оставшиеся циклы, не применяя циклические ограничения. Для слияния достаточно двух геометрически согласованных матриц. Если маленький цикл геометрически конфликтует с более крупной матрицей, цикл разбивается на составляющие подциклы более низкого порядка. Если два маленьких цикла одинаковой размерности конфликтуют, то используется цикл с меньшим средним значением несовместимости по всем смежным деталям в нём. Объединение не производится, если результирующая матрица больше пазла по какой-либо стороне или матрицы не связаны друг с другом. После всех слияний у матрицы-результата отрезаются почти пустые края и жадно заполняются пустые позиции.

Для более точного определения вводится функция объединения для двух матриц ω_x, ω_y:

$$f_m(\omega_x, \omega_y) = \begin{cases} \omega_x \oplus \omega_y, & \text{если } \omega_x \sim \omega_y \\ \omega_x, & \text{если } \omega_x \perp \omega_y \wedge f_p(\omega_x) \geqslant f_p(\omega_y) \\ \omega_y, & \text{если } \omega_x \perp \omega_y \wedge f_p(\omega_x) < f_p(\omega_y) \\ \omega_x, \omega_y, & \text{если } \omega_x || \omega_y \end{cases} \tag{16}$$

где f_p — функция приоритета матрицы:

$$\begin{cases} f_p(\omega_x) \geqslant f_p(\omega_y), & \text{если } |\omega_x| > |\omega_y| \\ f_p(\omega_x) \geqslant f_p(\omega_y), & \text{если } |\omega_x| = |\omega_y| \wedge \overline{\omega_x} < \overline{\omega_y} \\ f_p(\omega_x) < f_p(\omega_y), & \text{иначе} \end{cases} \tag{17}$$

где $|\omega|$ — количество непустых элементов в матрице ω, а $\overline{\omega}$ — среднее значение несовместимости по всем соседним деталям в матрице.

На данном этапе уже вычислены множества маленьких циклов $\{\Omega_1, \Omega_2, \ldots, \Omega_M\}$. Из каждого Ω_i и Λ_{i+1} создаётся Λ_i, множество матриц-результатов объединений, итеративно для всех $i \in [1, M]$, по убыванию i. Изначально $\Lambda_i = \Lambda_{i+1} \cup \Omega_i$, затем выполняются объединения элементов Λ_i, пока это возможно: пусть $\omega_x, \omega_y \in \Lambda_i$ — такие матрицы, что $\omega_x \sim \omega_y$ или $\omega_x \perp \omega_y$, тогда эта пара заменяется на результат их объединения: $\Lambda_i = \Lambda_i \setminus \{\omega_x, \omega_y\}$, $\Lambda_i = \Lambda_i \cup \{f_m(\omega_x, \omega_y)\}$. При создании Λ_1 для объединения матриц достаточно одного, а не двух общих элементов, так как на этой итерации присоединяются неиспользованные пары-кандидаты. Из Λ_1 выбирается матрица с наибольшим приоритетом в качестве предварительного результата работы алгоритма.

Так как матрица, полученная на предыдущем этапе, может не соответствовать размеру изображения или иметь пропуски, то отрезаются выступающие края и заполняются пустые позиции. Для удаления выбираются строки или столбцы, которые содержат небольшую долю (не больше $TRIM\_RATE = 0.1$) непустых позиций. Удаления происходят до тех пор, пока есть хотя бы одна подходящая строка или столбец. Затем жадно заполняются пропуски: для этого для каждой пустой позиции с наибольшим количеством деталей-соседей выбирается такая неиспользованная деталь, что её несовместимость с соседями минимальна.

В приложении Г приведена реализация алгоритма с циклическими ограничениями, вызываемая с помощью функции `algorithm_step`, создающей решение для изображения по уже вычисленным несовместимостям деталей и парам-кандидатам. Перед выполнением алгоритма для изображения должны быть найдены пары-кандидаты с помощью функции `find_match_candidates`. Алгоритм сначала вызывает функцию `small_loops`, которая находит все маленькие циклы, используя для этого функции `small_loops_matches` (опре-

деление совместимостей маленьких циклов) и `merge_loops` (объединение четырёх циклов в один более высокого порядка). При создании циклов и определении их совместимостей цикл выбора одного из элементов выполняется параллельно. Второй этап алгоритма реализован в виде функции `merge_matrices_groups`, объединяющей маленькие циклы. При выполнении поддерживается множество совместимых и несовместимых пар матриц, упорядоченное по приоритетам матриц. Так как при добавлении новой матрицы необходимо проверить её совместимость с остальными, и результаты для одной матрицы не зависят от другой, то цикл выполняется параллельно. В качестве вспомогательных используются функции `matrix_priority` (вычисление приоритета матрицы как пары из количества непустых позиций и среднего значения несовместимости деталей), `can_merge_matrices` (проверка матриц на возможность объединения — либо они совместимы, и для них вычисляется сдвиг между общими деталями и приоритет объединённой матрицы, либо не совместимы, либо не связаны друг с другом), `merge_matrices` (объединение матриц по известному сдвигу между ними). Для получения итогового решения к лучшей матрице применяются методы `trim` (отрезание почти пустых краёв) и `fill_greedy` (жадное заполнение почти пустых позиций).

1.3 Сравнение алгоритмов

1.3.1 Методы оценки результатов

В работе [11] были введены следующие две основные меры для оценки точности сборки пазла, неоднократно использованные в последующих работах: прямое сравнение, в котором измеряется доля деталей, расположенных в правильных позициях, и сравнение с соседями, в котором для каждой детали определяется доля правильных соседей (с которыми деталь имеет общую грань), и от этих значений вычисляется среднее. Прямой метод считается менее точным и менее содержательным из-за неадекватной оценки слегка сдвинутых решений [12]. Можно заметить, что собранный пазл, набравший 100% по одному из методов, является полной реконструкцией исходного изображения, и получает максимальную оценку по другому методу.

1.3.2 Наборы изображений

В литературе представлены несколько наборов изображений, ставших стандартными для тестирования алгоритмов автоматической сборки пазлов с квадратными деталями: 20 изображений, каждое из $N = 432$ деталей, из статьи [11], по 20 изображений с $N = 540$ и $N = 805$ и по 3 изображения с $N = 2360$ и $N = 3300$ из работы [12]. Количества деталей указаны из предположения о стандартном их размере: $K = 28$. Эти наборы достаточно разнообразны. Для некоторых изображений камера идеально выровнена по линии горизонта, а края изображения (например, границы зданий) точно совпадают с краями пазла. Некоторые детали содержат недостаточно информации (однородные области, такие как небо, вода и снег), а другие содержат повторяющиеся текстуры (искусственные поверхности и окна). В результате метрики попарной совместимости деталей возвращают много ложноположительных и ложноотрицательных результатов для этих изображений.

1.3.3 Определение параметров генетического алгоритма

Для работы генетического алгоритма сборки пазлов требуется найти значения нескольких параметров. Для некоторых, таких как количество хромосом, отбираемых непосредственно в следующее поколение, и вероятности мутаций на разных шагах алгоритма значения были подобраны, основываясь на величинах из работы [28]. Но для размера популяции и количества поколений было проведено сравнение на некоторых наборах изображений. Время выполнения алгоритмов измерялось для данной работы на системе с процессором Ryzen 5 4600H (4 ГГц) и ОС Linux (Manjaro 21.2.6). Для каждой конфигурации параметров алгоритма выполнялось 5 запусков с начальными значениями генератора случайных чисел от 1 до 5, и по всем данным выбиралось среднее.

На рисунках 1 (для $N = 432$) и 2 (для $N = 3300$) представлены средняя точность работы (прямое сравнение и по соседям) и среднее время выполнения по каждому изображению в наборе при разных размерах популяции. Для определения точности сборки выбиралось последнее поколение, использовался метод LAB SSD, было зафиксировано количество поколений: $G = 100$. Размеры популяции выбирались из множества $P \in \{10, 30, 100, 300, 1000\}$. При выполнении использовались все доступные потоки процессора (12). По графикам видно, что с увеличением популяции качество улучшается всё меньше,

а время работы значительно растёт. Таким образом, чтобы сохранить разумное время выполнения и качество, для дальнейшего рассмотрения были выбраны значения $P_1 = 100$ и $P_2 = 300$.

Аналогично на рисунках 3 (для $N = 432$) и 4 (для $N = 3300$) представлены средняя точность работы и среднее время выполнения по каждому изображению в наборе при разном количестве поколений, при этом был зафиксирован размер популяции: $P = 300$. Количество поколений выбиралось из отрезка $G \in [1, 100]$. По графикам видно, что с увеличением количества поколений качество улучшается всё меньше, а общее время работы растёт, но при этом время вычисления одного поколения уменьшается (наиболее заметно при $N = 3300$). Это объясняется увеличением доли деталей, размещённых на фазе 1 алгоритма (и кешированием подходящих для этого позиций), из-за достаточной схожести хромосом-предков между собой и с правильным решением. Таким образом, чтобы сохранить разумное время выполнения и качество, для дальнейшего рассмотрения были выбраны значения $G_1 = 30$ и $G_2 = 100$.

В таблицах 1 (с методом SSD) и 2 (с методом MGC) представлены результаты работы генетического алгоритма с разными методами сравнения деталей и конфигурациями ($P_1 = 100$, $G_1 = 30$ и $P_2 = 300$, $G_2 = 100$): средняя точность работы и среднее время выполнения по каждому изображению в наборе для всех наборов изображений, а также среднеквадратические отклонения этих величин. Можно заметить, что увеличение количества поколений и размера популяции привело к незначительному росту точности (не более 1%), при этом время выполнения увеличилось в несколько раз (от 2 до 8, в зависимости от количества деталей). Также видна приблизительно квадратичная зависимость времени выполнения от количества деталей. Использование метода MGC позволяет улучшить точность работы на 1-7% (наибольший прирост для изображений с большим количеством деталей), но при этом происходит замедление в 2-3 раза (для первой конфигурации) или 1.1-1.25 раза (для второй). Следовательно, для итогового сравнения алгоритмов были выбраны параметры: метод MGC, размер популяции $P_1 = 100$, количество поколений $G_1 = 30$.

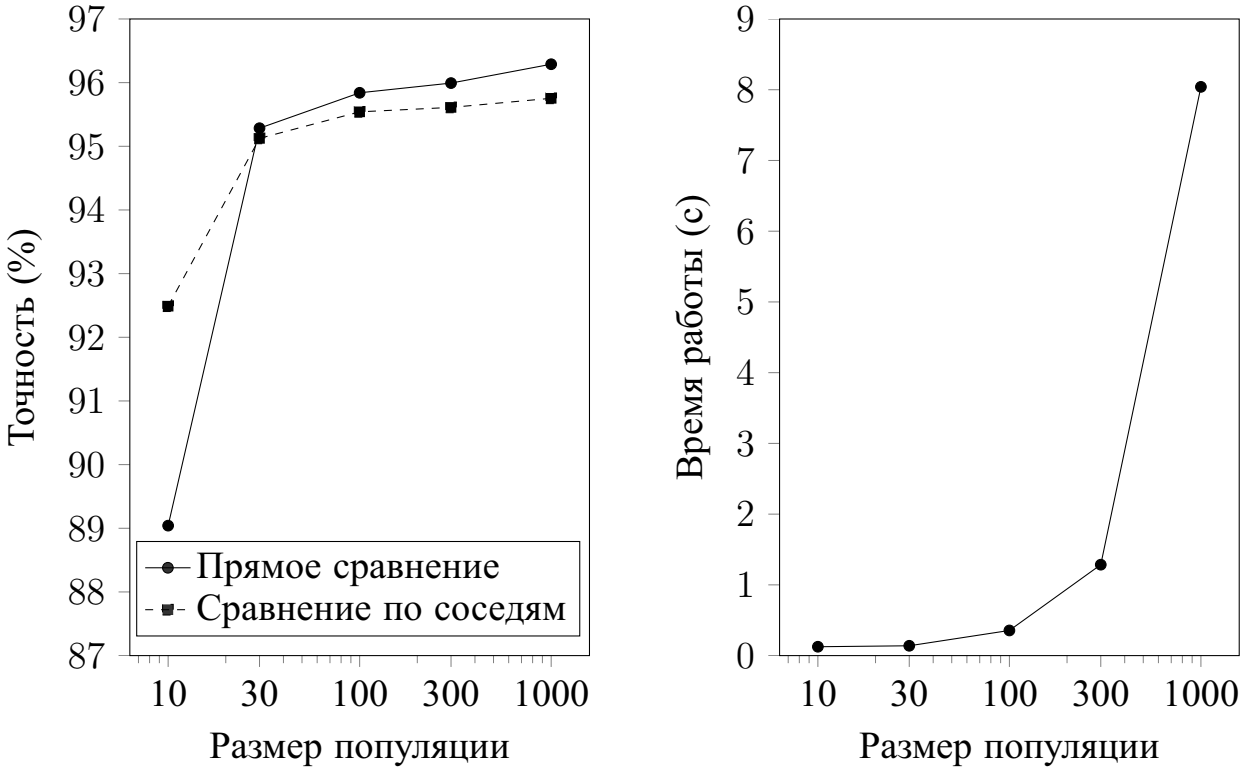

Рисунок 1 – Точность и время работы генетического алгоритма в зависимости от размера популяции на наборе изображений с $N = 432$ и $K = 28$

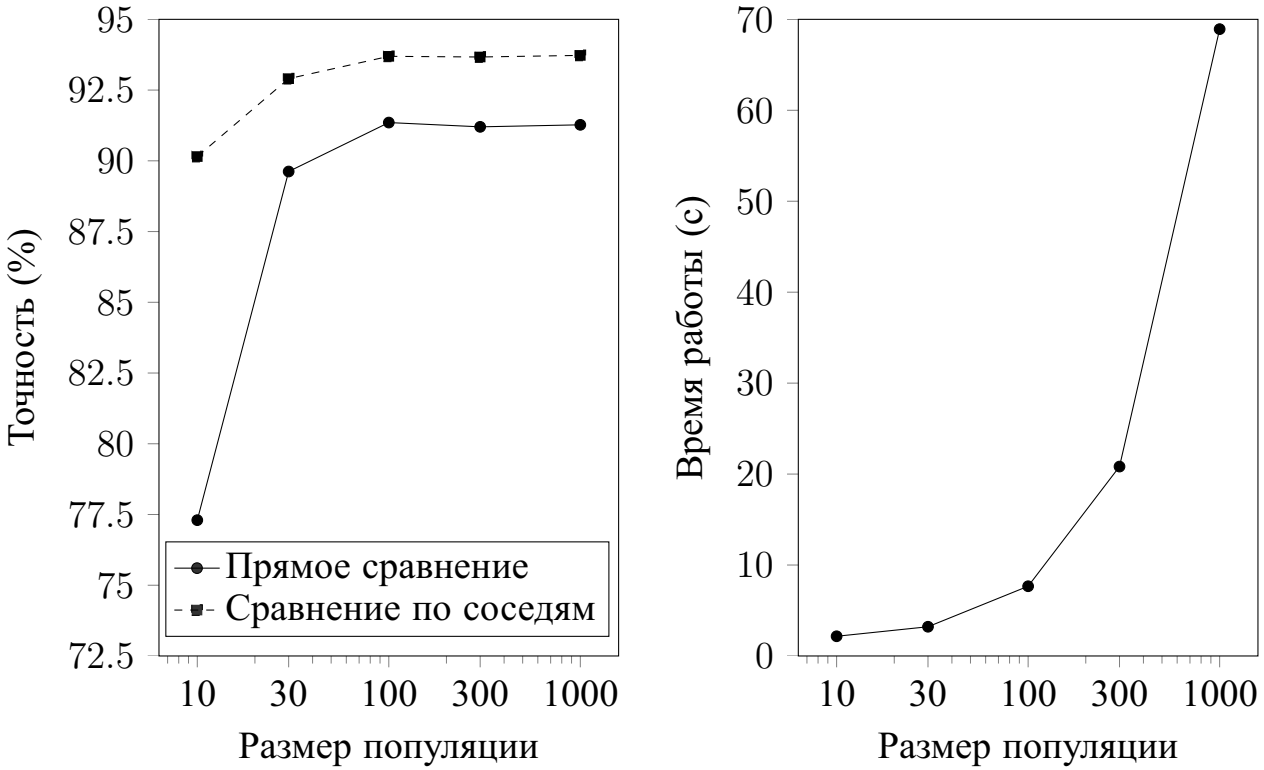

Рисунок 2 – Точность и время работы генетического алгоритма в зависимости от размера популяции на наборе изображений с $N = 3300$ и $K = 28$

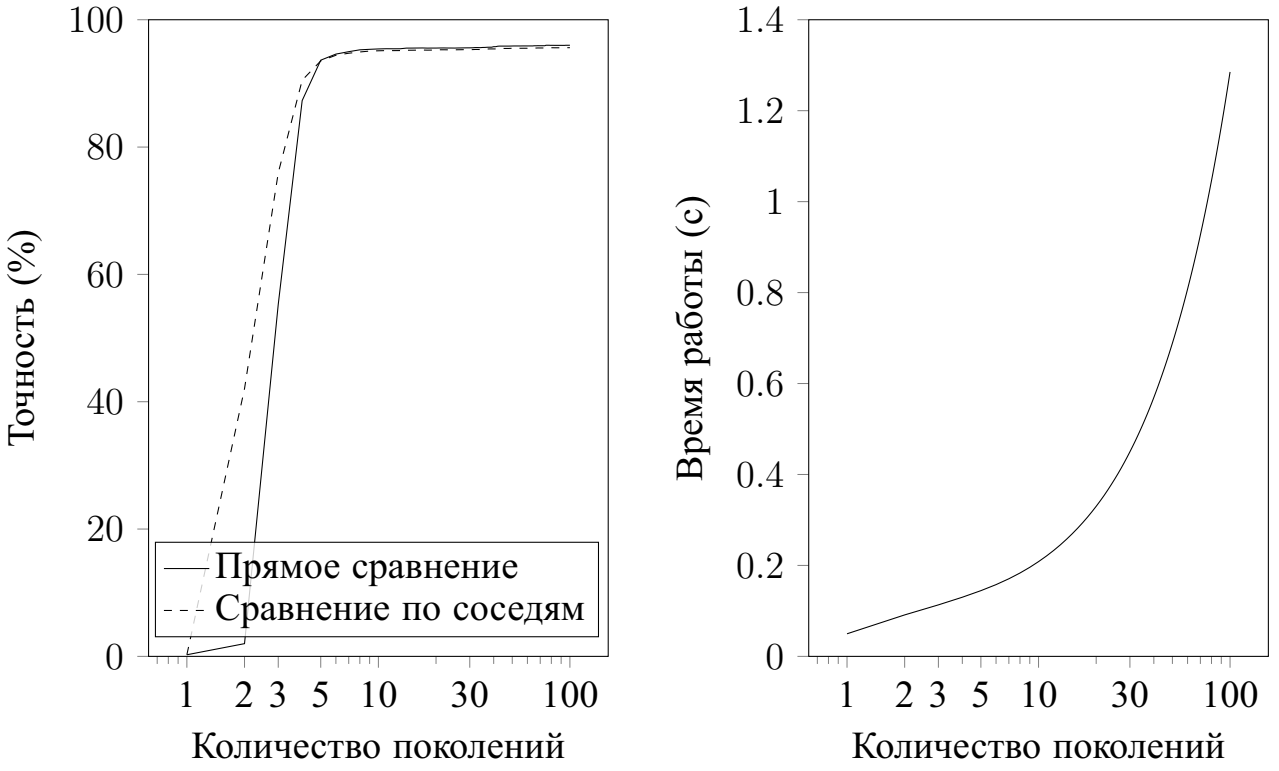

Рисунок 3 – Точность и время работы генетического алгоритма в зависимости от количества поколений на наборе изображений с $N = 432$ и $K = 28$

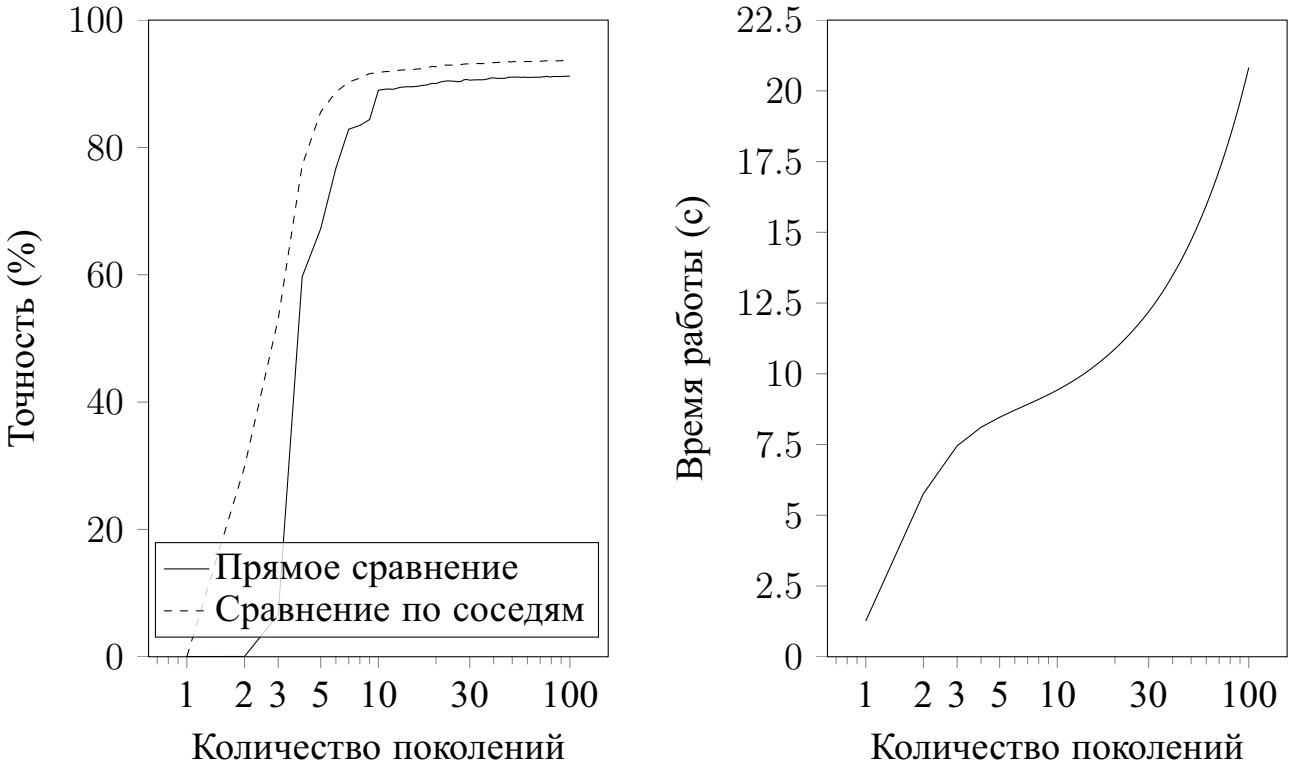

Рисунок 4 – Точность и время работы генетического алгоритма в зависимости от количества поколений на наборе изображений с $N = 3300$ и $K = 28$

Таблица 1 – Точность и время работы генетического алгоритма с методом сравнения деталей LAB SSD при параметрах $P_1 = 100$, $G_1 = 30$ и $P_2 = 300$, $G_2 = 100$

N	$P_1 = 100$, $G_1 = 30$			$P_2 = 300$, $G_2 = 100$		
	Прямое сравнение (%)	Ср. по соседям (%)	Время работы (с)	Прямое сравнение (%)	Ср. по соседям (%)	Время работы (с)
432	95.71 ± 0.62	95.39 ± 0.42	0.1464 ± 0.0095	95.99 ± 0.60	95.61 ± 0.41	1.2852 ± 0.0574
540	90.61 ± 0.93	93.25 ± 0.67	0.1963 ± 0.0125	90.80 ± 1.09	93.43 ± 0.54	1.5246 ± 0.0495
805	94.36 ± 1.00	95.42 ± 0.54	0.3394 ± 0.0170	95.06 ± 0.63	95.70 ± 0.38	2.1743 ± 0.0603
2360	86.31 ± 0.42	87.74 ± 0.57	2.7296 ± 0.0647	86.72 ± 0.58	88.13 ± 0.46	13.4461 ± 0.5043
3300	90.47 ± 0.96	92.83 ± 0.73	4.8070 ± 0.1348	91.20 ± 0.14	93.67 ± 0.21	20.8256 ± 0.5604

Таблица 2 – Точность и время работы генетического алгоритма с методом сравнения деталей MGC при параметрах $P_1 = 100$, $G_1 = 30$ и $P_2 = 300$, $G_2 = 100$

N	$P_1 = 100$, $G_1 = 30$			$P_2 = 300$, $G_2 = 100$		
	Прямое сравнение (%)	Ср. по соседям (%)	Время работы (с)	Прямое сравнение (%)	Ср. по соседям (%)	Время работы (с)
432	96.22 ± 0.20	95.72 ± 0.15	0.3228 ± 0.0218	96.31 ± 0.09	95.77 ± 0.09	1.4344 ± 0.0609
540	94.04 ± 0.73	96.29 ± 0.42	0.4312 ± 0.0326	94.06 ± 0.63	96.46 ± 0.32	1.7366 ± 0.0579
805	95.26 ± 0.39	95.90 ± 0.29	0.8580 ± 0.0248	95.37 ± 0.30	96.05 ± 0.26	2.6372 ± 0.0591
2360	93.07 ± 0.61	94.76 ± 0.40	7.5897 ± 0.0663	93.05 ± 0.92	94.97 ± 0.34	15.4857 ± 0.3672
3300	93.75 ± 0.44	96.62 ± 0.08	14.2809 ± 0.0771	93.71 ± 1.19	96.63 ± 0.17	25.9687 ± 0.4778

1.3.4 Сравнение всех алгоритмов

В таблице 3 приведены результаты работы алгоритма циклических ограничений с использованием метода сравнения деталей MGC в том же формате, что и в предыдущих таблицах. Можно видеть, что точность на 1-10% меньше по прямому сравнению, и на 1-5% меньше по сравнению по соседям, чем у генетического алгоритма. Для изображений с меньшим количеством деталей ($N \leqslant 805$) время работы приблизительно совпадает, а при большем N — больше в 1.5 раза.

В таблицах 4 и 5 представлены средняя точность (по методу сравнения по соседям) и время работы всех реализованных алгоритмов в данной работе и статьях [28], [24]. Генетический алгоритм не уступает алгоритму из статьи, а на больших изображениях — превосходит в качестве. Точность алгоритма циклических ограничений немного хуже. Оба алгоритма оказываются намного быстрее (на два порядка) алгоритмов из статей. Это можно объяснить уменьшенным размером популяции и количеством поколений, реализацией на более быстром языке программирования, множеством оптимизаций, а также параллелизацией и более производительным процессором. Также в таблице 6 приведено время выполнения алгоритмов при использовании программой 1 или 12 потоков. Можно видеть, что достигается ускорение в 4.5-5.5 раз при $N = 432$ и увеличивается до 8-8.5 раз для больших изображений, что свидетельствует о высокой эффективности распараллеливания.

Таким образом, наилучший результат по точности и времени работы достигается при использовании генетического алгоритма с методом сравнения деталей MGC, количеством поколений $G = 30$ и размером популяции $P = 100$. Результаты алгоритма на небольших изображениях близки к теоретическому максимуму (меньше 100%, так как невозможно определить положение нескольких полностью совпадающих деталей), и многие пазлы удаётся собрать полностью правильно.

Таблица 3 – Точность и время работы алгоритма циклических ограничений с методом сравнения деталей MGC

N	Прямое сравнение (%)	Сравнение по соседям (%)	Время работы (с)
432	94.98 ± 0.38	94.69 ± 0.12	0.3543 ± 0.0585
540	83.78 ± 0.99	92.67 ± 0.21	0.4745 ± 0.0144
805	88.73 ± 0.28	92.66 ± 0.33	0.9435 ± 0.0290
2360	88.12 ± 1.18	90.15 ± 0.87	11.6076 ± 0.2145
3300	88.92 ± 0.00	94.06 ± 0.03	23.6131 ± 0.0596

Таблица 4 – Точность (по методу сравнения по соседям, %) всех алгоритмов

N	Генетический алгоритм (работа [28])	Алгоритм циклических ограничений (работа [24])	Генетический алгоритм (данная работа)	Алгоритм циклических ограничений (данная работа)
432	96.16	95.5	95.72	94.69
540	95.96	95.2	96.29	92.67
805	96.26	94.9	95.90	92.66
2360	88.86	96.4	94.76	90.15
3300	92.76	96.4	96.62	94.06

Таблица 5 – Время работы (в секундах) всех алгоритмов

N	Генетический алгоритм (работа [28])	Алгоритм циклических ограничений (работа [24])	Генетический алгоритм (данная работа)	Алгоритм циклических ограничений (данная работа)
432	48.73	140	0.3228	0.3543
540	64.06	Н/Д	0.4312	0.4745
805	116.18	Н/Д	0.8580	0.9435
2360	1056	Н/Д	7.5897	11.6076
3300	1814.4	Н/Д	14.2809	23.6131

Таблица 6 – Время работы (в секундах) и ускорение алгоритмов при использовании 1 и 12 потоков

N	Генетический алгоритм			Алг. циклических ограничений		
	Время работы (1 поток)	Время работы (12 потоков)	Ускорение	Время работы (1 поток)	Время работы (12 потоков)	Ускорение
432	1.7260	0.3228	5.35	1.6512	0.3543	4.66
540	2.7940	0.4312	6.48	2.8290	0.4745	5.96
805	7.3459	0.8580	8.56	7.5731	0.9435	8.03
2360	63.7986	7.5897	8.41	95.5544	11.6076	8.23
3300	122.6072	14.2809	8.59	191.7477	23.6131	8.12

2 Детали реализации

Проект состоит из двух частей: библиотеки, содержащей описанные выше алгоритмы и вспомогательные функции, и приложения, реализующего графический интерфейс для выполнения этих алгоритмов. Такая модульная архитектура позволяет отделить интерфейс от логики алгоритмов. Для написания программы был выбран язык Rust из-за сочетания высокой производительности и безопасной работы с памятью и потоками [30]. В качестве зависимостей используются несколько библиотек:

- `iced` — кроссплатформенный фреймворк для создания графического интерфейса;
- `iced_aw` — дополнительные виджеты для `iced`;
- `native-dialog` — кроссплатформенная библиотека для создания диалоговых окон открытия файлов и папок, сохранения файлов;
- `image` — библиотека для работы с изображениями;
- `lab` — библиотека работы с цветовым пространством L*a*b*, используется для конвертации из RGB;
- `alphanumeric-sort` — библиотека для сортировки названий файлов в естественном порядке;
- `rand` — библиотека для генерации случайных чисел;
- `rand_xoshiro` — реализация быстрого генератора случайных чисел Xoshiro;
- `float-ord` — обёртка над числами с плавающей запятой, позволяющая их сортировать;
- `rayon` — библиотека параллелизма данных;
- `indexmap` — реализация хеш-таблицы, позволяющей индексирование;
- `fxhash` — быстрый, небезопасный алгоритм хеширования;
- `dark-light` — библиотека для определения предпочитаемой темы (светлой или тёмной).

Также для обработки статистики выполнения алгоритмов, создания таблиц и графиков использовался язык Python в интерактивной среде Jupyter Notebook и библиотеки:

- `pandas` для обработки и анализа данных;
- `matplotlib` и `seaborn` для создания графиков;
- `tikzplotlib` для экспорта графиков в формат PGF/TikZ.

2.1 Библиотека

Библиотека состоит из четырёх модулей: `lib`, `genetic_algorithm`, `loop_constraints_algorithm` и `image_processing`.

В модуле `lib` определён тип для решения пазла — двумерный (представленный как одномерный) массив, содержащий детали — пары из строк и столбцов; приведены функции сравнения деталей — `calculate_lab_ssd` (приведена в приложении А, описана в разделе 1.1.1) и `calculate_mgc` (приведена в приложении Б, описана в разделе 1.1.2); функции оценки решений, описанные в разделе 1.3 — `image_direct_comparison` (прямое сравнение) и `image_neighbour_comparison` (сравнение по соседям); и вспомогательные функции для работы с решениями — `generate_random_solution` для создания случайного решения, `apply_permutation_to_solution` для применения перестановки к решению, `solution_compatibility` для вычисления суммы несовместимостей всех смежных деталей в решении.

В модулях `genetic_algorithm` (код приведён в приложении В) и `loop_constraints_algorithm` (код приведён в приложении Г) реализованы алгоритмы сборки пазлов — генетический и циклических ограничений. Их реализация была подробно описана в разделах 1.2.1 и 1.2.2.

В модуле `image_processing` приведены вспомогательные функции для работы с изображениями: `get_image_handle` для преобразования из формата RGBA в BGRA и создания указателя на изображение для фреймворка `iced`, `get_solution_image` для получения изображения решения с возможностью отображения некорректных деталей, `get_rgb_image` для преобразования из формата RGBA в RGB и `get_lab_image` для преобразования в цветовое пространство L*a*b*.

2.2 Приложение

Приложение состоит из двух основных модулей `app` и `algorithms_async`, а также `main`, в котором соответствующая функция создаёт и запускает приложение.

Модуль `app` реализует интерфейс и его логику. В нём определены структуры `AppState` для состояния приложения (выбранные значения в интерфейсе, состояния загрузки изображений и выполнения алгоритмов), `ErrorModalState` для состояния окна сообщения об ошибке, а также пе-

речисления AppMessage для сообщений от интерфейса (нажатия кнопок, изменения текстовых полей) и ErrorModalMessage для сообщений от окна для ошибок. Для AppState реализованы функции update_with_result для обработки сообщений, load_images_start для открытия окна выбора папки с изображениями, save_results для сохранения статистики работы алгоритмов в файл, save_image для сохранения текущего изображения в файл, algorithm_start для запуска выбранного алгоритма и load_selected_image для обновления отображаемого изображения.

Также app содержит подмодули app_ui, images_loader и style. В app_ui определена структура AppUIState, сохраняющая состояния виджетов, для которой реализованы функции new (создание структуры и определение темы интерфейса), reset_state (сброс состояния выбранного изображения); а также приведена функция view для AppState, генерирующая интерфейс для текущего состояния. В подмодуле style заданы темы: светлая (стандартная) и тёмная (реализована как подмодуль). В images_loader реализована асинхронная загрузка изображений: определены структуры LoadImagesData (данные текущего состояния — пути, изображения, их имена), LoadImagesResponseData (данные результата одного этапа загрузки) и перечисления LoadImagesRequest (запрос на определение путей всех файлов в папке или загрузку одного изображения), LoadImagesResponse (результат запроса — все пути или изображение), LoadImagesState (текущее состояние — не загружено, подготовка путей, загрузка определённого изображения или завершено), LoadImagesMessage (сообщение о результате для интерфейса). Также создана функция load_images_next, асинхронно выполняющая следующий этап загрузки изображений по заданному запросу.

Модуль algorithms_async позволяет асинхронно работать с реализованными алгоритмами сборки пазлов, содержащихся в библиотеке. Модуль содержит перечисления Algorithm и CompatibilityMeasure, описывающие два алгоритма и два метода сравнения деталей, а также AlgorithmData (текущее состояние алгоритма), AlgorithmDataRequest (данные запроса на выполнение алгоритма), AlgorithmDataResponse (данные результата выполнения), AlgorithmState (состояние алгоритма), AlgorithmMessage (сообщение о результате выполнения), AlgorithmError (ошибка выполнения алгоритма). Для них реализованы вспомогательные функции, перена-

правляющие вызовы к соответствующим структурам алгоритмов. Также реализована функция `algorithm_next`, вызывающая следующий этап алгоритма: для генетического — создание нового поколения, для алгоритма циклических ограничений — обработка нового изображения. В подмодулях `genetic_algorithm` и `loop_constraints_algorithm` для каждого алгоритма созданы структуры и функции, соответствующие перечислениям и функциям из основного модуля.

2.3 Интерфейс

Интерфейс приложения состоит из одного окна, как показано на рисунке 5. В левой части находится кнопка загрузки изображений из папки, переключатели для выбора алгоритма сборки и метода сравнения деталей, поля для ввода параметров алгоритма, кнопки запуска, сохранения результатов (статистики выполнения) алгоритма, и сохранения текущего отображаемого изображения. Снизу отображается текущий статус приложения. После запуска возможно только одно действие — выбор папки с изображениями. После их загрузки в правой части появится их список с миниатюрами и названиями файлов (рисунок 6). Далее можно настроить необходимые параметры и запустить алгоритм, причём в строке статуса будет отображаться номер обрабатываемого изображения (и поколения для генетического алгоритма). После завершения работы кнопки изображений и сохранения результатов станут доступны. При выборе какого-либо изображения можно просмотреть результат сборки (для генетического алгоритма — лучший результат по каждому поколению) в центре окна, причём возможно включить отображение неправильных деталей по одному из способов (прямое сравнение или по соседям), также снизу выводится точность сборки по каждому методу (рисунок 7). Для обработки другими программами можно сохранить CSV-таблицу с точностью и временем работы для каждого изображения и поколения. Приложение поддерживает светлую и тёмную темы — можно выбрать с помощью переменной окружения `THEME` (значения `light` и `dark`), а при её отсутствии выбирается системная тема. Также возможно использование переменной окружения `RAYON_NUM_THREADS` для изменения количества доступных для алгоритмов потоков выполнения.

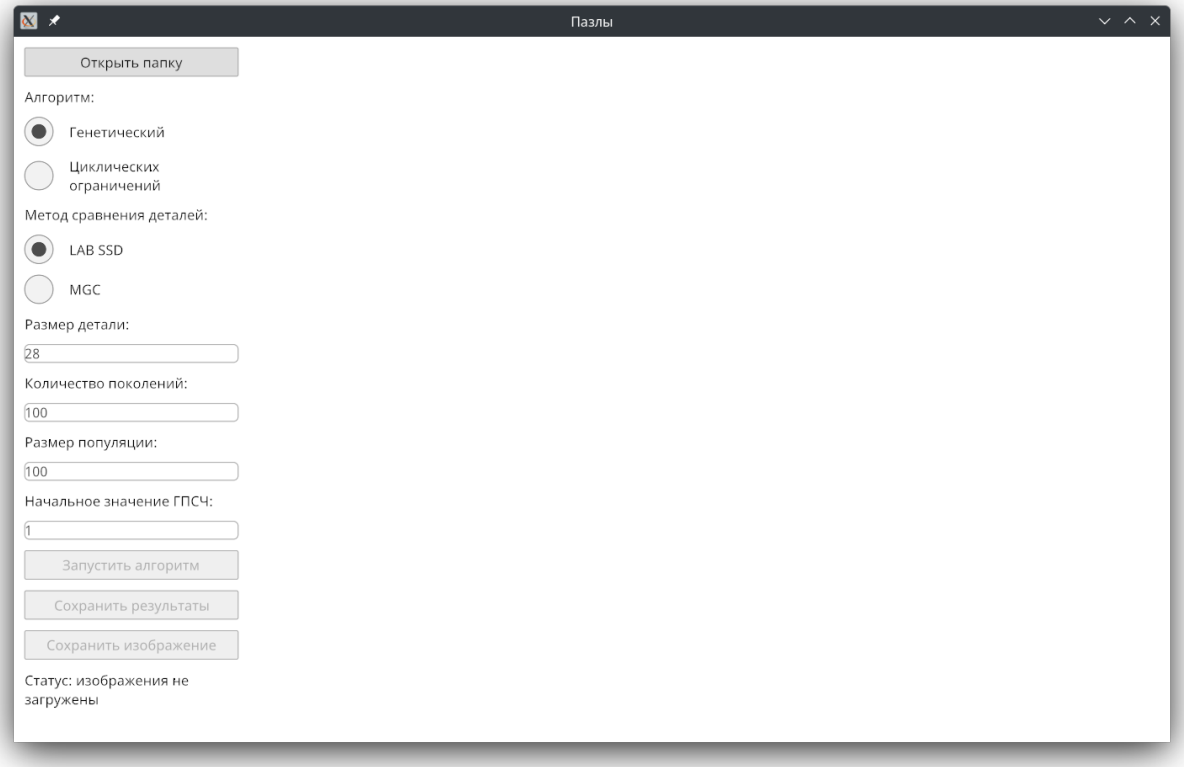

Рисунок 5 – Окно приложения после запуска

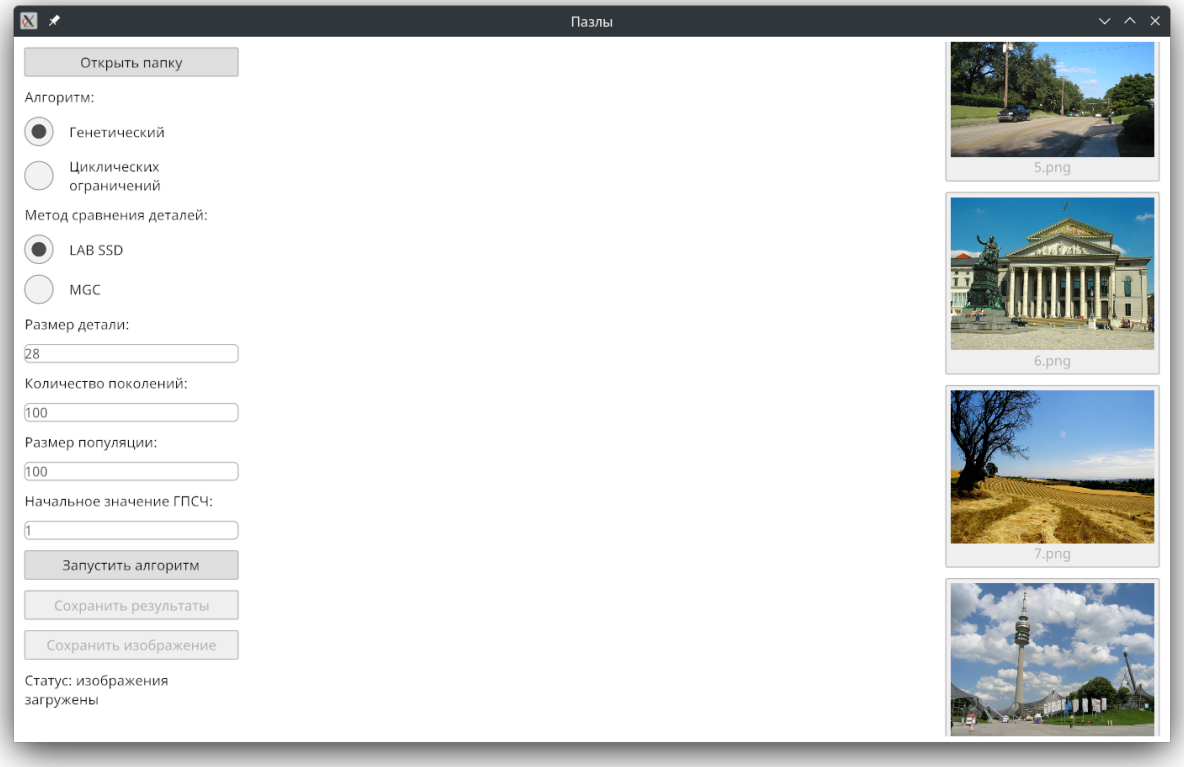

Рисунок 6 – Окно приложения после загрузки изображений

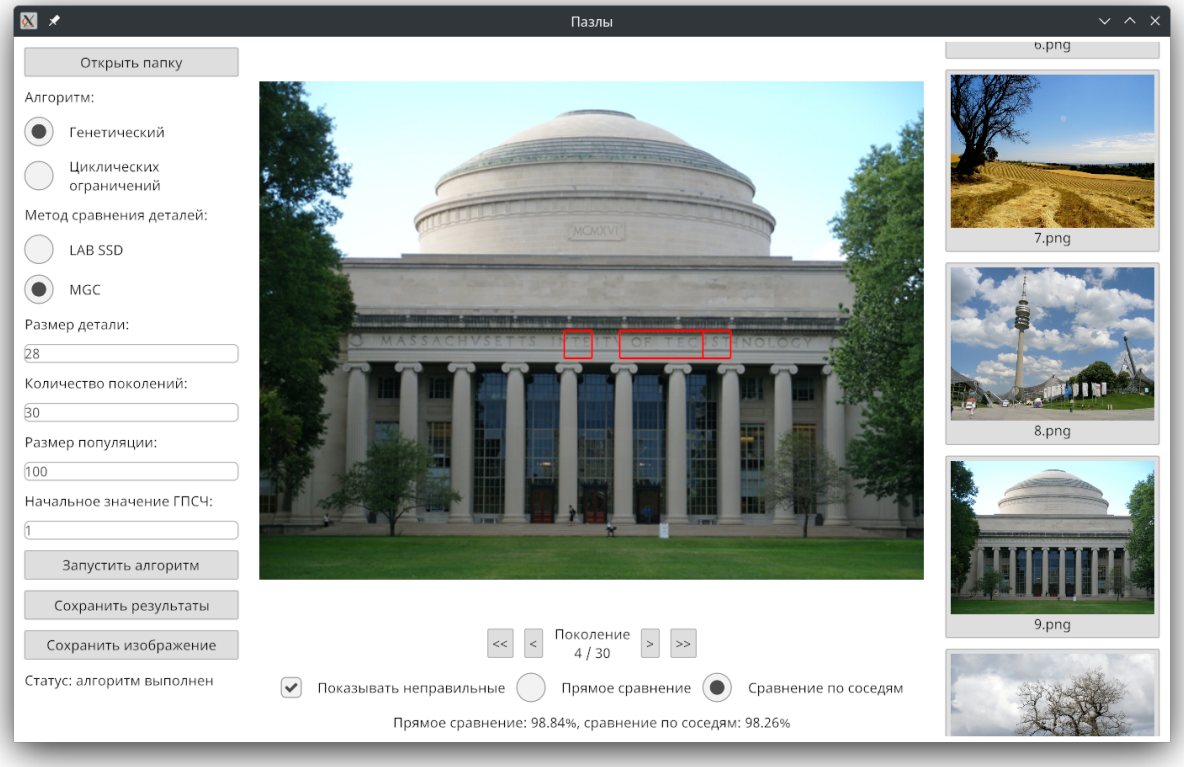

Рисунок 7 – Окно приложения после выполнения алгоритма — просмотр результатов

ЗАКЛЮЧЕНИЕ

В данной работе создано приложение для запуска и отображения результатов работы алгоритмов автоматической сборки пазлов. Была изучена литература и выбраны два лучших метода сравнения деталей: LAB SSD и MGC, и два алгоритма сборки: генетический и циклических ограничений; для них были созданы оптимизированные реализации. После проведён выбор оптимальных параметров генетического алгоритма, сравнение SSD и MGC по времени работы и качеству, а затем итоговое сравнение алгоритмов на разнообразных наборах изображений по двум способам оценки точности результатов и по затраченному времени, в том числе для однопоточных и параллельных версий. Выяснилось, что удалось добиться высокого качества сборки и значительно меньшего времени выполнения, чем в известных статьях.

В ходе написания работы были решены следующие задачи:

- реализованы различные методы сравнения деталей;
- изучены и реализованы два алгоритма сборки пазлов;
- создано приложение для выполнения и визуализации результатов работы алгоритмов;
- найдены методы оценки результатов;
- проведено сравнение реализованных алгоритмов и методов сравнения деталей по критериям точности и времени работы.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. *Willis A. R.*, *Cooper D. B.* Computational reconstruction of ancient artifacts // IEEE Signal Processing Magazine. — 2008. — Vol. 25, no. 4. — P. 65–83.
2. *Son K.*, *Almeida E. B.*, *Cooper D. B.* Axially symmetric 3D pots configuration system using axis of symmetry and break curve // Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition. — 2013. — P. 257–264.
3. *Zhu L.*, *Zhou Z.*, *Hu D.* Globally consistent reconstruction of ripped-up documents // IEEE Transactions on pattern analysis and machine intelligence. — 2007. — Vol. 30, no. 1. — P. 1–13.
4. *Cao S.*, *Liu H.*, *Yan S.* Automated assembly of shredded pieces from multiple photos // 2010 IEEE International Conference on Multimedia and Expo. — IEEE. 2010. — P. 358–363.
5. *Garfinkel S. L.* Digital forensics research: The next 10 years // Digital Investigation. — 2010. — Vol. 7. — S64–S73.
6. *Memon N.*, *Pal A.* Automated reassembly of file fragmented images using greedy algorithms // IEEE transactions on image processing. — 2006. — Vol. 15, no. 2. — P. 385–393.
7. *Cho T. S.*, *Avidan S.*, *Freeman W. T.* The patch transform // IEEE transactions on pattern analysis and machine intelligence. — 2009. — Vol. 32, no. 8. — P. 1489–1501.
8. A puzzle solver and its application in speech descrambling / Y.-X. Zhao [et al.] // WSEAS International Conference on Computer Engineering and Applications. — Citeseer. 2007. — P. 171–176.
9. *Freeman H.*, *Garder L.* Apictorial jigsaw puzzles: The computer solution of a problem in pattern recognition // IEEE Transactions on Electronic Computers. — 1964. — No. 2. — P. 118–127.
10. Solving jigsaw puzzles by computer / H. Wolfson [et al.] // Annals of Operations Research. — 1988. — Vol. 12, no. 1. — P. 51–64.
11. *Cho T. S.*, *Avidan S.*, *Freeman W. T.* A probabilistic image jigsaw puzzle solver // 2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition. — IEEE. 2010. — P. 183–190.

12. *Pomeranz D.*, *Shemesh M.*, *Ben-Shahar O.* A fully automated greedy square jigsaw puzzle solver // CVPR 2011. — IEEE. 2011. — P. 9–16.

13. *Demaine E. D.*, *Demaine M. L.* Jigsaw puzzles, edge matching, and polyomino packing: Connections and complexity // Graphs and Combinatorics. — 2007. — Vol. 23, no. 1. — P. 195–208.

14. An automatic jigsaw puzzle solver / D. A. Kosiba [et al.] // Proceedings of 12th International Conference on Pattern Recognition. Vol. 1. — IEEE. 1994. — P. 616–618.

15. *Yang X.*, *Adluru N.*, *Latecki L. J.* Particle filter with state permutations for solving image jigsaw puzzles // CVPR 2011. — IEEE. 2011. — P. 2873–2880.

16. *Alajlan N.* Solving square jigsaw puzzles using dynamic programming and the hungarian procedure // American Journal of Applied Sciences. — 2009. — Vol. 6, no. 11. — P. 1941.

17. *Makridis M.*, *Papamarkos N.* A new technique for solving a jigsaw puzzle // 2006 International Conference on Image Processing. — IEEE. 2006. — P. 2001–2004.

18. *Nielsen T. R.*, *Drewsen P.*, *Hansen K.* Solving jigsaw puzzles using image features // Pattern Recognition Letters. — 2008. — Vol. 29, no. 14. — P. 1924–1933.

19. *Sagiroglu M. S.*, *Erçil A.* A texture based matching approach for automated assembly of puzzles // 18th International Conference on Pattern Recognition (ICPR'06). Vol. 3. — IEEE. 2006. — P. 1036–1041.

20. *Yao F.-H.*, *Shao G.-F.* A shape and image merging technique to solve jigsaw puzzles // Pattern Recognition Letters. — 2003. — Vol. 24, no. 12. — P. 1819–1835.

21. *Criminisi A.*, *Perez P.*, *Toyama K.* Object removal by exemplar-based inpainting // 2003 IEEE Computer Society Conference on Computer Vision and Pattern Recognition, 2003. Proceedings. Vol. 2. — IEEE. 2003. — P. II–II.

22. *Gallagher A. C.* Jigsaw puzzles with pieces of unknown orientation // 2012 IEEE Conference on Computer Vision and Pattern Recognition. — IEEE. 2012. — P. 382–389.

23. *Sholomon D.*, *David O.*, *Netanyahu N.* A generalized genetic algorithm-based solver for very large jigsaw puzzles of complex types // Proceedings of the AAAI Conference on Artificial Intelligence. Vol. 28. — 2014.

24. *Son K.*, *Hays J.*, *Cooper D. B.* Solving square jigsaw puzzles with loop constraints // European Conference on Computer Vision. — Springer. 2014. — P. 32–46.

25. *Weiss Y.*, *Freeman W. T.* What makes a good model of natural images? // 2007 IEEE Conference on Computer Vision and Pattern Recognition. — IEEE. 2007. — P. 1–8.

26. *Jain A. K.* Fundamentals of digital image processing. — Prentice-Hall, Inc., 1989. — ISBN 0-13-336165-9.

27. Assembly of puzzles using a genetic algorithm / F. Toyama [et al.] // Object recognition supported by user interaction for service robots. Vol. 4. — IEEE. 2002. — P. 389–392.

28. *Sholomon D.*, *David O. E.*, *Netanyahu N. S.* An automatic solver for very large jigsaw puzzles using genetic algorithms // Genetic Programming and Evolvable Machines. — 2016. — Vol. 17, no. 3. — P. 291–313.

29. *Емельянов В. В.*, *Курейчик В. В.*, *Курейчик В. М.* Теория и практика эволюционного моделирования. — М.: Физматлит, 2003. — 432 с. — ISBN 5-9221-0337-7.

30. *Блэнди Д.*, *Орендорф Д.* Программирование на языке Rust / пер. с анг. А. А. Слинкина. — М.: ДМК Пресс, 2018. — 550 с. — ISBN 978-5-97060-236-2.

ПРИЛОЖЕНИЕ А

Программный код метода сравнения деталей SSD

```
// Вычисление SSD (sum of squared differences, сумма квадратов разностей)
// в цветовом пространстве L*a*b* для деталей
pub fn calculate_lab_ssd(
  image: &RgbaImage,
  img_width: usize,
  img_height: usize,
  piece_size: usize,
) -> [Vec<Vec<f32>>; 2] {
  // Пиксели изображения в цветовом пространстве L*a*b*
  let lab_pixels = get_lab_image(image);
  // Ширина изображения в пикселях
  let image_width = img_width * piece_size;

  // Несходство пикселей - разность цветов в пространстве L*a*b*
  let two_pixels_dissimilarity =
    |i: usize, j: usize| lab_pixels[i].squared_distance(&lab_pixels[j]);

  // Вычисление несходства детали i (строка i_r, столбец i_c),
  // находящейся слева от детали j (строка j_r, столбец j_c)
  let right_dissimilarity: Vec<Vec<f32>> = (0..img_height)
    .into_par_iter()
    .flat_map_iter(|i_r| {
      (0..img_width)
        .map(|i_c| {
          (0..img_height)
            .flat_map(|j_r| {
              (0..img_width)
                .map(|j_c| {
                  (0..piece_size)
                    .map(|k| {
                      two_pixels_dissimilarity(
                        // k-й элемент последнего столбца i-й детали
                        (i_r * piece_size + k) * image_width
                          + i_c * piece_size
                          + piece_size
                          - 1,
                        // k-й элемент первого столбца j-й детали
                        (j_r * piece_size + k) * image_width
                          + j_c * piece_size,
                      )
                    })
                    .sum::<f32>()
                    .sqrt()
                })
                .collect::<Vec<_>>()
```

```
            })
            .collect()
        })
        .collect::<Vec<_>>()
    })
    .collect();
  // Вычисление несходства детали i (строка i_r, столбец i_c),
  // находящейся сверху от детали j (строка j_r, столбец j_c)
  let down_dissimilarity: Vec<Vec<f32>> = (0..img_height)
    .into_par_iter()
    .flat_map_iter(|i_r| {
      (0..img_width)
        .map(|i_c| {
          (0..img_height)
            .flat_map(|j_r| {
              (0..img_width)
                .map(|j_c| {
                  (0..piece_size)
                    .map(|k| {
                      two_pixels_dissimilarity(
                        // k-й элемент последней строки i-й детали
                        (i_r * piece_size + piece_size - 1) * image_width
                          + i_c * piece_size
                          + k,
                        // k-й элемент первой строки j-й детали
                        j_r * piece_size * image_width
                          + j_c * piece_size
                          + k,
                      )
                    })
                    .sum::<f32>()
                    .sqrt()
                })
                .collect::<Vec<_>>()
            })
            .collect()
        })
        .collect::<Vec<_>>()
    })
    .collect();
  [right_dissimilarity, down_dissimilarity]
}
```

ПРИЛОЖЕНИЕ Б

Программный код метода сравнения деталей MGC

```
// Вычисление MGC (Mahalanobis Gradient Compatibility, совместимость градиентов Махаланобиса) для деталей
pub fn calculate_mgc(
  image: &RgbaImage,
  img_width: usize,
  img_height: usize,
  piece_size: usize,
) -> [Vec<Vec<f32>>; 2] {
  // Пиксели изображения в цветовом пространстве RGB
  let rgb_pixels = get_rgb_image(image);
  // Ширина изображения в пикселях
  let image_width = img_width * piece_size;

  // Вычисление MGC для одной стороны (по градиентам в одной из деталей и градиенту между ними)
  let calc_mgc_part = |grad_side: Vec<[f32; 3]>, grad_mid: Vec<[f32; 3]>| {
    const EPS: f32 = 1e-6;

    let sz = grad_side.len() as f32;
    // Средний градиент по каждому цветовому каналу
    let means = [0, 1, 2].map(|i| grad_side.iter().map(|v| v[i]).sum::<f32>() / sz);
    // Коэффициент ковариации
    let cov_denom = 1.0 / (sz - 1.0);
    // Вычисление ковариации для разных величин
    let get_cov = |i, j| {
      grad_side
        .iter()
        .map(|v| (v[i] - means[i]) * (v[j] - means[j]))
        .sum::<f32>()
        * cov_denom
    };
    // Вычисление ковариации величины с самой собой (дисперсии)
    let get_cov_sqr = |i: usize| {
      grad_side
        .iter()
        .map(|v| (v[i] - means[i]).powi(2))
        .sum::<f32>()
        * cov_denom
    };
    // Матрица ковариаций между градиентами по каждому цветовому каналу:
    // [[a b c]]
    // [[d e f]]
    // [[g h i]]
    // Так как она симметрична, то d = b, g = c, h = f, и они не вычисляются
    // К элементам на диагонали добавляется EPS для численной стабильности инвертирования матрицы
```

```
    let a = get_cov_sqr(0) + EPS;
    let e = get_cov_sqr(1) + EPS;
    let i = get_cov_sqr(2) + EPS;
    let b = get_cov(0, 1);
    let c = get_cov(0, 2);
    let f = get_cov(1, 2);
    // Вычисление обратной к матрице ковариаций:
    //               1                    [[(ei - fh) (ch - bi) (bf - ce)]]
    // ---------------------------------  [[(fg - di) (ai - cg) (cd - af)]]
    // a * ei_ff + b * cf_bi + c * bf_ce  [[(dh - eg) (bg - ah) (ae - bd)]]
    // Элементы матрицы (d, g, h заменены)
    let ei_ff = e * i - f * f;
    let cf_bi = c * f - b * i;
    let bf_ce = b * f - c * e;
    let bc_af = b * c - a * f;
    let ai_cc = a * i - c * c;
    let ae_bb = a * e - b * b;
    // Коэффициент обратной матрицы
    let denom = 1.0 / (a * ei_ff + b * cf_bi + c * bf_ce);
    // Умножение на коэффициент и на 2 (при необходимости для следующих формул)
    let ei_ff = ei_ff * denom;
    let cf_bi = 2.0 * (cf_bi * denom);
    let bf_ce = 2.0 * (bf_ce * denom);
    let bc_af = 2.0 * (bc_af * denom);
    let ai_cc = ai_cc * denom;
    let ae_bb = ae_bb * denom;
    // Разность между градиентом из одной детали в другую и средним градиентом в детали
    let grad_diff: Vec<_> = grad_mid
      .iter()
      .map(|v| [v[0] - means[0], v[1] - means[1], v[2] - means[2]])
      .collect();
    // Вычисление результата (. - умножение матриц):
    // sz
    // SUM grad_diff[i] . covariations_inv . (grad_diff[i])^T
    // i=0
    // Можно преобразовать как сумму элементов матрицы вида (* - поточечное умножение):
    // grad_diff . covariations_inv * grad_diff
    // Эта формула вычисляется построчно для grad_diff и подставляются элементы матрицы,
    // обратной к матрице ковариаций
    let res = grad_diff
      .iter()
      .map(|v| {
        (v[0] * ei_ff + v[1] * cf_bi + v[2] * bf_ce) * v[0]
          + (v[1] * ai_cc + v[2] * bc_af) * v[1]
          + ae_bb * v[2] * v[2]
      })
      .sum::<f32>();
    f32::max(0.0, res).sqrt()
```

```
92  };
93
94  // Вычисление MGC для детали i (строка i_r, столбец i_c),
95  // находящейся слева от детали j (строка j_r, столбец j_c)
96  let right_mgc: Vec<Vec<f32>> = (0..img_height)
97    .into_par_iter()
98    .flat_map_iter(|i_r| {
99      (0..img_width)
100       .map(|i_c| {
101         (0..img_height)
102           .flat_map(|j_r| {
103             (0..img_width)
104               .map(|j_c| {
105                 // Градиент из предпоследнего столбца i-й детали в её последний столбец
106                 let grad_l = (0..piece_size)
107                   .map(|k| {
108                     [0, 1, 2].map(|c| {
109                       (rgb_pixels[(i_r * piece_size + k) * image_width
110                         + i_c * piece_size
111                         + piece_size
112                         - 1][c]
113                         - rgb_pixels[(i_r * piece_size + k)
114                           * image_width
115                           + i_c * piece_size
116                           + piece_size
117                           - 2][c])
118                         as f32
119                     })
120                   })
121                   .collect::<Vec<_>>();
122                 // Градиент из последнего столбца i-й детали в первый столбец j-й детали
123                 let grad_lr = (0..piece_size)
124                   .map(|k| {
125                     [0, 1, 2].map(|c| {
126                       (rgb_pixels[(j_r * piece_size + k) * image_width
127                         + j_c * piece_size][c]
128                         - rgb_pixels[(i_r * piece_size + k)
129                           * image_width
130                           + i_c * piece_size
131                           + piece_size
132                           - 1][c])
133                         as f32
134                     })
135                   })
136                   .collect::<Vec<_>>();
137                 // Градиент из первого столбца j-й детали в последний столбец i-й детали
138                 let grad_rl = grad_lr
139                   .iter()
```

```
                    .map(|x: &[f32; 3]| [-x[0], -x[1], -x[2]])
                    .collect::<Vec<_>>();
                  // Градиент из второго столбца j-й детали в её первый столбец
                  let grad_r = (0..piece_size)
                    .map(|k| {
                      [0, 1, 2].map(|c| {
                        (rgb_pixels[(j_r * piece_size + k) * image_width
                          + j_c * piece_size][c]
                          - rgb_pixels[(j_r * piece_size + k)
                            * image_width
                            + j_c * piece_size
                            + 1][c])
                          as f32
                      })
                    })
                    .collect::<Vec<_>>();
                  // Вычисление MGC как суммы MGC слева направо и справа налево
                  calc_mgc_part(grad_l, grad_lr) + calc_mgc_part(grad_r, grad_rl)
                })
                .collect::<Vec<_>>()
            })
            .collect()
        })
        .collect::<Vec<_>>()
    })
    .collect();
  // Вычисление MGC для детали i (строка i_r, столбец i_c),
  // находящейся сверху от детали j (строка j_r, столбец j_c)
  let down_mgc: Vec<Vec<f32>> = (0..img_height)
    .into_par_iter()
    .flat_map_iter(|i_r| {
      (0..img_width)
        .map(|i_c| {
          (0..img_height)
            .flat_map(|j_r| {
              (0..img_width)
                .map(|j_c| {
                  // Градиент из предпоследней строки i-й детали в её последнюю строку
                  let grad_u = (0..piece_size)
                    .map(|k| {
                      [0, 1, 2].map(|c| {
                        (rgb_pixels[(i_r * piece_size + piece_size - 1)
                          * image_width
                          + i_c * piece_size
                          + k][c]
                          - rgb_pixels[(i_r * piece_size + piece_size
                            - 2)
                            * image_width
```

```
                        + i_c * piece_size
                        + k][c])
                    as f32
                })
            })
            .collect::<Vec<_>>();
        // Градиент из последней строки i-й детали в первую строку j-й детали
        let grad_ud = (0..piece_size)
            .map(|k| {
                [0, 1, 2].map(|c| {
                    (rgb_pixels[j_r * piece_size * image_width
                        + j_c * piece_size
                        + k][c]
                        - rgb_pixels[(i_r * piece_size + piece_size
                            - 1)
                            * image_width
                            + i_c * piece_size
                            + k][c])
                        as f32
                })
            })
            .collect::<Vec<_>>();
        // Градиент из первого столбца j-й детали в последний столбец i-й детали
        let grad_du = grad_ud
            .iter()
            .map(|x: &[f32; 3]| [-x[0], -x[1], -x[2]])
            .collect::<Vec<_>>();
        // Градиент из второй строки j-й детали в её первую строку
        let grad_d = (0..piece_size)
            .map(|k| {
                [0, 1, 2].map(|c| {
                    (rgb_pixels[j_r * piece_size * image_width
                        + j_c * piece_size
                        + k][c]
                        - rgb_pixels[(j_r * piece_size + 1)
                            * image_width
                            + j_c * piece_size
                            + k][c])
                        as f32
                })
            })
            .collect::<Vec<_>>();
        // Вычисление MGC как суммы MGC сверху вниз и снизу вверх
        calc_mgc_part(grad_u, grad_ud) + calc_mgc_part(grad_d, grad_du)
    })
    .collect::<Vec<_>>()
})
.collect()
```

```
236             })
237             .collect::<Vec<_>>()
238         })
239         .collect();
240     [right_mgc, down_mgc]
241 }
```

ПРИЛОЖЕНИЕ В

Программный код генетического алгоритма

```
const ELITISM_COUNT: usize = 4;
const MUTATION_RATE_1: f32 = 0.001;
const MUTATION_RATE_3: f32 = 0.005;

// Нахождение "лучших приятелей"
pub fn find_best_buddies(
  img_width: usize,
  pieces_compatibility: &[Vec<Vec<f32>>; 2],
) -> [Vec<(usize, usize)>; 4] {
  // Нахождение приятелей справа и снизу от детали
  let get_buddies = |ind: usize| {
    pieces_compatibility[ind]
      .par_iter()
      .enumerate()
      .map(|(i, v)| {
        v.iter()
          .enumerate()
          .filter(|(j, _)| i != *j)
          .reduce(|x, y| if x.1 <= y.1 { x } else { y })
          .map(|(j, _)| (j / img_width, j % img_width))
          .unwrap()
      })
      .collect::<Vec<_>>()
  };
  let right_buddies = get_buddies(0);
  let down_buddies = get_buddies(1);

  // Нахождение приятелей слева и сверху от детали
  let get_opposite_buddies = |ind: usize| {
    (0..pieces_compatibility[ind].len())
      .into_par_iter()
      .map(|i| {
        pieces_compatibility[ind]
          .iter()
          .enumerate()
          .map(|(j, v)| (j, v[i]))
          .filter(|(j, _)| i != *j)
          .reduce(|x, y| if x.1 <= y.1 { x } else { y })
          .map(|(j, _)| (j / img_width, j % img_width))
          .unwrap()
      })
      .collect::<Vec<_>>()
  };
  let left_buddies = get_opposite_buddies(0);
  let up_buddies = get_opposite_buddies(1);
```

```
46
47    [right_buddies, down_buddies, left_buddies, up_buddies]
48  }
49
50  // Скрещивание
51  fn chromosomes_crossover(
52    img_width: usize,
53    img_height: usize,
54    pieces_compatibility: &[Vec<Vec<f32>>; 2],
55    pieces_buddies: &[Vec<(usize, usize)>; 4],
56    chromosome_1: &Solution,
57    chromosome_2: &Solution,
58    mut rng: Xoshiro256PlusPlus,
59  ) -> Solution {
60    // Случайная начальная деталь
61    let start_piece = (
62      (0..img_height).choose(&mut rng).unwrap(),
63      (0..img_width).choose(&mut rng).unwrap(),
64    );
65
66    // Положение каждой детали в каждом предке
67    let mut pos_in_chromosome_1 = vec![(usize::MAX, usize::MAX); img_width * img_height];
68    let mut pos_in_chromosome_2 = pos_in_chromosome_1.clone();
69    for r in 0..img_height {
70      for c in 0..img_width {
71        let (i, j) = chromosome_1[r * img_width + c];
72        pos_in_chromosome_1[i * img_width + j] = (r, c);
73
74        let (i, j) = chromosome_2[r * img_width + c];
75        pos_in_chromosome_2[i * img_width + j] = (r, c);
76      }
77    }
78
79    // Свободные детали
80    let mut free_pieces: IndexSet<_, FxBuildHasher> = (0..img_height)
81      .flat_map(|r| (0..img_width).map(move |c| (r, c)))
82      .collect();
83    // Свободные позиции, подходящие для фазы 1, и детали, которые можно туда поставить
84    let mut free_positions_phase_1 = IndexMap::with_hasher(FxBuildHasher::default());
85    // Свободные позиции, подходящие для фазы 2, и детали, которые можно туда поставить
86    let mut free_positions_phase_2 = IndexMap::with_hasher(FxBuildHasher::default());
87    // Свободные позиции, подходящие для фазы 3
88    let mut free_positions_phase_3 = IndexSet::with_hasher(FxBuildHasher::default());
89    // Свободные позиции, не подходящие для фазы 1, которые должны быть рассмотрены в фазе 2
    ↪  или 3
90    let mut free_positions_not_in_phase_1 = IndexSet::with_hasher(FxBuildHasher::default());
91    // Свободные нерассмотренные позиции
92    let mut free_positions_unknown: IndexSet<(usize, usize), _> =
```

```
93        IndexSet::with_hasher(FxBuildHasher::default());
94      // Позиции, не подходящие для фазы 1
95      let mut bad_positions_phase_1 = vec![false; 2 * img_width * 2 * img_height];
96      // Позиции, не подходящие для фазы 2
97      let mut bad_positions_phase_2 = vec![false; 2 * img_width * 2 * img_height];
98      // Соответствие деталям позиций, подходящих для фазы 1
99      let mut piece_to_pos_phase_1: IndexMap<(usize, usize), Vec<(usize, usize)>, _> =
100       IndexMap::with_hasher(FxBuildHasher::default());
101     // Соответствие деталям позиций, подходящих для фазы 2
102     let mut piece_to_pos_phase_2: IndexMap<(usize, usize), Vec<(usize, usize)>, _> =
103       IndexMap::with_hasher(FxBuildHasher::default());
104
105     // Новая хромосома, получаемая в результате скрещивания
106     // Так как неизвестно положение начальной детали в исходном изображении,
107     // и построение может идти в любую сторону, высота и ширина в 2 раза больше необходимой
108     let mut new_chromosome: Solution =
109       vec![(usize::MAX, usize::MAX); 2 * img_width * 2 * img_height];
110
111     // Текущие грани построенного изображения
112     let (mut min_r, mut max_r, mut min_c, mut max_c) =
113       (img_height, img_height, img_width, img_width);
114     // Флаг, обозначающий необходимость добавления начальной детали в центр
115     let mut start_flag = true;
116     // Пока есть свободные позиции
117     while start_flag
118       || !free_positions_phase_1.is_empty()
119       || !free_positions_phase_2.is_empty()
120       || !free_positions_phase_3.is_empty()
121       || !free_positions_not_in_phase_1.is_empty()
122       || !free_positions_unknown.is_empty()
123     {
124       // Выбираемая на данной итерации позиция и деталь
125       let mut selected_pos = None;
126       let mut selected_piece = None;
127
128       // Начальная деталь
129       if start_flag {
130         start_flag = false;
131         selected_pos = Some((img_height, img_width));
132         selected_piece = Some(start_piece);
133       }
134
135       // Фаза 1 (у обоих предков одинаковая деталь в определённом направлении от позиции)
136       if selected_pos.is_none() {
137         // Обработка нерассмотренных свободных позиций
138         let tmp = free_positions_unknown.iter().map(|(pos_r, pos_c)| {
139           let (pos_r, pos_c) = (*pos_r, *pos_c);
140           assert!(new_chromosome[pos_r * 2 * img_width + pos_c].0 == usize::MAX);
```

```
141
142         // Обработка детали слева
143         if pos_c != 0 {
144           // Деталь слева в новой хромосоме
145           let (left_piece_r, left_piece_c) =
146             new_chromosome[pos_r * 2 * img_width + pos_c - 1];
147           if left_piece_r != usize::MAX {
148             // Положение этой детали в предках
149             let (pos_1_r, pos_1_c) =
150               pos_in_chromosome_1[left_piece_r * img_width + left_piece_c];
151             let (pos_2_r, pos_2_c) =
152               pos_in_chromosome_2[left_piece_r * img_width + left_piece_c];
153             // Если справа от левой детали в обоих предках одна и та же свободная деталь, то
    ↪  выбираем её
154             if pos_1_c != img_width - 1
155               && pos_2_c != img_width - 1
156               && chromosome_1[pos_1_r * img_width + pos_1_c + 1]
157                 == chromosome_2[pos_2_r * img_width + pos_2_c + 1]
158               && free_pieces
159                 .contains(&chromosome_1[pos_1_r * img_width + pos_1_c + 1])
160             {
161               return Ok((
162                 (pos_r, pos_c),
163                 chromosome_1[pos_1_r * img_width + pos_1_c + 1],
164               ));
165             }
166           }
167         }
168         // Обработка детали справа
169         if pos_c != 2 * img_width - 1 {
170           // Деталь справа в новой хромосоме
171           let (right_piece_r, right_piece_c) =
172             new_chromosome[pos_r * 2 * img_width + pos_c + 1];
173           if right_piece_r != usize::MAX {
174             // Положение этой детали в предках
175             let (pos_1_r, pos_1_c) =
176               pos_in_chromosome_1[right_piece_r * img_width + right_piece_c];
177             let (pos_2_r, pos_2_c) =
178               pos_in_chromosome_2[right_piece_r * img_width + right_piece_c];
179             // Если слева от правой детали в обоих предках одна и та же свободная деталь, то
    ↪  выбираем её
180             if pos_1_c != 0
181               && pos_2_c != 0
182               && chromosome_1[pos_1_r * img_width + pos_1_c - 1]
183                 == chromosome_2[pos_2_r * img_width + pos_2_c - 1]
184               && free_pieces
185                 .contains(&chromosome_1[pos_1_r * img_width + pos_1_c - 1])
186             {
```

```
187                 return Ok((
188                   (pos_r, pos_c),
189                   chromosome_1[pos_1_r * img_width + pos_1_c - 1],
190                 ));
191               }
192             }
193           }
194           // Обработка детали сверху
195           if pos_r != 0 {
196             // Деталь сверху в новой хромосоме
197             let (up_piece_r, up_piece_c) =
198               new_chromosome[(pos_r - 1) * 2 * img_width + pos_c];
199             if up_piece_r != usize::MAX {
200               // Положение этой детали в предках
201               let (pos_1_r, pos_1_c) =
202                 pos_in_chromosome_1[up_piece_r * img_width + up_piece_c];
203               let (pos_2_r, pos_2_c) =
204                 pos_in_chromosome_2[up_piece_r * img_width + up_piece_c];
205               // Если снизу от верхней детали в обоих предках одна и та же свободная деталь,
    ↪ то выбираем её
206               if pos_1_r != img_height - 1
207                 && pos_2_r != img_height - 1
208                 && chromosome_1[(pos_1_r + 1) * img_width + pos_1_c]
209                   == chromosome_2[(pos_2_r + 1) * img_width + pos_2_c]
210                 && free_pieces
211                   .contains(&chromosome_1[(pos_1_r + 1) * img_width + pos_1_c])
212               {
213                 return Ok((
214                   (pos_r, pos_c),
215                   chromosome_1[(pos_1_r + 1) * img_width + pos_1_c],
216                 ));
217               }
218             }
219           }
220           // Обработка детали снизу
221           if pos_r != 2 * img_height - 1 {
222             // Деталь снизу в новой хромосоме
223             let (down_piece_r, down_piece_c) =
224               new_chromosome[(pos_r + 1) * 2 * img_width + pos_c];
225             if down_piece_r != usize::MAX {
226               // Положение этой детали в предках
227               let (pos_1_r, pos_1_c) =
228                 pos_in_chromosome_1[down_piece_r * img_width + down_piece_c];
229               let (pos_2_r, pos_2_c) =
230                 pos_in_chromosome_2[down_piece_r * img_width + down_piece_c];
231               // Если сверху от нижней детали в обоих предках одна и та же свободная деталь,
    ↪ то выбираем её
232               if pos_1_r != 0
```

```
233                 && pos_2_r != 0
234                 && chromosome_1[(pos_1_r - 1) * img_width + pos_1_c]
235                   == chromosome_2[(pos_2_r - 1) * img_width + pos_2_c]
236                 && free_pieces
237                   .contains(&chromosome_1[(pos_1_r - 1) * img_width + pos_1_c])
238               {
239                 return Ok((
240                   (pos_r, pos_c),
241                   chromosome_1[(pos_1_r - 1) * img_width + pos_1_c],
242                 ));
243               }
244             }
245           }
246           // Позиция не подходит для фазы 1
247           Err((pos_r, pos_c))
248         });
249
250         for res in tmp {
251           match res {
252             // Если позиция подходит для фазы 1, то добавление её
253             // в список позиций фазы 1 и соответствие деталей позициям фазы 1
254             Ok((pos, piece)) => {
255               if let Some(v) = piece_to_pos_phase_1.get_mut(&piece) {
256                 v.push(pos);
257               } else {
258                 piece_to_pos_phase_1.insert(piece, vec![pos]);
259               }
260               assert!(free_positions_phase_1.insert(pos, piece).is_none());
261             }
262             // Если не подходит, то добавление позиции в список не подходящих для фазы 1
263             // и тех, которые должны быть рассмотрены в фазе 2 или 3
264             Err(pos) => {
265               bad_positions_phase_1[pos.0 * 2 * img_width + pos.1] = true;
266               free_positions_not_in_phase_1.insert(pos);
267             }
268           }
269         }
270         // Нерассмотренных позиций нет
271         free_positions_unknown.clear();
272
273         // Если есть хотя бы одна подходящая позиция
274         if !free_positions_phase_1.is_empty() {
275           // Выбор случайной позиции
276           let ind = rng.gen_range(0..free_positions_phase_1.len());
277           let (pos, mut piece) = free_positions_phase_1.get_index(ind).unwrap();
278
279           // С небольшой вероятностью происходит мутация: выбирается случайная деталь
280           if rng.gen_range(0.0f32..1.0) <= MUTATION_RATE_1 {
```

```
281             let ind = rng.gen_range(0..free_pieces.len());
282             piece = free_pieces.get_index(ind).unwrap();
283           }
284
285           assert!(new_chromosome[pos.0 * 2 * img_width + pos.1].0 == usize::MAX);
286           assert!(free_pieces.contains(piece));
287           // Позиция и деталь на данной итерации выбраны
288           selected_pos = Some(*pos);
289           selected_piece = Some(*piece);
290         }
291       }
292
293       // Фаза 2 (две детали являются "лучшими приятелями" друг друга)
294       if selected_pos.is_none() {
295         // Обработка свободных позиций, не подходящих для фазы 1
296         let tmp = free_positions_not_in_phase_1.iter().map(|(pos_r, pos_c)| {
297           let (pos_r, pos_c) = (*pos_r, *pos_c);
298           assert!(new_chromosome[pos_r * 2 * img_width + pos_c].0 == usize::MAX);
299
300           // Обработка детали слева
301           if pos_c != 0 {
302             // Деталь слева в новой хромосоме
303             let (left_piece_r, left_piece_c) =
304               new_chromosome[pos_r * 2 * img_width + pos_c - 1];
305             if left_piece_r != usize::MAX {
306               // "Лучший приятель" этой детали
307               let best_buddy = pieces_buddies[0][left_piece_r * img_width + left_piece_c];
308               // Положение левой детали в предках
309               let (pos_1_r, pos_1_c) =
310                 pos_in_chromosome_1[left_piece_r * img_width + left_piece_c];
311               let (pos_2_r, pos_2_c) =
312                 pos_in_chromosome_2[left_piece_r * img_width + left_piece_c];
313               // Если найденный "лучший приятель" также считает левую деталь "лучшим
    ↪  приятелем",
314               // и справа от левой детали хотя бы в одном из предков есть этот "лучший
    ↪  приятель",
315               // и он свободен, то выбираем его
316               if pieces_buddies[2][best_buddy.0 * img_width + best_buddy.1]
317                 == (left_piece_r, left_piece_c)
318                 && ((pos_1_c != img_width - 1
319                   && chromosome_1[pos_1_r * img_width + pos_1_c + 1] == best_buddy)
320                   || (pos_2_c != img_width - 1
321                     && chromosome_2[pos_2_r * img_width + pos_2_c + 1]
322                       == best_buddy))
323                 && free_pieces.contains(&best_buddy)
324               {
325                 return Ok(((pos_r, pos_c), best_buddy));
326               }
```

```
327             }
328           }
329           // Обработка детали справа
330           if pos_c != 2 * img_width - 1 {
331             // Деталь справа в новой хромосоме
332             let (right_piece_r, right_piece_c) =
333               new_chromosome[pos_r * 2 * img_width + pos_c + 1];
334             if right_piece_r != usize::MAX {
335               // "Лучший приятель" этой детали
336               let best_buddy =
337                 pieces_buddies[2][right_piece_r * img_width + right_piece_c];
338               // Положение правой детали в предках
339               let (pos_1_r, pos_1_c) =
340                 pos_in_chromosome_1[right_piece_r * img_width + right_piece_c];
341               let (pos_2_r, pos_2_c) =
342                 pos_in_chromosome_2[right_piece_r * img_width + right_piece_c];
343               // Если найденный "лучший приятель" также считает правую деталь "лучшим
    ↪  приятелем",
344               // и слева от правой детали хотя бы в одном из предков есть этот "лучший
    ↪  приятель",
345               // и он свободен, то выбираем его
346               if pieces_buddies[0][best_buddy.0 * img_width + best_buddy.1]
347                 == (right_piece_r, right_piece_c)
348                 && ((pos_1_c != 0
349                   && chromosome_1[pos_1_r * img_width + pos_1_c - 1] == best_buddy)
350                   || (pos_2_c != 0
351                     && chromosome_2[pos_2_r * img_width + pos_2_c - 1]
352                       == best_buddy))
353                 && free_pieces.contains(&best_buddy)
354               {
355                 return Ok(((pos_r, pos_c), best_buddy));
356               }
357             }
358           }
359           // Обработка детали сверху
360           if pos_r != 0 {
361             // Деталь сверху в новой хромосоме
362             let (up_piece_r, up_piece_c) =
363               new_chromosome[(pos_r - 1) * 2 * img_width + pos_c];
364             if up_piece_r != usize::MAX {
365               // "Лучший приятель" этой детали
366               let best_buddy = pieces_buddies[1][up_piece_r * img_width + up_piece_c];
367               // Положение верхней детали в предках
368               let (pos_1_r, pos_1_c) =
369                 pos_in_chromosome_1[up_piece_r * img_width + up_piece_c];
370               let (pos_2_r, pos_2_c) =
371                 pos_in_chromosome_2[up_piece_r * img_width + up_piece_c];
```

```
372             // Если найденный "лучший приятель" также считает верхнюю деталь "лучшим
    ↪ приятелем",
373             // и снизу от верхней детали хотя бы в одном из предков есть этот "лучший
    ↪ приятель",
374             // и он свободен, то выбираем его
375             if pieces_buddies[3][best_buddy.0 * img_width + best_buddy.1]
376               == (up_piece_r, up_piece_c)
377               && ((pos_1_r != img_height - 1
378                 && chromosome_1[(pos_1_r + 1) * img_width + pos_1_c] == best_buddy)
379                 || (pos_2_r != img_height - 1
380                   && chromosome_2[(pos_2_r + 1) * img_width + pos_2_c]
381                     == best_buddy))
382               && free_pieces.contains(&best_buddy)
383             {
384               return Ok(((pos_r, pos_c), best_buddy));
385             }
386           }
387         }
388         // Обработка детали снизу
389         if pos_r != 2 * img_height - 1 {
390           // Деталь снизу в новой хромосоме
391           let (down_piece_r, down_piece_c) =
392             new_chromosome[(pos_r + 1) * 2 * img_width + pos_c];
393           if down_piece_r != usize::MAX {
394             // "Лучший приятель" этой детали
395             let best_buddy = pieces_buddies[3][down_piece_r * img_width + down_piece_c];
396             // Положение нижней детали в предках
397             let (pos_1_r, pos_1_c) =
398               pos_in_chromosome_1[down_piece_r * img_width + down_piece_c];
399             let (pos_2_r, pos_2_c) =
400               pos_in_chromosome_2[down_piece_r * img_width + down_piece_c];
401             // Если найденный "лучший приятель" также считает нижнюю деталь "лучшим
    ↪ приятелем",
402             // и сверху от нижней детали хотя бы в одном из предков есть этот "лучший
    ↪ приятель",
403             // и он свободен, то выбираем его
404             if pieces_buddies[1][best_buddy.0 * img_width + best_buddy.1]
405               == (down_piece_r, down_piece_c)
406               && ((pos_1_r != 0
407                 && chromosome_1[(pos_1_r - 1) * img_width + pos_1_c] == best_buddy)
408                 || (pos_2_r != 0
409                   && chromosome_2[(pos_2_r - 1) * img_width + pos_2_c]
410                     == best_buddy))
411               && free_pieces.contains(&best_buddy)
412             {
413               return Ok(((pos_r, pos_c), best_buddy));
414             }
415           }
```

```
416           }
417           // Позиция не подходит для фазы 2
418           Err((pos_r, pos_c))
419         });
420
421         for res in tmp {
422           match res {
423             // Если позиция подходит для фазы 2, то добавление её
424             // в список позиций фазы 2 и соответствие деталей позициям фазы 2
425             Ok((pos, piece)) => {
426               if let Some(v) = piece_to_pos_phase_2.get_mut(&piece) {
427                 v.push(pos);
428               } else {
429                 piece_to_pos_phase_2.insert(piece, vec![pos]);
430               }
431               assert!(free_positions_phase_2.insert(pos, piece).is_none());
432             }
433             // Если не подходит, то добавление позиции в список не подходящих для фазы 2
434             // и подходящих для фазы 3
435             Err(pos) => {
436               bad_positions_phase_2[pos.0 * 2 * img_width + pos.1] = true;
437               free_positions_phase_3.insert(pos);
438             }
439           }
440         }
441         // Нерассмотренных позиций нет
442         free_positions_not_in_phase_1.clear();
443
444         // Если есть хотя бы одна подходящая позиция
445         if !free_positions_phase_2.is_empty() {
446           // Выбор случайной позиции
447           let ind = rng.gen_range(0..free_positions_phase_2.len());
448           let (pos, piece) = free_positions_phase_2.get_index(ind).unwrap();
449
450           assert!(new_chromosome[pos.0 * 2 * img_width + pos.1].0 == usize::MAX);
451           assert!(free_pieces.contains(piece));
452           // Позиция и деталь на данной итерации выбраны
453           selected_pos = Some(*pos);
454           selected_piece = Some(*piece);
455         }
456       }
457
458       // Фаза 3 (наиболее подходящая деталь для позиции)
459       if selected_pos.is_none() {
460         assert!(!free_positions_phase_3.is_empty());
461         assert!(!free_pieces.is_empty());
462
463         // Выбор случайной позиции
```

```
464         let ind = rng.gen_range(0..free_positions_phase_3.len());
465         let (pos_r, pos_c) = *free_positions_phase_3.get_index(ind).unwrap();
466
467         // Наиболее подходящая деталь
468         let mut best_piece = (usize::MAX, usize::MAX);
469         // С небольшой вероятностью происходит мутация: выбирается случайная деталь
470         if rng.gen_range(0.0f32..1.0) <= MUTATION_RATE_3 {
471           let ind = rng.gen_range(0..free_pieces.len());
472           best_piece = *free_pieces.get_index(ind).unwrap();
473         } else {
474           // Деталь слева в новой хромосоме
475           let (left_piece_r, left_piece_c) = if pos_c == 0 {
476             (usize::MAX, usize::MAX)
477           } else {
478             new_chromosome[pos_r * 2 * img_width + pos_c - 1]
479           };
480           // Деталь справа в новой хромосоме
481           let (right_piece_r, right_piece_c) = if pos_c == 2 * img_width - 1 {
482             (usize::MAX, usize::MAX)
483           } else {
484             new_chromosome[pos_r * 2 * img_width + pos_c + 1]
485           };
486           // Деталь сверху в новой хромосоме
487           let (up_piece_r, up_piece_c) = if pos_r == 0 {
488             (usize::MAX, usize::MAX)
489           } else {
490             new_chromosome[(pos_r - 1) * 2 * img_width + pos_c]
491           };
492           // Деталь снизу в новой хромосоме
493           let (down_piece_r, down_piece_c) = if pos_r == 2 * img_height - 1 {
494             (usize::MAX, usize::MAX)
495           } else {
496             new_chromosome[(pos_r + 1) * 2 * img_width + pos_c]
497           };
498
499           // Наименьшая несовместимость
500           let mut best_compatibility = f32::INFINITY;
501           // "Лучшие приятели"
502           let mut buddies = Vec::new();
503           if left_piece_r != usize::MAX {
504             buddies.push(pieces_buddies[0][left_piece_r * img_width + left_piece_c]);
505           }
506           if right_piece_r != usize::MAX {
507             buddies.push(pieces_buddies[2][right_piece_r * img_width + right_piece_c]);
508           }
509           if up_piece_r != usize::MAX {
510             buddies.push(pieces_buddies[1][up_piece_r * img_width + up_piece_c]);
511           }
```

```
512         if down_piece_r != usize::MAX {
513           buddies.push(pieces_buddies[3][down_piece_r * img_width + down_piece_c]);
514         }
515         let buddies = buddies.iter().filter(|piece| free_pieces.contains(*piece));
516
517         // Обработка всех свободных деталей
518         for (piece_r, piece_c) in buddies.chain(free_pieces.iter()) {
519           // Несовместимость
520           let mut res = 0.0f32;
521           // Обработка детали слева
522           if left_piece_r != usize::MAX {
523             res += pieces_compatibility[0][left_piece_r * img_width + left_piece_c]
524               [piece_r * img_width + piece_c];
525             if res >= best_compatibility {
526               continue;
527             }
528           }
529           // Обработка детали справа
530           if right_piece_r != usize::MAX {
531             res += pieces_compatibility[0][piece_r * img_width + piece_c]
532               [right_piece_r * img_width + right_piece_c];
533             if res >= best_compatibility {
534               continue;
535             }
536           }
537           // Обработка детали сверху
538           if up_piece_r != usize::MAX {
539             res += pieces_compatibility[1][up_piece_r * img_width + up_piece_c]
540               [piece_r * img_width + piece_c];
541             if res >= best_compatibility {
542               continue;
543             }
544           }
545           // Обработка детали снизу
546           if down_piece_r != usize::MAX {
547             res += pieces_compatibility[1][piece_r * img_width + piece_c]
548               [down_piece_r * img_width + down_piece_c];
549             if res >= best_compatibility {
550               continue;
551             }
552           }
553
554           // Обновление наименее несовместимой детали
555           best_piece = (*piece_r, *piece_c);
556           best_compatibility = res;
557         }
558       }
559
```

```
560         assert!(new_chromosome[pos_r * 2 * img_width + pos_c].0 == usize::MAX);
561         assert!(free_pieces.contains(&best_piece));
562         // Позиция и деталь на данной итерации выбраны
563         selected_pos = Some((pos_r, pos_c));
564         selected_piece = Some(best_piece);
565       }
566
567       // Выбранные позиция и деталь
568       let selected_pos = selected_pos.unwrap();
569       let selected_piece = selected_piece.unwrap();
570       let (selected_pos_r, selected_pos_c) = selected_pos;
571       // Установка детали в новой хромосоме
572       new_chromosome[selected_pos_r * 2 * img_width + selected_pos_c] = selected_piece;
573       // Удаление позиции из списков свободных позиций
574       free_positions_phase_1.remove(&selected_pos);
575       free_positions_phase_2.remove(&selected_pos);
576       free_positions_phase_3.remove(&selected_pos);
577       free_positions_unknown.remove(&selected_pos);
578       // Удаление детали из списка свободных
579       free_pieces.remove(&selected_piece);
580       // Если выбранной детали соответствуют позиции в фазе 1, то нужно рассмотреть их заново
581       if let Some(v) = piece_to_pos_phase_1.get(&selected_piece) {
582         for pos in v {
583           if free_positions_phase_1.contains_key(pos) {
584             free_positions_phase_1.remove(pos);
585             free_positions_phase_3.remove(pos);
586             free_positions_not_in_phase_1.remove(pos);
587             bad_positions_phase_1[pos.0 * 2 * img_width + pos.1] = false;
588             bad_positions_phase_2[pos.0 * 2 * img_width + pos.1] = false;
589             if new_chromosome[pos.0 * 2 * img_width + pos.1].0 == usize::MAX {
590               free_positions_unknown.insert(*pos);
591             }
592           }
593         }
594         piece_to_pos_phase_1.remove(&selected_piece);
595       }
596       // Если выбранной детали соответствуют позиции в фазе 2, то нужно рассмотреть их заново
597       if let Some(v) = piece_to_pos_phase_2.get(&selected_piece) {
598         for pos in v {
599           if free_positions_phase_2.contains_key(pos) {
600             free_positions_phase_2.remove(pos);
601             free_positions_phase_3.remove(pos);
602             free_positions_not_in_phase_1.remove(pos);
603             bad_positions_phase_1[pos.0 * 2 * img_width + pos.1] = false;
604             bad_positions_phase_2[pos.0 * 2 * img_width + pos.1] = false;
605             if new_chromosome[pos.0 * 2 * img_width + pos.1].0 == usize::MAX {
606               free_positions_unknown.insert(*pos);
607             }
```

```
608           }
609         }
610         piece_to_pos_phase_2.remove(&selected_piece);
611       }
612
613       // Обновление верхней границы
614       if selected_pos_r < min_r {
615         min_r = selected_pos_r;
616         // Если достигнута полная высота изображения, удалить все позиции, выходящие за
    ↪ границы
617         if 1 + max_r - min_r == img_height {
618           for c in 0..(2 * img_width) {
619             if min_r >= 1 {
620               free_positions_phase_1.remove(&(min_r - 1, c));
621               free_positions_phase_2.remove(&(min_r - 1, c));
622               free_positions_phase_3.remove(&(min_r - 1, c));
623               free_positions_not_in_phase_1.remove(&(min_r - 1, c));
624               free_positions_unknown.remove(&(min_r - 1, c));
625               bad_positions_phase_1[(min_r - 1) * 2 * img_width + c] = false;
626               bad_positions_phase_2[(min_r - 1) * 2 * img_width + c] = false;
627             }
628
629             if max_r + 1 < 2 * img_height {
630               free_positions_phase_1.remove(&(max_r + 1, c));
631               free_positions_phase_2.remove(&(max_r + 1, c));
632               free_positions_phase_3.remove(&(max_r + 1, c));
633               free_positions_not_in_phase_1.remove(&(max_r + 1, c));
634               free_positions_unknown.remove(&(max_r + 1, c));
635               bad_positions_phase_1[(max_r + 1) * 2 * img_width + c] = false;
636               bad_positions_phase_2[(max_r + 1) * 2 * img_width + c] = false;
637             }
638           }
639         }
640       }
641       // Обновление нижней границы
642       else if selected_pos_r > max_r {
643         max_r = selected_pos_r;
644         // Если достигнута полная высота изображения, удалить все позиции, выходящие за
    ↪ границы
645         if 1 + max_r - min_r == img_height {
646           for c in 0..(2 * img_width) {
647             if min_r >= 1 {
648               free_positions_phase_1.remove(&(min_r - 1, c));
649               free_positions_phase_2.remove(&(min_r - 1, c));
650               free_positions_phase_3.remove(&(min_r - 1, c));
651               free_positions_not_in_phase_1.remove(&(min_r - 1, c));
652               free_positions_unknown.remove(&(min_r - 1, c));
653               bad_positions_phase_1[(min_r - 1) * 2 * img_width + c] = false;
```

```
654             bad_positions_phase_2[(min_r - 1) * 2 * img_width + c] = false;
655           }
656
657           if max_r + 1 < 2 * img_height {
658             free_positions_phase_1.remove(&(max_r + 1, c));
659             free_positions_phase_2.remove(&(max_r + 1, c));
660             free_positions_phase_3.remove(&(max_r + 1, c));
661             free_positions_not_in_phase_1.remove(&(max_r + 1, c));
662             free_positions_unknown.remove(&(max_r + 1, c));
663             bad_positions_phase_1[(max_r + 1) * 2 * img_width + c] = false;
664             bad_positions_phase_2[(max_r + 1) * 2 * img_width + c] = false;
665           }
666         }
667       }
668     }
669     // Обновление левой границы
670     if selected_pos_c < min_c {
671       min_c = selected_pos_c;
672       // Если достигнута полная ширина изображения, удалить все позиции, выходящие за
    ↪  границы
673       if 1 + max_c - min_c == img_width {
674         for r in 0..(2 * img_height) {
675           if min_c >= 1 {
676             free_positions_phase_1.remove(&(r, min_c - 1));
677             free_positions_phase_2.remove(&(r, min_c - 1));
678             free_positions_phase_3.remove(&(r, min_c - 1));
679             free_positions_not_in_phase_1.remove(&(r, min_c - 1));
680             free_positions_unknown.remove(&(r, min_c - 1));
681             bad_positions_phase_1[r * 2 * img_width + min_c - 1] = false;
682             bad_positions_phase_2[r * 2 * img_width + min_c - 1] = false;
683           }
684
685           if max_c + 1 < 2 * img_width {
686             free_positions_phase_1.remove(&(r, max_c + 1));
687             free_positions_phase_2.remove(&(r, max_c + 1));
688             free_positions_phase_3.remove(&(r, max_c + 1));
689             free_positions_not_in_phase_1.remove(&(r, max_c + 1));
690             free_positions_unknown.remove(&(r, max_c + 1));
691             bad_positions_phase_1[r * 2 * img_width + max_c + 1] = false;
692             bad_positions_phase_2[r * 2 * img_width + max_c + 1] = false;
693           }
694         }
695       }
696     }
697     // Обновление правой границы
698     else if selected_pos_c > max_c {
699       max_c = selected_pos_c;
```

```
700       // Если достигнута полная ширина изображения, удалить все позиции, выходящие за
    ↪ границы
701       if 1 + max_c - min_c == img_width {
702         for r in 0..(2 * img_height) {
703           if min_c >= 1 {
704             free_positions_phase_1.remove(&(r, min_c - 1));
705             free_positions_phase_2.remove(&(r, min_c - 1));
706             free_positions_phase_3.remove(&(r, min_c - 1));
707             free_positions_not_in_phase_1.remove(&(r, min_c - 1));
708             free_positions_unknown.remove(&(r, min_c - 1));
709             bad_positions_phase_1[r * 2 * img_width + min_c - 1] = false;
710             bad_positions_phase_2[r * 2 * img_width + min_c - 1] = false;
711           }
712
713           if max_c + 1 < 2 * img_width {
714             free_positions_phase_1.remove(&(r, max_c + 1));
715             free_positions_phase_2.remove(&(r, max_c + 1));
716             free_positions_phase_3.remove(&(r, max_c + 1));
717             free_positions_not_in_phase_1.remove(&(r, max_c + 1));
718             free_positions_unknown.remove(&(r, max_c + 1));
719             bad_positions_phase_1[r * 2 * img_width + max_c + 1] = false;
720             bad_positions_phase_2[r * 2 * img_width + max_c + 1] = false;
721           }
722         }
723       }
724     }
725
726     // Обработка позиций сверху и снизу от выбранной
727     for dr in [-1isize, 1] {
728       let new_r = ((selected_pos_r as isize) + dr) as usize;
729       if 1 + max(max_r, new_r) - min(min_r, new_r) > img_height {
730         continue;
731       }
732       // Если позиция свободна и не подходит для фазы 1 или 2 (нет в списках подходящих для
    ↪ этих фаз
733       // или уже была рассмотрена и оказалась неподходящей), то добавить её к рассмотрению,
734       // удалив из всех списков рассмотренных позиций
735       if new_chromosome[new_r * 2 * img_width + selected_pos_c].0 == usize::MAX
736         && ((!free_positions_phase_1.contains_key(&(new_r, selected_pos_c))
737           && !free_positions_phase_2.contains_key(&(new_r, selected_pos_c)))
738           || bad_positions_phase_1[new_r * 2 * img_width + selected_pos_c]
739           || bad_positions_phase_2[new_r * 2 * img_width + selected_pos_c])
740       {
741         free_positions_phase_1.remove(&(new_r, selected_pos_c));
742         free_positions_phase_2.remove(&(new_r, selected_pos_c));
743         free_positions_phase_3.remove(&(new_r, selected_pos_c));
744         free_positions_not_in_phase_1.remove(&(new_r, selected_pos_c));
745         bad_positions_phase_1[new_r * 2 * img_width + selected_pos_c] = false;
```

```
746           bad_positions_phase_2[new_r * 2 * img_width + selected_pos_c] = false;
747           free_positions_unknown.insert((new_r, selected_pos_c));
748         }
749       }
750       // Обработка позиций слева и справа от выбранной
751       for dc in [-1isize, 1] {
752         let new_c = ((selected_pos_c as isize) + dc) as usize;
753         if 1 + max(max_c, new_c) - min(min_c, new_c) > img_width {
754           continue;
755         }
756         // Если позиция свободна и не подходит для фазы 1 или 2 (нет в списках подходящих для
    ↪ этих фаз
757         // или уже была рассмотрена и оказалась неподходящей), то добавить её к рассмотрению,
758         // удалив из всех списков рассмотренных позиций
759         if new_chromosome[selected_pos_r * 2 * img_width + new_c].0 == usize::MAX
760           && ((!free_positions_phase_1.contains_key(&(selected_pos_r, new_c))
761             && !free_positions_phase_2.contains_key(&(selected_pos_r, new_c)))
762             || bad_positions_phase_1[selected_pos_r * 2 * img_width + new_c]
763             || bad_positions_phase_2[selected_pos_r * 2 * img_width + new_c])
764         {
765           free_positions_phase_1.remove(&(selected_pos_r, new_c));
766           free_positions_phase_2.remove(&(selected_pos_r, new_c));
767           free_positions_phase_3.remove(&(selected_pos_r, new_c));
768           free_positions_not_in_phase_1.remove(&(selected_pos_r, new_c));
769           bad_positions_phase_1[selected_pos_r * 2 * img_width + new_c] = false;
770           bad_positions_phase_2[selected_pos_r * 2 * img_width + new_c] = false;
771           free_positions_unknown.insert((selected_pos_r, new_c));
772         }
773       }
774     }
775
776     assert!(free_pieces.is_empty());
777     assert_eq!(1 + max_r - min_r, img_height);
778     assert_eq!(1 + max_c - min_c, img_width);
779     // Вырезание результата из новой хромосомы
780     (min_r..=max_r)
781       .flat_map(|r| {
782         (min_c..=max_c)
783           .map(|c| new_chromosome[r * 2 * img_width + c])
784           .collect::<Vec<_>>()
785       })
786       .collect()
787   }
788
789   // Шаг алгоритма
790   pub fn algorithm_step(
791     population_size: usize,
792     rng: &mut Xoshiro256PlusPlus,
```

```
793     img_width: usize,
794     img_height: usize,
795     image_generations_processed: usize,
796     pieces_compatibility: &[Vec<Vec<f32>>; 2],
797     pieces_buddies: &[Vec<(usize, usize)>; 4],
798     current_generation: &[Solution],
799   ) -> Vec<Solution> {
800     // Создание нового поколения
801     let mut new_generation: Vec<Solution> = if image_generations_processed == 0 {
802       (0..population_size)
803         .map(|_| generate_random_solution(img_width, img_height, rng))
804         .collect()
805     } else {
806       // Отбор лучших хромосом
807       let mut best_chromosomes: Vec<_> = current_generation
808         .iter()
809         .take(ELITISM_COUNT)
810         .cloned()
811         .collect();
812 
813       let indexes: Vec<usize> = (0..population_size).collect();
814       // Оценки хромосом текущего поколения
815       let curr_gen_compatibilities: Vec<_> = current_generation
816         .iter()
817         .map(|chromosome| {
818           solution_compatibility(img_width, img_height, pieces_compatibility, chromosome)
819         })
820         .collect();
821       // Наименьшая оценка
822       let min_compatibility = curr_gen_compatibilities
823         .iter()
824         .cloned()
825         .reduce(f32::min)
826         .unwrap();
827       // Наибольшая оценка
828       let max_compatibility = curr_gen_compatibilities
829         .iter()
830         .cloned()
831         .reduce(f32::max)
832         .unwrap();
833       // Разность оценок
834       let diff_compatibility = if min_compatibility == max_compatibility {
835         1.0
836       } else {
837         max_compatibility - min_compatibility
838       };
839 
840       // Выбранные для скрещивания предки
```

```
841       let parents: Vec<_> = (0..(population_size - ELITISM_COUNT))
842         .map(|_| {
843           let mut iter = indexes
844             .choose_multiple_weighted(rng, 2, |i| {
845               (-(curr_gen_compatibilities[*i] - min_compatibility) / diff_compatibility)
846                 * 0.9
847                 + 0.95
848             })
849             .unwrap();
850           (*iter.next().unwrap(), *iter.next().unwrap())
851         })
852         .collect();
853       // Генераторы случайных чисел для каждого скрещивания
854       let rngs: Vec<_> = (0..(population_size - ELITISM_COUNT))
855         .map(|_| {
856           let tmp = rng.clone();
857           rng.jump();
858           tmp
859         })
860         .collect();
861       // Получение нового поколения скрещиванием
862       let mut other_chromosomes: Vec<_> = parents
863         .into_par_iter()
864         .zip(rngs.into_par_iter())
865         .map(|((i, j), rng)| {
866           chromosomes_crossover(
867             img_width,
868             img_height,
869             pieces_compatibility,
870             pieces_buddies,
871             &current_generation[i],
872             &current_generation[j],
873             rng,
874           )
875         })
876         .collect();
877
878       // Объединение
879       best_chromosomes.append(&mut other_chromosomes);
880       best_chromosomes
881     };
882     // Сортировка хромосом по возрастанию оценки
883     new_generation.sort_by_cached_key(|chromosome| {
884       FloatOrd(solution_compatibility(
885         img_width,
886         img_height,
887         pieces_compatibility,
888         chromosome,
```

```
889       ))
890     });
891     new_generation
892   }
```

ПРИЛОЖЕНИЕ Г

Программный код алгоритма циклических ограничений

```rust
// Матрица - двумерный массив, в каждой позиции которого находится деталь (пара из строки и столбца)
type Matrix = Vec<Vec<(usize, usize)>>;
// Приоритет матрицы - количество непустых позиций (со знаком минус) и значение совместимости между деталями
type MatrixPriority = (isize, FloatOrd<f32>);

// Совместимость матриц
enum MatrixCompatibility {
  // Совместимы: сдвиг между общими деталями, приоритет объединённой матрицы
  Compatible(isize, isize, MatrixPriority),
  // Не совместимы (есть геометрический конфликт)
  Incompatible,
  // Не связаны друг с другом (не более одной общей детали)
  NotRelated,
}

// Соотношение несовместимости, при котором две детали считаются парой-кандидатом
const CANDIDATE_MATCH_RATIO: f32 = 1.15;
// Соотношение несовместимости для отсечения одинаковых деталей
const CANDIDATE_MATCH_RATIO_EQUAL: f32 = 1.0001;
// Максимальное количество пар-кандидатов, которое может образовывать одна деталь в одном направлении
const MAX_CANDIDATES: usize = 10;
// То же, но для одинаковых деталей
const MAX_CANDIDATES_EQUAL: usize = 7;
// Соотношение деталей к пустым позициям в строке/столбце, при котором она/он вырезается
const TRIM_RATE: f32 = 0.1;

// Нахождение пар-кандидатов
pub fn find_match_candidates(
  img_width: usize,
  pieces_compatibility: &[Vec<Vec<f32>>; 2],
) -> [Vec<Vec<(usize, usize)>>; 2] {
  // Нахождение пар-кандидатов, в которых вторая деталь справа или снизу
  let get_candidates = |ind: usize| {
    // Минимальные значения несовместимости для каждой детали в заданном направлении
    let min_scores = pieces_compatibility[ind]
      .par_iter()
      .enumerate()
      .map(|(i, v)| {
        v.iter()
          .enumerate()
          .filter(|(j, _)| i != *j)
          .reduce(|x, y| if x.1 <= y.1 { x } else { y })
```

```
43            .map(|(_, x)| *x)
44            .unwrap()
45        })
46        .collect::<Vec<_>>();
47      // Нахождение пар-кандидатов для каждой детали
48      pieces_compatibility[ind]
49        .par_iter()
50        .zip(min_scores)
51        .enumerate()
52        .map(|(i, (v, min_score))| {
53          // Все детали, для которых соотношение несовместимости не больше порога
54          let mut tmp: Vec<_> = v
55            .iter()
56            .enumerate()
57            .filter(|(j, x)| i != *j && **x <= CANDIDATE_MATCH_RATIO * min_score)
58            .collect();
59          // Выбор лучших пар с количеством не больше заданного порога
60          tmp.sort_by_key(|x| FloatOrd(*x.1));
61          tmp.into_iter()
62            .take(MAX_CANDIDATES)
63            .enumerate()
64            .filter_map(|(ind, x)| {
65              if ind < MAX_CANDIDATES_EQUAL
66                || *x.1 > CANDIDATE_MATCH_RATIO_EQUAL * min_score
67              {
68                Some(x)
69              } else {
70                None
71              }
72            })
73            .map(|(j, _)| (j / img_width, j % img_width))
74            .collect::<Vec<_>>()
75        })
76        .collect::<Vec<_>>()
77    };
78    let right_candidates = get_candidates(0);
79    let down_candidates = get_candidates(1);
80
81    // Нахождение пар-кандидатов, в которых вторая деталь слева или сверху
82    let get_opposite_candidates = |ind: usize| {
83      // Минимальные значения несовместимости для каждой детали в заданном направлении
84      let min_scores = (0..pieces_compatibility[ind].len())
85        .into_par_iter()
86        .map(|i| {
87          pieces_compatibility[ind]
88            .iter()
89            .enumerate()
90            .map(|(j, v)| (j, v[i]))
```

```
91            .filter(|(j, _)| i != *j)
92            .reduce(|x, y| if x.1 <= y.1 { x } else { y })
93            .map(|(_, x)| x)
94            .unwrap()
95        })
96        .collect::<Vec<_>>();
97      // Нахождение пар-кандидатов для каждой детали
98      (0..pieces_compatibility[ind].len())
99        .into_par_iter()
100       .zip(min_scores)
101       .map(|(i, min_score)| {
102         // Все детали, для которых соотношение несовместимости не больше порога
103         let mut tmp: Vec<_> = pieces_compatibility[ind]
104           .iter()
105           .enumerate()
106           .map(|(j, v)| (j, v[i]))
107           .filter(|(j, x)| i != *j && *x <= CANDIDATE_MATCH_RATIO * min_score)
108           .collect();
109         // Выбор лучших пар с количеством не больше заданного порога
110         tmp.sort_by_key(|x| FloatOrd(x.1));
111         tmp.into_iter()
112           .take(MAX_CANDIDATES)
113           .enumerate()
114           .filter_map(|(ind, x)| {
115             if ind < MAX_CANDIDATES_EQUAL
116               || x.1 > CANDIDATE_MATCH_RATIO_EQUAL * min_score
117             {
118               Some(x)
119             } else {
120               None
121             }
122           })
123           .map(|(j, _)| (j / img_width, j % img_width))
124           .collect::<Vec<_>>()
125       })
126       .collect::<Vec<_>>()
127   };
128   let left_candidates = get_opposite_candidates(0);
129   let up_candidates = get_opposite_candidates(1);
130
131   // Выбор таких пар, в которых и первая деталь считает вторую своей парой, и наоборот
132   let get_buddies = |candidates: Vec<Vec<(usize, usize)>>,
133                      opposite_candidates: Vec<Vec<(usize, usize)>>| {
134     candidates
135       .into_iter()
136       .enumerate()
137       .map(|(i, v)| {
138         let x = (i / img_width, i % img_width);
```

```
        v.into_iter()
          .filter(|y| opposite_candidates[y.0 * img_width + y.1].contains(&x))
          .collect::<Vec<_>>()
      })
      .collect::<Vec<_>>()
  };
  let right_buddies = get_buddies(right_candidates, left_candidates);
  let down_buddies = get_buddies(down_candidates, up_candidates);

  [right_buddies, down_buddies]
}

// Определение совместимостей маленьких циклов
fn small_loops_matches(size: usize, sl: &[Matrix]) -> [Vec<Vec<usize>>; 2] {
  // Два маленьких цикла размера i x i слева направо: должны пересекаться в области размера
  ↪ i x (i - 1)
  let right_matches = sl
    .par_iter()
    .map(|sl_left| {
      sl.iter()
        .enumerate()
        .filter_map(|(sl_right_i, sl_right)| {
          for r in 0..size {
            for c in 0..(size - 1) {
              if sl_left[r][c + 1] != sl_right[r][c] {
                return None;
              }
            }
          }
          for v_left in sl_left {
            for v_right in sl_right {
              if v_left[0] == v_right[size - 1] {
                return None;
              }
            }
          }
          Some(sl_right_i)
        })
        .collect::<Vec<_>>()
    })
    .collect::<Vec<_>>();

  // Два маленьких цикла размера i x i сверху вниз: должны пересекаться в области размера (i
  ↪ - 1) x i
  let down_matches = sl
    .par_iter()
    .map(|sl_up| {
      sl.iter()
```

```
185           .enumerate()
186           .filter_map(|(sl_down_i, sl_down)| {
187             for r in 0..(size - 1) {
188               for c in 0..size {
189                 if sl_up[r + 1][c] != sl_down[r][c] {
190                   return None;
191                 }
192               }
193             }
194             for x_left in &sl_up[0] {
195               for x_right in &sl_down[size - 1] {
196                 if x_left == x_right {
197                   return None;
198                 }
199               }
200             }
201             Some(sl_down_i)
202           })
203           .collect::<Vec<_>>()
204       })
205       .collect::<Vec<_>>();
206
207     [right_matches, down_matches]
208   }
209
210   // Объединение четырёх маленьких циклов размера i x i в один размера (i + 1) x (i + 1)
211   fn merge_loops(
212     size: usize,
213     left_up: &Matrix,
214     right_up: &Matrix,
215     left_down: &Matrix,
216     right_down: &Matrix,
217   ) -> Matrix {
218     let mut new_loop = vec![vec![(usize::MAX, usize::MAX); size + 1]; size + 1];
219     for r in 0..size {
220       for c in 0..size {
221         new_loop[r][c] = left_up[r][c];
222       }
223     }
224     for r in 0..size {
225       new_loop[r][size] = right_up[r][size - 1];
226     }
227     for c in 0..size {
228       new_loop[size][c] = left_down[size - 1][c];
229     }
230     new_loop[size][size] = right_down[size - 1][size - 1];
231     new_loop
232   }
```

```
233
234 // Нахождение маленьких циклов
235 fn small_loops(
236   img_width: usize,
237   img_height: usize,
238   pieces_match_candidates: &[Vec<Vec<(usize, usize)>>; 2],
239 ) -> Vec<Vec<Matrix>> {
240   // Циклы порядка 1 (пары-кандидаты)
241   let sl_1_right = (0..img_height).flat_map(|left_r| {
242     (0..img_width)
243       .flat_map(|left_c| {
244         let left_i = left_r * img_width + left_c;
245         pieces_match_candidates[0][left_i]
246           .iter()
247           .map(|right| vec![vec![(left_r, left_c), *right]])
248           .collect::<Vec<_>>()
249       })
250       .collect::<Vec<_>>()
251   });
252   let sl_1_down = (0..img_height).flat_map(|up_r| {
253     (0..img_width)
254       .flat_map(|up_c| {
255         let up_i = up_r * img_width + up_c;
256         pieces_match_candidates[1][up_i]
257           .iter()
258           .map(|down| vec![vec![(up_r, up_c)], vec![*down]])
259           .collect::<Vec<_>>()
260       })
261       .collect::<Vec<_>>()
262   });
263   let sl_1 = sl_1_right.chain(sl_1_down).collect::<Vec<_>>();
264
265   // Циклы порядка 2 (из пар-кандидатов)
266   let sl_2 = (0..img_height)
267     .into_par_iter()
268     .flat_map(|left_up_r| {
269       (0..img_width)
270         .flat_map(|left_up_c| {
271           let left_up_i = left_up_r * img_width + left_up_c;
272           pieces_match_candidates[0][left_up_i]
273             .iter()
274             .flat_map(|right_up| {
275               let right_up_i = right_up.0 * img_width + right_up.1;
276               if right_up_i == left_up_i {
277                 return Vec::new();
278               }
279
280               pieces_match_candidates[1][left_up_i]
```

```
                .iter()
                .flat_map(|left_down| {
                  let left_down_i = left_down.0 * img_width + left_down.1;
                  if left_down_i == left_up_i || left_down_i == right_up_i {
                    return Vec::new();
                  }

                  pieces_match_candidates[0][left_down_i]
                    .iter()
                    .flat_map(|right_down_0| {
                      if *right_down_0 == (left_up_r, left_up_c)
                        || right_down_0 == right_up
                        || right_down_0 == left_down
                      {
                        return Vec::new();
                      }
                      pieces_match_candidates[1][right_up_i]
                        .iter()
                        .find_map(|right_down_1| {
                          if right_down_0 == right_down_1 {
                            Some(vec![
                              vec![(left_up_r, left_up_c), *right_up],
                              vec![*left_down, *right_down_0],
                            ])
                          } else {
                            None
                          }
                        })
                        .into_iter()
                        .collect::<Vec<_>>()
                    })
                    .collect::<Vec<_>>()
                })
                .collect::<Vec<_>>()
            })
            .collect::<Vec<_>>()
        })
        .collect::<Vec<_>>()
    })
    .collect::<Vec<_>>();

  // Построение циклов следующих порядков
  let mut sl_all = vec![sl_1, sl_2];
  loop {
    let sl_last = sl_all.last().unwrap();
    let sl_size = sl_all.len();
    let matches = small_loops_matches(sl_size, sl_last);
    let sl_next = (0..sl_last.len())
```

```
      .into_par_iter()
      .flat_map(|left_up_i| {
        matches[0][left_up_i]
          .iter()
          .flat_map(|right_up_i| {
            matches[1][left_up_i]
              .iter()
              .flat_map(|left_down_i| {
                matches[0][*left_down_i]
                  .iter()
                  .flat_map(|right_down_0_i| {
                    matches[1][*right_up_i]
                      .iter()
                      .filter_map(|right_down_1_i| {
                        if right_down_0_i == right_down_1_i
                          && sl_last[left_up_i][0][0]
                            != sl_last[*right_down_0_i][sl_size - 1]
                              [sl_size - 1]
                          && sl_last[*right_up_i][0][sl_size - 1]
                            != sl_last[*left_down_i][sl_size - 1][0]
                        {
                          let sl = merge_loops(
                            sl_size,
                            &sl_last[left_up_i],
                            &sl_last[*right_up_i],
                            &sl_last[*left_down_i],
                            &sl_last[*right_down_0_i],
                          );
                          Some(sl)
                        } else {
                          None
                        }
                      })
                      .collect::<Vec<_>>()
                  })
                  .collect::<Vec<_>>()
              })
              .collect::<Vec<_>>()
          })
          .collect::<Vec<_>>()
      })
      .collect::<Vec<_>>();
    if sl_next.is_empty() {
      break;
    }
    sl_all.push(sl_next);
  }
  sl_all
```

```
377 }
378
379 // Приоритет матрицы
380 fn matrix_priority(
381   img_width: usize,
382   pieces_compatibility: &[Vec<Vec<f32>>; 2],
383   m: &Matrix,
384 ) -> MatrixPriority {
385   let cnt = m.iter().flatten().filter(|x| x.0 != usize::MAX).count();
386
387   // Совместимости деталей по направлению вправо
388   let right_compatibility = (0..m.len())
389     .map(|i_r| {
390       (0..(m[i_r].len() - 1))
391         .map(|i_c| {
392           let (j_r, j_c) = m[i_r][i_c];
393           let (k_r, k_c) = m[i_r][i_c + 1];
394           if j_r == usize::MAX || k_r == usize::MAX {
395             return 0.0;
396           }
397           pieces_compatibility[0][j_r * img_width + j_c][k_r * img_width + k_c]
398         })
399         .sum::<f32>()
400     })
401     .sum::<f32>();
402   // Совместимости деталей по направлению вниз
403   let down_compatibility = (0..(m.len() - 1))
404     .map(|i_r| {
405       (0..m[i_r].len())
406         .map(|i_c| {
407           let (j_r, j_c) = m[i_r][i_c];
408           let (k_r, k_c) = m[i_r + 1][i_c];
409           if j_r == usize::MAX || k_r == usize::MAX {
410             return 0.0;
411           }
412           pieces_compatibility[1][j_r * img_width + j_c][k_r * img_width + k_c]
413         })
414         .sum::<f32>()
415     })
416     .sum::<f32>();
417   (
418     -(cnt as isize),
419     FloatOrd((right_compatibility + down_compatibility) / (cnt as f32)),
420   )
421 }
422
423 // Проверка матриц на совместимость (возможность объединения)
424 fn can_merge_matrices(
```

```
425   img_width: usize,
426   img_height: usize,
427   pieces_compatibility: &[Vec<Vec<f32>>; 2],
428   m_x: &Matrix,
429   m_y: &Matrix,
430 ) -> MatrixCompatibility {
431   // Отсортированные детали второй матрицы
432   let mut tmp_y: Vec<_> = m_y
433     .iter()
434     .flatten()
435     .cloned()
436     .enumerate()
437     .filter_map(|(i, x)| {
438       if x.0 != usize::MAX {
439         Some((x, i))
440       } else {
441         None
442       }
443     })
444     .collect();
445   tmp_y.sort_unstable();
446   // Количество деталей, встречающихся в обоих матрицах
447   let mut cnt = 0;
448   // Сдвиг между общими деталями в первой и второй матрицах
449   let mut shared_shift = None;
450   // Проверка существования детали из первой матрицы во второй
451   for (x_ind, x) in m_x.iter().flatten().enumerate() {
452     if x.0 == usize::MAX {
453       continue;
454     }
455     if let Ok(y_ind_tmp) = tmp_y.binary_search_by_key(&x, |pr| &pr.0) {
456       cnt += 1;
457       // Вычисление общего сдвига
458       if shared_shift.is_none() {
459         let y_ind = tmp_y[y_ind_tmp].1;
460         let (x_r, x_c) = (x_ind / m_x[0].len(), x_ind % m_x[0].len());
461         let (y_r, y_c) = (y_ind / m_y[0].len(), y_ind % m_y[0].len());
462         shared_shift = Some((
463           (y_r as isize) - (x_r as isize),
464           (y_c as isize) - (x_c as isize),
465         ));
466       }
467       // Для проверки достаточно двух общих деталей
468       if cnt == 2 {
469         break;
470       }
471     }
472   }
```

```
473     // Если не больше одной общей детали, то матрицы не связаны друг с другом
474     // Исключение для матриц-пар
475     if cnt < 2
476       && !(cnt == 1
477         && ((m_x.len() == 1 || m_x[0].len() == 1) ^ (m_y.len() == 1 || m_y[0].len() == 1)))
478     {
479       return MatrixCompatibility::NotRelated;
480     }
481     let (shared_shift_r, shared_shift_c) = shared_shift.unwrap();
482
483     // Размеры матриц x и y
484     let (m_x_r, m_x_c) = (m_x.len(), m_x[0].len());
485     let (m_y_r, m_y_c) = (m_y.len(), m_y[0].len());
486     // Переход из системы координат матрицы x к с. к. новой матрицы
487     let (d_r, d_c) = (
488       max(0, shared_shift_r) as usize,
489       max(0, shared_shift_c) as usize,
490     );
491     // Переход из системы координат матрицы y к с. к. новой матрицы
492     let (d_minus_shift_r, d_minus_shift_c) = (
493       ((d_r as isize) - shared_shift_r) as usize,
494       ((d_c as isize) - shared_shift_c) as usize,
495     );
496     // Размер новой матрицы
497     let (m_new_r, m_new_c) = (
498       max(m_x_r + d_r, m_y_r + d_minus_shift_r),
499       max(m_x_c + d_c, m_y_c + d_minus_shift_c),
500     );
501     // Матрица должна быть не больше изображения
502     if m_new_r > img_height || m_new_c > img_width {
503       return MatrixCompatibility::NotRelated;
504     }
505
506     // Новая матрица
507     let mut m_new = vec![vec![(usize::MAX, usize::MAX); m_new_c]; m_new_r];
508     // Копирование в неё матрицы x
509     for (r_x, v_x) in m_x.iter().enumerate() {
510       for (c_x, x) in v_x.iter().enumerate() {
511         let (r_new, c_new) = (r_x + d_r, c_x + d_c);
512         m_new[r_new][c_new] = *x;
513       }
514     }
515     // Копирование в неё матрицы y
516     for (r_y, v_y) in m_y.iter().enumerate() {
517       for (c_y, y) in v_y.iter().enumerate() {
518         let (r_new, c_new) = (r_y + d_minus_shift_r, c_y + d_minus_shift_c);
519         if m_new[r_new][c_new].0 != usize::MAX && m_new[r_new][c_new] != *y {
520           return MatrixCompatibility::Incompatible;
```

```
521         }
522         m_new[r_new][c_new] = *y;
523       }
524     }
525     // Если новая матрица совпадает с одной из объединяемых, то объединять не нужно
526     if m_new == *m_x || m_new == *m_y {
527       return MatrixCompatibility::Incompatible;
528     }
529
530     // Приоритет новой матрицы
531     let m_new_priority = matrix_priority(img_width, pieces_compatibility, &m_new);
532
533     // Отсортированные детали новой матрицы
534     let mut tmp_new: Vec<_> = m_new
535       .into_iter()
536       .flatten()
537       .filter(|x| x.0 != usize::MAX)
538       .collect();
539     tmp_new.sort_unstable();
540     let tmp_new_len = tmp_new.len();
541     tmp_new.dedup();
542     // Если в матрице повторялись детали, то объединение невозможно
543     if tmp_new.len() != tmp_new_len {
544       return MatrixCompatibility::Incompatible;
545     }
546     // Матрицы совместимы
547     MatrixCompatibility::Compatible(shared_shift_r, shared_shift_c, m_new_priority)
548   }
549
550   // Объединение матриц
551   fn merge_matrices(
552     m_x: &Matrix,
553     m_y: &Matrix,
554     shared_shift_r: isize,
555     shared_shift_c: isize,
556   ) -> Matrix {
557     // Размеры матриц x и y
558     let (m_x_r, m_x_c) = (m_x.len(), m_x[0].len());
559     let (m_y_r, m_y_c) = (m_y.len(), m_y[0].len());
560     // Переход из системы координат матрицы x к с. к. новой матрицы
561     let (d_r, d_c) = (
562       max(0, shared_shift_r) as usize,
563       max(0, shared_shift_c) as usize,
564     );
565     // Переход из системы координат матрицы y к с. к. новой матрицы
566     let (d_minus_shift_r, d_minus_shift_c) = (
567       ((d_r as isize) - shared_shift_r) as usize,
568       ((d_c as isize) - shared_shift_c) as usize,
```

```
569     );
570     // Размер новой матрицы
571     let (m_new_r, m_new_c) = (
572       max(m_x_r + d_r, m_y_r + d_minus_shift_r),
573       max(m_x_c + d_c, m_y_c + d_minus_shift_c),
574     );
575
576     // Новая матрица
577     let mut m_new = vec![vec![(usize::MAX, usize::MAX); m_new_c]; m_new_r];
578     // Копирование в неё матрицы x
579     for (r_x, v_x) in m_x.iter().enumerate() {
580       for (c_x, x) in v_x.iter().enumerate() {
581         let (r_new, c_new) = (r_x + d_r, c_x + d_c);
582         m_new[r_new][c_new] = *x;
583       }
584     }
585     // Копирование в неё матрицы y
586     for (r_y, v_y) in m_y.iter().enumerate() {
587       for (c_y, y) in v_y.iter().enumerate() {
588         let (r_new, c_new) = (r_y + d_minus_shift_r, c_y + d_minus_shift_c);
589         m_new[r_new][c_new] = *y;
590       }
591     }
592     m_new
593   }
594
595   // Объединение матриц
596   fn merge_matrices_groups(
597     img_width: usize,
598     img_height: usize,
599     pieces_compatibility: &[Vec<Vec<f32>>; 2],
600     sl_all: Vec<Vec<Matrix>>,
601   ) -> Vec<Matrix> {
602     type AvailablePairsType = BTreeMap<MatrixPriority, Vec<(MatrixCompatibility, usize,
    ↪  usize)>>;
603
604     // Пары совместимых или несовместимых матриц в текущем множестве
605     // Сопоставление приоритету матрицы (результата объединения матриц или одной матрицы,
    ↪  выбранной из двух)
606     // кортежей из совместимостей матриц и их индексов
607     let mut available_pairs: AvailablePairsType = BTreeMap::new();
608     // Матрицы, удалённые из множества
609     let mut used = Vec::new();
610     // Множество-результат объединения пар матриц
611     let mut matrices_last = Vec::new();
612     // Обработка новой матрицы
613     let add_new_matrix = |available_pairs: &mut AvailablePairsType,
614                           used: &mut Vec<bool>,
```

```
615             matrices_last: &mut Vec<(Matrix, MatrixPriority)>,
616             m_x: Matrix,
617             check: bool| {
618     let m_x_i = matrices_last.len();
619     let m_x_priority = matrix_priority(img_width, pieces_compatibility, &m_x);
620     if check {
621       // Обработка пар из текущей матрицы и всех остальных в множестве
622       let tmp: Vec<_> = matrices_last
623         .par_iter()
624         .enumerate()
625         .filter_map(|(m_y_i, (m_y, m_y_priority))| {
626           // Матрица уже удалена
627           if used[m_y_i] {
628             return None;
629           }
630           let order = m_x_priority < *m_y_priority;
631           // Определение совместимости матриц
632           let can_merge = if order {
633             can_merge_matrices(img_width, img_height, pieces_compatibility, &m_x, m_y)
634           } else {
635             can_merge_matrices(img_width, img_height, pieces_compatibility, m_y, &m_x)
636           };
637
638           // Приоритет матрицы-результата
639           let pairs_key = match can_merge {
640             // Если матрицы совместимы, то приоритет матрицы-объединения
641             MatrixCompatibility::Compatible(_, _, m_new_priority) => m_new_priority,
642             MatrixCompatibility::Incompatible => {
643               // Если матрицы не совместимы, выбирается более приоритетная
644               if order {
645                 m_x_priority
646               } else {
647                 *m_y_priority
648               }
649             }
650             // Если матрицы не связаны, то пара не добавляется
651             MatrixCompatibility::NotRelated => return None,
652           };
653
654           if order {
655             Some((pairs_key, (can_merge, m_x_i, m_y_i)))
656           } else {
657             Some((pairs_key, (can_merge, m_y_i, m_x_i)))
658           }
659         })
660         .collect();
661       // Добавление всех пар
662       for (pairs_key, x) in tmp {
```

```
663         let v = match available_pairs.get_mut(&pairs_key) {
664           Some(v) => v,
665           None => {
666             available_pairs.insert(pairs_key, Vec::new());
667             available_pairs.get_mut(&pairs_key).unwrap()
668           }
669         };
670         v.push(x);
671       }
672     }
673     // Добавление матрицы в множество
674     used.push(false);
675     matrices_last.push((m_x, m_x_priority));
676   };
677
678   // Обработка всех маленьких циклов по убыванию порядка
679   for sl_curr in sl_all.into_iter().rev() {
680     for sl in sl_curr.into_iter() {
681       add_new_matrix(
682         &mut available_pairs,
683         &mut used,
684         &mut matrices_last,
685         sl,
686         true,
687       );
688     }
689
690     // Пока возможно, выбирается пара совместимых или не совместимых матриц и заменяется
    ↪  одной
691     while !available_pairs.is_empty() {
692       let (pairs_key, pairs_v) = available_pairs.iter_mut().next().unwrap();
693       if pairs_v.is_empty() {
694         let pairs_key = *pairs_key;
695         available_pairs.remove(&pairs_key);
696         continue;
697       }
698       let (m_comp, x_i, y_i) = pairs_v.pop().unwrap();
699       if used[x_i] || used[y_i] {
700         continue;
701       }
702
703       match m_comp {
704         // Если матрицы совместимы, то оригинальные матрицы удаляются и добавляется их
    ↪  объединение
705         MatrixCompatibility::Compatible(shared_shift_r, shared_shift_c, _) => {
706           let m_new = merge_matrices(
707             &matrices_last[x_i].0,
708             &matrices_last[y_i].0,
```

```
709               shared_shift_r,
710               shared_shift_c,
711             );
712
713             used[x_i] = true;
714             used[y_i] = true;
715             add_new_matrix(
716               &mut available_pairs,
717               &mut used,
718               &mut matrices_last,
719               m_new,
720               true,
721             );
722           }
723           // Если матрицы не совместимы, то удаляется менее приоритетная
724           MatrixCompatibility::Incompatible => {
725             used[y_i] = true;
726           }
727           _ => unreachable!(),
728         }
729       }
730
731       // Удаление всех матриц, помеченных к удалению
732       let matrices_next: Vec<_> = matrices_last
733         .iter()
734         .enumerate()
735         .filter_map(|(i, m)| if used[i] { None } else { Some(m.0.clone()) })
736         .collect();
737       available_pairs.clear();
738       used.clear();
739       matrices_last.clear();
740       for m in matrices_next {
741         add_new_matrix(
742           &mut available_pairs,
743           &mut used,
744           &mut matrices_last,
745           m,
746           false,
747         );
748       }
749     }
750
751     // Результирующее множество матриц, отсортированных по приоритету
752     matrices_last.sort_by_key(|m| m.1);
753     matrices_last.into_iter().map(|m| m.0).collect()
754   }
755
756   // Отрезание почти пустых краёв
```

```
757 fn trim(solution_r: usize, solution_c: usize, solution: Solution) -> (usize, usize,
    ↪ Solution) {
758   // Границы изображения
759   let (mut min_r, mut max_r, mut min_c, mut max_c) = (0, solution_r - 1, 0, solution_c - 1);
760   loop {
761     // Количество непустых деталей в верхней строке
762     let up_count = (min_c..=max_c)
763       .map(|c| solution[min_r * solution_c + c])
764       .filter(|x| x.0 != usize::MAX)
765       .count() as f32;
766     // Если слишком мало, то строка вырезается
767     if up_count / ((max_c - min_c + 1) as f32) <= TRIM_RATE {
768       min_r += 1;
769       continue;
770     }
771
772     // Количество непустых деталей в нижней строке
773     let down_count = (min_c..=max_c)
774       .map(|c| solution[max_r * solution_c + c])
775       .filter(|x| x.0 != usize::MAX)
776       .count() as f32;
777     // Если слишком мало, то строка вырезается
778     if down_count / ((max_c - min_c + 1) as f32) <= TRIM_RATE {
779       max_r -= 1;
780       continue;
781     }
782
783     // Количество непустых деталей в левом столбце
784     let left_count = (min_r..=max_r)
785       .map(|r| solution[r * solution_c + min_c])
786       .filter(|x| x.0 != usize::MAX)
787       .count() as f32;
788     // Если слишком мало, то столбец вырезается
789     if left_count / ((max_r - min_r + 1) as f32) <= TRIM_RATE {
790       min_c += 1;
791       continue;
792     }
793
794     // Количество непустых деталей в правом столбце
795     let right_count = (min_r..=max_r)
796       .map(|r| solution[r * solution_c + max_c])
797       .filter(|x| x.0 != usize::MAX)
798       .count() as f32;
799     // Если слишком мало, то столбец вырезается
800     if right_count / ((max_r - min_r + 1) as f32) <= TRIM_RATE {
801       max_c -= 1;
802       continue;
803     }
```

```
804
805       break;
806     }
807
808     // Вырезание результата
809     (
810       max_r - min_r + 1,
811       max_c - min_c + 1,
812       (min_r..=max_r)
813         .flat_map(|r| {
814           (min_c..=max_c)
815             .map(|c| solution[r * solution_c + c])
816             .collect::<Vec<_>>()
817         })
818         .collect(),
819     )
820   }
821
822   // Жадное заполнение пустых позиций
823   fn fill_greedy(
824     img_width: usize,
825     img_height: usize,
826     pieces_compatibility: &[Vec<Vec<f32>>; 2],
827     old_solution_r: usize,
828     old_solution_c: usize,
829     old_solution: Solution,
830   ) -> Solution {
831     let (missing_r, missing_c) = (img_height - old_solution_r, img_width - old_solution_c);
832     // Так как дополнение изображения может идти в любую сторону,
833     // высота и ширина больше необходимой на количество отсутствующих строк и столбцов
834     let (solution_r, solution_c) = (
835       old_solution_r + 2 * missing_r,
836       old_solution_c + 2 * missing_c,
837     );
838     let mut solution = vec![(usize::MAX, usize::MAX); solution_r * solution_c];
839     // Копирование старого решения в новое
840     for r in 0..old_solution_r {
841       for c in 0..old_solution_c {
842         solution[(r + missing_r) * solution_c + c + missing_c] =
843           old_solution[r * old_solution_c + c];
844       }
845     }
846     // Текущие грани построенного изображения
847     let (mut min_r, mut max_r, mut min_c, mut max_c) =
848       (missing_r, img_height - 1, missing_c, img_width - 1);
849
850     // Неиспользованные детали
851     let mut free_pieces: IndexSet<_, FxBuildHasher> = (0..img_height)
```

```
    .flat_map(|r| (0..img_width).map(move |c| (r, c)))
    .collect();
  for piece in solution.iter() {
    free_pieces.remove(piece);
  }

  // Свободные позиции (по количеству свободных соседей)
  let mut free_positions: [IndexSet<(usize, usize), FxBuildHasher>; 5] = [
    IndexSet::with_hasher(FxBuildHasher::default()),
    IndexSet::with_hasher(FxBuildHasher::default()),
    IndexSet::with_hasher(FxBuildHasher::default()),
    IndexSet::with_hasher(FxBuildHasher::default()),
    IndexSet::with_hasher(FxBuildHasher::default()),
  ];
  // Добавление позиции в множество свободных
  let add_to_free_positions = |solution: &mut Solution,
                free_positions: &mut [IndexSet<(usize, usize), FxBuildHasher>;
                     5],
                min_r: usize,
                max_r: usize,
                min_c: usize,
                max_c: usize,
                r: usize,
                c: usize| {
    if solution[r * solution_c + c].0 != usize::MAX {
      return;
    }
    // Вычисление количества свободных соседей
    let mut cnt = 0;
    for dr in [-1isize, 1] {
      for dc in [-1isize, 1] {
        let (new_r, new_c) = ((r as isize) + dr, (c as isize) + dc);
        if new_r < 0 || new_c < 0 {
          continue;
        }
        let (new_r, new_c) = (new_r as usize, new_c as usize);
        if 1 + max(max_r, new_r) - min(min_r, new_r) > img_height
          || 1 + max(max_c, new_c) - min(min_c, new_c) > img_width
        {
          continue;
        }
        if solution[new_r * solution_c + new_c].0 == usize::MAX {
          cnt += 1;
        }
      }
    }
    free_positions[cnt].insert((r, c));
  };
```

```
900     // Добавление всех свободных позиций
901     for r in 0..solution_r {
902       for c in 0..solution_c {
903         add_to_free_positions(
904           &mut solution,
905           &mut free_positions,
906           min_r,
907           max_r,
908           min_c,
909           max_c,
910           r,
911           c,
912         );
913       }
914     }
915
916     // Пока есть неиспользованные детали, найти для каждой позицию
917     while !free_pieces.is_empty() {
918       // Выбирается позиция с наименьшим количеством свободных соседей
919       for cnt in 0..=4 {
920         if free_positions[cnt].is_empty() {
921           continue;
922         }
923         let pos = free_positions[cnt].iter().find(|&&(r, c)| {
924           1 + max(max_r, r) - min(min_r, r) <= img_height
925             && 1 + max(max_c, c) - min(min_c, c) <= img_width
926         });
927         if pos.is_none() {
928           continue;
929         }
930         let (pos_r, pos_c) = *pos.unwrap();
931
932         // Деталь слева
933         let (left_piece_r, left_piece_c) = if pos_c == 0 {
934           (usize::MAX, usize::MAX)
935         } else {
936           solution[pos_r * solution_c + pos_c - 1]
937         };
938         // Деталь справа
939         let (right_piece_r, right_piece_c) = if pos_c == solution_c - 1 {
940           (usize::MAX, usize::MAX)
941         } else {
942           solution[pos_r * solution_c + pos_c + 1]
943         };
944         // Деталь сверху
945         let (up_piece_r, up_piece_c) = if pos_r == 0 {
946           (usize::MAX, usize::MAX)
947         } else {
```

```
948           solution[(pos_r - 1) * solution_c + pos_c]
949         };
950         // Деталь снизу
951         let (down_piece_r, down_piece_c) = if pos_r == solution_r - 1 {
952           (usize::MAX, usize::MAX)
953         } else {
954           solution[(pos_r + 1) * solution_c + pos_c]
955         };
956
957         // Наименьшая несовместимость
958         let mut best_compatibility = f32::INFINITY;
959         // Наиболее подходящая деталь
960         let mut best_piece = (usize::MAX, usize::MAX);
961         // Обработка всех свободных деталей
962         for (piece_r, piece_c) in &free_pieces {
963           // Несовместимость
964           let mut res = 0.0f32;
965           // Обработка детали слева
966           if left_piece_r != usize::MAX {
967             res += pieces_compatibility[0][left_piece_r * img_width + left_piece_c]
968               [piece_r * img_width + piece_c];
969             if res >= best_compatibility {
970               continue;
971             }
972           }
973           // Обработка детали справа
974           if right_piece_r != usize::MAX {
975             res += pieces_compatibility[0][piece_r * img_width + piece_c]
976               [right_piece_r * img_width + right_piece_c];
977             if res >= best_compatibility {
978               continue;
979             }
980           }
981           // Обработка детали сверху
982           if up_piece_r != usize::MAX {
983             res += pieces_compatibility[1][up_piece_r * img_width + up_piece_c]
984               [piece_r * img_width + piece_c];
985             if res >= best_compatibility {
986               continue;
987             }
988           }
989           // Обработка детали снизу
990           if down_piece_r != usize::MAX {
991             res += pieces_compatibility[1][piece_r * img_width + piece_c]
992               [down_piece_r * img_width + down_piece_c];
993             if res >= best_compatibility {
994               continue;
995             }
```

```
996           }
997
998           // Обновление наименее несовместимой детали
999           best_piece = (*piece_r, *piece_c);
1000          best_compatibility = res;
1001        }
1002
1003        // Назначение детали, удаление позиции из свободных
1004        free_pieces.remove(&best_piece);
1005        free_positions[cnt].remove(&(pos_r, pos_c));
1006        solution[pos_r * solution_c + pos_c] = best_piece;
1007
1008        // Обновление верхней границы
1009        if pos_r < min_r {
1010          min_r = pos_r;
1011        }
1012        // Обновление нижней границы
1013        else if pos_r > max_r {
1014          max_r = pos_r;
1015        }
1016        // Обновление левой границы
1017        if pos_c < min_c {
1018          min_c = pos_c;
1019        }
1020        // Обновление правой границы
1021        else if pos_c > max_c {
1022          max_c = pos_c;
1023        }
1024
1025        // Обновление свободной позиции слева
1026        if pos_c != 0 {
1027          for cnt_adj in 0..=4 {
1028            if free_positions[cnt_adj].remove(&(pos_r, pos_c - 1)) {
1029              add_to_free_positions(
1030                &mut solution,
1031                &mut free_positions,
1032                min_r,
1033                max_r,
1034                min_c,
1035                max_c,
1036                pos_r,
1037                pos_c - 1,
1038              );
1039              break;
1040            }
1041          }
1042        }
1043        // Обновление свободной позиции справа
```

```
1044        if pos_c != solution_c - 1 {
1045          for cnt_adj in 0..=4 {
1046            if free_positions[cnt_adj].remove(&(pos_r, pos_c + 1)) {
1047              add_to_free_positions(
1048                &mut solution,
1049                &mut free_positions,
1050                min_r,
1051                max_r,
1052                min_c,
1053                max_c,
1054                pos_r,
1055                pos_c + 1,
1056              );
1057              break;
1058            }
1059          }
1060        }
1061        // Обновление свободной позиции сверху
1062        if pos_r != 0 {
1063          for cnt_adj in 0..=4 {
1064            if free_positions[cnt_adj].remove(&(pos_r - 1, pos_c)) {
1065              add_to_free_positions(
1066                &mut solution,
1067                &mut free_positions,
1068                min_r,
1069                max_r,
1070                min_c,
1071                max_c,
1072                pos_r - 1,
1073                pos_c,
1074              );
1075              break;
1076            }
1077          }
1078        }
1079        // Обновление свободной позиции снизу
1080        if pos_r != solution_r - 1 {
1081          for cnt_adj in 0..=4 {
1082            if free_positions[cnt_adj].remove(&(pos_r + 1, pos_c)) {
1083              add_to_free_positions(
1084                &mut solution,
1085                &mut free_positions,
1086                min_r,
1087                max_r,
1088                min_c,
1089                max_c,
1090                pos_r + 1,
1091                pos_c,
```

```
            );
            break;
          }
        }
      }

      break;
    }
  }

  assert!(free_pieces.is_empty());
  assert_eq!(1 + max_r - min_r, img_height);
  assert_eq!(1 + max_c - min_c, img_width);
  // Вырезание результата
  (min_r..=max_r)
    .flat_map(|r| {
      (min_c..=max_c)
        .map(|c| solution[r * solution_c + c])
        .collect::<Vec<_>>()
    })
    .collect()
}

// Шаг алгоритма (обработка одного изображения)
pub fn algorithm_step(
  img_width: usize,
  img_height: usize,
  pieces_compatibility: &[Vec<Vec<f32>>; 2],
  pieces_match_candidates: &[Vec<Vec<(usize, usize)>>; 2],
) -> Vec<Solution> {
  // Нахождение маленьких циклов
  let sl_all = small_loops(img_width, img_height, pieces_match_candidates);
  // Объединение матриц
  let matrices_last = merge_matrices_groups(img_width, img_height, pieces_compatibility,
  ↪  sl_all);
  // Выбор лучшей матрицы в качестве решения
  let matrices_last_first = matrices_last.first().unwrap();
  let (solution_r, solution_c) = (matrices_last_first.len(), matrices_last_first[0].len());
  let mut solution = vec![(usize::MAX, usize::MAX); solution_r * solution_c];
  for r in 0..matrices_last_first.len() {
    for c in 0..matrices_last_first[r].len() {
      if matrices_last_first[r][c].0 != usize::MAX {
        solution[r * solution_c + c] = matrices_last_first[r][c];
      }
    }
  }
  // Отрезание почти пустых краёв и жадное заполнение пустых позиций
  let (solution_r, solution_c, solution) = trim(solution_r, solution_c, solution);
```

```
    let new_solution = fill_greedy(
      img_width,
      img_height,
      pieces_compatibility,
      solution_r,
      solution_c,
      solution,
    );
    vec![new_solution]
  }
```