



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
(Группа)

(Подпись, дата)

К. Э. Ковалец
(И. О. Фамилия)

Руководитель ВКР

(Подпись, дата)

Н. В. Новик
(И. О. Фамилия)

Нормоконтролер

(Подпись, дата)

Д. Ю. Мальцева
(И. О. Фамилия)

2025 г.

РЕФЕРАТ

Расчетно-пояснительная записка к выпускной квалификационной работе «Метод сжатия статических изображений без потерь на основе алгоритма Хаффмана» содержит 83 страниц, 4 части, 18 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

РЕФЕРАТ	5
ВВЕДЕНИЕ	8
1 Аналитическая часть	9
1.1 Анализ предметной области	9
1.2 Методы сжатия статических изображений без потерь	10
1.2.1 Метод RLE	10
1.2.2 Словарные методы	11
1.2.3 Унарное кодирование	12
1.2.4 Метод Хаффмана	13
1.2.5 Арифметическое кодирование	14
1.2.6 Сравнение методов сжатия без потерь	15
1.3 Методы сжатия статических изображений с потерями	16
1.3.1 Метод сжатия JPEG	16
1.3.2 Wavelet сжатие	17
1.3.3 Фрактальный метод	18
1.3.4 Сравнение методов сжатия с потерями	19
1.4 Цветовые модели изображений	20
1.4.1 Анализ цветовых моделей изображений	20
1.4.2 Сравнение цветовых моделей изображений	21
1.5 Постановка задачи	22
2 Конструкторская часть	24
2.1 Требования к разрабатываемому методу сжатия изображений .	24
2.2 Проектирование метода сжатия изображений	24
2.3 Требования к разрабатываемому ПО	26
2.4 Схемы разрабатываемого гибридного метода сжатия изображений	27
2.4.1 Схема гибридного метода сжатия	27
2.4.2 Схема метода LZW для первичного сжатия данных . . .	28
2.4.3 Схема построения дерева кодов Хаффмана	29
2.4.4 Схема метода Хаффмана для повторного сжатия данных	30

3	Технологическая часть	31
3.1	Используемые программные средства для реализации метода	31
3.2	Формат входных и выходных данных	31
3.3	Структура разработанного ПО	32
3.3.1	Описание этапов гибридного метода сжатия	32
3.3.2	Описание модулей разработанного ПО	33
3.4	Результаты работы ПО	35
4	Исследовательская часть	43
4.1	Критерии оценки методов сжатия изображений	43
4.2	Сравнение разработанного метода сжатия с аналогами	44
4.2.1	Сравнение по степени сжатия изображений	44
4.2.2	Сравнение по размеру информации для распаковки изображений	45
	ЗАКЛЮЧЕНИЕ	48
	СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ	51
	ПРИЛОЖЕНИЕ А Реализация гибридного метода сжатия статических изображений без потерь	52
	ПРИЛОЖЕНИЕ Б Реализация сравнения методов сжатия статических изображений без потерь	80
	ПРИЛОЖЕНИЕ В	83

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 [1]. 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 [2].

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 получаемого 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 [3].

1.2 Методы сжатия статических изображений без потерь

1.2.1 Метод RLE

Метод RLE [4] 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 [5].

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 счетчик повторений.
 - Если отличается или достигнут конец строки:
 - в результирующую строку записать значение предыдущего пикселя и количество его повторений;

— сбросить счетчик и начать отсчет 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) [6], основаны на использовании специального словаря, в котором повторяющиеся последовательности заменяются более короткими кодами. Такой подход позволяет существенно сократить объем данных за счет 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 определенный код. После создания словаря эти коды используются для замены изначальных последовательностей, что и обеспечивает сжатие.

Изначально словарь содержит все возможные односимвольные последовательности (пиксель изображения считается за один символ) [7]. Далее LZW начинает проходить по всем значениям входной строки, формируя текущую последовательность символов. Как только сформируется цепочка значений, отсутствующая в словаре:

- текущую последовательность добавится в словарь с уникальным кодом;
- в выходной поток запишется код уже известной цепочки.

Этот процесс продолжается до тех пор, пока весь поток данных не будет обработан. При декодировании используется тот же словарь значений.

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:
 - присвоить W значение $W + K$.
 - 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 полученный результат в виде последовательности кодов.

1.2.3 Унарное кодирование

Унарное кодирование [8] представляет собой префиксный код, основанный на идее представления целых положительных чисел n в виде последовательности из $(n - 1)$ битов со значением 1, и одного бита со значением 0, следующим за цепочкой единиц. Также возможно обратное представление, где в начале кода идет последовательность из $(n - 1)$ нулей, заканчивающаяся единицей.

Данный метод следует использовать только для представления очень малых чисел, в противном случае унарное кодирование может привести к увеличению размера исходного файла. Примером эффективного использования является сжатие изображений с малой цветовой палитрой. Например, при сжатии черно-белой фотографии с малым количеством оттенков серого (с 8 оттенками для примера) самый часто встречающийся пиксель получит значение унарного кода 0 (соответствующее $n = 1$), а самый редко встречающийся будет представлен значением 11111110 (соответствующим $n = 8$).

Унарное кодирование редко применяется в чистом виде для сжатия изображений, однако используется в качестве одного из этапов более сложных гибридных методов сжатия.

Примеры унарного кодирования:

- представление числа 1 (первый вариант) — 0;
- представление числа 2 (первый вариант) — 10;
- представление числа 3 (первый вариант) — 110;
- представление числа 4 (первый вариант) — 1110.

1.2.4 Метод Хаффмана

Метод Хаффмана [9] представляет собой метод сжатия данных без потерь, в основе которого лежит замена часто встречающихся символов более короткими кодами, а редко встречающихся — более длинными. При использовании данного метода для сжатия изображений пиксели будут выступать в роли кодируемых символов.

При реализации метода Хаффмана необходимо вычислить вес каждого пикселя на основе вероятности вхождения символов в сообщение. Полученная таблица частот будет использоваться при построении дерева Хаффмана.

Алгоритм построения дерева Хаффмана [10].

1. Создается список узлов из всех уникальных символов сообщения.
2. Каждому узлу присваивается вес, равный частоте появления символа в сообщении.
3. Из списка выбираются два узла с наименьшими весами.
4. Создается новый узел-родитель, объединяющий выбранные два, с весом, равным сумме весов дочерних узлов.
5. В направлении к потомку с меньшим весом назначается бит 1, с большим весом — бит 0.
6. Два потомка удаляются из списка, а на их место ставится новый родительский узел.

7. Шаги 2–5 повторяются до тех пор, пока в списке не останется один узел — корень дерева.

Построенное дерево используется для получения кодов символов (пикселей). Для этого нужно выбрать лист дерева Хаффмана, соответствующий текущему значению, и построить путь от него до корня дерева, добавляя биты при каждом переходе от одного узла к другому. Полученная последовательность битов будет представлять собой код данного символа (пикселя), записанный в обратном порядке.

Благодаря уникальным префиксам полученных кодов, каждый символ может быть однозначно декодирован, несмотря на то, что коды Хаффмана имеют переменную длину.

Для восстановления исходного изображения требуется доступ к таблице частот пикселей, по которой можно воссоздать дерево Хаффмана и декодировать сжатые данные. Хранение такой таблицы или структуры дерева приводит к увеличению размера сжатого файла. Также к недостатку рассмотренного метода относится необходимость в дополнительном проходе по изображению:

- первый — для построения дерева Хаффмана;
- второй — для кодирования данных.

1.2.5 Арифметическое кодирование

Арифметическое кодирование [11] — это блочный метод сжатия данных, при котором все входное сообщение преобразуется в единый код, уникальный для данной последовательности символов. При таком подходе нельзя разбить код на отдельные части, соответствующие отдельным символам.

В основе метода лежит представление всего сообщения в виде двоичной дроби в диапазоне от 0 до 1. По мере добавления символов интервал, соответствующий текущему сообщению, сужается, требуя все больше битов для точного его описания.

Для работы алгоритма необходимо заранее определить вероятности появления каждого символа. Значения с большей вероятностью появления уменьшают текущий интервал в меньшей степени, в результате чего к окончательному коду добавляется меньше битов. В то время как редкие символы

требуют большего уточнения интервала и, следовательно, увеличивают длину кода.

Каждому символу сопоставляется определенный интервал на числовой оси, длина которого соответствует вероятности его появления. Эти интервалы располагаются последовательно один за другим и в сумме образуют диапазон от 0 до 1. При кодировании каждый новый символ «вырезает» из текущего интервала подинтервал, соответствующий своей вероятности и положению на оси. В результате конечный интервал однозначно представляет все сообщение.

В отличие от метода Хаффмана, арифметическое кодирование не требует дополнительного прохода по входной строке как при кодировании, так и при восстановлении данных.

1.2.6 Сравнение методов сжатия без потерь

Сравнение предлагается проводить по следующим критериям.

1. Возможность кодирования данных за один проход.
2. Отсутствие необходимости 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 Experts Group) [12] — это метод сжатия изображений с потерями, широко используемый для хранения полноцветных фотографий. Свою популярность JPEG приобрел благодаря способности эффективно уменьшать объем изображений без значительного ухудшения визуального качества. Данный метод работает с блоками пикселей размером 8×8 , яркость и цвет в которых, как правило, изменяются плавно. После разбиения изображения на блоки, каждая полученная матрица раскладывается в двойной ряд по косинусам, что позволяет выделить наиболее значимые коэффициенты. Такой подход позволяет осуществить плавное изменение цветов в изображении, что и приводит к сжатию.

Алгоритм JPEG можно описать следующим образом.

1. **Разделение изображения на блоки** — исходное изображение делится на небольшие квадратные фрагменты по 8×8 пикселей.
2. **Преобразование цветовой модели** — преобразование исходной цветовой модели (чаще всего RGB) в YCbCr, где Y обозначает яркость, а Cb и Cr — цветовые составляющие.
3. **Дискретное косинусное преобразование (DCT)** — применяется к каждому блоку, преобразуя его в набор частотных коэффициентов, отражающих яркостные и цветовые колебания.
4. **Квантование** — полученные коэффициенты округляются с использованием таблицы квантования, в результате чего незначительные детали отбрасываются, что и приводит к потерям.
5. **Энтропийное кодирование** — к оставшимся данным применяется метод энтропийного кодирования, например, алгоритм Хаффмана, где часто встречающиеся значения получают более короткие коды.
6. **Формирование файла** — все закодированные данные объединяются в структуру JPEG-файла, пригодного для хранения и передачи.

7. **Декодирование** — восстановление изображения происходит в обратной последовательности: распаковка, декодирование, обратное квантование, обратное DCT и преобразование в исходную цветовую модель.

Полученное изображение хоть и теряет часть информации, но сохраняет достаточное качество для большинства практических задач.

1.3.2 Wavelet сжатие

Вейвлет-преобразование [13] — это метод обработки сигналов, позволяющий анализировать их частотные характеристики. В отличие от алгоритмов сжатия, таких как JPEG или фрактальные методы, вейвлеты не требуют предварительного разбиения изображения на блоки, так как могут применяться ко всему изображению целиком, что помогает избежать появления артефактов на границах блоков.

Например, рассмотрим одномерное преобразование Хаара. Оно работает с парами элементов сигнала, вычисляя их полусумму и полуразность. Эти два значения можно использовать для восстановления исходных данных, что делает преобразование обратимым. В результате сигнал разобьется на:

- приближенную часть (низкочастотную составляющую), содержащую основную информацию;
- уточняющую часть (высокочастотную составляющую), отражающую мелкие детали.

Двумерное вейвлет-преобразование Хаара строится на применении одномерного преобразования сначала к строкам матрицы изображения, затем к его столбцам. В итоге получится четыре подматрицы:

- одна матрица с пониженным разрешением, представляющая приближенное изображение;
- три матрицы, содержащие уточняющие детали по горизонтали, вертикали и диагонали.

Сжатие достигается за счет обнуления наименее значимых коэффициентов из уточняющих подматриц, что позволяет существенно уменьшить объем данных при сохранении важных деталей изображения.

Рассмотренный метод хорошо подходит для сжатия изображений с плавными переходами, что может быть полезно для работы с медицинскими снимками.

1.3.3 Фрактальный метод

Фрактальное сжатие изображений [14] — это метод, основанный на использовании фрактальных кодов и принципа самоподобия, заключающегося в том, что отдельные участки изображения могут быть приближенно выражены через другие его части.

Ход работы алгоритма можно описать следующим образом.

1. **Разбиение изображения на фрагменты.** Исходное изображение делится на небольшие блоки пикселей (диапазонные блоки), размер которых может составлять 4×4 или 8×8 . Каждый из таких блоков впоследствии сравнивается с другими частями изображения для выявления самоподобия.
2. **Определение базисных блоков.** Среди множества участков изображения выбираются опорные блоки (фракталы [15]), которые будут использоваться в качестве шаблонов для аппроксимации других квадратов.
3. **Поиск наилучших совпадений.** Для каждого участка изображения ищется наиболее похожий базисный блок. Сходство между блоками оценивается с помощью метрик, таких как среднеквадратичное отклонение или евклидово расстояние.
4. **Применение преобразований.** Для определения точного соответствия между рассматриваемым фракталом и диапазонным блоком, к первому применяются различные преобразования, такие как поворот, масштабирование и отражение.
5. **Формирование фрактального описания.** Для каждого блока изображения сохраняется информация о подобранном фрактале и параметрах его преобразования. Эти данные составляют сжатое представление изображения — фрактальный код.

6. **Декодирование изображения.** Восстановление блоков изображения происходит путем поочередного применения к базисным фракталам сохраненных преобразований.

Фрактальное сжатие следует использовать для изображений, содержащих множество повторяющихся структур, таких как текстуры, элементы природы и архитектуры. Данный метод обладает высокой вычислительной сложностью, что затрудняет его использование для работы с изображениями большого размера.

Пример изображения с фракталами приведен на рисунке 1.1.

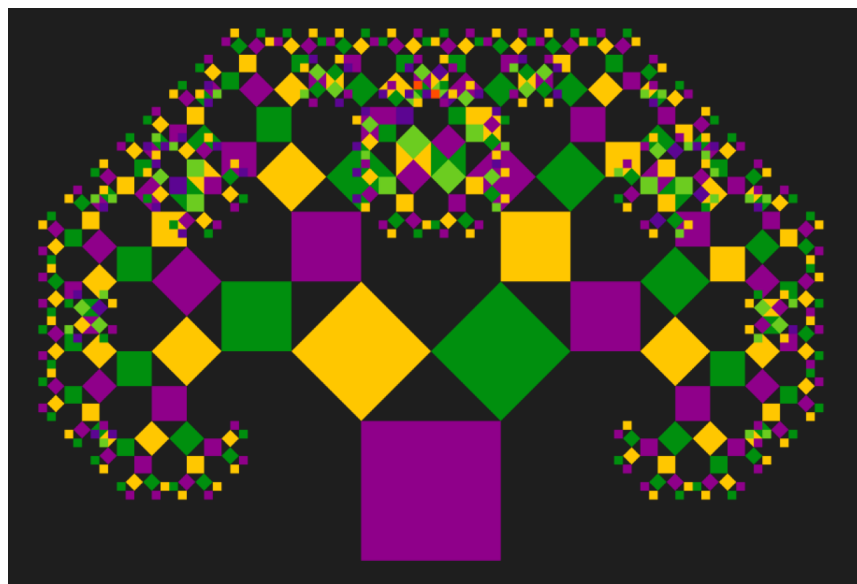


Рисунок 1.1 – Пример изображения с фракталами

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 YCbCr.

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 представляют собой математические способы представления цветов в таком формате, в котором цифровые устройства могут их интерпретировать и отобразить, выполнить над ними различные манипуляции или перевести из одного формата в другой. Данные модели определяют структуру хранения цветов. Далее представлены основные цветовые модели изображений.

1. **RGB (Red, Green, Blue)** [16] — это аддитивная модель цвета, основанная на смешивании трех основных компонентов: красного, зеленого и синего. Она широко применяется в электронных устройствах, где изображение формируется путем добавления света. Каждый цветовой канал принимает значения от 0 до 255, что дает более 16 миллионов возможных оттенков. RGB считается стандартной моделью для отображения информации на экранах.
2. **RGBA (Red, Green, Blue, Alpha)** [16] — это расширенная версия модели RGB, включающая альфа-канал, который задает степень прозрачности. Значение альфа-канала варьируется от 0 (полностью прозрачно) до 255 (полностью непрозрачно). Эта модель особенно полезна в

компьютерной графике и анимации, где требуется реализовать эффекты прозрачности и наложения.

3. **CMYK (Cyan, Magenta, Yellow, Key/Black)** [17] — субтрактивная модель, используемая преимущественно в печатных технологиях. Цвета формируются путём вычитания света при наложении цветных чернил на белую поверхность. В отличие от RGB, CMYK использует голубой, пурпурный, жёлтый и чёрный цвета. Такая модель обеспечивает высокую точность цветопередачи при печати и применяется в типографике и издательском деле.
4. **LAB (Lightness, A, B)** [18] — модель, описывающая цвет на основе восприятия человеческим глазом. Она состоит из компонента яркости (Lightness) и двух цветовых осей: A (от зелёного к красному) и B (от синего к жёлтому). LAB не зависит от конкретного устройства, что делает её полезной при преобразовании изображений между различными цветовыми пространствами и для точной цветокоррекции. Данная цветовая модель охватывает более широкий спектр цветов, чем RGB или CMYK.
5. **HSB (Hue, Saturation, Brightness)** [19] — цветовая модель, описывающая цвет с точки зрения оттенка, насыщенности и яркости. Основной цвет определяется оттенком, насыщенность показывает его чистоту и интенсивность, а яркость отражает степень светлоты. Благодаря своей интуитивности, HSB широко используется в приложениях для работы с цветом, таких как графические и дизайнерские программы.

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. Отсутствие 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 1.2.

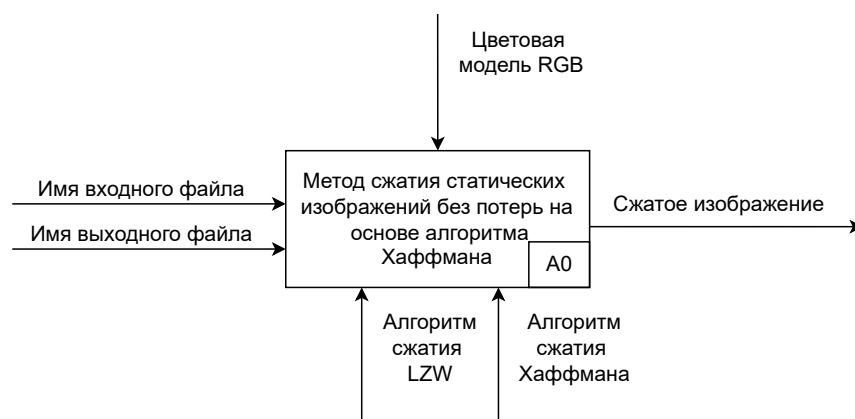


Рисунок 1.2 – Формальная постановка задачи в нотации IDEF0

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. Также были рассмотрены основные цветовые модели изображений.

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 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 [20]. 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 гибридный 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.

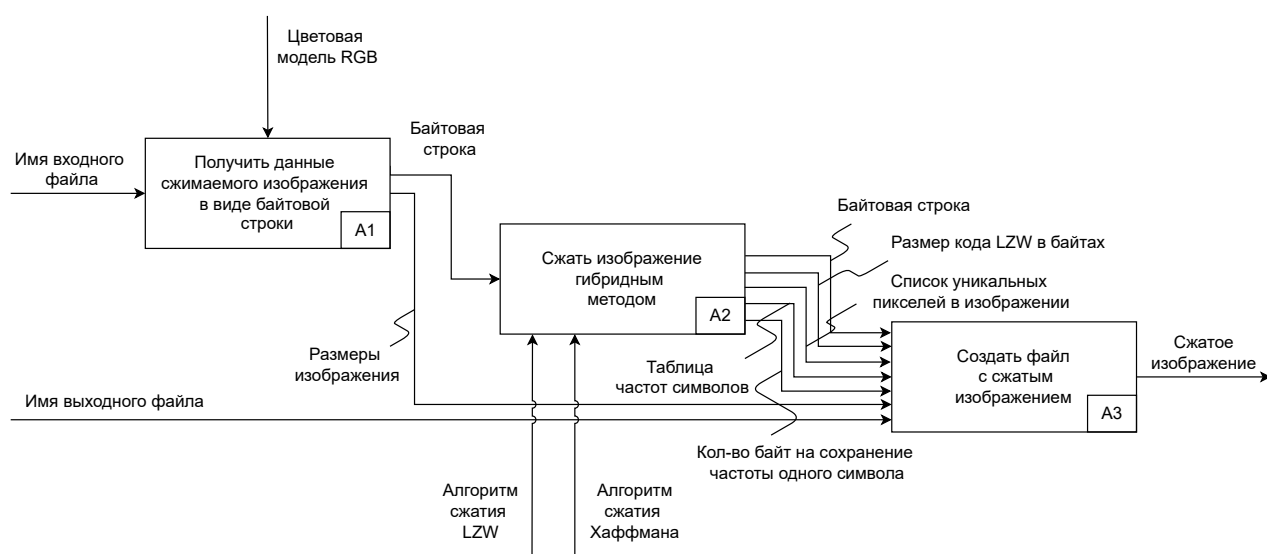


Рисунок 2.1 – Детализированная IDEF0-диаграмма гибридного метода сжатия изображений первого уровня

Диаграмма уровня А2 (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.

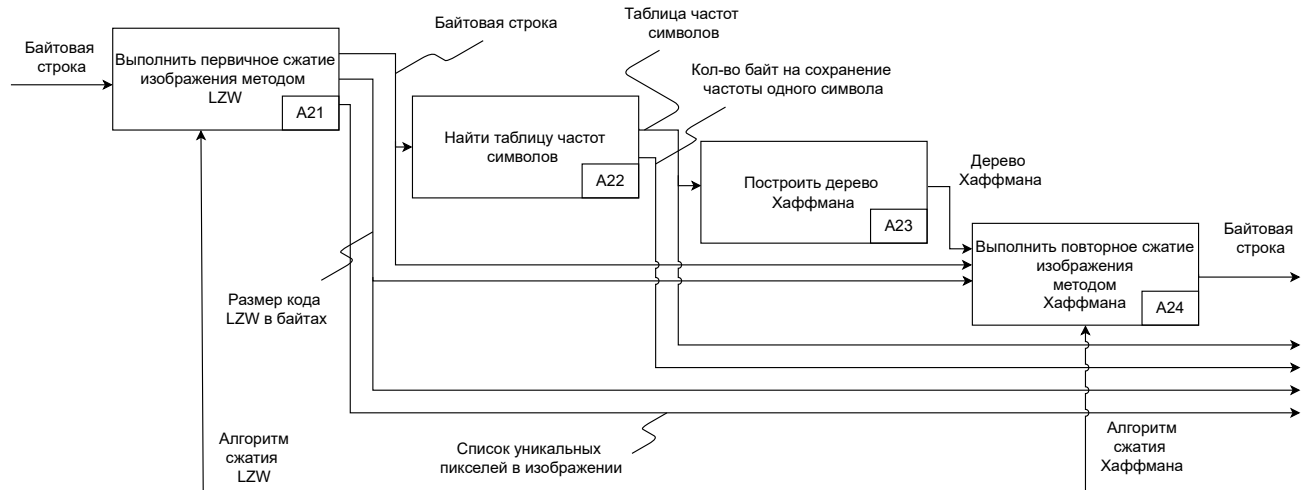


Рисунок 2.2 – Детализированная IDEF0-диаграмма гибридного метода сжатия изображений уровня А2

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 положить 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.

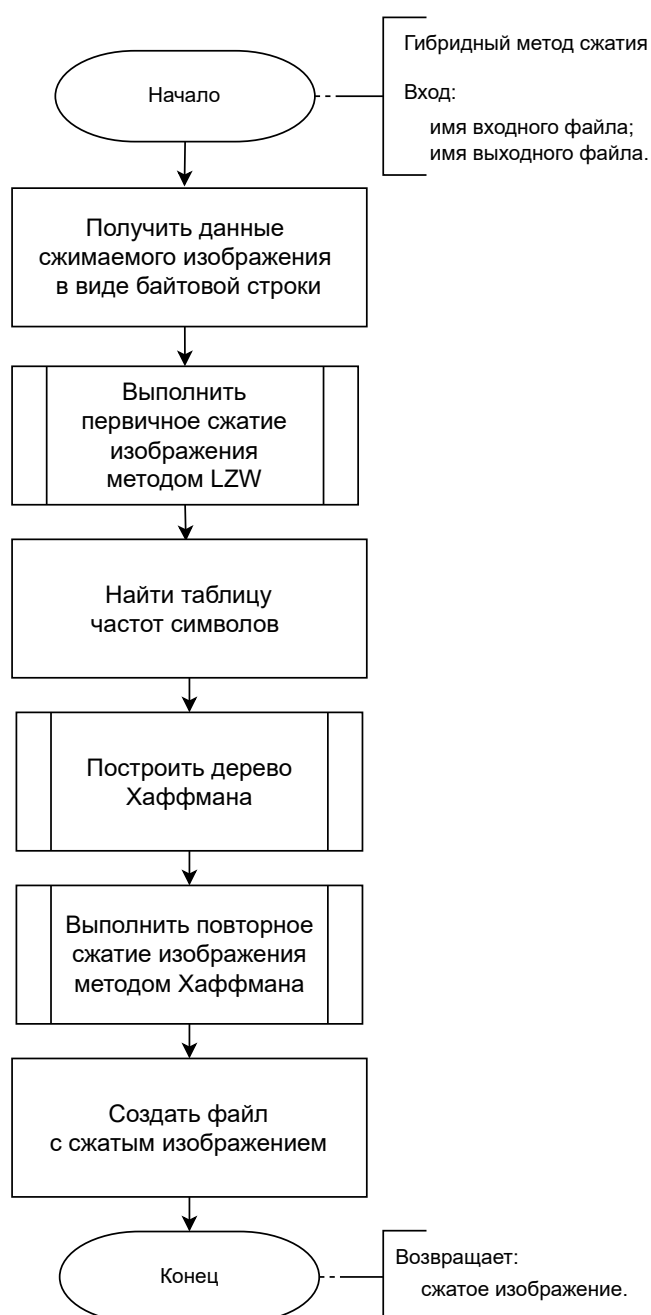


Рисунок 2.3 – Схема гибридного метода сжатия изображений

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.

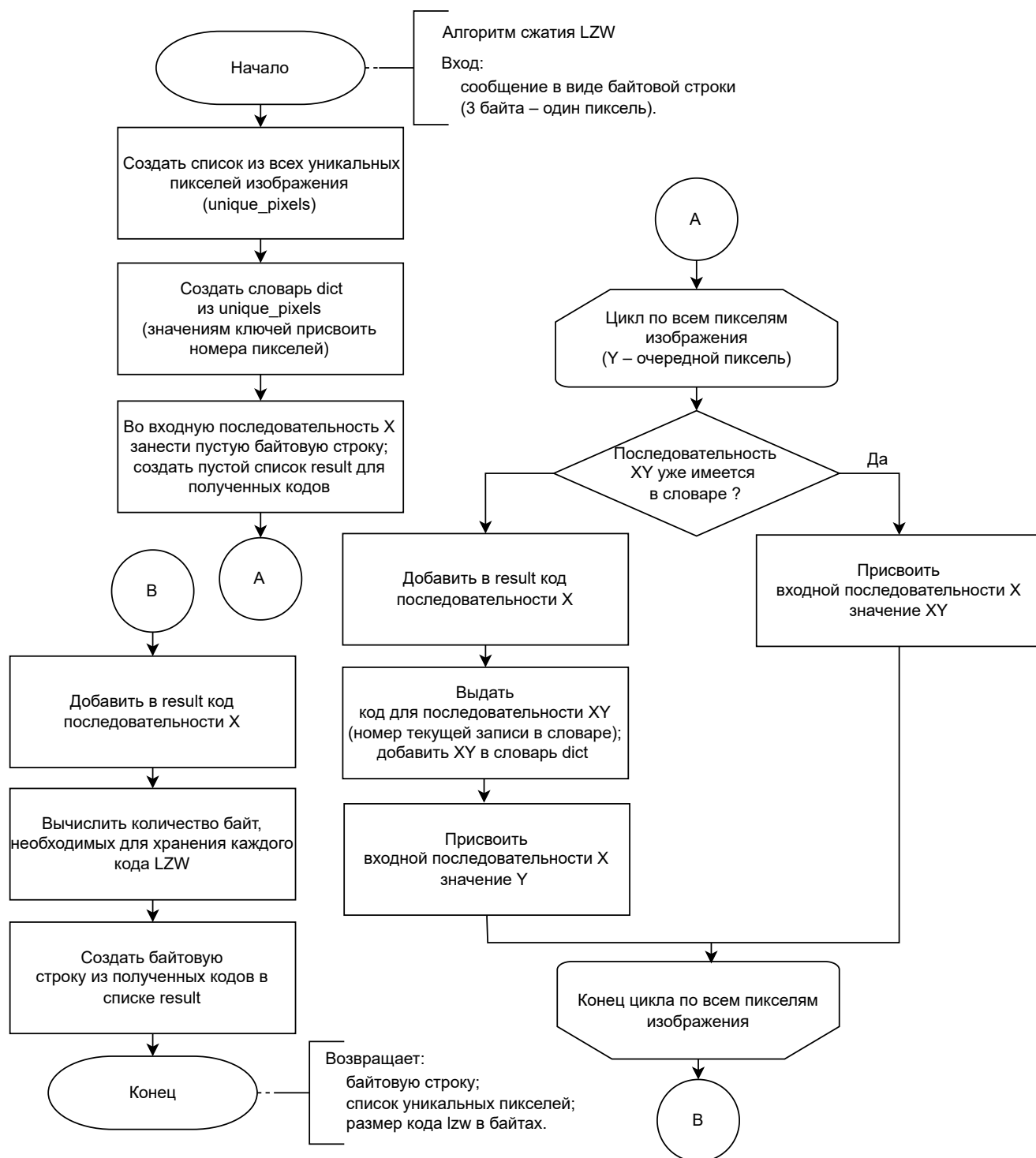


Рисунок 2.4 – Схема метода LZW для первичного сжатия изображений

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 произведено сжатие байтовой строки, полученной на этапе первичного сжатия изображения методом LZW.

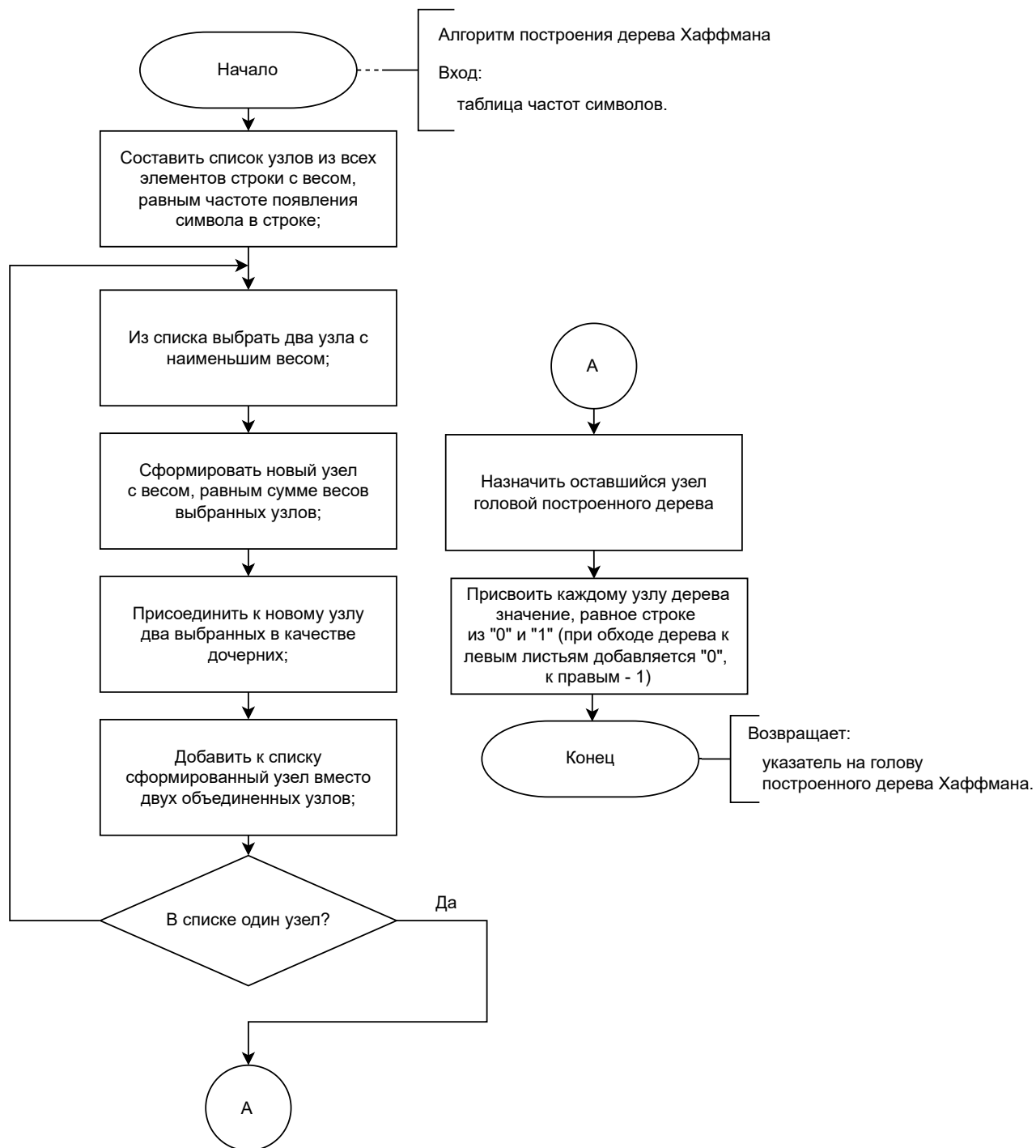


Рисунок 2.5 – Схема построения дерева кодов Хаффмана

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.

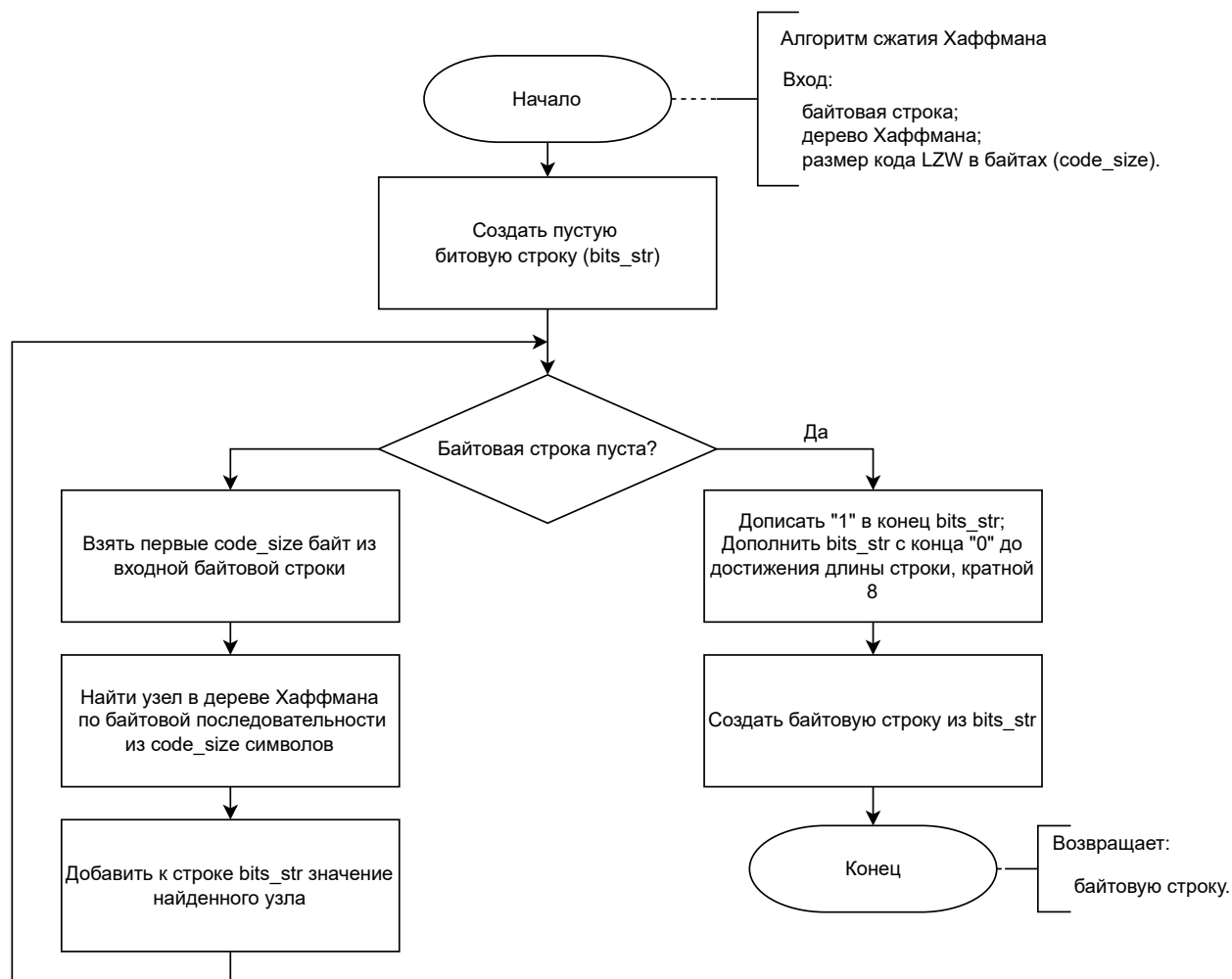


Рисунок 2.6 – Схема метода Хаффмана для повторного сжатия данных

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 [21]. 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` [22]. Она является кроссплатформенной и включена в стандартную библиотеку языка Python в виде отдельного модуля. Для визуализации сравнения работы методов сжатия изображений использовалась библиотека `matplotlib` [23] с следующими модулями:

- `matplotlib.pyplot` [24] — модуль, предоставляющий функции для создания графиков и визуализации данных, использовался для построения столбчатых диаграмм при сравнении методов сжатия изображений;
- `matplotlib.offsetbox` [25] — модуль, предоставляющий возможность размещения текстовых и графических элементов на построенных графиках, использовался для добавления сжимаемых изображений под диаграммами сравнения методов.

Для работы с массивами битов при сжатии данных методом Хаффмана была использована библиотека `bitarray` [26], для отображения прогресса этапов сжатия и распаковки изображения использовалась библиотека `progress` [27]. Для получения списка файлов, доступных для сжатия, был использован модуль `subprocess` [28].

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 изображение. 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, тем больше цепочек байт можно заменить на первом этапе метода). При распаковке 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 отличие 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 [29] компонентов разработанного программного обеспечения представлена на рисунке 3.1. Она показывает структуру зависимостей между основными модулями программы.

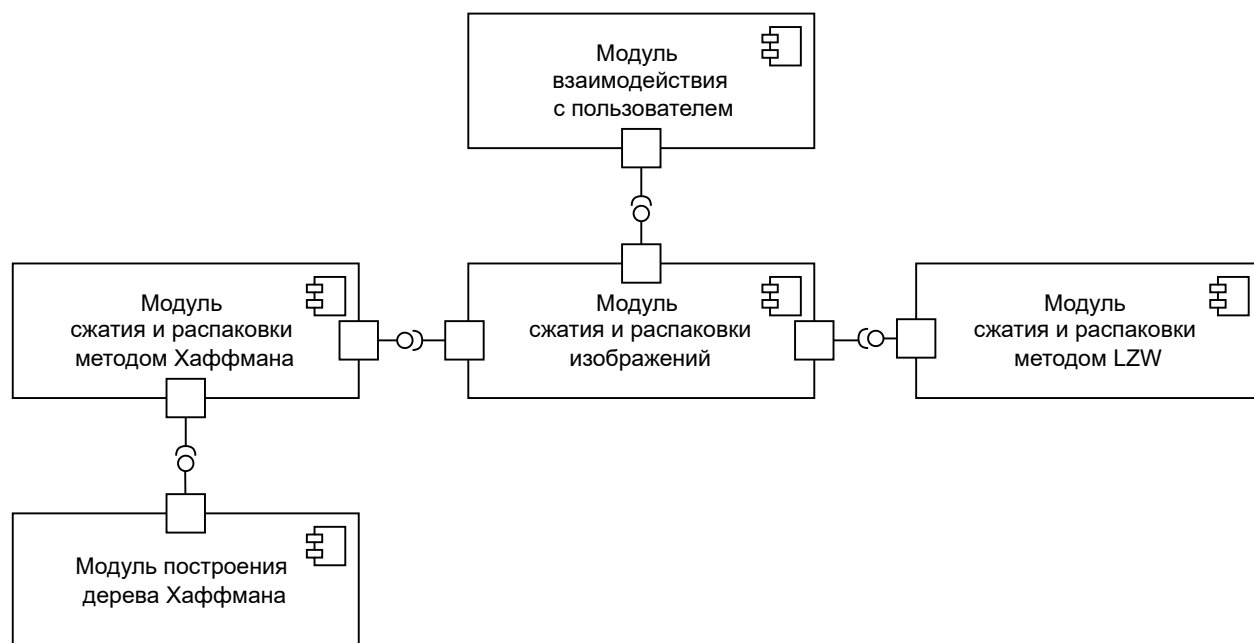


Рисунок 3.1 – UML-диаграмма компонентов разработанного ПО

Разработанное программное обеспечение состоит из следующих модулей.

1. **Модуль сжатия и распаковки изображений.** Реализует основной функционал сжатия и распаковки изображений разработанным гибридным методом (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 (листинг А.5) с использованием библиотеки `tkinter`. Позволяет пользователю выбрать входное изображение, метод сжатия (LZW, Хаффман или гибридный), а также директорию для сохранения резуль-

татов. 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 в окно программы.

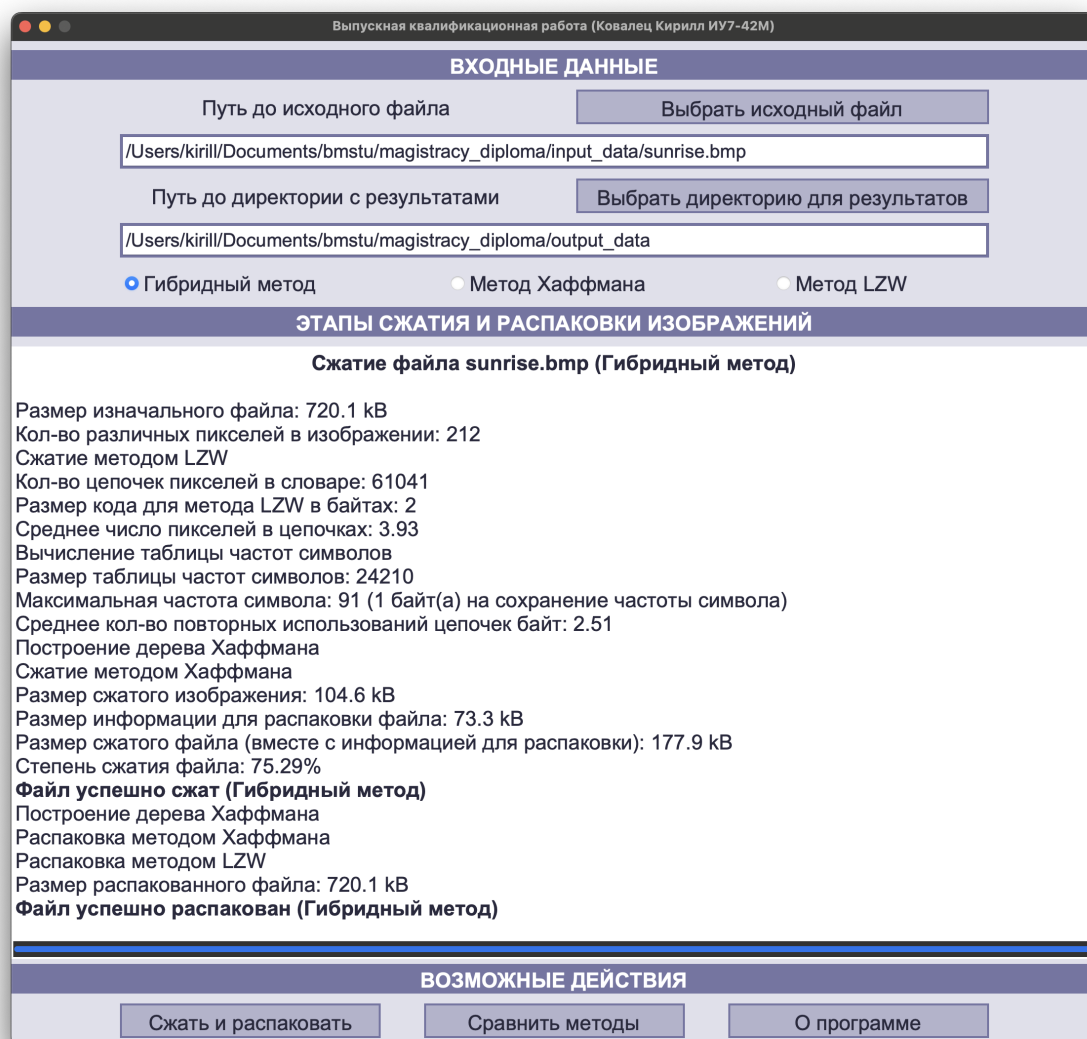


Рисунок 3.2 – Интерфейс программы для сжатия изображений

Для демонстрации работы гибридного метода сжатия было выбрано изображение `sunrise.bmp`, представленное на рисунке 3.3.



Рисунок 3.3 – Исходное изображение восхода солнца

После сжатия файла пользователю выводится график сравнения размеров полученного изображения с исходным в виде столбчатой диаграммы (рисунок 3.4). Данный график позволяет оценить коэффициент сжатия файла и размер метаданных, необходимых для распаковки изображения.

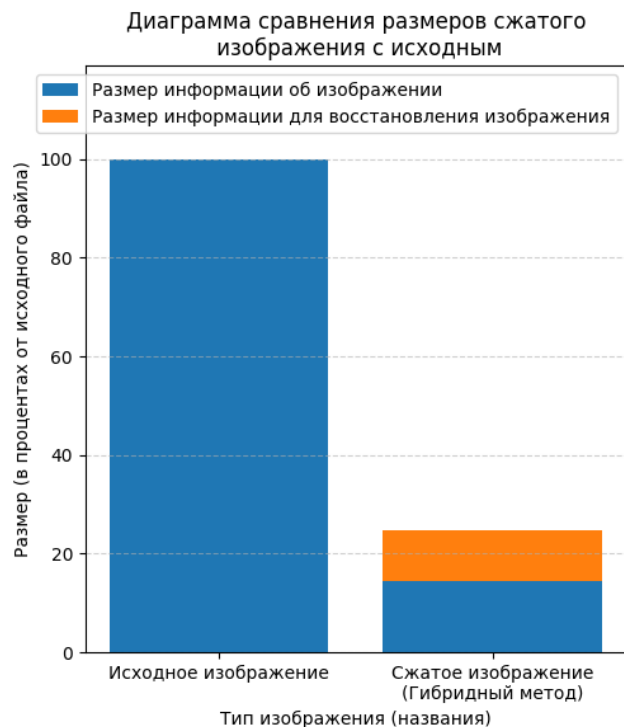


Рисунок 3.4 – Результаты сравнения размеров сжатого изображения с исходным (изображение восхода солнца)

Подробная информация об этапах сжатия и распаковки изображения восхода солнца продемонстрирована в листинге 3.1.

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

```
1  Сжатие файла sunrise.bmp (Гибридный метод)
2
3  Размер изначального файла: 720.1 kB
4
5  Кол-во различных пикселей в изображении: 212
6
7  Сжатие методом LZW |*****| 240000/240000
8
9  Кол-во цепочек пикселей в словаре: 61041
10
11 Размер кода для метода LZW в байтах: 2
12
13 Среднее число пикселей в цепочках: 3.93
14
15 Вычисление таблицы частот символов |*****| 60830/60830
16
17 Размер таблицы частот символов: 24210
18
19 Максимальная частота символа: 91 (1 байт(а) на сохранение частоты)
20
21 Среднее кол-во повторных использований цепочек байт: 2.51
22
23 Построение дерева Хаффмана |*****| 24209/24209
24
25 Сжатие методом Хаффмана |*****| 60830/60830
26
27 Размер сжатого изображения: 104.6 kB
28
29 Размер информации для распаковки файла: 73.3 kB
30
31 Размер сжатого файла (вместе с информацией для распаковки): 177.9 kB
32
33 Степень сжатия файла: 75.29%
34
35 Файл успешно сжат (Гибридный метод)
36
37 Построение дерева Хаффмана |*****| 24209/24209
38
39 Распаковка методом Хаффмана |*****| 836834/836834
40
41 Распаковка методом LZW |*****| 60829/60829
```

Продолжение листинга 3.1

```
42  
43     Размер распакованного файла: 720.1 кВ  
44  
45     Файл успешно распакован (Гибридный метод)
```

Распакованное изображение `sunrise.bmp` с сохранением всех деталей исходного представлено на рисунке 3.5.



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.

Для следующего примера работы гибридного метода сжатия было выбрано изображение `girl.bmp`, представленное на рисунке 3.6.



Рисунок 3.6 – Исходное изображение девушки

Сравнение размеров сжатого файла с исходным для изображения `girl.bmp` представлено на рисунке 3.7.

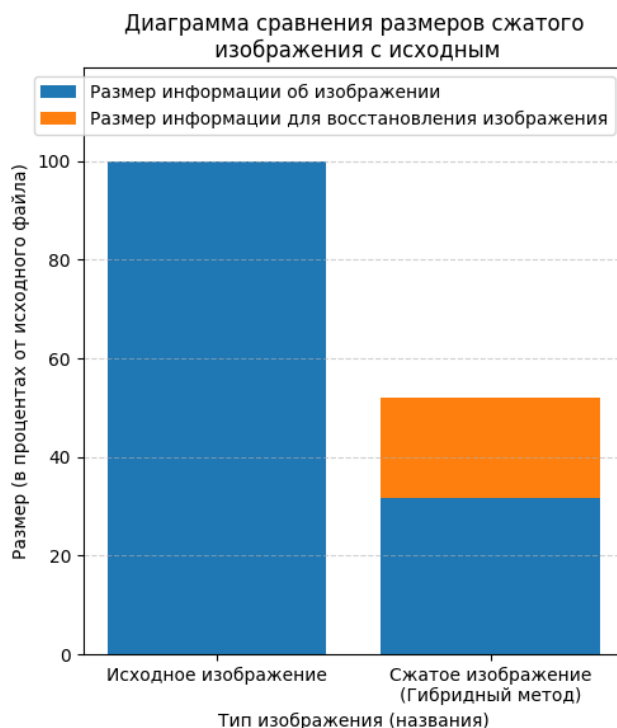


Рисунок 3.7 – Результаты сравнения размеров сжатого изображения с исходным (изображение девушки)

Подробная информация об этапах сжатия и распаковки изображения девушки продемонстрирована в листинге 3.1.

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

```
1  Сжатие файла girl.bmp (Гибридный метод)
2
3  Размер изначального файла: 491.3 kB
4
5  Кол-во различных пикселей в изображении: 2643
6
7  Сжатие методом LZW |*****| 163450/163450
8
9  Кол-во цепочек пикселей в словаре: 96886
10
11 Размер кода для метода LZW в байтах: 3
12
13 Среднее число пикселей в цепочках: 1.69
14
15 Вычисление таблицы частот символов |*****| 94244/94244
16
17 Размер таблицы частот символов: 22942
18
19 Максимальная частота символа: 98 (1 байт(а) на сохранение частоты)
20
21 Среднее кол-во повторных использований цепочек байт: 4.11
22
23 Построение дерева Хаффмана |*****| 22941/22941
24
25 Сжатие методом Хаффмана |*****| 94244/94244
26
27 Размер сжатого изображения: 155.5 kB
28
29 Размер информации для распаковки файла: 99.7 kB
30
31 Размер сжатого файла (вместе с информацией для распаковки): 255.3 kB
32
33 Степень сжатия файла: 48.05%
34
35 Файл успешно сжат (Гибридный метод)
36
37 Построение дерева Хаффмана |*****| 22941/22941
38
39 Распаковка методом Хаффмана |*****| 1244263/1244263
40
41 Распаковка методом LZW |*****| 94243/94243
```

Продолжение листинга 3.2

```
42  
43     Размер распакованного файла: 491.3 kB  
44  
45     Файл успешно распакован (Гибридный метод)
```

Распакованное изображение `girl.bmp` с сохранением всех деталей исходного представлено на рисунке 3.8.



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, описана реализация гибридного метода сжатия статических изображений и приведены результаты работы программы. Также было представлено описание структуры разработанного ПО.

В примере изображения с восходом солнца степень сжатия 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 на 133% (3.93 против 1.69). 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.

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

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 [30]. 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

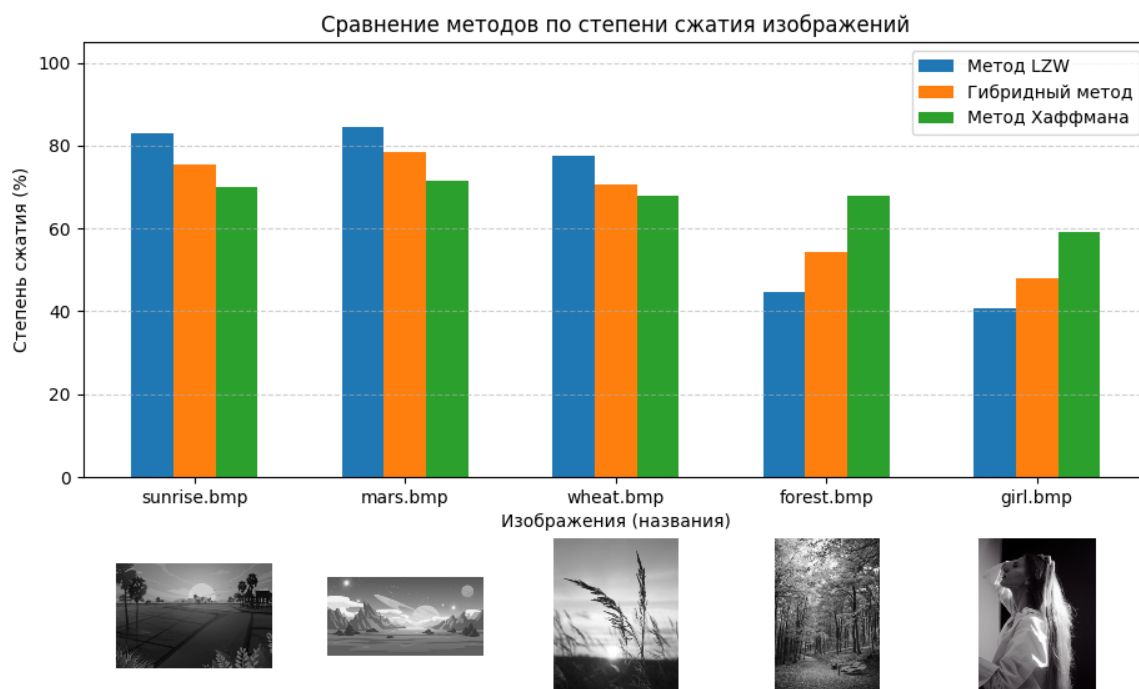


Рисунок 4.1 – Сравнения методов сжатия статических изображений без потерь по степени сжатия

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.

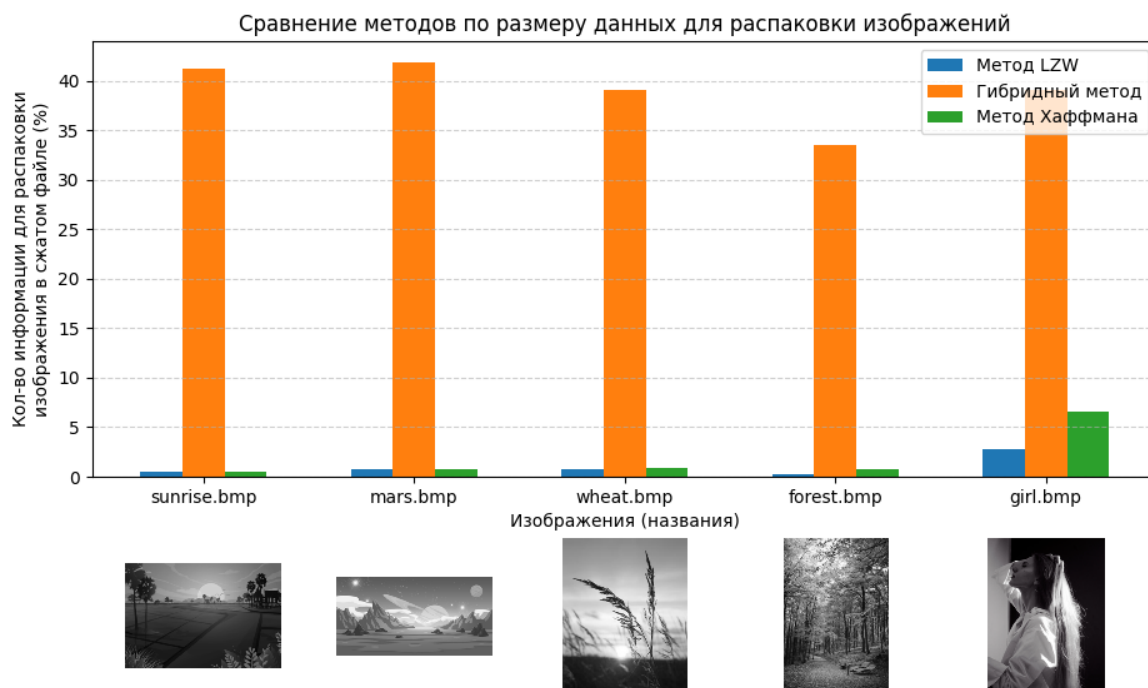


Рисунок 4.2 – Сравнение методов сжатия статических изображений без потерь по количеству информации для распаковки

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 большой 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

В ходе выполнения выпускной квалификационной работы разработан метод сжатия статических изображений без потерь на основе алгоритма Хаффмана. Поставленная цель была достигнута.

Проведен аналитический обзор известных методов сжатия статических изображений. Были рассмотрены алгоритмы сжатия без потерь (RLE, LZW, унарное кодирование, Хаффман и арифметическое кодирование) и с потерями (JPEG, Wavelet, фрактальное сжатие), проведено сравнение методов по выделенным критериям. Рассмотрены основные цветовые модели изображений (RGB, RGBA, CMYK, LAB, HSB).

Разработан метод сжатия статических изображений без потерь на основе алгоритма Хаффмана. Он представляет собой гибридную реализацию, где для первичной обработки изображения используется словарный метод LZW.

Разработано программное обеспечение для демонстрации работы созданного метода. Реализованная программа представляет собой графическое приложение, предоставляющее пользователю возможность выбора изображения для сжатия и распаковки.

Программное обеспечение позволяет провести сравнение гибридного метода с Хаффманом и LZW по степени сжатия изображений и размеру информации, необходимой для их распаковки. Гибридный метод показал более стабильный результат сжатия, минимизировав зависимость от особенностей входных изображений, несмотря на большую величину метаданных, необходимых для восстановления исходных файлов.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. *Лежнев В. Г., Н. М. А.* Математические алгоритмы сжатия изображений. — Краснодар: Кубанский гос. ун-т, 2015. — С. 55.
2. *Тропченко А. Ю.* Методы сжатия изображений, аудиосигналов и видео. — СПб: СПбГУ ИТМО, 2009. — С. 108.
3. *Равский А. А., Левенец А. В.* Современные тенденции развития алгоритмов сжатия // Вестник Тихоокеанского государственного университета. — 2021. — № 3. — С. 29—36.
4. Run-Length Encoding (RLE) [Электронный ресурс]. — Режим доступа URL: https://www.fileformat.info/mirror/egff/ch09_03.htm (Дата обращения: 29.11.2024).
5. *Бакулина М. П.* Повышение эффективности сжатия изображений на основе метода RLE // Проблемы информатики. — 2023. — № 4. — С. 73—77.
6. LZW (Lempel–Ziv–Welch) Compression Technique [Электронный ресурс]. — Режим доступа URL: <https://www.scaler.com/topics/lzw-compression/> (Дата обращения: 29.11.2024).
7. *Kaur H.* Image Compression Techniques with LZW method // International Journal for Research in Applied Science and Engineering Technology. — 2022. — Pp. 1773—1777.
8. [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: 29.11.2024).
9. Huffman Coding Algorithm [Электронный ресурс]. — Режим доступа URL: <https://www.programiz.com/dsa/huffman-coding> (Дата обращения: 29.11.2024).
10. *Виноградова М. С., Ткачева О. С.* Сжатие данных. Алгоритм Хаффмана // Modern European Research. — 2022. — Т. 1, № 3. — С. 60—69.

11. Data Compression with Arithmetic Coding [Электронный ресурс]. — Режим доступа URL: <https://www.scaler.com/topics/data-compression-with-arithmetic-coding/> (Дата обращения: 29.11.2024).
12. JPEG Compression Explained [Электронный ресурс]. — Режим доступа URL: <https://www.baeldung.com/cs/jpeg-compression> (Дата обращения: 29.11.2024).
13. 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: 29.11.2024).
14. Optimization of Fractal Image Compression [Электронный ресурс]. — Режим доступа URL: <https://www.intechopen.com/chapters/72917> (Дата обращения: 29.11.2024).
15. Fractal geometry [Электронный ресурс]. — Режим доступа URL: <https://www.ibm.com/history/fractal-geometry> (Дата обращения: 29.11.2024).
16. HTML RGB and RGBA Colors [Электронный ресурс]. — Режим доступа URL: https://www.w3schools.com/html/html_colors_rgb.asp (Дата обращения: 29.11.2024).
17. Colors CMYK [Электронный ресурс]. — Режим доступа URL: https://www.w3schools.com/colors/colors_cmyk.asp (Дата обращения: 29.11.2024).
18. Lab Color — MATLAB and Simulink [Электронный ресурс]. — Режим доступа URL: <https://www.mathworks.com/discovery/lab-color.html> (Дата обращения: 29.11.2024).
19. 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: 29.11.2024).
20. Ковалец К. Э., Новик Н. В. Метод сжатия статических изображений на основе алгоритма Хаффмана // Вестник Российского нового университета. Серия: Сложные системы: модели, анализ и управление. — 2025. — № 2.

21. Welcome to Python [Электронный ресурс]. — Режим доступа URL: <https://www.python.org> (Дата обращения: 18.01.2025).
22. Tkinter — Python interface to Tcl/Tk [Электронный ресурс]. — Режим доступа URL: <https://docs.python.org/3/library/tkinter.html> (Дата обращения: 18.01.2025).
23. Matplotlib — Visualization with Python [Электронный ресурс]. — Режим доступа URL: <https://matplotlib.org> (Дата обращения: 18.01.2025).
24. 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).
25. 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).
26. bitarray [Электронный ресурс]. — Режим доступа URL: <https://pypi.org/project/bitarray/> (Дата обращения: 18.01.2025).
27. progress [Электронный ресурс]. — Режим доступа URL: <https://pypi.org/project/progress/> (Дата обращения: 18.01.2025).
28. 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).
29. UML diagrams [Электронный ресурс]. — Режим доступа URL: <https://www.ibm.com/docs/en/dma?topic=diagrams-uml> (Дата обращения: 18.01.2024).
30. *Фролов А.* Форматы GPEG, BMP, Gif и их особенности // Системный администратор. — 2024. — № 10. — С. 48—50.

ПРИЛОЖЕНИЕ А

Реализация гибридного метода сжатия статических изображений без потерь

Листинг А.1 — Реализация модуля сжатия изображений

```
1  import re
2  from PIL import Image
3  from os.path import getsize
4  from humanize import naturalsize
5  from tkinter import Text, END
6  from tkinter.ttk import Progressbar
7
8  from constants import CompressionMethods, BYTES_AMOUNT_PER_PIXEL
9  from huffman import Huffman
10 from lzw import LZW
11 from comparison import plot_comparison_bar_chart
12 from color import *
13
14
15 class Compression():
16     compression_methods_names = {
17         CompressionMethods.HYBRID: "Гибридный метод",
18         CompressionMethods.HUFFMAN: "Хаффман",
19         CompressionMethods.LZW: "LZW",
20     }
21
22     def __init__(
23         self,
24         method: CompressionMethods,
25         text_editor: Text,
26         progressbar: Progressbar,
27     ) -> None:
28         self.lzw = LZW(text_editor, progressbar)
29         self.huffman = Huffman(text_editor, progressbar)
30         self.text_editor = text_editor
31         self.method = method
32
33     def compress(self, input_file_name: str, output_file_name: str) -> None:
34         short_filename = input_file_name.split("/")[-1]
35         method_name = self.compression_methods_names[self.method]
36         size = getsize(input_file_name)
37         size_str = naturalsize(size)
38
39         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 =
53             ↪ self.lzw.compress(data)
54             frequency_table, frequency_size =
55             ↪ self.huffman.build_frequency_table(
56                 bytes_str=lzw_compressed,
57                 code_size=lzw_code_size
58             )
59             tree = self.huffman.build_tree(frequency_table)
60             compressed = self.huffman.compress(lzw_compressed, lzw_code_size,
61             ↪ tree)
62
63             data_to_decompress = (
64                 self.width.to_bytes(4, byteorder='big') +
65                 self.height.to_bytes(4, byteorder='big') +
66                 lzw_code_size.to_bytes(4, byteorder='big') +
67                 len(unique_pixels).to_bytes(4, byteorder='big') +
68                 b''.join(unique_pixels) +
69                 frequency_size.to_bytes(4, byteorder='big') +
70                 len(frequency_table).to_bytes(4, byteorder='big') +
71                 self.__convert_frequency_table_to_bytes(frequency_table,
72                 ↪ frequency_size)
73             )
74         case CompressionMethods.HUFFMAN:
75             frequency_table, frequency_size =
76             ↪ self.huffman.build_frequency_table(
77                 bytes_str=data,
78                 code_size=BYTES_AMOUNT_PER_PIXEL
79             )
80             tree = self.huffman.build_tree(frequency_table)
81             compressed = self.huffman.compress(data, BYTES_AMOUNT_PER_PIXEL,
82             ↪ tree)
```

Продолжение листинга А.1

```

78         data_to_decompress = (
79             self.width.to_bytes(4, byteorder='big') +
80             self.height.to_bytes(4, byteorder='big') +
81             frequency_size.to_bytes(4, byteorder='big') +
82             len(frequency_table).to_bytes(4, byteorder='big') +
83             self.__convert_frequency_table_to_bytes(frequency_table,
84                 ↪ frequency_size)
85         )
86         case _:
87             compressed, lzw_code_size, unique_pixels = self.lzw.compress(data)
88
89             data_to_decompress = (
90                 self.width.to_bytes(4, byteorder='big') +
91                 self.height.to_bytes(4, byteorder='big') +
92                 lzw_code_size.to_bytes(4, byteorder='big') +
93                 len(unique_pixels).to_bytes(4, byteorder='big') +
94                 b''.join(unique_pixels)
95             )
96
97         with open(output_file_name, "wb") as f:
98             f.write(data_to_decompress + compressed)
99
100         compressed_file_size = getsize(output_file_name)
101         compressed_file_size_str = naturalsize(compressed_file_size)
102         compressed_data_size_str = naturalsize(len(compressed))
103         size_data_to_decompress_str = naturalsize(len(data_to_decompress))
104         compression_ratio = (size - compressed_file_size) / size * 100
105
106         self.text_editor.insert(END, f"Размер сжатого изображения:
107             ↪ {compressed_data_size_str}\n")
108         self.text_editor.insert(END, f"Размер информации для распаковки файла:
109             ↪ {size_data_to_decompress_str}\n")
110         self.text_editor.insert(END, f"Размер сжатого файла (вместе с информацией
111             ↪ для распаковки): {compressed_file_size_str}\n")
112         self.text_editor.insert(END, "Степень сжатия файла:
113             ↪ {:.2f}%\n".format(compression_ratio))
114         self.text_editor.insert(END, f"Файл успешно сжат ({method_name})\n",
115             ↪ ("bold",))
116         self.text_editor.update()
117         print(f"\nРазмер сжатого изображения: {compressed_data_size_str}")
118         print(f"\nРазмер информации для распаковки файла:
119             ↪ {size_data_to_decompress_str}")
120         print(f"\nРазмер сжатого файла (вместе с информацией для распаковки):
121             ↪ {compressed_file_size_str}")
122         print(f"\nСтепень сжатия файла: {:.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],
133                                     ↪ byteorder='big')
134                 start += 4
135                 height = int.from_bytes(bytes_str[start:start + 4],
136                                     ↪ byteorder='big')
137                 start += 4
138                 lzw_code_size = int.from_bytes(bytes_str[start:start + 4],
139                                     ↪ byteorder='big')
140                 start += 4
141                 unique_pixels_count = int.from_bytes(bytes_str[start:start + 4],
142                                     ↪ byteorder='big')
143                 start += 4
144                 unique_pixels = self.__convert_bytes_to_list_of_unique_pixels(
145                     byte_string=bytes_str[start:start + unique_pixels_count *
146                                     ↪ BYTES_AMOUNT_PER_PIXEL],
147                 )
148                 start += unique_pixels_count * BYTES_AMOUNT_PER_PIXEL
149                 frequency_size = int.from_bytes(bytes_str[start:start + 4],
150                                     ↪ byteorder='big')
151                 start += 4
152                 frequency_table_size = int.from_bytes(bytes_str[start:start + 4],
153                                     ↪ byteorder='big')
154                 start += 4
155                 frequency_table = self.__convert_bytes_to_frequency_table(
156                     byte_string=bytes_str[start:start + frequency_table_size *
157                                     ↪ (lzw_code_size + frequency_size)],
158                     code_size=lzw_code_size,
159                     frequency_size=frequency_size,
```

Продолжение листинга А.1

```

152         )
153         start += frequency_table_size * (lzw_code_size + frequency_size)
154
155         tree = self.huffman.build_tree(frequency_table)
156         huffman_decompressed = self.huffman.decompress(bytes_str[start:],
157             ↪ tree)
158         decompressed = self.lzw.decompress(
159             data=huffman_decompressed,
160             code_size=lzw_code_size,
161             unique_pixels=unique_pixels,
162         )
163     case CompressionMethods.HUFFMAN:
164         width = int.from_bytes(bytes_str[start:start + 4],
165             ↪ byteorder='big')
166         start += 4
167         height = int.from_bytes(bytes_str[start:start + 4],
168             ↪ byteorder='big')
169         start += 4
170         frequency_size = int.from_bytes(bytes_str[start:start + 4],
171             ↪ byteorder='big')
172         start += 4
173         frequency_table_size = int.from_bytes(bytes_str[start:start + 4],
174             ↪ byteorder='big')
175         start += 4
176         frequency_table = self.__convert_bytes_to_frequency_table(
177             byte_string=bytes_str[start:start + frequency_table_size *
178             ↪ (BYTES_AMOUNT_PER_PIXEL + frequency_size)],
179             code_size=BYTES_AMOUNT_PER_PIXEL,
180             frequency_size=frequency_size,
181         )
182         start += frequency_table_size * (BYTES_AMOUNT_PER_PIXEL +
183             ↪ frequency_size)
184
185         tree = self.huffman.build_tree(frequency_table)
186         decompressed = self.huffman.decompress(bytes_str[start:], tree)
187     case _:
188         width = int.from_bytes(bytes_str[start:start + 4],
189             ↪ byteorder='big')
190         start += 4
191         height = int.from_bytes(bytes_str[start:start + 4],
192             ↪ byteorder='big')
193         start += 4
194         lzw_code_size = int.from_bytes(bytes_str[start:start + 4],
195             ↪ byteorder='big')
196         start += 4

```

Продолжение листинга A.1

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

Продолжение листинга А.1

```
229     ) -> dict[bytes, int]:
230         frequency_table = {}
231         for i in range(0, len(byte_string), code_size + frequency_size):
232             code = byte_string[i:i + code_size]
233             frequency = byte_string[i + code_size:i + code_size + frequency_size]
234             frequency_table[code] = int.from_bytes(frequency, byteorder='big')
235
236         return frequency_table
237
238     def __convert_bytes_to_list_of_unique_pixels(
239         self,
240         byte_string: bytes,
241     ) -> list[bytes]:
242         return re.findall(
243             rb"[\x00-\xff]{%d}" % BYTES_AMOUNT_PER_PIXEL,
244             byte_string,
245         )
```

Листинг А.2 — Реализация модуля для построения дерева Хаффмана

```
1     from tkinter import Text, END
2     from tkinter.ttk import Progressbar
3     from progress.bar import IncrementalBar
4
5
6     class Node():
7         def __init__(
8             self,
9             symbols: list[bytes],
10            frequency: int,
11            left=None,
12            right=None
13        ):
14            self.symbols = symbols
15            self.frequency = frequency
16            self.value = ""
17
18            self.left = left
19            self.right = right
20
21
22     class Tree():
23         def __init__(
24             self,
25             frequency_table: dict,
```

Продолжение листинга A.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.right != 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,
25             ↪ bytes_str)
26         size_data = len(codes)
27
28         frequency_table = {}
29
30         bar = self.__init_progressbar(
31             name="Вычисление таблицы частот символов",
32             size=size_data,
33         )
34         for i, code in enumerate(codes):
35             if code not in frequency_table:
36                 frequency_table[code] = 1
37             else:
38                 frequency_table[code] += 1
39
40             self.__update_progressbar(
41                 iteration=i + 1,
42                 size=size_data,
43             )
44             bar.next()
45         bar.finish()
46
47         size_table = len(frequency_table)
48         max_frequency = max(frequency_table.values())
49         frequency_size = self.__calculate_number_size_in_bytes(max_frequency)
50
51         self.text_editor.insert(END, f"Размер таблицы частот символов:
52             ↪ {size_table}\n")
53         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

```
193         iteration=i + 1,
194         size=size_data,
195     )
196     bar.next()
197 bar.finish()
198
199     bits = self.__add_missing_bits(
200         bits=bitarray(bits_str),
201         multiplicity=8,
202     )
203
204     return self.__to_bytes(bits)
205
206 def decompress(self, data: bytes, tree: Tree) -> bytes:
207     bits = self.__to_bits(data)
208     bits_str = self.__remove_extra_bits(bits).to01()
209
210     size_data = len(bits_str)
211     bar = self.__init_progressbar(
212         name="Распаковка методом Хаффмана",
213         size=size_data,
214     )
215     bytes_str = bytes()
216     amount_processed_chars = 0
217     while amount_processed_chars < size_data:
218         byte_sequence, len_symbol = self.__get_decompressed_symbol(
219             bits_str=bits_str,
220             initial_index=amount_processed_chars,
221             tree=tree,
222         )
223         bytes_str += byte_sequence
224         amount_processed_chars += len_symbol
225
226         self.__update_progressbar(
227             iteration=amount_processed_chars,
228             size=size_data,
229         )
230         bar.next(n=len_symbol)
231     bar.finish()
232
233     return bytes_str
234
235 def __add_missing_bits(
236     self,
237     bits: bitarray,
```

Продолжение листинга А.3

```
138         multiplicity: int,
139     ) -> bytearray:
140         bits += bytearray("1")
141         return bits + bytearray("0" * (len(bits) % multiplicity))
142
143     def __remove_extra_bits(self, bits: bytearray) -> bytearray:
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: bytearray) -> bytes:
154         return bits.tobytes()
155
156     def __to_bits(self, bytes_str: bytes) -> bytearray:
157         bits = bytearray()
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}" %
24                 ↪ BYTES_AMOUNT_PER_PIXEL, data)
25             size_data = len(codes)
26
27             unique_pixels = self.__get_unique_pixels(codes)
28             dictionary = self.__get_initial_dictionary(unique_pixels)
29             chain_count = len(dictionary)
30
31             curr_msg = bytes()
```

Продолжение листинга А.4

```

31         result = []
32
33         bar = self.__init_progressbar(
34             name="Сжатие методом LZW",
35             size=size_data,
36         )
37         for i, code in enumerate(codes):
38             if curr_msg + code in dictionary:
39                 curr_msg += code
40             else:
41                 # Добавляем код текущей последовательности в результат
42                 result.append(dictionary[curr_msg])
43                 # Добавляем новую цепочку в словарь
44                 dictionary[curr_msg + code] = chain_count
45                 chain_count += 1
46                 curr_msg = code
47
48             self.__update_progressbar(
49                 iteration=i + 1,
50                 size=size_data
51             )
52             bar.next()
53         bar.finish()
54
55         # Добавляем последний код, если есть
56         if curr_msg:
57             result.append(dictionary[curr_msg])
58
59         code_size = self.__calculate_code_size(chain_count)
60
61         self.text_editor.insert(END, f"Кол-во цепочек пикселей в словаре:
62         ↪ {chain_count}\n")
63         self.text_editor.insert(END, f"Размер кода для метода LZW в байтах:
64         ↪ {code_size}\n")
65         self.text_editor.insert(END, "Среднее число пикселей в цепочках:
66         ↪ {:.2f}\n".format(size_data / chain_count))
67         self.text_editor.update()
68         print(f"\nКол-во цепочек пикселей в словаре: {chain_count}")
69         print(f"\nРазмер кода для метода LZW в байтах: {code_size}")
70         print(f"\nСреднее число пикселей в цепочках: {:.2f}".format(size_data /
71         ↪ chain_count))
72
73         # Преобразуем результат в байты
74         compressed_data = b''.join(
75             code.to_bytes(code_size, byteorder='big') for code in result

```


Продолжение листинга A.4

```

72         )
73         return compressed_data, code_size, unique_pixels
74
75     def decompress(self, data: bytes, code_size: int, unique_pixels: list[bytes])
76     ↪ -> bytes:
77         """Распаковка данных с 3-байтовыми RGB-последовательностями."""
78         inverted_dict = self.__get_inverted_initial_dictionary(unique_pixels)
79         chain_count = len(inverted_dict)
80
81         # Разбиваем данные на n-байтовые коды
82         codes = [
83             int.from_bytes(data[i:i + code_size], byteorder='big')
84             for i in range(0, len(data), code_size)
85         ]
86
87         # Восстанавливаем данные
88         result = bytearray()
89         prev_chain = inverted_dict[codes[0]]
90         result.extend(prev_chain)
91
92         size_data = len(codes) - 1
93         bar = self.__init_progressbar(
94             name="Распаковка методом LZW",
95             size=size_data,
96         )
97         for i, code in enumerate(codes[1:]):
98             if code in inverted_dict:
99                 chain = inverted_dict[code]
100             elif code == chain_count:
101                 # Специальный случай: новый код, равный следующему индексу
102                 chain = prev_chain + prev_chain[:BYTES_AMOUNT_PER_PIXEL]
103             else:
104                 raise ValueError("Неверный код в сжатых данных")
105
106             result.extend(chain)
107
108             # Добавляем новую цепочку в словарь
109             inverted_dict[chain_count] = prev_chain +
110             ↪ chain[:BYTES_AMOUNT_PER_PIXEL]
111             chain_count += 1
112             prev_chain = chain
113
114             self.__update_progressbar(
115                 iteration=i + 1,
116                 size=size_data

```

Продолжение листинга A.4

```
115         )
116         bar.next()
117     bar.finish()
118
119     return bytes(result)
120
121     def __get_unique_pixels(self, codes: list[bytes]) -> list[bytes]:
122         pixels = []
123         for code in codes:
124             if code not in pixels:
125                 pixels.append(code)
126
127         size = len(pixels)
128         self.text_editor.insert(END, f"Кол-во различных пикселей в изображении:
129         ↪ {size}\n")
130         self.text_editor.update()
131         print(f"\nКол-во различных пикселей в изображении: {size}")
132
133         return pixels
134
135     def __get_initial_dictionary(self, pixels: list[bytes]) -> dict[bytes, int]:
136         dictionary = {}
137         chain_count = 0
138         for pixel in pixels:
139             dictionary[pixel] = chain_count
140             chain_count += 1
141
142         return dictionary
143
144     def __get_inverted_initial_dictionary(self, pixels: list[bytes]) -> dict[int,
145     ↪ bytes]:
146         dictionary = {}
147         chain_count = 0
148         for pixel in pixels:
149             dictionary[chain_count] = pixel
150             chain_count += 1
151
152         return dictionary
153
154     def __init_progressbar(self, name: str, size: int) -> IncrementalBar:
155         self.text_editor.insert(END, f"{name}\n")
156         self.text_editor.update()
157
158         self.progressbar.step(0)
159         self.progressbar.update()
```

Продолжение листинга A.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
```

Листинг A.5 — Реализация модуля взаимодействия с пользователем

```
1     import os
2     from tkinter import (
3         Tk,
4         Label,
5         Entry,
6         Radiobutton,
7         IntVar,
8         Button,
9         messagebox,
10        filedialog,
11        Text,
12        END,
13        DISABLED,
14    )
15    from tkinter.ttk import Progressbar
16
17    from compression import Compression
18    from comparison import plot_comparison_graph
19    from constants import *
20    from color import *
21
22
23    class Window():
24        window: Tk
25        codeSizeEntry: Entry
26        inputFilenameEntry: Entry
27        outputDirectoryEntry: Entry
28        textEditor: Text
29        progressbar: Progressbar
30
```

Продолжение листинга А.5

```
31     def __init__(self, windowWidth: int, windowHeight: int):
32         self.window = self.createWindow(windowWidth, windowHeight)
33         self.createInterface(windowWidth, windowHeight)
34
35     def createWindow(self, windowWidth: int, windowHeight: int):
36         window = Tk()
37         window.title("Выпускная квалификационная работа (Ковалец Кирилл ИУ7-42М)")
38         window.geometry("{0}x{1}".format(windowWidth, windowHeight))
39         window.resizable(False, False)
40         window["bg"] = PURPLE_LIGHT
41
42         return window
43
44     def createInterface(self, windowWidth: int, windowHeight: int):
45         Label(
46             text = "ВХОДНЫЕ ДАННЫЕ",
47             font = ("Arial", 16, "bold"),
48             bg = PURPLE_DARK,
49             fg = "white",
50         ).place(
51             width = windowWidth,
52             height = 30,
53             x = 0,
54             y = 10,
55         )
56
57         Label(
58             text = "Путь до исходного файла",
59             font = ("Arial", 16),
60             bg = PURPLE_LIGHT,
61             fg = PURPLE_SUPER_DARK,
62         ).place(
63             width = windowWidth * 0.38,
64             height = 35,
65             x = windowWidth * 0.1,
66             y = 50,
67         )
68
69         Button(
70             highlightbackground = PURPLE_DARK,
71             highlightthickness = 30,
72             fg = PURPLE_LIGHT,
73             state = DISABLED,
74         ).place(
75             width = windowWidth * 0.38,
```

Продолжение листинга A.5

```
76         height = 35,
77         x = windowWidth * 0.52,
78         y = 50,
79     )
80     Button(
81         text = "Выбрать исходный файл",
82         font = ("Arial", 16),
83         fg = PURPLE_SUPER_DARK,
84         highlightbackground = PURPLE,
85         highlightthickness = 30,
86         command = lambda: self.setInputFilenameEntry(),
87     ).place(
88         width = windowWidth * 0.38 - 4,
89         height = 31,
90         x = windowWidth * 0.52 + 2,
91         y = 52,
92     )
93
94     self.inputFilenameEntry = Entry(
95         font = ("Arial", 14),
96         bg = "white",
97         fg = PURPLE_SUPER_DARK,
98         highlightbackground = PURPLE_DARK,
99     )
100     self.inputFilenameEntry.place(
101         width = windowWidth * 0.8,
102         height = 35,
103         x = windowWidth * 0.1,
104         y = 95,
105     )
106
107     Label(
108         text = "Путь до директории с результатами",
109         font = ("Arial", 16),
110         bg = PURPLE_LIGHT,
111         fg = PURPLE_SUPER_DARK,
112     ).place(
113         width = windowWidth * 0.38,
114         height = 35,
115         x = windowWidth * 0.1,
116         y = 140,
117     )
118
119     Button(
120         highlightbackground = PURPLE_DARK,
```

Продолжение листинга А.5

```
121         highlightthickness = 30,
122         fg = PURPLE_LIGHT,
123         state = DISABLED,
124     ).place(
125         width = windowWidth * 0.38,
126         height = 35,
127         x = windowWidth * 0.52,
128         y = 140,
129     )
130     Button(
131         text = "Выбрать директорию для результатов",
132         font = ("Arial", 16),
133         fg = PURPLE_SUPER_DARK,
134         highlightbackground = PURPLE,
135         highlightthickness = 30,
136         command = lambda: self.setOutputDirectoryEntry(),
137     ).place(
138         width = windowWidth * 0.38 - 4,
139         height = 31,
140         x = windowWidth * 0.52 + 2,
141         y = 142,
142     )
143
144     self.outputDirectoryEntry = Entry(
145         font = ("Arial", 14),
146         bg = "white",
147         fg = PURPLE_SUPER_DARK,
148         highlightbackground = PURPLE_DARK,
149     )
150     self.outputDirectoryEntry.place(
151         width = windowWidth * 0.8,
152         height = 35,
153         x = windowWidth * 0.1,
154         y = 185,
155     )
156
157     self.methodVar = IntVar()
158
159     Radiobutton(
160         text = "Разработанный гибридный метод",
161         variable = self.methodVar, value = CompressionMethods.HYBRID.value,
162         font = ("Arial", 16),
163         bg = PURPLE_LIGHT,
164         fg = PURPLE_SUPER_DARK,
165         anchor = "w",
```

Продолжение листинга А.5

```
166         ).place(
167             width = windowWidth * 0.35,
168             height = 30,
169             x = windowWidth * 0.1,
170             y = 230,
171         )
172     Radiobutton(
173         text = "Метод Хаффмана",
174         variable = self.methodVar, value = CompressionMethods.HUFFMAN.value,
175         font = ("Arial", 16),
176         bg = PURPLE_LIGHT,
177         fg = PURPLE_SUPER_DARK,
178         anchor = "w",
179     ).place(
180         width = windowWidth * 0.25,
181         height = 30,
182         x = windowWidth * 0.45,
183         y = 230,
184     )
185     Radiobutton(
186         text = "Метод LZW",
187         variable = self.methodVar, value = CompressionMethods.LZW.value,
188         font = ("Arial", 16),
189         bg = PURPLE_LIGHT,
190         fg = PURPLE_SUPER_DARK,
191         anchor = "w",
192     ).place(
193         width = windowWidth * 0.2,
194         height = 30,
195         x = windowWidth * 0.7,
196         y = 230,
197     )
198
199     Label(
200         text = "ЭТАПЫ СЖАТИЯ И РАСПАКОВКИ ИЗОБРАЖЕНИЙ",
201         font = ("Arial", 16, "bold"),
202         bg = PURPLE_DARK,
203         fg = "white",
204     ).place(
205         width = windowWidth,
206         height = 30,
207         x = 0,
208         y = 270,
209     )
210     self.textEditor = Text(
```

Продолжение листинга А.5

```
211         font = ("Arial", 16),
212         bg = "white",
213         fg = PURPLE_SUPER_DARK,
214         highlightbackground = "white",
215     )
216     self.textEditor.place(
217         width = windowWidth,
218         height = windowHeight - 110 - 310,
219         x = 0,
220         y = 310,
221     )
222     self.textEditor.tag_configure("bold", font=("Arial", 16, "bold"))
223     self.textEditor.tag_configure("center", justify="center")
224
225     progressbarLabel = Label(
226         bg = "white",
227     )
228     progressbarLabel.place(
229         width = windowWidth,
230         x = 0,
231         y = windowHeight - 110,
232     )
233     self.progressbar = Progressbar(
234         progressbarLabel,
235         orient="horizontal",
236         length=windowWidth,
237         maximum=100,
238     )
239     self.progressbar.pack()
240     self.progressbar.step(0)
241
242     Label(
243         text = "ВОЗМОЖНЫЕ ДЕЙСТВИЯ",
244         font = ("Arial", 16, "bold"), bg = PURPLE_DARK, fg = "white",
245     ).place(
246         width = windowWidth,
247         height = 30,
248         x = 0 ,
249         y = windowHeight - 85,
250     )
251
252     Button(
253         highlightbackground = PURPLE_DARK,
254         highlightthickness = 30,
255         fg = PURPLE_LIGHT,
```


Продолжение листинга А.5

```
256         state = DISABLED,
257     ).place(
258         width = windowWidth * 0.24,
259         height = 35,
260         x = windowWidth * 0.1,
261         y = windowHeight - 45,
262     )
263     Button(
264         text = "Сжать и распаковать",
265         font = ("Arial", 16),
266         fg = PURPLE_SUPER_DARK,
267         highlightbackground = PURPLE,
268         highlightthickness = 30,
269         command = lambda: self.startEncryption(),
270     ).place(
271         width = windowWidth * 0.24 - 4,
272         height = 31,
273         x = windowWidth * 0.1 + 2,
274         y = windowHeight - 43,
275     )
276
277     Button(
278         highlightbackground = PURPLE_DARK,
279         highlightthickness = 30,
280         fg = PURPLE_LIGHT,
281         state = DISABLED,
282     ).place(
283         width = windowWidth * 0.24,
284         height = 35,
285         x = windowWidth * 0.38,
286         y = windowHeight - 45,
287     )
288     Button(
289         text = "Сравнить методы",
290         font = ("Arial", 16),
291         fg = PURPLE_SUPER_DARK,
292         highlightbackground = PURPLE,
293         highlightthickness = 30,
294         command = self.compareCompressionMethods
295     ).place(
296         width = windowWidth * 0.24 - 4,
297         height = 31,
298         x = windowWidth * 0.38 + 2,
299         y = windowHeight - 43,
300     )
```

Продолжение листинга А.5

```
301
302         Button(
303             highlightbackground = PURPLE_DARK,
304             highlightthickness = 30,
305             fg = PURPLE_LIGHT,
306             state = DISABLED,
307         ).place(
308             width = windowWidth * 0.24,
309             height = 35,
310             x = windowWidth * 0.66,
311             y = windowHeight - 45,
312         )
313         Button(
314             text = "0 nporпамме",
315             font = ("Arial", 16),
316             fg = PURPLE_SUPER_DARK,
317             highlightbackground = PURPLE,
318             highlightthickness = 30,
319             command = lambda: self.aboutProgram(),
320         ).place(
321             width = windowWidth * 0.24 - 4,
322             height = 31,
323             x = windowWidth * 0.66 + 2,
324             y = windowHeight - 43,
325         )
326
327     def setInputFilenameEntry(self) -> None:
328         filepath = filedialog.askopenfilename()
329         if filepath != "":
330             self.inputFilenameEntry.delete(0, END)
331             self.inputFilenameEntry.insert(0, filepath)
332
333     def setOutputDirectoryEntry(self) -> None:
334         filepath = filedialog.askdirectory()
335         if filepath != "":
336             self.outputDirectoryEntry.delete(0, END)
337             self.outputDirectoryEntry.insert(0, filepath)
338
339     def startEncryption(self) -> None:
340         method = CompressionMethods(self.methodVar.get())
341
342         inputFile = self.getInputFile()
343         if inputFile is None:
344             return
345
```

Продолжение листинга А.5

```
346         outputDirectory = self.getOutputDirectory()
347         if outputDirectory is None:
348             return
349
350         compressor = Compression(
351             method=method,
352             text_editor=self.textEditor,
353             progressbar=self.progressbar,
354         )
355         compressor.compress(
356             inputFile,
357             outputDirectory + "/compressed.bin",
358         )
359         compressor.decompress(
360             outputDirectory + "/compressed.bin",
361             outputDirectory + "/decompressed.bmp",
362         )
363
364     def getOutputDirectory(self) -> str | None:
365         path = self.outputDirectoryEntry.get()
366         if not os.path.exists(path) or not os.path.isdir(path):
367             messagebox.showwarning(
368                 "Ошибка",
369                 "Директории с таким именем не существует!"
370             )
371             return
372
373         return path
374
375     def getInputFile(self) -> str | None:
376         path = self.inputFilenameEntry.get()
377         if not os.path.exists(path) or not os.path.isfile(path):
378             messagebox.showwarning(
379                 "Ошибка",
380                 "Файла с таким именем не существует!"
381             )
382             return
383
384         return path
385
386     def compareCompressionMethods(self) -> None:
387         plot_comparison_graph(
388             image_paths=IMAGE_PATHS,
389             compression_rates=COMPRESSION_RATES,
390             title="Сравнение методов по степени сжатия изображений",
```

Продолжение листинга А.5

```
391         y_label="Степень сжатия (%)",
392         x_label="Изображения (названия)",
393         y_lim=105,
394     )
395     plot_comparison_graph(
396         image_paths=IMAGE_PATHS,
397         compression_rates=INFORMATION_TO_DECOMPRESS,
398         title="Сравнение методов по размеру данных для распаковки
↵ изображений",
399         y_label="Кол-во информации для распаковки\низображения в сжатом файле
↵ (%)",
400         x_label="Изображения (названия)",
401         y_lim=44,
402     )
403
404     def aboutProgram(self):
405         messagebox.showinfo(
406             "О программе",
407             "Реализация метода сжатия статических изображений "\
408             "без потерь на основе алгоритма Хаффмана\n\n"\
409             "Ковалец Кирилл ИУ7-42М (2025)"
410         )
411
412     def run(self):
413         self.methodVar.set(CompressionMethods.HYBRID.value)
414
415         self.inputFilenameEntry.insert(0,
↵ "/Users/kirill/Documents/bmstu/magistracy_diploma/input_data/heart
↵ .bmp")
416         self.outputDirectoryEntry.insert(0,
↵ "/Users/kirill/Documents/bmstu/magistracy_diploma/output_data")
417
418         self.window.mainloop()
```

ПРИЛОЖЕНИЕ Б

Реализация сравнения методов сжатия статических изображений без потерь

Листинг Б.1 — Модуль для сравнения методов сжатия

```
1  from PIL import Image
2  import matplotlib.pyplot as plt
3  import matplotlib.offsetbox as offsetbox
4
5  from constants import CompressionMethods
6
7
8  def plot_comparison_graph(
9      image_paths: list[str],
10     compression_rates: dict[CompressionMethods, list[float]],
11     title: str,
12     y_label: str,
13     x_label: str,
14     y_lim: int,
15 ) -> None:
16     fig, ax = plt.subplots(figsize=(11, 6))
17     fig.canvas.manager.set_window_title("Выпускная квалификационная работа
18     ↳ (Ковалец Кирилл ИУ7-42М)")
19     plt.subplots_adjust(bottom=0.3)
20     plt.title(title)
21
22     x_positions = range(len(image_paths))
23     bar_width = 0.2
24
25     ax.bar(
26         [x - bar_width for x in x_positions],
27         compression_rates[CompressionMethods.LZW],
28         width=bar_width,
29         label='Метод LZW'
30     )
31     ax.bar(
32         x_positions,
33         compression_rates[CompressionMethods.HYBRID],
34         width=bar_width,
35         label='Гибридный метод'
36     )
37     ax.bar(
38         [x + bar_width for x in x_positions],
39         compression_rates[CompressionMethods.HUFFMAN],
40         width=bar_width,
```

Продолжение листинга Б.1

```
40     label='Метод Хаффмана'
41 )
42
43 ax.grid(True, axis='y', alpha=0.6, linestyle='--')
44 ax.legend()
45 ax.set_ylabel(y_label)
46 ax.set_xlabel(x_label)
47
48 ax.set_xticks(x_positions)
49 ax.set_xticklabels([img.split('/')[ -1] for img in image_paths])
50
51 ax.set_ylim(0, y_lim)
52
53 for i, image_path in enumerate(image_paths):
54     img = Image.open(image_path)
55     img.thumbnail((300, 300))
56     imagebox = offsetbox.AnnotationBbox(
57         offsetbox=offsetbox.OffsetImage(img, zoom=0.3),
58         xy=(i, 0),
59         xybox=(i, -32 * y_lim / 100),
60         frameon=False,
61     )
62     ax.add_artist(imagebox)
63
64 plt.show()
65
66
67 def plot_comparison_bar_chart(
68     image_sizes: list[int],
69     data_to_decompress_sises: list[int],
70     method: str,
71 ) -> None:
72     fig, ax = plt.subplots(figsize=(5.5, 6))
73     fig.canvas.manager.set_window_title("Сравнение значений")
74     plt.title("Диаграмма сравнения размеров сжатого\n изображения с исходным")
75
76     labels = ["Исходное изображение", f"Сжатое изображение\n ({method})"]
77
78     ax.bar(
79         labels,
80         image_sizes,
81         label='Размер информации об изображении',
82     )
83     ax.bar(
84         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