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

ФАКУЛЬТЕТ ИУ «Информатика и системы управления»

КАФЕДРА ИУ-7 «Программное обеспечение ЭВМ и информационные технологии»

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА

К ВЫПУСКНОЙ КВАЛИФИКАЦИОННОЙ РАБОТЕ

НА ТЕМУ:

«Метод сжатия статических изображений без потерь на основе алгоритма Хаффмана»

Студент ИУ7-42М _____ К. Э. Ковалец
(Группа) (Подпись, дата) (И. О. Фамилия)

Руководитель ВКР _____ Н. В. Новик
(Подпись, дата) (И. О. Фамилия)

Нормоконтролер _____ Д. Ю. Мальцева
(Подпись, дата) (И. О. Фамилия)

2025 г.

РЕФЕРАТ

Рассчетно-пояснительная записка к выпускной квалификационной работе «Метод сжатия статических изображений без потерь на основе алгоритма Хаффмана» содержит 73 страниц, 4 части, 17 рисунков, 5 таблиц и список используемых источников из 26 наименования.

Ключевые слова: сжатие изображений, алгоритм Хаффмана, дерево Хаффмана алгоритм LZW, сжатие без потерь, статические изображения, bmp-файлы.

Объект разработки — метод сжатия статических изображений без потерь.

Цель работы: разработать метод сжатия статических изображений без потерь на основе алгоритма Хаффмана.

В первой части работы рассмотрены основные методы сжатия данных без потерь. Сформулированы критерии сравнения методов сжатия. Выполнен сравнительный анализ исследуемых методов по выделенным критериям. Описана формальная постановка задачи в виде IDEF0-диаграммы.

Во второй части разработан метод сжатия статических изображений на основе алгоритма Хаффмана. Описаны основные особенности предлагаемого метода. Изложены ключевые этапы метода в виде схем алгоритмов.

В третьей части обоснован выбор программных средств для реализации предложенного метода. Описан формат входных и выходных данных. Разработано программное обеспечение, реализующее описанный метод. Описано взаимодействие пользователя с программным обеспечением.

В четвертой части в рамках исследования проведено сравнение разработанного метода сжатия статических изображений без потерь с рассмотренными аналогами. В качестве критериев сравнения использовались полученная степень сжатия файла и размер информации, необходимой для распаковки изображения.

Разработанный метод сжатия статических изображений без потерь может применяться в системах хранения и передачи данных, где важна высокая степень сжатия изображений без потери их качества.

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Методы сжатия статических изображений активно применяются для хранения и передачи растровых изображений. За счет уменьшения размера файла, методы сжатия позволяют достичь увеличения скорости передачи данных, а также уменьшения занимаемого на диске места.

Например, одной из областей применения методов сжатия изображений является медицина, где важно передавать снимки КТ, МРТ, УЗИ, рентгеновские снимки в медицинских информационных системах с привлечением минимальных ресурсов.

Также методы сжатия изображений активно применяются в интернет-магазинах, где скорость загрузки снимков товаров на странице с ассортиментом является ключевой.

Задача сжатия изображений остается актуальной, так как файлов с ними с каждым днем становится все больше. Совершенствование методов сжатия и разработка новых алгоритмов остается важной задачей для обеспечения эффективного хранения и передачи изображений.

Целью выпускной квалификационной работы является разработка метода сжатия статических изображений без потерь на основе алгоритма Хаффмана.

Для достижения поставленной цели необходимо выполнить следующие задачи:

— провести аналитический обзор известных методов сжатия статических изображений;

— разработать метод сжатия статических изображений без потерь на основе алгоритма Хаффмана;

— разработать программное обеспечение для демонстрации работы созданного метода;

— провести сравнение разработанного метода с аналогами по степени сжатия изображений.

1 Аналитическая часть

1.1 Анализ предметной области

Все изображения можно разделить на две группы.

1. **Статические изображения** — это визуальные представления, не содержащие анимации или элементов взаимодействия. Они остаются неизменными и могут быть представлены в виде рисунков, фотографий, диаграмм или любых других неподвижных изображений сцен или объектов.

2. **Динамические изображения** — это визуальные представления, содержащие движения или изменения в течение времени. В отличие от статических, они могут быть анимированными, интерактивными (реагировать на клики или наведение) или включать переходы. Примеры: анимация, видео, GIF-файлы, интерактивная графика.

Существуют два основных типа сжатия изображений — с потерями и без потерь. Сжатие без потерь позволяет уменьшить размер файла с сохранением всех деталей исходного изображения. Сжатие с потерями приводит к более сильному уменьшению размера исходного файла ценой потери его деталей, следствием чего является потеря качества изображения.

Разработаны различные форматы файлов изображений, каждый из которых предназначен для определенных задач и использует в своей реализации собственные способы сжатия и сохранения информации. Например, формат JPEG используется для сжатия с потерями, тогда как формат PNG предназначен для сжатия изображений без потерь.

Уменьшить размер сжатого файла можно путем изменения параметров сжатия, таких как качество изображения и его разрешение. Более высокие значения этих параметров приведут к увеличению размера полученнного файла.

Сокращение числа цветов в изображении также будет способствовать уменьшению размера сжатого файла, так как приведет к снижению числа байт, необходимых для представления каждого пикселя.

1.2 Методы сжатия статических изображений без потерь

1.2.1 Метод RLE

Метод RLE основан на идее кодирования последовательностей повторяющихся значений. Задача данного метода заключается в нахождении цепочек одинаковых символов и замене их на одно значение из последовательности и количество повторений. Такой алгоритм подходит для сжатия как текстовых файлов, так и изображений, где в роли каждого символа выступает набор байт, необходимый для хранения одного пикселя.

Например, если изображение содержит несколько подряд идущих пикселей одного цвета, то рассматриваемый метод закодирует данную последовательность в виде одного пикселя и количества его повторений. Из описания алгоритма можно сделать вывод о том, что он эффективен для изображений с большим количеством повторяющихся участков или частыми областями одного цвета. Однако RLE не подходит для изображений с большим количеством деталей, где, помимо низкой степени сжатия, может привести к увеличению размера файла.

Рассматриваемый метод относится к сжатию без потерь и может использоваться как отдельно, так и в комбинации с другими алгоритмами для достижения лучшего результата.

Процесс работы алгоритма можно описать следующим образом.

1. Создать пустую строку для хранения результата.
2. Произвести проход по всем пикселям изображения (по всем строкам слева направо).
3. Для каждого пикселя.
    - Если он совпадает с предыдущим, увеличить счётчик повторений.
    - Если отличается или достигнут конец строки:
        - В результирующую строку записать значение предыдущего пикселя и количество его повторений.
        - Сбросить счётчик и начать отсчёт заново.

4. После завершения прохода добавить данные о последнем пикселе и количестве его повторений.

5. Вернуть строку с закодированными данными.

Пример работы алгоритма:

— входная строка: `FFFFCCBBBDAA`;

— закодированные данные: `4F2C3B1D2A`.

1.2.2 Словарные методы

Словарные методы сжатия, например LZW (Lempel-Ziv-Welch), основаны на использовании специального словаря, в котором повторяющиеся последовательности заменяются более короткими кодами. Такой подход позволяет существенно сократить объём данных за счёт замены часто встречающихся шаблонов уникальными кодами из словаря.

Принцип работы алгоритма LZW заключается в том, что он анализирует входные данные и постепенно строит словарь, где каждой повторяющейся последовательности символов присваивается определённый код. После создания словаря эти коды используются для замены изначальных последовательностей, что и обеспечивает сжатие.

Изначально словарь содержит все возможные односимвольные последовательности. Алгоритм считывает входной поток слева направо, формируя текущую последовательность символов. Как только встречается цепочка значений, отсутствующая в словаре, она добавляется туда, а в выходной поток записывается код уже известной части. Этот процесс продолжается до тех пор, пока весь поток данных не будет обработан. При декодировании используется тот же самый словарь для восстановления исходной информации.

Пошаговый алгоритм метода LZW можно описать следующим образом.

1. Создать словарь со всеми возможными односимвольными значениями (пикселями).

2. Установить начальное значение последовательности W равным первому символу входных данных.

3. Последовательно считывать символа K из входного потока.

— Проверить, содержится ли комбинация W + K в текущем словаре.

— Если да, то W := W + K, после чего продолжить анализ.

— Если нет.

– Добавить код для текущей последовательности W в результат.

– Добавить новую комбинацию W + K в словарь с уникальным кодом.

– Присвоить W значение K.

4. Добавить код последней оставшейся в буфере последовательности W.

5. Вернуть полученную рузультат в виде последовательности кодов.

1.2.3 Унарное кодирование

Унарное кодирование — это метод статистического кодирования, который может использоваться в сжатии изображений для представления значений пикселей, особенно в случае, когда значения пикселей ограничены небольшим диапазоном.

Если есть изображение в оттенках серого, где значения пикселей находятся в диапазоне от 0 до 7 (0 соответствует самому частому оттенку серого, 7 — самому редкому), то можно использовать унарное кодирование для представления этих значений.

В данном алгоритме каждое значение представляется последовательностью единиц, за которой следует ноль. Количество единиц в последовательности определяет значение пикселя. Например, значение 3 будет представлено как 1110, где три единицы обозначают значение 3, а ноль обозначает конец кода. Значение 0 будет представлено как 0.

Унарное кодирование может применяться для сжатия изображений, если значения пикселей имеют ограниченный диапазон и малую вариацию. В случае большого диапазона значений или большой вариации, унарное кодирование может привести к увеличению размера файла. В таких случаях следует использовать другие методы сжатия изображений.

1.2.4 Метод Хаффмана

Алгоритм Хаффмана — метод сжатия данных, основанный на замене часто встречающихся символов более короткими кодами, а реже встречающих-

ся — более длинными. Данный алгоритм может использоваться для сжатия изображений без потерь, в этом случае пиксели выступают в роли кодируемых символов.

Для реализации метода Хаффмана необходимо знать вероятности вхождения символов в сообщение, которые будут использоваться для наделения весом каждого пикселя.

Алгоритм построения дерева Хаффмана.

1. Строится список свободных узлов на основе символов входного алфавита. Для этого каждый лист наделяется весом, равным вероятности вхождений символа в ожидаемое сообщение.

2. Из свободных узлов дерева выбираются 2 с наименьшими весами.

3. Для выбранных узлов создается родитель с суммарным весом.

4. Дуге к потомку с меньшим весом ставится в соответствие бит 1, с большим весом — бит 0.

5. Родитель заменяет в списке свободных узлов двое своих потомков.

6. Пункты 2–5 повторяются до тех пор, пока в списке свободных узлов не останется только один узел, который станет корнем дерева.

Построенное дерево будет использоваться для определения кодов символов (пикселей) из полученного сообщения (изображения). Для этого необходимо пройти путь от листа дерева, соответствующего текущему символу, до корня, дописывая биты при каждом переходе по ветвям дерева. Таким образом, полученная последовательность битов будет соответствовать записанному в обратном порядке коду данного символа (пикселя).

Несмотря на то, что коды Хаффмана имеют переменную длину, они могут быть однозначно декодированы, так как обладают уникальными префиксами.

Для восстановления исходного изображения декодер должен иметь доступ к таблице частот, которая использовалась при кодировании. Это означает, что длина сжатого сообщения увеличится на длину таблицы частот, которая должна быть передана перед данными. Также к недостатку рассматриваемого

метода можно отнести необходимость повторного прохода по изображению: один для построения дерева Хаффмана, другой для кодирования.

1.2.5 Арифметическое кодирование

Арифметическое кодирование является блочным и имеет уникальный выходной код для каждого возможного входного сообщения. Нельзя разбить его на коды отдельных символов, в отличие от кодов Хаффмана, которые присваиваются отдельным буквам входного алфавита.

При использовании арифметического кодирования, слово можно представить в виде двоичной дроби из интервала от 0 до 1. С увеличением длины слова, этот интервал сокращается, требуя больше битов для его определения.

Алгоритм арифметического кодирования нуждается в распределении частот символов входного сообщения. Более вероятные символы добавляют меньше битов к общей длине закодированного слова, так как приводят к уменьшению интервала на меньшую величину, по сравнению с маловероятными символами.

Арифметическое кодирование представляет каждый символ исходного текста с помощью числового интервала на оси, причем длина этого интервала соответствует вероятности появления символа. Начало интервала совпадает с концом интервала предыдущего символа в алфавите. Сумма всех интервалов равна единице. При поступлении нового символа, из текущего интервала «вырезается» подинтервал пропорционально длине и положению этого символа на оси.

Следует отметить, что арифметическое кодирование требует только одного прохода по данным для кодирования и декодирования, в отличие от алгоритма Хаффмана.

1.2.6 Сравнение методов сжатия без потерь

Сравнение предлагается проводить по следующим критериям.

1. Возможность кодирования данных за один проход.
2. Необходимость в таблице частот пикселей сжимаемого изображения.
3. Наличие в зашифрованном сообщении информации для дешифровщика (распаковщика).

4. Наличие у каждого сжатого пикселя своего кода.

Результаты сравнения методов сжатия изображений без потерь приведены в таблице 1.1.

Таблица 1.1 – Сравнение рассмотренных методов сжатия изображений без потерь

Метод сжатия	**Кр. 1**	**Кр. 2**	**Кр. 3**	**Кр. 4**
Алгоритм RLE	+	–	–	–
Словарные алгоритмы	+	–	+	–
Унарное кодирование	+	+	+	+
Алгоритм Хаффмана	–	+	+	+
Арифметическое кодирование	+	+	+	–

1.3 Методы сжатия статических изображений с потерями

1.3.1 Метод сжатия JPEG

JPEG (Joint Photographic Expert Group) — алгоритм сжатия изображений с потерями, который является неофициальным стандартом для полноцветных изображений. Он использует области пикселей размером 8x8, в которых яркость и цвет изменяются плавно. Из-за этого, при разложении матрицы такой области в двойной ряд по косинусам, только первые коэффициенты оказываются значимыми. Это позволяет осуществлять сжатие за счет плавных изменений цвета в изображении.

Алгоритм JPEG можно описать следующим образом.

1. **Разделение изображения на блоки.** Изображение разбивается на квадратные блоки пикселей размером 8x8 пикселей.

2. **Преобразование цветовой модели.** Изначально изображение может быть представлено в различных цветовых моделях, таких как RGB. Алгоритм JPEG преобразует изображение из цветовой модели RGB в цветовую модель YCbCr, где Y представляет яркость, а Cb и Cr представляют различные цветовые компоненты.

3. **Преобразование Фурье.** Применение двумерного дискретного преобразования Фурье (DCT) к каждому блоку пикселей 8x8. DCT преобразует блоки пикселей в блоки коэффициентов, представляющих частоты различных компонент изображения.

4. **Квантование.** Коэффициенты DCT квантуются путем деления на заранее заданную таблицу квантования. Значениями таблицы квантования являются коэффициенты, определяющие степень потери информации.

5. **Кодирование.** Коэффициенты квантования кодируются с использованием энтропийного кодирования, такого как алгоритм Хаффмана. Это сжатие используется для эффективного представления значений блоков пикселей в виде битовых последовательностей с переменной длиной.

6. **Упаковка данных.** Кодированные данные объединяются и упаковываются в структуру, известную как JPEG-файл. Этот файл может быть сохранен или передан по сети.

7. **Декодирование.** Для восстановления изображения из JPEG-файла процесс выполняется в обратном порядке. Данные распаковываются, декодируются с использованием алгоритма Хаффмана, применяется обратное квантование и обратное DCT преобразование. Затем изображение восстанавливается в цветовую модель RGB.

В результате алгоритма JPEG получается сжатое изображение с потерями, где некоторая информация оригинального изображения теряется, но с сохранением достаточного качества для большинства приложений.

1.3.2 Wavelet сжатие

Вейвлеты — это математические функции, которые используются для анализа частотных компонент данных. По сравнению с форматами сжатия, такими как JPEG и фрактальный алгоритм, вейвлеты выделяются тем, что не требуют предварительного разбиения исходного изображения на блоки, а могут быть применены к изображению в целом.

Для примера, возьмем одномерное преобразование Хаара. При нем каждой паре элементов сигнала сопоставляются два числа: полусумма элементов

и их полуразность. Так как из них можно получить исходную пару чисел, то преобразование будет обратимым.

В результате сигнал разложится на две составляющие: приближенное значение исходного сигнала (разрешение которого уменьшится в два раза) и уточняющую информацию.

Двумерное преобразование Хаара представляет собой объединение одномерных преобразований. В случае, если исходные данные будут записаны в виде матрицы, сначала выполнятся преобразования для каждой строки, а затем для полученных матриц выполнятся преобразования для каждого столбца. Это позволит получить четыре матрицы. Одна из которых будет содержать аппроксимацию исходного изображения (частота дискретизации которой уменьшится), в то время как три другие матрицы будут содержать уточняющую информацию.

Само сжатие достигается путем удаления некоторых коэффициентов из уточняющих матриц, что позволяет значительно уменьшить размер данных, сохраняя при этом важные детали изображения.

Данный метод был разработан специально для цветных и черно-белых изображений с плавными переходами, из-за чего подходит для обработки рентгеновских снимков и МРТ.

1.3.3 Фрактальный метод

Фрактальное сжатие изображений — это метод сжатия данных, основанный на использовании фрактальных кодов. Алгоритм использует идею самоподобия в изображении, чтобы создать компактное представление оригинала.

Алгоритм фрактального сжатия можно описать следующим образом.

1. **Деление изображения на блоки.** Изображение разбивается на небольшие блоки пикселей, называемые фракталами. Обычно используются группы размером 2x2, 4x4 или 8x8 пикселей. Это нужно, чтобы каждый блок мог быть рассмотрен как потенциально самоподобная часть изображения.

2. **Выбор базисного фрактала.** Из множества фракталов выбирается один в качестве базисного блока. Он будет использоваться для аппрок-

симации других фракталов.

3. **Поиск подходящих фракталов.** Для каждого блока изображения ищутся наиболее похожие фракталы из множества доступных блоков. Похожесть определяется с помощью различных метрик, таких как евклидово расстояние или среднеквадратичная ошибка.

4. **Аппроксимация фракталов.** Найденные похожие блоки аппроксимируются с использованием базисного фрактала. Аппроксимация может быть выполнена с помощью различных методов, например, с использованием аффинных преобразований или кодирования Хаффмана.

5. **Создание фрактального кода.** Для каждого блока сохраняется информация о выбранном базисном фрактале и преобразованиях, используемых для его аппроксимации. Эта информация составляет фрактальный код.

6. **Восстановление изображения.** Фрактальный код декодируется, чтобы получить информацию о выбранных базисных фракталах и преобразованиях. Эта информация используется для восстановления исходных блоков и соединения их в изображение.

Фрактальное сжатие обеспечивает высокую степень сжатия при сохранении деталей для изображений, содержащих много повторяющихся узоров, таких как текстуры или некоторые естественные изображения. Тем не менее, алгоритм сложен в реализации и может занимать много времени для сжатия и декомпрессии больших изображений.

1.3.4 Сравнение методов сжатия с потерями

Сравнение предлагается проводить по следующим критериям.

1. Идея, на которой строится алгоритм сжатия.

2. Тип артефактов, возникающих при больших коэффициентах сжатия.

3. Необходимость разбиения исходного изображения на блоки.

4. Необходимость преобразование изображения из цветовой модели RGB в цветовую модель YCbC.

Результаты сравнения методов сжатия изображений без потерь приведены в таблице 1.2.

Таблица 1.2 – Сравнение рассмотренных методов сжатия изображений без потерь

Метод сжатия	**Кр. 1**	**Кр. 2**	**Кр. 3**	**Кр. 4**
JPEG	Дискретное косинусное преобразование	Блочные артефакты	+	+
Wavelet	Вейвлет-преобразование	Кольцевые артефакты	–	–
Фрактальный	Самоподобие множеств	Артефакты реконструкции	+	–

1.4 Цветовые модели изображений

1.4.1 Анализ цветовых моделей изображений

Цветовые модели изображений — это системы, которые описывают, как цвета представлены в цифровых изображениях. Они определяют способ кодирования и хранения цветовой информации, а также позволяют преобразовывать цвета между различными форматами и устройствами отображения.

1. RGB (Red, Green, Blue) — это аддитивная цветовая модель, основанная на смешении трех основных цветов: красного, зеленого и синего. Она широко используется в цифровых устройствах, таких как мониторы, телевизоры и камеры, где цвета создаются путем добавления света. Каждый цвет представлен значением интенсивности от 0 до 255, что позволяет получить более 16 миллионов оттенков. RGB является стандартом для экранного отображения.

2. RGBA (Red, Green, Blue, Alpha) — это расширение модели RGB, которое добавляет четвертый компонент — альфа-канал. Альфа-канал определяет прозрачность цвета, где 0 означает полную прозрачность, а 255 — полную непрозрачность. Эта модель используется в графике и анимации для создания эффектов наложения и прозрачности.

3. CMYK (Cyan, Magenta, Yellow, Key/Black) — это субтрактивная цветовая модель, используемая в полиграфии и печати. Она основана на вычитании света из белого фона с помощью чернил. CMYK работает с четырьмя цветами: голубым, пурпурным, желтым и черным. Эта модель оптимальна для печатных материалов, так как позволяет точно воспроизводить цвета на бумаге.

4. LAB (Lightness, A, B) — это цветовая модель, которая описывает цвет с точки зрения его яркости (Lightness) и двух цветовых компонентов: A и B. Компонент A представляет собой ось от зеленого до красного, а компонент B — ось от синего до желтого.

   LAB является устройственно-независимой моделью, что делает её полезной для задач цветокоррекции и преобразования между различными цветовыми пространствами. Она основана на восприятии цвета человеческим глазом и охватывает большее количество цветов, чем RGB или CMYK.

   LAB часто используется в графических редакторах и системах управления цветом для точной настройки цвета и обеспечения согласованности между устройствами отображения и печати.

5. HSB (Hue, Saturation, Brightness) — это цветовая модель, которая описывает цвет с точки зрения его оттенка (Hue), насыщенности (Saturation) и яркости (Brightness). Оттенок определяет основной цвет (например, красный или синий), насыщенность — интенсивность цвета, а яркость — его светлоту. HSB удобна для работы с цветами в графических редакторах, так как позволяет интуитивно изменять их параметры.

1.4.2 Сравнение цветовых моделей изображений

Сравнение предлагается проводить по следующим критериям.

1. Класс метода по принципу действия.
2. Количество байт для кодирования одного пикселя.
3. Наличие поддержки альфа-канала.
4. Наличие отдельного канала для яркости.

Результаты сравнения цветовых моделей изображений приведены в таблице 1.3.

Таблица 1.3 – Сравнение рассмотренных цветовых моделей изображений

Цветовая модель	**Кр. 1**	**Кр. 2**	**Кр. 3**	**Кр. 4**
RGB	аддитивный	3	–	–
RGBA	аддитивный	4	+	–
CMYK	субтрактивный	4	–	–
LAB	перцепционный	3	–	+
HSB	перцепционный	3	–	+

1.5 Постановка задачи

В рамках выполнения научно-исследовательской работы требуется разработать метод сжатия статических изображений без потерь на основе алгоритма Хаффмана. При создании метода необходимо определить:

— входные и выходные данные метода сжатия;

— способ хранения информации, необходимой для восстановления исходного качества изображений.

В разрабатываемом гибридном методе сжатия улучшение будет производиться за счет первичной обработки изображения другим методом сжатия, а именно словарным методом LZW. Формальная постановка задачи в виде IDEF0-диаграммы представлена на рисунке 1.1.

Вывод

В данном разделе была проведена классификация основных методов сжатия статических изображений по следующим категориям: сжатие с потерями и сжатие без потерь. В каждой из них было проведено сравнение описанных методов по выделенным критериям.

Унарное кодирование может применяться для сжатия изображений в тех случаях, когда значения пикселей имеют ограниченный диапазон и малую вариацию (черно-белые снимки). Алгоритмы RLE и LZW могут быть полезны для изображений с большими областями одного цвета или повторяющихся участков. Арифметическое кодирование следует использовать для

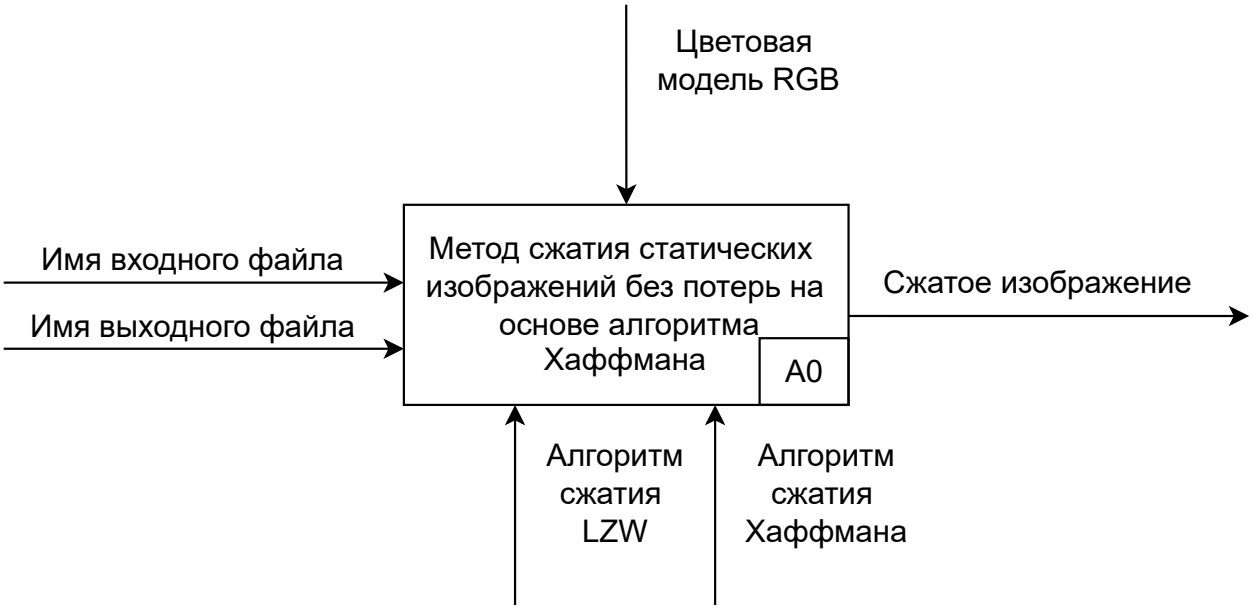

Рисунок 1.1 – Формальная постановка задачи в нотации IDEF0

изображений с большим количеством текста, например, для отсканированных документов, где вероятностное распределение частоты появления символов может быть использовано в качестве основы для сжатия. Алгоритм Хаффмана подойдет для сжатия стандартных изображений без потерь, также он может использоваться внутри сжимающих форматов изображений, таких как JPEG для оптимизации размера файлов.

JPEG является одним из наиболее широко используемых методов сжатия изображений, и обычно используется для фотографий и непрерывных тональных изображений. Wavelet сжатие было специально разработано для цветных и черно-белых изображений с плавными переходами, из-за чего подходит для обработки рентгеновских снимков и МРТ. Изображения, представляющие природные сцены, такие как пейзажи, горы, облака, водопады, хорошо подходят для фрактального сжатия, поскольку обладают повторяющимися узорами и самоподобием.

2 Конструкторская часть

2.1 Требования к разрабатываемому методу сжатия изображений

Для гибридного метода сжатия изображений были выдвинуты следующие требования:

— на вход разрабатываемый метод должен получать путь до файла со статическим изображением и путь до директории, куда необходимо сохранить сжатый файл;

— результатом работы метода должно стать сжатое изображение, сохраненное в указанной директории;

— сжатое изображение должно содержать всю информацию, необходимую для его восстановления;

— сжатие должно сохранять всю информацию об исходном изображении.

2.2 Проектирование метода сжатия изображений

Разрабатываемый метод сжатия статических изображений будет представлять собой гибридный метод на основе алгоритма сжатия Хаффмана. Улучшение данного метода будет производиться за счет первичной обработки изображения другим алгоритмом сжатия. В качестве такого метода был выбран LZW из-за:

— возможности кодирования данных за один проход;

— отсутствия необходимости в таблице частот пикселей сжимаемого изображения.

Метод LZW удаляет избыточность из последовательности пикселей изображения. Он заменяет повторяющиеся подстроки уникальными кодами, что значительно уменьшает размер изображения и приводит к уменьшению размера дерева кодов Хаффмана.

Разрабатываемый гибридной метод сжатия будет состоять из следующих этапов:

— получение данных сжимаемого изображения в виде байтовой строки, которая будет использоваться в качестве входных данных для алгоритма LZW;

— выполнение первичного сжатия изображения алгоритмом LZW;

— нахождение таблицы частот символов;

— построение дерева кодов Хаффмана на основе вычисленной таблицы частот символов;

— выполнение повторного сжатия изображения алгоритмом Хаффмана;

— создание файла с сжатым изображением и информацией для его распаковки.

Таким образом, использование первичной обработки изображения в гибридном методе сжатия позволяет подготовить данные для метода Хаффмана путем уменьшения количества обрабатываемых символов. Такой подход приводит к более эффективному сжатию Хаффмана и, следовательно, к более высокой общей степени сжатия.

Диаграмма уровня A1 (рисунок 2.1) иллюстрирует общую структуру гибридного метода сжатия: преобразование изображения в байтовую строку, этап сжатия и создание итогового файла с сжатым изображением.

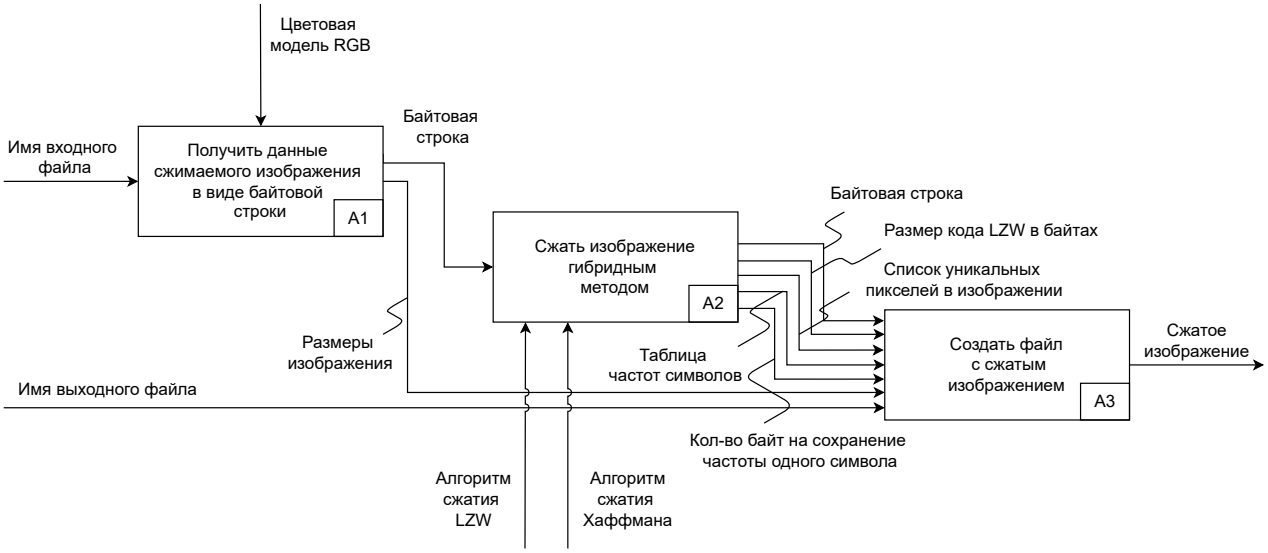

Рисунок 2.1 – Детализированная IDEF0-диаграмма гибридного метода сжатия изображений первого уровня

Диаграмма уровня A2 (рисунок 2.1) раскрывает детали этапа сжатия, выделяя первичную обработку изображения методом LZW, построение таблицы частот символов, генерацию дерева Хаффмана и повторное кодирование методом Хаффмана.

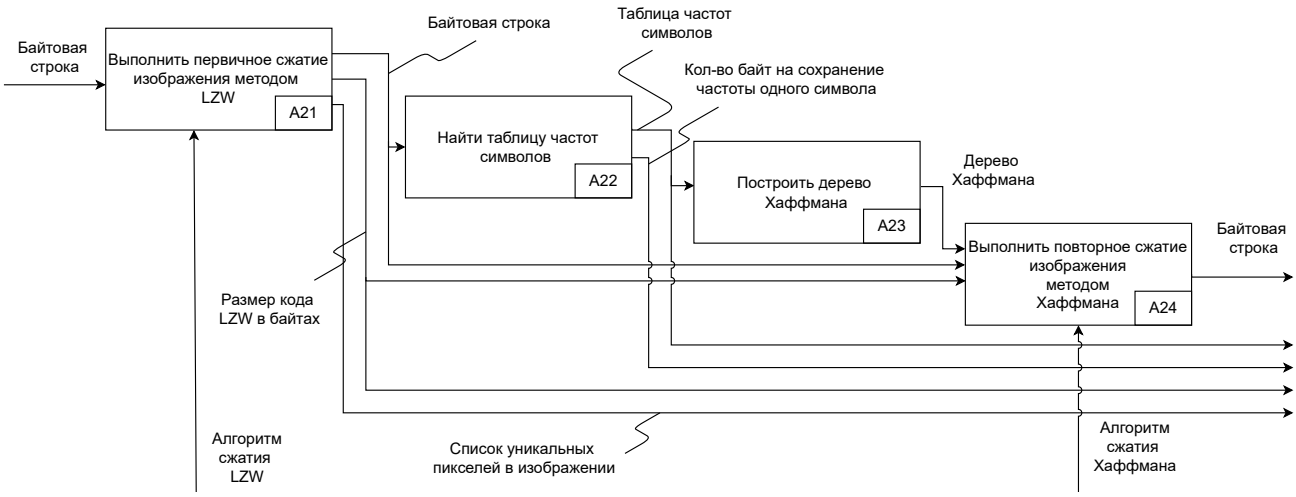

Рисунок 2.2 – Детализированная IDEF0-диаграмма гибридного метода сжатия изображений уровня A2

2.3 Требования к разрабатываемому ПО

Для демонстрации работы гибридного метода необходимо разработать ПО со следующими требованиями:

— взаимодействие пользователя с ПО должно осуществляться с помощью графического интерфейса;

— необходимо предусмотреть возможность выбора сжимаемого изображения через файловый менеджер;

— необходимо предусмотреть возможность восстановления сжатых изображений;

— пользователь должен иметь возможность сравнения гибридного метода сжатия изображений с базовыми, на основе которых он был разработан.

— сравнение должно проводиться по степени сжатия исходного файла, а также по размеру информации для распаковки в сжатом изображении.

Также необходимо подготовить список тестовых изображений для сжатия и пложить их в директорию `input_data`;

2.4 Схемы разрабатываемого гибридного метода сжатия изображений

2.4.1 Схема гибридного метода сжатия

Схема гибридного метода сжатия статических изображений представлена на рисунке 2.3. Она состоит из шести основных пунктов, три из которых далее будут рассмотрены подробно.

Рисунок 2.3 – Схема гибридного метода сжатия изображений

2.4.2 Схема метода LZW для первичного сжатия данных

Схема метода первичного сжатия LZW представлена на рисунке 2.4. На данном этапе происходит удаление избыточности данных и уменьшение количества обрабатываемых символов.

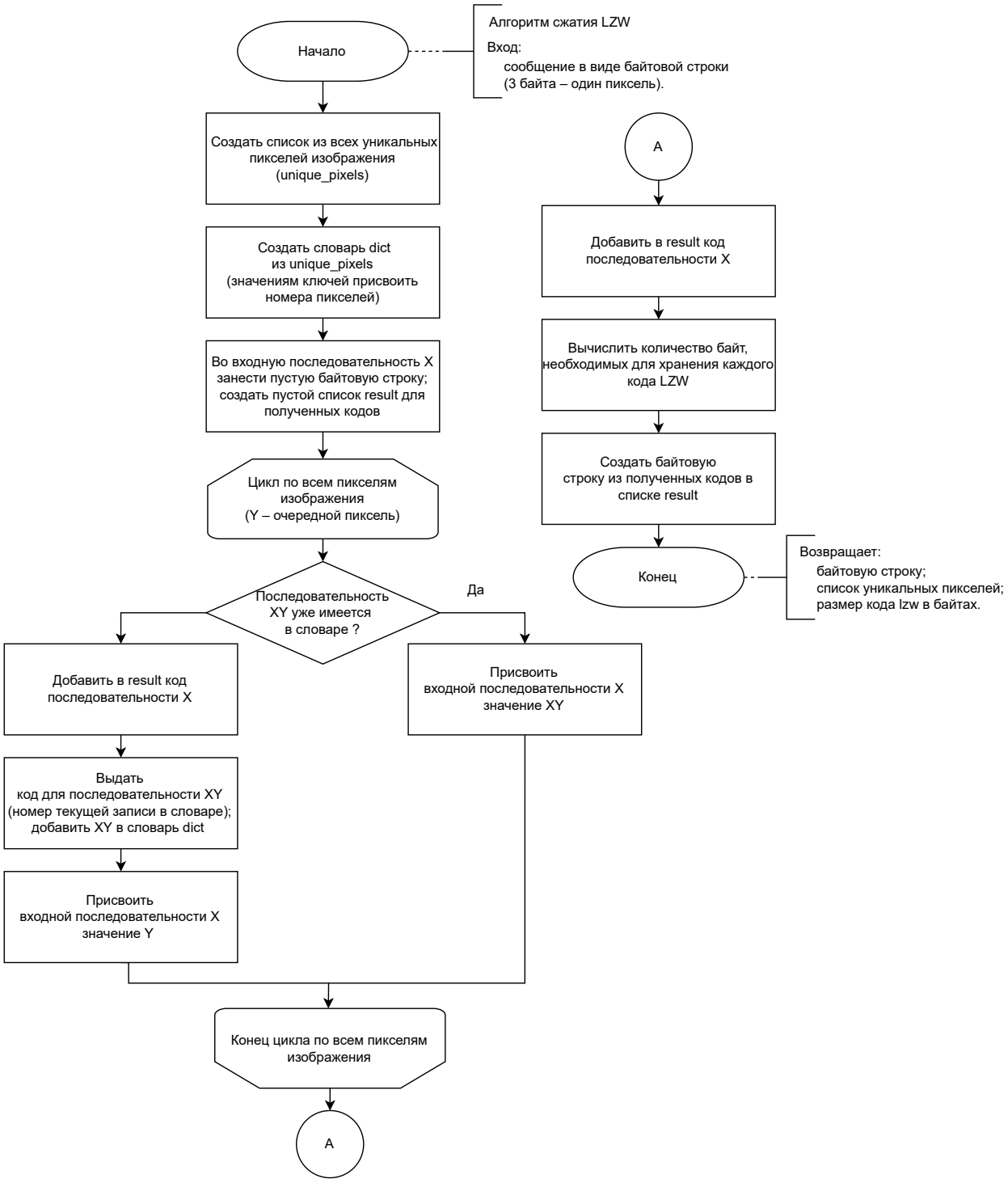

Рисунок 2.4 – Схема метода LZW для первичного сжатия изображений

2.4.3 Схема построения дерева кодов Хаффмана

Схема построения дерева кодов Хаффмана на основе таблицы частот символов представлена на рисунке 2.6. На основе этого дерева будет поизведено сжатие байтовой строки, полученной на этапе первичного сжатия изображения методом LZW.

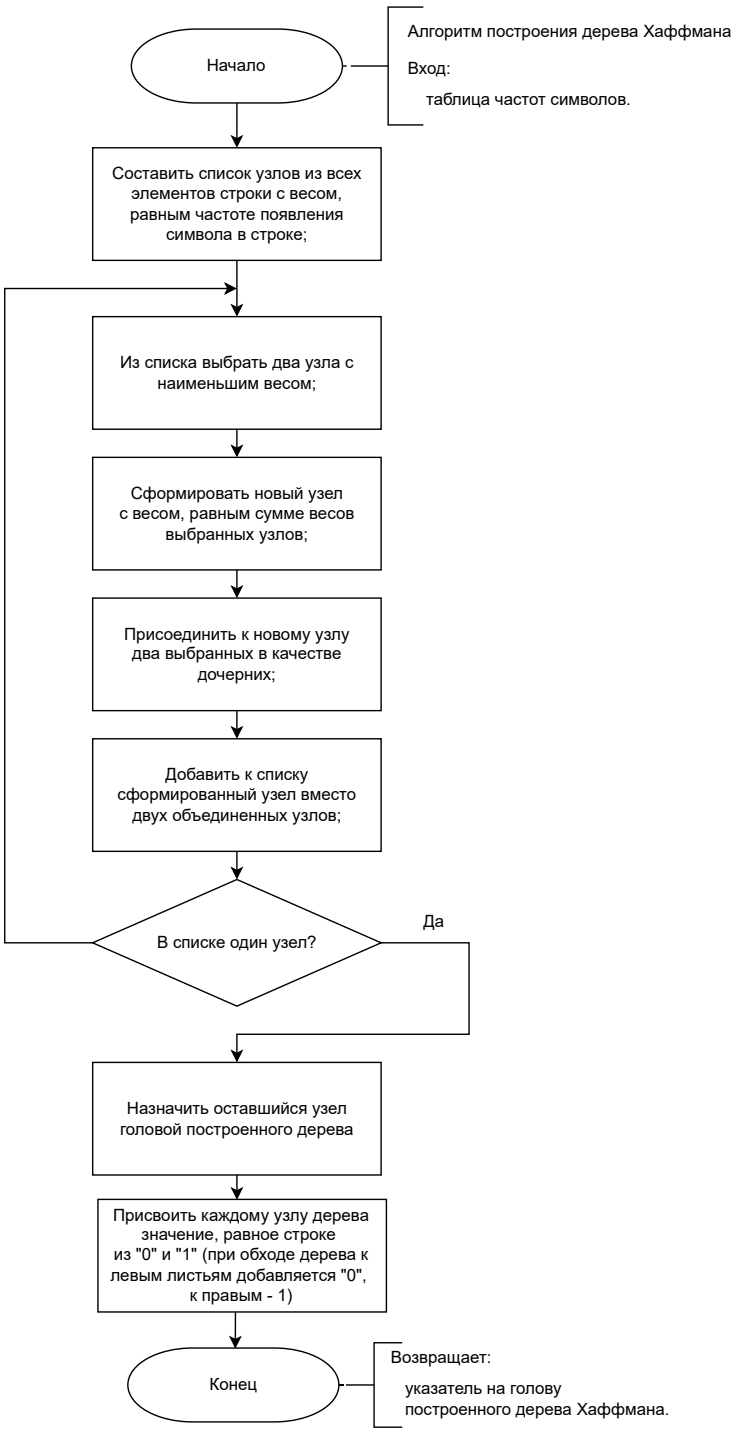

Рисунок 2.5 – Схема построения дерева кодов Хаффмана

2.4.4 Схема метода Хаффмана для повторного сжатия данных

Схема метода Хаффмана для повторного сжатия данных представлена на рисунке 2.6. Это основной этап гибридного метода, в результате которого будет получена байтовая строка с итоговым сжатым изображением.

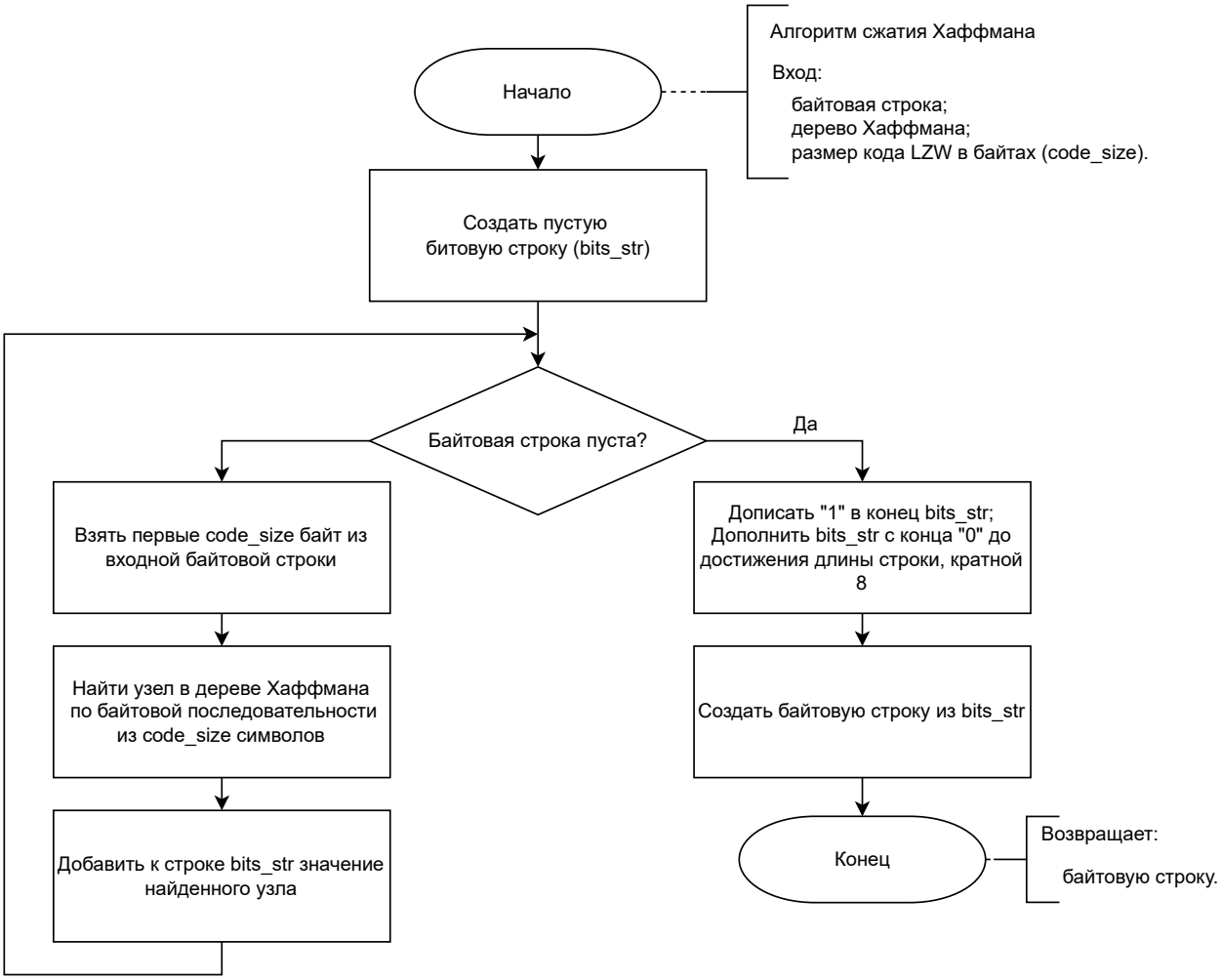

Рисунок 2.6 – Схема метода Хаффмана для повторного сжатия данных

Вывод

В данном разделе были предъявлены требования к разрабатываемому методу сжатия статических изображений и к разрабатываемому ПО, произведено проектирование метода сжатия. Для первичного сжатия, удаления избыточности и уменьшения количества обрабатываемых символов был выбран метод LZW. Кроме того, в данном разделе были построены схемы для реализации гибридного метода сжатия.

3 Технологическая часть

3.1 Используемые программные средства для реализации метода

В качестве языка программирования был выбран `Python` [1]. Это обусловлено наличием опыта работы с выбранным языком. Также для `Python` существует большое количество библиотек и документация на русском языке, а сам язык поддерживает объектно-ориентированную парадигму программирования.

При создании графического интерфейса для программного обеспечения была использована библиотека `tkinter` [2]. Она является кроссплатформенной и включена в стандартную библиотеку языка `Python` в виде отдельного модуля. Для визуализации сравнения работы методов сжатия изображений использовалась библиотека `matplotlib` [3] с следующими модулями:

— `matplotlib.pyplot` [4] — модуль, предоставляющий функции для создания графиков и визуализации данных. Использовался для построения столбчатых диаграмм при сравнении методов сжатия изображений.

— `matplotlib.offsetbox` [5] — модуль, предоставляющий возможность размещения текстовых и графические элементов на построенных графиках. Использовалась для добавления сжимаемых изображений под диаграммами сравнения методов.

Для работы с массивами битов при сжатии данных методом Хаффмана была использована библиотека `bitarray` [6], для отображения прогресса этапов сжатия и распаковки изображения использовалась библиотека `progress` [7]. Для получения списка файлов, доступных для сжатия, был использован модуль `subprocess` [8].

3.2 Формат входных и выходных данных

В качестве входных данных разработанный программный комплекс получает путь до изображения в формате BMP, TIFF, PNG или JPEG, а также путь до директории, куда необходимо сохранить сжатое и распакованное изображения. Также пользователю предоставляется возможность выбрать один из

трех методов сжатия: LZW, Хаффман или гибридный метод, разработанный в данной работе.

На выходе в директории с результатами создаются два файла с расширениями .bin (для сжатого изображения) и .bmp (для распакованного). В консоль выводится подробная информация об этапах сжатия и распаковки изображения, размеры исходного и полученного файлов, а также итоговая степень сжатия.

3.3 Структура разработанного ПО

3.3.1 Описание этапов гибридного метода сжатия

Реализация гибридного метода сжатия статических изображений без потерь состоит из следующих основных этапов.

1. Первичное сжатие изображения методом lZW.
2. Создание таблицы частот символов.
3. Построение дерева Хаффмана.
4. Применение метода сжатия Хаффмана к байтовой строке, полученной после первого этапа алгоритма.
5. Создание файла с сжатым изображением и информацией для его распаковки.

На первом этапе метода проводится первичное сжатие изображения методом LZW. В процессе обработки пикселей входного изображения создается словарь повторяющихся цепочек байт. Выделенные последовательности пикселей заменяются на уникальные коды фиксированной длины. Размер таких кодов зависит от количества заменяемых последовательностей (чем длиннее код, чем больше цепочек байт можно заменить на первом этапе метода). При распаковки сжатого изображения используется тот же словарь повторяющихся цепочек пикселей.

На втором этапе подсчитывается количество каждого уникального кода в полученной байтовой строке, строится таблица частот символов.

Третий этап включает в себя построение дерева Хаффмана, задача которого заключается в присвоении часто встречающимся символам более

коротких кодов, а редко встречающимся — более длинных. В отличии от классического дерева Хаффмана, в разработанном методе за один символ принимается не один байт, а уникальный код, состоящий из заданного числа байт.

На четвертом этапе происходит применение метода сжатия Хаффмана к байтовой строке, полученной после первого этапа алгоритма. На основе построенного дерева каждой цепочке байт (уникальному коду из метода LZW) присваивается код Хаффмана переменной длины, который за счет уникального префикса может быть однозначно декодирован.

На заключительном этапе метода происходит формирование байтовой строки со сжатым изображением и информацией для его распаковки, которая включает в себя таблицу частот символов (для восстановления дерева Хаффмана) и уникальные пиксели исходного изображения (для воссоздания словаря повторяющихся цепочек байт). Полученная байтовая строка является результатом сжатия статического изображения разработанным гибридным методом.

3.3.2 Описание модулей разработанного ПО

UML-диаграмма компонентов разработанного программного обеспечения представлена на рисунке 3.1. Она показывает структору зависимостей между основными модулями программы.

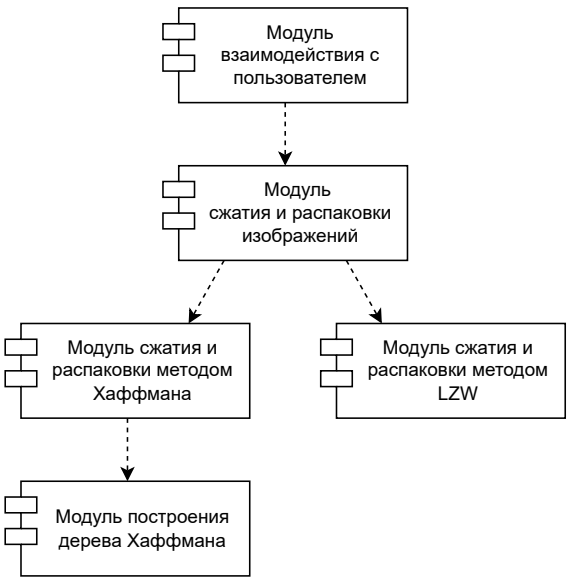

Рисунок 3.1 – UML-диаграмма компонентов разработанного ПО

Разработанное программное обеспечение состоит из следующих модулей.

1. **Модуль сжатия и распаковки изображений**. Реализует основной функционал сжатия и распаковки изображений с использованием методов LZW, Хаффмана и их гибридной реализации (листинг А.1). Основной класс модуля, `Compression`, отвечает за выполнение всех этапов сжатия и восстановления данных. При сжатии входное изображение представляется в виде байтовой строки, к которой применяется выбранный метод (LZW, Хаффман или гибридный), после чего формируется файл с сжатым изображением и метаданными для его распаковки.

2. **Модуль сжатия и распаковки методом LZW**. Реализует алгоритм сжатия и распаковки данных методом LZW (листинг А.4). Включает создание словаря повторяющихся последовательностей байт и их замену уникальными кодами фиксированной длины. При сжатии генерирует сжатую байтовую строку и список уникальных пикселей, а при распаковке восстанавливает исходные данные на основе словаря, воссозданного по списку пикселей.

3. **Модуль сжатия и распаковки методом Хаффмана**. Реализует алгоритм сжатия и распаковки данных методом Хаффмана (листинг А.3). Включает построение таблицы частот символов, создание дерева Хаффмана и генерацию кодов переменной длины. При сжатии преобразует данные в битовую строку на основе построенного дерева, а при распаковке восстанавливает исходные данные на основе сохраненной таблице частот символов.

4. **Модуль построения дерева Хаффмана**. Предоставляет вспомогательные классы и функции для работы с деревом Хаффмана (листинг А.2). Включает создание узлов дерева, объединение их на основе частот символов и генерацию кодов Хаффмана. Модуль используется в `huffman.py` для построения дерева и кодирования данных, а также для восстановления исходной информации при распаковке.

5. **Модуль взаимодействия с пользователем**. Реализует графический интерфейс с использованием библиотеки `tkinter`. Позволяет пользователю выбрать входное изображение, метод сжатия (LZW, Хаффман

или гибридный), а также директорию для сохранения результатов. Модуль отображает прогресс выполнения операций, результаты сжатия и распаковки, а также предоставляет визуализацию сравнения методов сжатия.

3.4 Результаты работы ПО

Разработанное программное обеспечение представляет собой приложение с графическим интерфейсом (рисунок 3.2), предоставляющее возможность выбора исходного изображения, метода сжатия и директории для сохранения результатов. Пользователь может сжать и распаковать выбранное изображение, посмотреть результаты сравнения доступных методов сжатия, а также получить информацию о данной программе. Подробная информация об этапах сжатия и распаковки выводится как в консоль, так и окно программы.

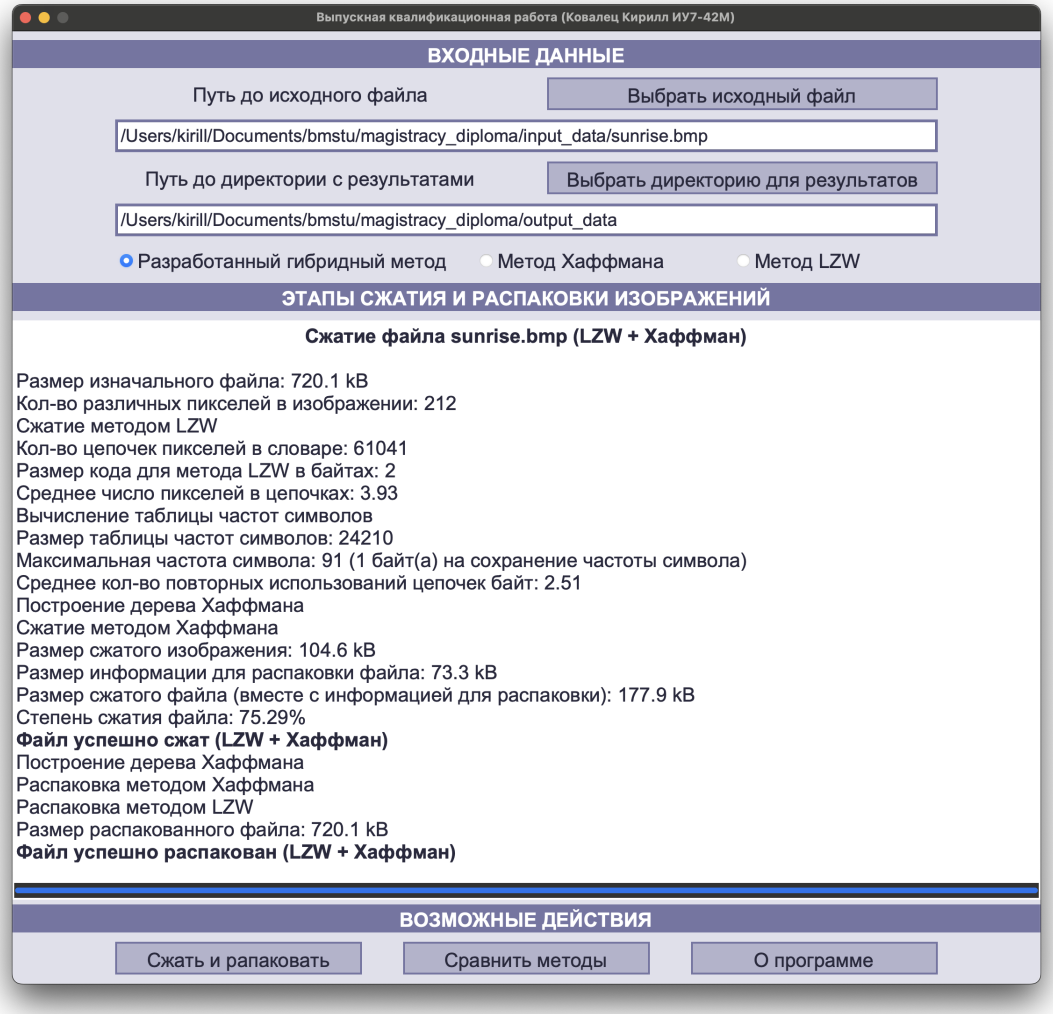

Рисунок 3.2 – Интерфейс программы для сжатия изображений

Примеры работы программы для изображения `sunrise.bmp` приведены на рисунках 3.3–3.5, а также в листинге 3.1.

Рисунок 3.3 – Исходное изображение восхода солнца

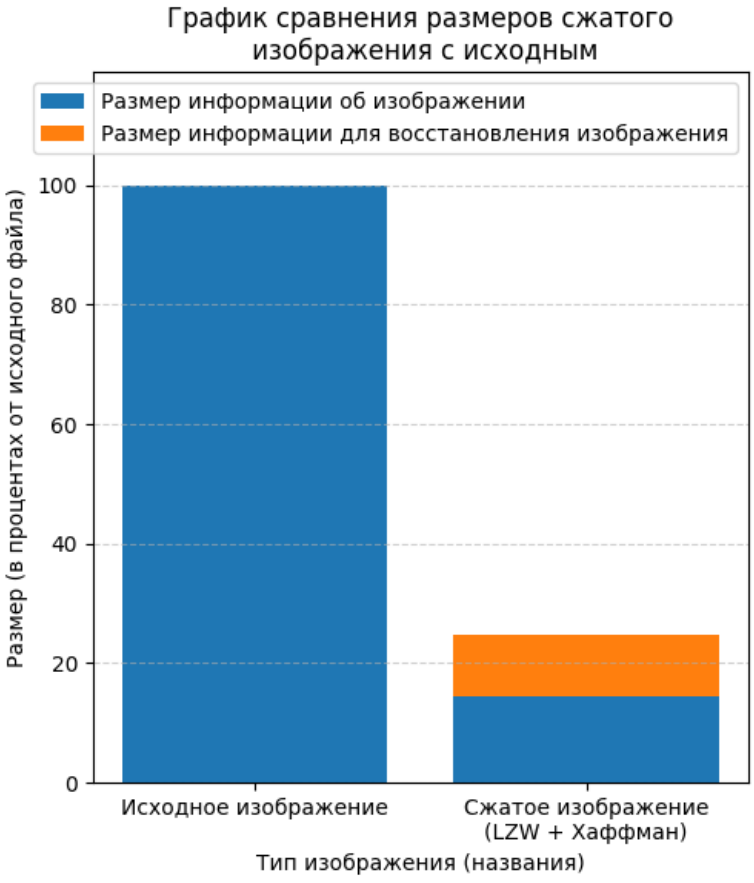

Рисунок 3.4 – Результаты сравнения размеров сжатого изображения с исходным (изображение восхода солнца)

Листинг 3.1 — Результаты сжатия и распаковки входного изображения восхода солнца гибридным методом

```
Сжатие файла sunrise.bmp (LZW + Хаффман)

Размер изначального файла: 720.1 kB

Кол-во различных пикселей в изображении: 212

Сжатие методом LZW |**********| 240000/240000

Кол-во цепочек пикселей в словаре: 61041

Размер кода для метода LZW в байтах: 2

Среднее число пикселей в цепочках: 3.93

Вычисление таблицы частот символов |**********| 60830/60830

Размер таблицы частот символов: 24210

Максимальная частота символа: 91 (1 байт(а) на сохранение частоты)

Среднее кол-во повторных использований цепочек байт: 2.51

Построение дерева Хаффмана |**********| 24209/24209

Сжатие методом Хаффмана |**********| 60830/60830

Размер сжатого изображения: 104.6 kB

Размер информации для распаковки файла: 73.3 kB

Размер сжатого файла (вместе с информацией для распаковки): 177.9 kB

Степень сжатия файла: 75.29%

Файл успешно сжат (LZW + Хаффман)

Построение дерева Хаффмана |**********| 24209/24209

Распаковка методом Хаффмана |**********| 836834/836834

Распаковка методом LZW |**********| 60829/60829

Размер распакованного файла: 720.1 kB

```

Продолжение листинга 3.1

```
45  Файл успешно распакован (LZW + Хаффман)
```

Рисунок 3.5 – Восстановленное изображение восхода солнца

Для изображения `sunrise.bmp` исходный размер файла составил 720.1 КБ, а количество различных пикселей в изображении — 212. После сжатия разработанным гибридным методом (LZW + Хаффман) размер сжатого изображения составил 104.6 КБ, а информация для его распаковки заняла 73.3 КБ, что в сумме дало размер сжатого файла 177.9 КБ. Степень сжатия файла составила 75.29%. Среднее число пикселей в цепочках, созданных методом LZW, составило 3.93, а среднее количество повторных использований цепочек байт — 2.51. Распаковка файла успешно восстановила исходное изображение с размером 720.1 КБ.

Примеры работы программы для изображения `girl.bmp` приведены на рисунках 3.6–3.8, а также в листинге 3.2.

Рисунок 3.6 – Исходное изображение девушки

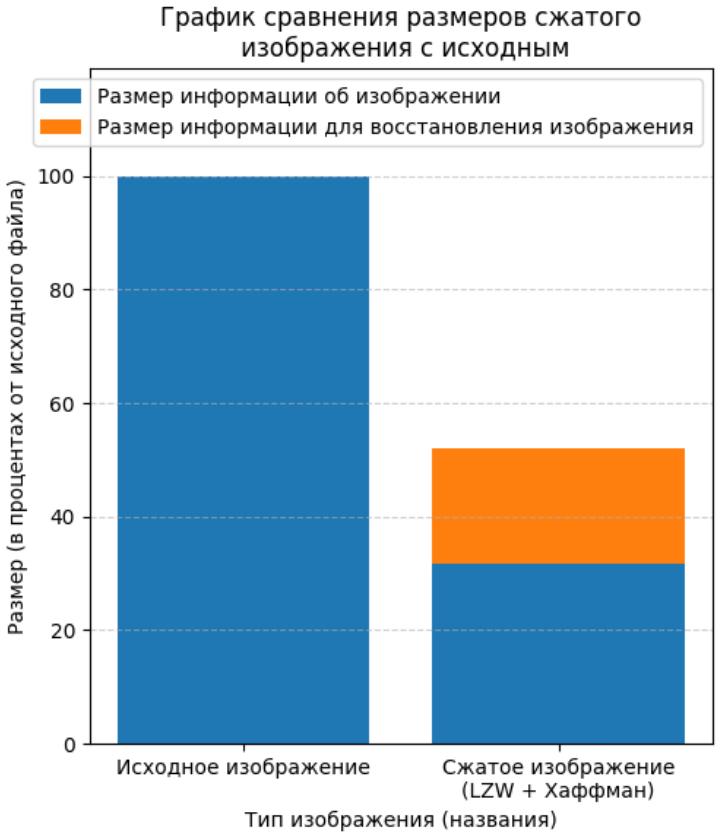

Рисунок 3.7 – Результаты сравнения размеров сжатого изображения с исходным (изображение девушки)

Листинг 3.2 — Результаты сжатия и распаковки входного изображения девушки гибридным методом

```
Сжатие файла girl.bmp (LZW + Хаффман)

Размер изначального файла: 491.3 kB

Кол-во различных пикселей в изображении: 2643

Сжатие методом LZW |**********| 163450/163450

Кол-во цепочек пикселей в словаре: 96886

Размер кода для метода LZW в байтах: 3

Среднее число пикселей в цепочках: 1.69

Вычисление таблицы частот символов |**********| 94244/94244

Размер таблицы частот символов: 22942

Максимальная частота символа: 98 (1 байт(а) на сохранение частоты)

Среднее кол-во повторных использований цепочек байт: 4.11

Построение дерева Хаффмана |**********| 22941/22941

Сжатие методом Хаффмана |**********| 94244/94244

Размер сжатого изображения: 155.5 kB

Размер информации для распаковки файла: 99.7 kB

Размер сжатого файла (вместе с информацией для распаковки): 255.3 kB

Степень сжатия файла: 48.05%

Файл успешно сжат (LZW + Хаффман)

Построение дерева Хаффмана |**********| 22941/22941

Распаковка методом Хаффмана |**********| 1244263/1244263

Распаковка методом LZW |**********| 94243/94243

Размер распакованного файла: 491.3 kB

```

Продолжение листинга 3.2

```
45  Файл успешно распакован (LZW + Хаффман)
```

Рисунок 3.8 – Восстановленное изображение девушки

Для изображения `girl.bmp` исходный размер файла составил 491.3 КБ, а количество уникальных пикселей в изображении — 2643. После применения гибридного метода сжатия (LZW + Хаффман) размер сжатого изображения составил 155.5 КБ, а данные для его восстановления заняли 99.7 КБ, что в сумме дало общий размер сжатого файла 255.3 КБ. Степень сжатия составила 48.05%.

Средняя длина цепочек пикселей, сформированных методом LZW, составила 1.69, а среднее количество повторений цепочек байт — 4.11. Размер таблицы частот символов, использованной для построения дерева Хаффмана, составил 22.9 КБ. Максимальная частота символа в таблице составила 98, что потребовало 1 байт для хранения частоты. Распаковка файла успешно восстановила исходное качество изображения.

Вывод

В данном разделе были рассмотрены используемые программные средства реализации метода, описан формат входных и выходных данных, реализован гибридный метод сжатия статических изображений и приведены

результаты работы программы. Также было представлено описание структуры разработанного ПО.

В приведенных примерах степень сжатия изображения с восходом солнца в 1.56 раз больше, чем в изображении с девушкой (75.29% против 48.05%). Это связано с тем, что в файле `sunrise.bmp` цепочки байт, полученные на этапе обработки изображения методом LZW, в среднем содержат больше пикселей (3.93 против 1.69, то есть на 133%). Также в изображении `girl.bmp` больше уникальных пикселей (2643 против 212, то есть в 12.47 раз). Эти факторы способствуют более эффективному сжатию изображения с восходом солнца разработанным гибридным методом.

4 Исследовательская часть

4.1 Критерии оценки методов сжатия изображений

Для оценки методов сжатия изображений использовались следующие критерии.

1. **Степень сжатия**: показывает, на сколько процентов от изначального размера файла удалось сжать изображение. Чем выше коэффициент, тем лучше удалось выполнить сжатие. При этом учитывается не только размер сжатого изображения, но и объем метаданных, необходимых для его восстановления. Степень сжатия рассчитывается по формуле.

$$\text{Степень сжатия} = \left(1 - \frac{\text{Размер сжатого изображения}}{\text{Размер исходного изображения}}\right) \times 100\%. \tag{4.1}$$

2. **Размер информации для распаковки**: показывает, какую часть сжатого изображения занимает информация, необходимая для восстановления исходного файла. Чем выше этот показатель, тем большую долю от сжатого файла занимают метаданные. Большой объем информации для распаковки может не дать достичь высокой степени сжатия изображения.

Для проведения исследования по выделенным критериям были выбраны изображения в формате BMP. Выбор файлов данного формата обусловлен следующими причинами:

— Файлы в формате BMP хранят информацию о каждом пикселе изображения в исходном качестве без сжатия.

— BMP-файлы широко используются на практике в различных приложениях и системах.

— Формат BMP подходит для работы как с черно-белыми, так и с цветными изображениями.

4.2 Сравнение разработанного метода сжатия с аналогами

4.2.1 Сравнение по степени сжатия изображений

Результаты сравнения методов сжатия статических изображений без потерь по степени сжатия приведены в таблице 4.1 и продемонстрированы на рисунке 4.1.

Таблица 4.1 – Результаты сравнения методов сжатия изображений по степени сжатия

Изображение	**Метод LZW, %**	**Гибридный метод, %**	**Метод Хаффмана, %**
sunrise.bmp	83.01	75.29	69.91
mars.bmp	84.61	78.30	71.59
wheat.bmp	77.67	70.52	68.02
forest.bmp	44.52	54.23	67.77
girl.bmp	40.84	48.05	59.23

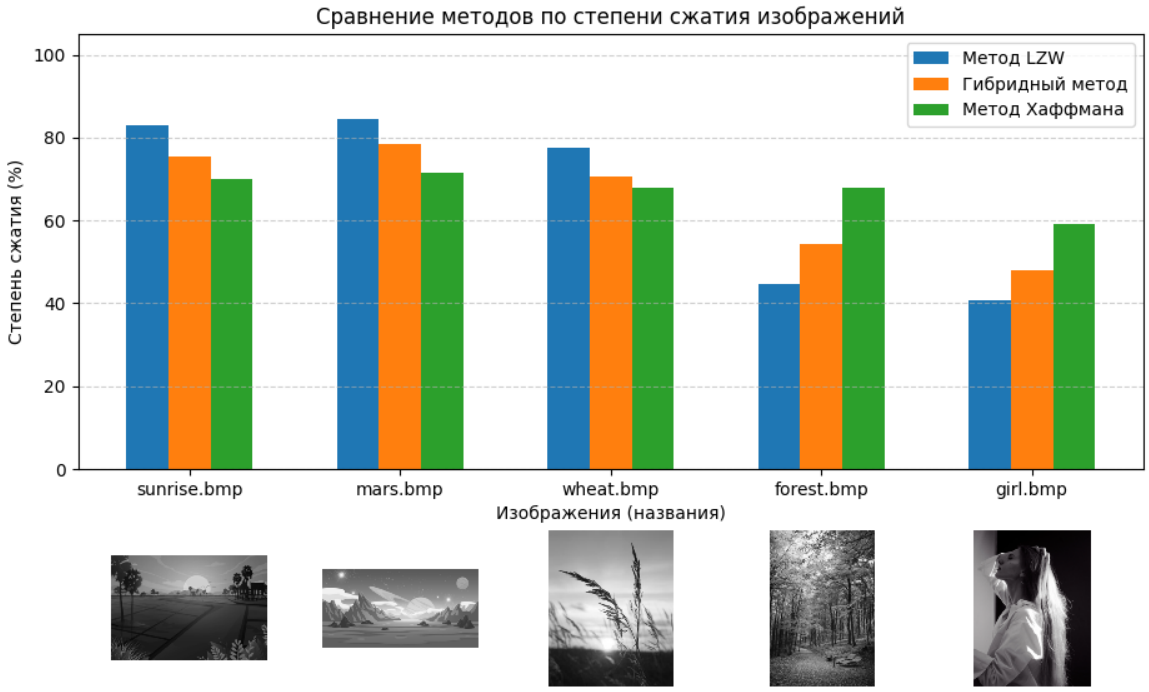

Рисунок 4.1 – Сравнения методов сжатия статических изображений без потерь по степени сжатия

На примерах видно, что степень сжатия зависит от типа данных: метод LZW лучше работает с длинными последовательностями одинаковых пикселей, показывая наивысшую степень сжатия для изображений *sunrise.bmp* (83.01%), *mars.bmp* (84.61%) и *wheat.bmp* (77.67%). Метод Хаффмана, напротив, демонстрирует лучшие результаты для изображений с неравномерным распределением цветов, таких как *forest.bmp* (67.77%) и *girl.bmp* (59.23%).

Гибридный метод является универсальным решением, которое позволяет минимизировать зависимость от особенностей входных изображений. Например, для изображения *forest.bmp* он показывает степень сжатия 54.23%, что выше, чем у метода LZW (44.52%), но ниже, чем у метода Хаффмана (67.77%). В то же время для изображения *mars.bmp* его результат (78.30%) уступает методу LZW (84.61%), но превосходит метод Хаффмана (71.59%).

Таким образом, гибридный метод обеспечивает более стабильный результат сжатия, выступая как компромиссное решение между высокой степенью сжатия и универсальностью.

4.2.2 Сравнение по размеру информации для распаковки изображений

Результаты сравнения методов сжатия статических изображений без потерь по количеству информации, необходимой для распаковки изображений, приведены в таблице 4.2.

Таблица 4.2 – Результаты сравнения методов сжатия по размеру информации для распаковки изображений

Изображение	**Метод LZW, %**	**Гибридный метод, %**	**Метод Хаффмана, %**
sunrise.bmp	0.53	41.20	0.51
mars.bmp	0.77	41.81	0.72
wheat.bmp	0.76	39.07	0.88
forest.bmp	0.25	33.50	0.73
girl.bmp	2.72	39.05	6.59

Визуализация результатов сравнения приведена на рисунке 4.2.

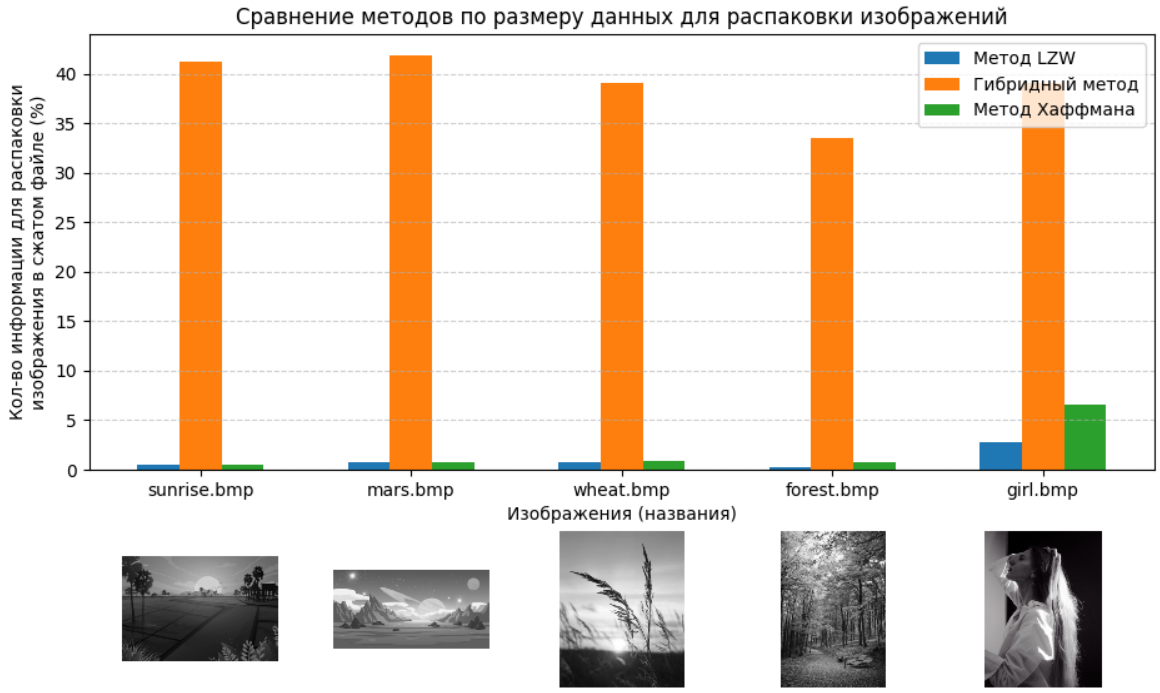

Рисунок 4.2 – Сравнение методов сжатия статических изображений без потерь по количеству информации для распаковки

Гибридный метод сжатия, сочетающий в себе алгоритмы LZW и Хаффмана, требует больше информации для распаковки изображения по сравнению с отдельными методами. Это связано с необходимостью сохранения данных, используемых на обоих этапах сжатия. Например, после применения LZW сохраняются уникальные пиксели изображения и размер кода LZW, а на этапе Хаффмана добавляются таблица частот символов и количество байт, необходимых для сохранения частоты одного символа. Каждый из этих компонентов увеличивает объем метаданных. Как видно из таблицы 4.2, для гибридного метода доля информации для распаковки составляет от 33.50% до 41.81%, тогда как для LZW и Хаффмана этот показатель не превышает 6.59%.

Выводы

Несмотря на больший объем метаданных в сжатом файле, гибридный метод обеспечивает более стабильный результат сжатия за счет сильных сторон обоих алгоритмов. LZW эффективно сжимает повторяющиеся последовательности (например, однородные участки изображений), а Хаффман оптимизирует кодирование частых символов. Это позволяет гибридному мето-

ду адаптироваться к разным типам изображений, минимизируя зависимость от их структуры. Например, для изображения forest.bmp гибридный метод показывает степень сжатия 54.23%, что лучше, чем у LZW (44.52%), и близко к результату Хаффмана (67.77%). Для изображения mars.bmp метод LZW демонстрирует степень сжатия 84.61%, что значительно лучше, чем у метода Хаффмана (71.59%). Гибридный метод показывает промежуточный результат (78.30%), сохраняя универсальность. Таким образом, компромисс между объемом метаданных и универсальностью делает гибридный метод подходящим для задач, где важна стабильность, а не абсолютная минимизация размера файла.

ЗАКЛЮЧЕНИЕ

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Welcome to Python [Электронный ресурс]. — Режим доступа URL: https://www.python.org (Дата обращения: 18.01.2025).
2. Tkinter — Python interface to Tcl/Tk [Электронный ресурс]. — Режим доступа URL: https://docs.python.org/3/library/tkinter.html (Дата обращения: 18.01.2025).
3. Matplotlib — Visualization with Python [Электронный ресурс]. — Режим доступа URL: https://matplotlib.org (Дата обращения: 18.01.2025).
4. matplotlib.pyplot — Matplotlib 3.5.3 documentation [Электронный ресурс]. — Режим доступа URL: https://matplotlib.org/3.5.3/api/_as_gen/matplotlib.pyplot.html (Дата обращения: 18.01.2025).
5. matplotlib.offsetbox — Matplotlib 3.10.3 documentation [Электронный ресурс]. — Режим доступа URL: https://matplotlib.org/stable/api/offsetbox_api.html (Дата обращения: 18.01.2025).
6. bitarray [Электронный ресурс]. — Режим доступа URL: https://pypi.org/project/bitarray/ (Дата обращения: 18.01.2025).
7. progress [Электронный ресурс]. — Режим доступа URL: https://pypi.org/project/progress/ (Дата обращения: 18.01.2025).
8. Subprocess management — Python 3.13.1 documentation [Электронный ресурс]. — Режим доступа URL: https://docs.python.org/3/library/subprocess.html (Дата обращения: 18.01.2025).
9. *Тропченко А. Ю.* МЕТОДЫ СЖАТИЯ ИЗОБРАЖЕНИЙ, АУДИОСИГНАЛОВ И ВИДЕО // Санкт-Петербург, ИТМО. — 2023. — С. 8—42.
10. *Лежнев В. Г.* МАТЕМАТИЧЕСКИЕ АЛГОРИТМЫ СЖАТИЯ ИЗОБРАЖЕНИЙ // Краснодар, КГУ. — 2021. — С. 20—26.
11. Digital image compression: An overview [Электронный ресурс]. — Режим доступа URL: https://www.educative.io/blog/digital-image-compression (Дата обращения: 29.11.2023).
12. What is image compression? [Электронный ресурс]. — Режим доступа URL: https://www.cloudflare.com/learning/performance/glossary/what-is-image-compression/ (Дата обращения: 29.11.2023).

13. Run-Length Encoding (RLE) [Электронный ресурс]. — Режим доступа URL: https://www.fileformat.info/mirror/egff/ch09_03.htm (Дата обращения: 29.11.2023).

14. LZW (Lempel–Ziv–Welch) Compression Technique [Электронный ресурс]. — Режим доступа URL: https://www.scaler.com/topics/lzw-compression/ (Дата обращения: 29.11.2023).

15. LZW compression [Электронный ресурс]. — Режим доступа URL: https://www.geeksforgeeks.org/lzw-lempel-ziv-welch-compression-technique/ (Дата обращения: 29.05.2024).

16. [MS-RDPEGFX]: Unary Encoding [Электронный ресурс]. — Режим доступа URL: https://learn.microsoft.com/en-us/openspecs/windows_protocols/ms-rdpegfx/11bf987b-eeba-4ed8-8fd2-0c89f632382b (Дата обращения: 29.11.2023).

17. Huffman Coding Algorithm [Электронный ресурс]. — Режим доступа URL: https://www.programiz.com/dsa/huffman-coding (Дата обращения: 29.11.2023).

18. Huffman Coding and Decoding Algorithm [Электронный ресурс]. — Режим доступа URL: https://www.topcoder.com/thrive/articles/huffman-coding-and-decoding-algorithm (Дата обращения: 29.05.2024).

19. Data Compression with Arithmetic Coding [Электронный ресурс]. — Режим доступа URL: https://www.scaler.com/topics/data-compression-with-arithmetic-coding/ (Дата обращения: 29.11.2023).

20. JPEG Compression Explained [Электронный ресурс]. — Режим доступа URL: https://www.baeldung.com/cs/jpeg-compression (Дата обращения: 29.11.2023).

21. Wavelet-Based Image Compression [Электронный ресурс]. — Режим доступа URL: https://www.clear.rice.edu/elec301/Projects00/wavelet_image_comp/img-compression-theory.html (Дата обращения: 29.11.2023).

22. Optimization of Fractal Image Compression [Электронный ресурс]. — Режим доступа URL: https://www.intechopen.com/chapters/72917 (Дата обращения: 29.11.2023).

23. HTML RGB and RGBA Colors [Электронный ресурс]. — Режим доступа URL: https://www.w3schools.com/html/html_colors_rgb.asp (Дата обращения: 29.11.2023).

24. Colors CMYK [Электронный ресурс]. — Режим доступа URL: https://www.w3schools.com/colors/colors_cmyk.asp (Дата обращения: 29.11.2023).

25. Lab Color — MATLAB and Simulink [Электронный ресурс]. — Режим доступа URL: https://www.mathworks.com/discovery/lab-color.html (Дата обращения: 29.11.2023).

26. The HSB Color System: A Practitioner's Primer [Электронный ресурс]. — Режим доступа URL: https://www.learnui.design/blog/the-hsb-color-system-practicioners-primer.html (Дата обращения: 29.11.2023).

ПРИЛОЖЕНИЕ А
Реализация гибридного метода сжатия статических изображений без потерь

Листинг А.1 — Реализация модуля сжатия изображений

```
import re
from PIL import Image
from os.path import getsize
from humanize import naturalsize
from tkinter import Text, END
from tkinter.ttk import Progressbar

from constants import CompressionMethods, BYTES_AMOUNT_PER_PIXEL
from huffman import Huffman
from lzw import LZW
from comparison import plot_comparison_bar_chart
from color import *


class Compression():
    compression_methods_names = {
        CompressionMethods.HYBRID: "LZW + Хаффман",
        CompressionMethods.HUFFMAN: "Хаффман",
        CompressionMethods.LZW: "LZW",
    }

    def __init__(
        self,
        method: CompressionMethods,
        text_editor: Text,
        progressbar: Progressbar,
    ) -> None:
        self.lzw = LZW(text_editor, progressbar)
        self.huffman = Huffman(text_editor, progressbar)
        self.text_editor = text_editor
        self.method = method

    def compress(self, input_file_name: str, output_file_name: str) -> None:
        short_filename = input_file_name.split("/")[-1]
        method_name = self.compression_methods_names[self.method]
        size = getsize(input_file_name)
        size_str = naturalsize(size)

        self.text_editor.insert(END, f"Сжатие файла {short_filename}
        ↪  ({method_name})\n\n", ("bold", "center"))
```

Продолжение листинга А.1

```
40          self.text_editor.insert(END, f"Размер изначального файла: {size_str}\n")
41          self.text_editor.update()
42          print(f"\n{blue}Сжатие файла {short_filename}
            ↪  ({method_name}){base_color}")
43          print(f"\nРазмер изначального файла: {size_str}")
44
45          image = Image.open(input_file_name)
46          image = image.convert("RGB")
47          data = image.tobytes()
48          self.width, self.height = image.size
49
50          match self.method:
51              case CompressionMethods.HYBRID:
52                  lzw_compressed, lzw_code_size, unique_pixels =
                    ↪  self.lzw.compress(data)
53                  frequency_table, frequency_size =
                    ↪  self.huffman.build_frequency_table(
54                      bytes_str=lzw_compressed,
55                      code_size=lzw_code_size
56                  )
57                  tree = self.huffman.build_tree(frequency_table)
58                  compressed = self.huffman.compress(lzw_compressed, lzw_code_size,
                    ↪  tree)
59
60                  data_to_decompress = (
61                      self.width.to_bytes(4, byteorder='big') +
62                      self.height.to_bytes(4, byteorder='big') +
63                      lzw_code_size.to_bytes(4, byteorder='big') +
64                      len(unique_pixels).to_bytes(4, byteorder='big') +
65                      b''.join(unique_pixels) +
66                      frequency_size.to_bytes(4, byteorder='big') +
67                      len(frequency_table).to_bytes(4, byteorder='big') +
68                      self.__convert_frequency_table_to_bytes(frequency_table,
                        ↪  frequency_size)
69                  )
70              case CompressionMethods.HUFFMAN:
71                  frequency_table, frequency_size =
                    ↪  self.huffman.build_frequency_table(
72                      bytes_str=data,
73                      code_size=BYTES_AMOUNT_PER_PIXEL
74                  )
75                  tree = self.huffman.build_tree(frequency_table)
76                  compressed = self.huffman.compress(data, BYTES_AMOUNT_PER_PIXEL,
                    ↪  tree)
77
```

Продолжение листинга А.1

```
                    data_to_decompress = (
                        self.width.to_bytes(4, byteorder='big') +
                        self.height.to_bytes(4, byteorder='big') +
                        frequency_size.to_bytes(4, byteorder='big') +
                        len(frequency_table).to_bytes(4, byteorder='big') +
                        self.__convert_frequency_table_to_bytes(frequency_table,
                        ↪  frequency_size)
                    )
                case _:
                    compressed, lzw_code_size, unique_pixels = self.lzw.compress(data)

                    data_to_decompress = (
                        self.width.to_bytes(4, byteorder='big') +
                        self.height.to_bytes(4, byteorder='big') +
                        lzw_code_size.to_bytes(4, byteorder='big') +
                        len(unique_pixels).to_bytes(4, byteorder='big') +
                        b''.join(unique_pixels)
                    )

            with open(output_file_name, "wb") as f:
                f.write(data_to_decompress + compressed)

            compressed_file_size = getsize(output_file_name)
            compressed_file_size_str = naturalsize(compressed_file_size)
            compressed_data_size_str = naturalsize(len(compressed))
            size_data_to_decompress_str = naturalsize(len(data_to_decompress))
            compression_ratio = (size - compressed_file_size) / size * 100

            self.text_editor.insert(END, f"Размер сжатого изображения:
            ↪ {compressed_data_size_str}\n")
            self.text_editor.insert(END, f"Размер информации для распаковки файла:
            ↪ {size_data_to_decompress_str}\n")
            self.text_editor.insert(END, f"Размер сжатого файла (вместе с информацией
            ↪  для распаковки): {compressed_file_size_str}\n")
            self.text_editor.insert(END, "Степень сжатия файла:
            ↪ {:2.2f}%\n".format(compression_ratio))
            self.text_editor.insert(END, f"Файл успешно сжат ({method_name})\n",
            ↪  ("bold",))
            self.text_editor.update()
            print(f"\nРазмер сжатого изображения: {compressed_data_size_str}")
            print(f"\nРазмер информации для распаковки файла:
            ↪ {size_data_to_decompress_str}")
            print(f"\nРазмер сжатого файла (вместе с информацией для распаковки):
            ↪ {compressed_file_size_str}")
            print("\nСтепень сжатия файла: {:2.2f}%".format(compression_ratio))
```

Продолжение листинга А.1

```
115             print(f"{purple}\nФайл успешно сжат ({method_name}){base_color}")
116
117             plot_comparison_bar_chart(
118                 image_sizes=[100, len(compressed) / size * 100],
119                 data_to_decompress_sises=[0, len(data_to_decompress) / size * 100],
120                 method=method_name,
121             )
122
123         def decompress(self, input_file_name: str, output_file_name: str) -> None:
124             with open(input_file_name, "rb") as f:
125                 bytes_str = f.read()
126                 if not bytes_str:
127                     return None
128
129             start = 0
130             match self.method:
131                 case CompressionMethods.HYBRID:
132                     width = int.from_bytes(bytes_str[start:start + 4],
                        ↪ byteorder='big')
133                     start += 4
134                     height = int.from_bytes(bytes_str[start:start + 4],
                        ↪ byteorder='big')
135                     start += 4
136                     lzw_code_size = int.from_bytes(bytes_str[start:start + 4],
                        ↪ byteorder='big')
137                     start += 4
138                     unique_pixels_count = int.from_bytes(bytes_str[start:start + 4],
                        ↪ byteorder='big')
139                     start += 4
140                     unique_pixels = self.__convert_bytes_to_list_of_unique_pixels(
141                         byte_string=bytes_str[start:start + unique_pixels_count *
                            ↪ BYTES_AMOUNT_PER_PIXEL],
142                     )
143                     start += unique_pixels_count * BYTES_AMOUNT_PER_PIXEL
144                     frequency_size = int.from_bytes(bytes_str[start:start + 4],
                        ↪ byteorder='big')
145                     start += 4
146                     frequency_table_size = int.from_bytes(bytes_str[start:start + 4],
                        ↪ byteorder='big')
147                     start += 4
148                     frequency_table = self.__convert_bytes_to_frequency_table(
149                         byte_string=bytes_str[start:start + frequency_table_size *
                            ↪ (lzw_code_size + frequency_size)],
150                         code_size=lzw_code_size,
151                         frequency_size=frequency_size,
```

Продолжение листинга А.1

```
                )
                start += frequency_table_size * (lzw_code_size + frequency_size)

                tree = self.huffman.build_tree(frequency_table)
                huffman_decompressed = self.huffman.decompress(bytes_str[start:],
                ↪  tree)
                decompressed = self.lzw.decompress(
                    data=huffman_decompressed,
                    code_size=lzw_code_size,
                    unique_pixels=unique_pixels,
                )
            case CompressionMethods.HUFFMAN:
                width = int.from_bytes(bytes_str[start:start + 4],
                ↪  byteorder='big')
                start += 4
                height = int.from_bytes(bytes_str[start:start + 4],
                ↪  byteorder='big')
                start += 4
                frequency_size = int.from_bytes(bytes_str[start:start + 4],
                ↪  byteorder='big')
                start += 4
                frequency_table_size = int.from_bytes(bytes_str[start:start + 4],
                ↪  byteorder='big')
                start += 4
                frequency_table = self.__convert_bytes_to_frequency_table(
                    byte_string=bytes_str[start:start + frequency_table_size *
                    ↪  (BYTES_AMOUNT_PER_PIXEL + frequency_size)],
                    code_size=BYTES_AMOUNT_PER_PIXEL,
                    frequency_size=frequency_size,
                )
                start += frequency_table_size * (BYTES_AMOUNT_PER_PIXEL +
                ↪  frequency_size)

                tree = self.huffman.build_tree(frequency_table)
                decompressed = self.huffman.decompress(bytes_str[start:], tree)
            case _:
                width = int.from_bytes(bytes_str[start:start + 4],
                ↪  byteorder='big')
                start += 4
                height = int.from_bytes(bytes_str[start:start + 4],
                ↪  byteorder='big')
                start += 4
                lzw_code_size = int.from_bytes(bytes_str[start:start + 4],
                ↪  byteorder='big')
                start += 4
```

Продолжение листинга А.1

```
187                 unique_pixels_count = int.from_bytes(bytes_str[start:start + 4],
                    ↪  byteorder='big')
188                 start += 4
189                 unique_pixels = self.__convert_bytes_to_list_of_unique_pixels(
190                     byte_string=bytes_str[start:start + unique_pixels_count *
                        ↪  BYTES_AMOUNT_PER_PIXEL],
191                 )
192                 start += unique_pixels_count * BYTES_AMOUNT_PER_PIXEL
193
194                 decompressed = self.lzw.decompress(
195                     data=bytes_str[start:],
196                     code_size=lzw_code_size,
197                     unique_pixels=unique_pixels,
198                 )
199
200             image = Image.frombytes("RGB", (width, height), decompressed)
201             image.save(output_file_name, "BMP")
202
203             size_str = naturalsize(getsize(output_file_name))
204
205             method_name = self.compression_methods_names[self.method]
206             self.text_editor.insert(END, f"Размер распакованного файла: {size_str}\n")
207             self.text_editor.insert(END, f"Файл успешно распакован
                ↪  ({method_name})\n\n", ("bold",))
208             self.text_editor.update()
209             print(f"\nРазмер распакованного файла: {size_str}")
210             print(f"{purple}\nФайл успешно распакован ({method_name}){base_color}\n")
211
212         def __convert_frequency_table_to_bytes(
213             self,
214             frequency_table: dict[bytes, int],
215             frequency_size: int,
216         ) -> bytes:
217             byte_string = bytes()
218             for code, frequency in frequency_table.items():
219                 byte_string += code
220                 byte_string += frequency.to_bytes(frequency_size, byteorder='big')
221
222             return byte_string
223
224         def __convert_bytes_to_frequency_table(
225             self,
226             byte_string: bytes,
227             code_size: int,
228             frequency_size: int,
```

Продолжение листинга А.1

```
    ) -> dict[bytes, int]:
        frequency_table = {}
        for i in range(0, len(byte_string), code_size + frequency_size):
            code = byte_string[i:i + code_size]
            frequency = byte_string[i + code_size:i + code_size + frequency_size]
            frequency_table[code] = int.from_bytes(frequency, byteorder='big')

        return frequency_table

    def __convert_bytes_to_list_of_unique_pixels(
        self,
        byte_string: bytes,
    ) -> list[bytes]:
        return re.findall(
            rb"[\x00-\xff]{%d}" % BYTES_AMOUNT_PER_PIXEL,
            byte_string,
        )
```

Листинг А.2 — Реализация модуля для построения дерева Хаффмана

```
from tkinter import Text, END
from tkinter.ttk import Progressbar
from progress.bar import IncrementalBar


class Node():
    def __init__(
        self,
        symbols: list[bytes],
        frequency: int,
        left=None,
        right=None
    ):
        self.symbols = symbols
        self.frequency = frequency
        self.value = ""

        self.left  = left
        self.right = right


class Tree():
    def __init__(
        self,
        frequency_table: dict,
```

Продолжение листинга А.2

```
26          text_editor: Text,
27          progressbar: Progressbar,
28      ) -> None:
29          self.text_editor = text_editor
30          self.progressbar = progressbar
31
32          self.nodes: list[Node] = list()
33          self.__add_nodes(frequency_table)
34
35          self.tree = self.__build_tree()
36          self.__fill_node_value(self.tree)
37
38      def get_code_by_symbol(self, symbol: bytes) -> str:
39          return self.__find_code_by_symbol(
40              symbol=symbol,
41              node=self.tree,
42          )
43
44      def get_symbol_by_code(self, code: str) -> bytes:
45          return self.__find_symbol_by_code(
46              code=code,
47              node=self.tree,
48          )
49
50      def __add_nodes(self, frequency_table: dict) -> None:
51          for key in frequency_table.keys():
52              if frequency_table[key] > 0:
53                  self.nodes.append(
54                      Node(
55                          symbols=[key],
56                          frequency=frequency_table[key],
57                      )
58                  )
59
60      def __find_index_of_min_elem(self) -> int:
61          index = 0
62          for i in range(1, len(self.nodes)):
63              if self.nodes[i].frequency < self.nodes[index].frequency:
64                  index = i
65
66          return index
67
68      def __build_tree(self) -> Node:
69          size_data = len(self.nodes) - 1
70          bar = self.__init_progressbar(
```

Продолжение листинга А.2

```
71              name="Построение дерева Хаффмана",
72              size=size_data,
73          )
74          i = 0
75          while len(self.nodes) > 1:
76              first_node = self.nodes.pop(self.__find_index_of_min_elem())
77              second_node = self.nodes.pop(self.__find_index_of_min_elem())
78              self.nodes.append(
79                  Node(
80                      symbols=first_node.symbols + second_node.symbols,
81                      frequency=first_node.frequency + second_node.frequency,
82                      left=first_node,
83                      right=second_node,
84                  )
85              )
86
87              i += 1
88              self.__update_progressbar(
89                  iteration=i,
90                  size=size_data,
91              )
92              bar.next()
93          bar.finish()
94
95          return self.nodes[0]
96
97      def __fill_node_value(self, node: Node) -> None:
98          if node.left != None:
99              node.left.value += node.value + "0"
100             self.__fill_node_value(node.left)
101
102         if node.right != None:
103             node.right.value += node.value + "1"
104             self.__fill_node_value(node.right)
105
106     def __find_code_by_symbol(self, symbol: bytes, node: Node) -> str:
107         if len(node.symbols) == 1 and node.symbols[0] == symbol:
108             code = node.value
109         # есть ли искомый символ в левой части дерева
110         elif symbol in node.left.symbols:
111             code = self.__find_code_by_symbol(symbol, node.left)
112         # есть ли искомый символ в правой части дерева
113         else:
114             code = self.__find_code_by_symbol(symbol, node.right)
115
```

Продолжение листинга А.2

```
116             return code
117
118         def __find_symbol_by_code(self, code: str, node: Node) -> bytes:
119             if len(code) == 0:
120                 # не дошли до конца дерева, надо взять больший код
121                 if node.left != None or node.left != None:
122                     symbol = None
123                 else:
124                     symbol = node.symbols[0]
125
126             # есть ли искомый символ в левой части дерева
127             elif node.left.value[-1] == code[0]:
128                 symbol = self.__find_symbol_by_code(code[1:], node.left)
129             # есть ли искомый символ в правой части дерева
130             else:
131                 symbol = self.__find_symbol_by_code(code[1:], node.right)
132
133             return symbol
134
135         def __init_progressbar(self, name: str, size: int) -> IncrementalBar:
136             self.text_editor.insert(END, f"{name}\n")
137             self.text_editor.update()
138
139             self.progressbar.step(0)
140             self.progressbar.update()
141             print()
142
143             return IncrementalBar(name, max=size)
144
145         def __update_progressbar(self, iteration: int, size: int) -> None:
146             percent = round(iteration / size * 100)
147             if self.progressbar['value'] + 5 <= percent:
148                 self.progressbar['value'] = percent
149                 self.progressbar.update()
```

Листинг А.3 — Реализация модуля сжатия методом Хаффмана

```
1   import re
2   from tkinter import Text, END
3   from tkinter.ttk import Progressbar
4   from bitarray import bitarray
5   from progress.bar import IncrementalBar
6
7   from tree import Tree
8
```

Продолжение листинга А.3

```
9
10  class Huffman():
11      def __init__(
12          self,
13          text_editor: Text,
14          progressbar: Progressbar,
15      ) -> None:
16          self.text_editor = text_editor
17          self.progressbar = progressbar
18
19      def build_frequency_table(
20          self,
21          bytes_str: bytes,
22          code_size: int,
23      ) -> tuple[dict[bytes, int], int]:
24          codes: list[bytes] = re.findall(rb"[\x00-\xff]{%d}" % code_size,
            ↪  bytes_str)
25          size_data = len(codes)
26
27          frequency_table = {}
28
29          bar = self.__init_progressbar(
30              name="Вычисление таблицы частот символов",
31              size=size_data,
32          )
33          for i, code in enumerate(codes):
34              if code not in frequency_table:
35                  frequency_table[code] = 1
36              else:
37                  frequency_table[code] += 1
38
39              self.__update_progressbar(
40                  iteration=i + 1,
41                  size=size_data,
42              )
43              bar.next()
44          bar.finish()
45
46          size_table = len(frequency_table)
47          max_frequency = max(frequency_table.values())
48          frequency_size = self.__calculate_number_size_in_bytes(max_frequency)
49
50          self.text_editor.insert(END, f"Размер таблицы частот символов:
            ↪  {size_table}\n")
51          self.text_editor.insert(
```

Продолжение листинга А.3

```
52              END,
53              "Максимальная частота символа: {} ({} байт(а) на сохранение частоты
                ↪  символа) \n".format(
54              max_frequency, frequency_size
55          ))
56          self.text_editor.insert(
57              END,
58              "Среднее кол-во повторных использований цепочек байт:
                ↪ {:.2f}\n".format(
59              size_data / size_table
60          ))
61          self.text_editor.update()
62          print(f"\nРазмер таблицы частот символов: {size_table}")
63          print("\nМаксимальная частота символа: {} ({} байт(а) на сохранение
            ↪  частоты)".format(
64              max_frequency, frequency_size
65          ))
66          print("\nСреднее кол-во повторных использований цепочек байт:
            ↪ {:.2f}".format(
67              size_data / size_table
68          ))
69
70          return frequency_table, frequency_size
71
72      def build_tree(self, frequency_table: dict[bytes, int]) -> Tree:
73          return Tree(
74              frequency_table=frequency_table,
75              text_editor=self.text_editor,
76              progressbar=self.progressbar,
77          )
78
79      def compress(self, data: bytes, code_size: int, tree: Tree) -> bytes:
80          codes: list[bytes] = re.findall(rb"[\x00-\xff]{%d}" % code_size, data)
81          size_data = len(codes)
82
83          bar = self.__init_progressbar(
84              name="Сжатие методом Хаффмана",
85              size=size_data,
86          )
87          bits_str = ""
88          for i, code in enumerate(codes):
89              # обход дерева в поисках кода переданного символа
90              bits_str += tree.get_code_by_symbol(code)
91
92              self.__update_progressbar(
```

Продолжение листинга А.3

```
                iteration=i + 1,
                size=size_data,
            )
            bar.next()
        bar.finish()

        bits = self.__add_missing_bits(
            bits=bitarray(bits_str),
            multiplicity=8,
        )

        return self.__to_bytes(bits)

    def decompress(self, data: bytes, tree: Tree) -> bytes:
        bits = self.__to_bits(data)
        bits_str = self.__remove_extra_bits(bits).to01()

        size_data = len(bits_str)
        bar = self.__init_progressbar(
            name="Распаковка методом Хаффмана",
            size=size_data,
        )
        bytes_str = bytes()
        amount_processed_chars = 0
        while amount_processed_chars < size_data:
            byte_sequence, len_symbol = self.__get_decompressed_symbol(
                bits_str=bits_str,
                initial_index=amount_processed_chars,
                tree=tree,
            )
            bytes_str += byte_sequence
            amount_processed_chars += len_symbol

            self.__update_progressbar(
                iteration=amount_processed_chars,
                size=size_data,
            )
            bar.next(n=len_symbol)
        bar.finish()

        return bytes_str

    def __add_missing_bits(
        self,
        bits: bitarray,
```

Продолжение листинга А.3

```
138             multiplicity: int,
139         ) -> bitarray:
140             bits += bitarray("1")
141             return bits + bitarray("0" * (len(bits) % multiplicity))
142 
143         def __remove_extra_bits(self, bits: bitarray) -> bitarray:
144             counter = 1
145             for i in range(len(bits) - 1, 0, -1):
146                 if bits[i] == 0:
147                     counter += 1
148                 else:
149                     break
150 
151             return bits[:-counter]
152 
153         def __to_bytes(self, bits: bitarray) -> bytes:
154             return bits.tobytes()
155 
156         def __to_bits(self, bytes_str: bytes) -> bitarray:
157             bits = bitarray()
158             bits.frombytes(bytes_str)
159             return bits
160 
161         def __get_decompressed_symbol(
162             self,
163             bits_str: str,
164             initial_index: int,
165             tree: Tree,
166         ) -> tuple[bytes, int] | None:
167             for i in range(initial_index, len(bits_str) + 1):
168                 # обход дерева в поисках символа переданного кода
169                 symbol = tree.get_symbol_by_code(bits_str[initial_index:i])
170                 if symbol != None:
171                     return symbol, i - initial_index
172                 # иначе не дошли до конца дерева, надо взять больший код
173 
174         def __init_progressbar(self, name: str, size: int) -> IncrementalBar:
175             self.text_editor.insert(END, f"{name}\n")
176             self.text_editor.update()
177 
178             self.progressbar.step(0)
179             self.progressbar.update()
180             print()
181 
182             return IncrementalBar(name, max=size)
```

Продолжение листинга А.3

```
183
184         def __update_progressbar(self, iteration: int, size: int) -> None:
185             percent = round(iteration / size * 100)
186             if self.progressbar['value'] + 5 <= percent:
187                 self.progressbar['value'] = percent
188                 self.progressbar.update()
189
190         def __calculate_number_size_in_bytes(self, number: int) -> int:
191             return (number.bit_length() + 7) // 8
192
```

Листинг А.4 — Реализация модуля для сжатия методом LZW

```
1   import re
2   from tkinter import Text, END
3   from tkinter.ttk import Progressbar
4   from progress.bar import IncrementalBar
5
6   from constants import BYTES_AMOUNT_PER_PIXEL
7
8
9   class LZW:
10      def __init__(
11          self,
12          text_editor: Text,
13          progressbar: Progressbar,
14      ) -> None:
15          self.text_editor = text_editor
16          self.progressbar = progressbar
17
18      def compress(self, data: bytes) -> tuple[bytes, int, list[bytes]]:
19          """Сжатие данных с 3-байтовыми последовательностями (RGB)."""
20          if len(data) % BYTES_AMOUNT_PER_PIXEL != 0:
21              raise ValueError(f"Размер данных должен быть кратен
                ↪ {BYTES_AMOUNT_PER_PIXEL} (RGB-пиксели).")
22
23          codes: list[bytes] = re.findall(b"[\x00-\xff]{%d}" %
            ↪ BYTES_AMOUNT_PER_PIXEL, data)
24          size_data = len(codes)
25
26          unique_pixels = self.__get_unique_pixels(codes)
27          dictionary = self.__get_initial_dictionary(unique_pixels)
28          chain_count = len(dictionary)
29
30          curr_msg = bytes()
```

Продолжение листинга А.4

```
        result = []

        bar = self.__init_progressbar(
            name="Сжатие методом LZW",
            size=size_data,
        )
        for i, code in enumerate(codes):
            if curr_msg + code in dictionary:
                curr_msg += code
            else:
                # Добавляем код текущей последовательности в результат
                result.append(dictionary[curr_msg])
                # Добавляем новую цепочку в словарь
                dictionary[curr_msg + code] = chain_count
                chain_count += 1
                curr_msg = code

            self.__update_progressbar(
                iteration=i + 1,
                size=size_data
            )
            bar.next()
        bar.finish()

        # Добавляем последний код, если есть
        if curr_msg:
            result.append(dictionary[curr_msg])

        code_size = self.__calculate_code_size(chain_count)

        self.text_editor.insert(END, f"Кол-во цепочек пикселей в словаре:
        ↪ {chain_count}\n")
        self.text_editor.insert(END, f"Размер кода для метода LZW в байтах:
        ↪ {code_size}\n")
        self.text_editor.insert(END, "Среднее число пикселей в цепочках:
        ↪ {:.2f}\n".format(size_data / chain_count))
        self.text_editor.update()
        print(f"\nКол-во цепочек пикселей в словаре: {chain_count}")
        print(f"\nРазмер кода для метода LZW в байтах: {code_size}")
        print("\nСреднее число пикселей в цепочках: {:.2f}".format(size_data /
        ↪ chain_count))

        # Преобразуем результат в байты
        compressed_data = b''.join(
            code.to_bytes(code_size, byteorder='big') for code in result
```

Продолжение листинга А.4

```
        )
        return compressed_data, code_size, unique_pixels

    def decompress(self, data: bytes, code_size: int, unique_pixels: list[bytes])
    ↪ -> bytes:
        """Распаковка данных с 3-байтовыми RGB-последовательностями."""
        inverted_dict = self.__get_inverted_initial_dictionary(unique_pixels)
        chain_count = len(inverted_dict)

        # Разбиваем данные на n-байтовые коды
        codes = [
            int.from_bytes(data[i:i + code_size], byteorder='big')
            for i in range(0, len(data), code_size)
        ]

        # Восстанавливаем данные
        result = bytearray()
        prev_chain = inverted_dict[codes[0]]
        result.extend(prev_chain)

        size_data = len(codes) - 1
        bar = self.__init_progressbar(
            name="Распаковка методом LZW",
            size=size_data,
        )
        for i, code in enumerate(codes[1:]):
            if code in inverted_dict:
                chain = inverted_dict[code]
            elif code == chain_count:
                # Специальный случай: новый код, равный следующему индексу
                chain = prev_chain + prev_chain[:BYTES_AMOUNT_PER_PIXEL]
            else:
                raise ValueError("Неверный код в сжатых данных")

            result.extend(chain)

            # Добавляем новую цепочку в словарь
            inverted_dict[chain_count] = prev_chain +
            ↪ chain[:BYTES_AMOUNT_PER_PIXEL]
            chain_count += 1
            prev_chain = chain

            self.__update_progressbar(
                iteration=i + 1,
                size=size_data
```

Продолжение листинга А.4

```
            )
            bar.next()
        bar.finish()

        return bytes(result)

    def __get_unique_pixels(self, codes: list[bytes]) -> list[bytes]:
        pixels = []
        for code in codes:
            if code not in pixels:
                pixels.append(code)

        size = len(pixels)
        self.text_editor.insert(END, f"Кол-во различных пикселей в изображении: {size}\n")
        self.text_editor.update()
        print(f"\nКол-во различных пикселей в изображении: {size}")

        return pixels

    def __get_initial_dictionary(self, pixels: list[bytes]) -> dict[bytes, int]:
        dictionary = {}
        chain_count = 0
        for pixel in pixels:
            dictionary[pixel] = chain_count
            chain_count += 1

        return dictionary

    def __get_inverted_initial_dictionary(self, pixels: list[bytes]) -> dict[int, bytes]:
        dictionary = {}
        chain_count = 0
        for pixel in pixels:
            dictionary[chain_count] = pixel
            chain_count += 1

        return dictionary

    def __init_progressbar(self, name: str, size: int) -> IncrementalBar:
        self.text_editor.insert(END, f"{name}\n")
        self.text_editor.update()

        self.progressbar.step(0)
        self.progressbar.update()
```

Продолжение листинга А.4

```
158             print()
159
160             return IncrementalBar(name, max=size)
161
162         def __update_progressbar(self, iteration: int, size: int) -> None:
163             percent = round(iteration / size * 100)
164             if self.progressbar['value'] + 5 <= percent:
165                 self.progressbar['value'] = percent
166                 self.progressbar.update()
167
168         def __calculate_code_size(self, number: int) -> int:
169             return (number.bit_length() + 7) // 8
```

ПРИЛОЖЕНИЕ Б

Реализация сравнения методов сжатия статических изображений без потерь

Листинг Б.1 — Модуль для сравнения методов сжатия

```python
from PIL import Image
import matplotlib.pyplot as plt
import matplotlib.offsetbox as offsetbox

from constants import CompressionMethods


def plot_comparison_graph(
    image_paths: list[str],
    compression_rates: dict[CompressionMethods, list[float]],
    title: str,
    y_label: str,
    x_label: str,
    y_lim: int,
) -> None:
    fig, ax = plt.subplots(figsize=(11, 6))
    fig.canvas.manager.set_window_title("Выпускная квалификационная работа (Ковалец Кирилл ИУ7-42М)")
    plt.subplots_adjust(bottom=0.3)
    plt.title(title)

    x_positions = range(len(image_paths))
    bar_width = 0.2

    ax.bar(
        [x - bar_width for x in x_positions],
        compression_rates[CompressionMethods.LZW],
        width=bar_width,
        label='Метод LZW'
    )
    ax.bar(
        x_positions,
        compression_rates[CompressionMethods.HYBRID],
        width=bar_width,
        label='Гибридный метод'
    )
    ax.bar(
        [x + bar_width for x in x_positions],
        compression_rates[CompressionMethods.HUFFMAN],
        width=bar_width,
```

Продолжение листинга Б.1

```
            label='Метод Хаффмана'
        )

        ax.grid(True, axis='y', alpha=0.6, linestyle='--')
        ax.legend()
        ax.set_ylabel(y_label)
        ax.set_xlabel(x_label)

        ax.set_xticks(x_positions)
        ax.set_xticklabels([img.split('/')[-1] for img in image_paths])

        ax.set_ylim(0, y_lim)

        for i, image_path in enumerate(image_paths):
            img = Image.open(image_path)
            img.thumbnail((300, 300))
            imagebox = offsetbox.AnnotationBbox(
                offsetbox=offsetbox.OffsetImage(img, zoom=0.3),
                xy=(i, 0),
                xybox=(i, -32 * y_lim / 100),
                frameon=False,
            )
            ax.add_artist(imagebox)

        plt.show()


    def plot_comparison_bar_chart(
        image_sizes: list[int],
        data_to_decompress_sises: list[int],
        method: str,
    ) -> None:
        fig, ax = plt.subplots(figsize=(5.5, 6))
        fig.canvas.manager.set_window_title("Сравнение значений")
        plt.title("График сравнения размеров сжатого\n изображения с исходным")

        labels = ["Исходное изображение", f"Сжатое изображение\n({method})"]

        ax.bar(
            labels,
            image_sizes,
            label='Размер информации об изображении',
        )
        ax.bar(
            labels,
```

Продолжение листинга Б.1

```
85          data_to_decompress_sises,
86          bottom=image_sizes,
87          label='Размер информации для восстановления изображения',
88      )
89
90      ax.set_ylim(0, 119)
91      ax.set_xticks(range(len(labels)))
92      ax.set_xticklabels(labels)
93      ax.set_ylabel('Размер (в процентах от исходного файла)')
94      ax.set_xlabel('Тип изображения (названия)')
95
96      ax.legend(loc='upper right')
97      ax.grid(axis='y', alpha=0.6, linestyle='--')
98
99      plt.show()
```

ПРИЛОЖЕНИЕ В