

МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ
БЕЛОРУССКИЙ НАЦИОНАЛЬНЫЙ ТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ

Факультет информационных технологий и робототехники
Кафедра программного обеспечения информационных систем и технологий

КУРСОВАЯ РАБОТА

по дисциплине «Структуры и алгоритмы обработки данных»

на тему «**Приложение, реализующее алгоритм Хаффмана сжатия информации
с графической иллюстрацией построения дерева Хаффмана**»

Выполнил:
студент группы 10702122

Солдатов Н. В.

Руководитель:
преподаватель

Воронич Л. В.

Минск 2023

СОДЕРЖАНИЕ

| | |
|---|----|
| ВВЕДЕНИЕ..... | 4 |
| СЖАТИЕ ИНФОРМАЦИИ С ПРИМЕНЕНИЕМ АЛГОРИТМА ХАФФМАНА..... | 6 |
| 1.1. Алгоритм Хаффмана..... | 6 |
| 1.2. Принципы сжатия данных..... | 8 |
| 1.3. Коэффициент сжатия..... | 9 |
| 1.4. Допустимость потерь..... | 9 |
| ОПИСАНИЕ ПРОГРАММЫ..... | 10 |
| ТЕСТИРОВАНИЕ ПРОГРАММЫ..... | 12 |
| ЗАКЛЮЧЕНИЕ..... | 16 |
| СПИСОК ЛИТЕРАТУРЫ..... | 17 |
| ПРИЛОЖЕНИЕ А..... | 18 |
| ПРИЛОЖЕНИЕ Б..... | 28 |
| ПРИЛОЖЕНИЕ В..... | 32 |
| ПРИЛОЖЕНИЕ Г..... | 33 |
| ПРИЛОЖЕНИЕ Д..... | 36 |
| ПРИЛОЖЕНИЕ Е..... | 38 |
| ПРИЛОЖЕНИЕ Ж..... | 40 |

ВВЕДЕНИЕ

Программирование — процесс создания компьютерных программ. В более широком смысле под программированием понимают весь спектр деятельности, связанный с созданием и поддержанием в рабочем состоянии программ — программного обеспечения. В наше время программирование очень популярно, а значит владеть этим навыком очень полезно. Это открывает для человека много возможностей для последующей рабочей деятельности. Программирование основывается на специальных алгоритмах, которые выполняет компьютер для решения определенной задачи. Одна из важных задач программирования – *сжатие информации*.

Целью курсовой работы является: закрепление и углубление знаний, полученных при изучении курса «Структуры и алгоритмы обработки данных» посредством изучения основных алгоритмов сжатия информации, развитие практических навыков работы со структурами данных и приложениями на их основе.

Для курсовой работы по реализации алгоритма Хаффмана с графической иллюстрацией построения дерева Хаффмана необходимо выполнить следующие задачи. В первую очередь, провести обзор литературы, изучив существующие методы сжатия данных, основные принципы работы алгоритма Хаффмана и примеры его применения в различных областях. На основе полученных знаний сформулировать функциональные и нефункциональные требования к разрабатываемому приложению. Затем определить язык программирования и технологии, которые будут использованы в процессе реализации. Следующим шагом является написание кода для построения кодового дерева Хаффмана и реализации механизма сжатия и декомпрессии данных. Важным аспектом работы будет создание графического интерфейса для визуализации процесса построения дерева Хаффмана и отображения кодов символов. После этого необходимо интегрировать механизм сжатия в приложение, предоставив пользователю возможность выбора файла для сжатия. Провести тестирование приложения на различных типах файлов и произвести отладку, выявив и устранив возможные ошибки. Далее, оценить производительность алгоритма Хаффмана, проанализировать время его работы на различных объемах данных, и сравнить полученные результаты с другими методами сжатия.

Актуальность разработки приложения, реализующего алгоритм Хаффмана для сжатия информации с графической иллюстрацией построения дерева, обусловлена рядом факторов. Во-первых, современное информационное общество сталкивается с постоянным ростом объема данных, передаваемых и хранимых в различных сферах, таких как медицина, телекоммуникации, интернет-технологии, анализ данных и другие. Применение эффективных методов сжатия становится ключевым элементом оптимизации передачи и хранения данных. Алгоритм Хаффмана, как один из наиболее распространенных и эффективных методов сжатия, сохраняет актуальность в данном контексте. Графическая иллюстрация построения

дерева Хаффмана в разрабатываемом приложении добавляет важный элемент визуализации, облегчая понимание процесса работы алгоритма. Это может быть полезным для студентов, обучающихся основам сжатия данных, а также для специалистов, применяющих сжатие в своей работе и желающих более наглядно понять принципы Хаффмановского кодирования. Таким образом, разработка подобного приложения не только соответствует современным требованиям по оптимизации работы с данными, но и может быть полезной в образовательных целях, способствуя более глубокому пониманию алгоритма Хаффмана.

1. СЖАТИЕ ИНФОРМАЦИИ С ПРИМЕНЕНИЕМ АЛГОРИТМА ХАФФМАНА.

1.1 Алгоритм Хаффмана

Алгоритм оптимального префиксного кодирования алфавита с минимальной избыточностью. Был разработан в 1952 году аспирантом Массачусетского технологического института Дэвидом Хаффманом при написании им курсовой работы. В настоящее время используется во многих программах сжатия данных.

Этот алгоритм кодирования информации был предложен Д.А. Хаффманом в 1952 году. Хаффмановское кодирование (сжатие) – это широко используемый метод сжатия, присваивающий символам алфавита коды переменной длины, основываясь на вероятностях появления этих символов.

Идея алгоритма состоит в следующем: зная вероятности вхождения символов в исходный текст, можно описать процедуру построения кодов переменной длины, состоящих из целого количества битов. Символам с большей вероятностью присваиваются более короткие коды. Таким образом, в этом методе при сжатии данных каждому символу присваивается оптимальный префиксный код, основанный на вероятности его появления в тексте.

Префиксный код – это код, в котором никакое кодовое слово не является префиксом любого другого кодового слова. Эти коды имеют переменную длину.

Оптимальный префиксный код – это префиксный код, имеющий минимальную среднюю длину. Алгоритм Хаффмана можно разделить на два этапа:

1. Определение вероятности появления символов в исходном тексте.

Первоначально необходимо прочитать исходный текст полностью и подсчитать вероятности (точнее частоты) появления символов в нем (иногда подсчитывают, сколько раз встречается каждый символ). Если при этом учитываются все 256 символов, то не будет разницы в сжатии текстового или файла иного формата.

2. Нахождение оптимального префиксного кода.

Далее находятся два символа *a* и *b* с наименьшими вероятностями появления и заменяются одним фиктивным символом *x*, который имеет вероятность появления, равную сумме вероятностей появления символов *a* и *b*. Затем, используя эту процедуру рекурсивно, находится оптимальный префиксный код для меньшего множества символов (где символы *a* и *b* заменены одним символом *x*). Код для исходного множества символов получается из кодов замещающих символов путем добавления 0 или 1 перед кодом замещающего символа, и эти два новых кода принимаются как коды заменяемых символов. Например, код символа *a* будет соответствовать коду *x* с добавленным нулем перед этим кодом, а для символа *b* перед кодом символа *x* будет добавлена единица.

Кроме того, метод Хаффмана часто сравнивают с другим широко используемым методом сжатия данных - LZW (Lempel-Ziv-Welch). В отличие от метода Хаффмана, который основан на предсказании вероятности символов и присваивании более коротких кодов более вероятным символам, LZW работает на основе поиска повторяющихся последовательностей символов.

Алгоритм LZW создает словарь, содержащий уже встреченные последовательности символов, и заменяет эти последовательности на более короткие коды из словаря. Этот метод эффективен для данных с часто повторяющимися паттернами, так как он строит словарь в процессе сжатия и использует его для замены повторяющихся участков данных.

Хотя оба метода применяются для сжатия данных, они имеют разные подходы и характеристики. Метод Хаффмана обычно эффективен для данных с разной вероятностью появления символов, в то время как LZW хорошо работает с данными, содержащими повторяющиеся участки или паттерны.

Таким образом, выбор между методом Хаффмана и LZW зависит от характера данных и требований к сжатию. Хаффман эффективен для данных с различной вероятностью появления символов, в то время как LZW может быть более эффективным для данных с повторяющимися участками.

Коды Хаффмана имеют уникальный префикс, что и позволяет однозначно их декодировать, несмотря на их переменную длину. Алгоритм Хаффмана универсальный, его можно применять для сжатия данных любых типов, но он малоэффективен для файлов маленьких размеров (за счет необходимости сохранения словаря). В настоящее время данный метод практически не применяется в чистом виде, обычно используется как один из этапов сжатия в более сложных схемах. Это единственный алгоритм, который не увеличивает размер исходных данных в худшем случае (если не считать необходимости хранить таблицу перекодировки вместе с файлом).

Пример сжатия методом Хаффмана представлен на рисунке 1.

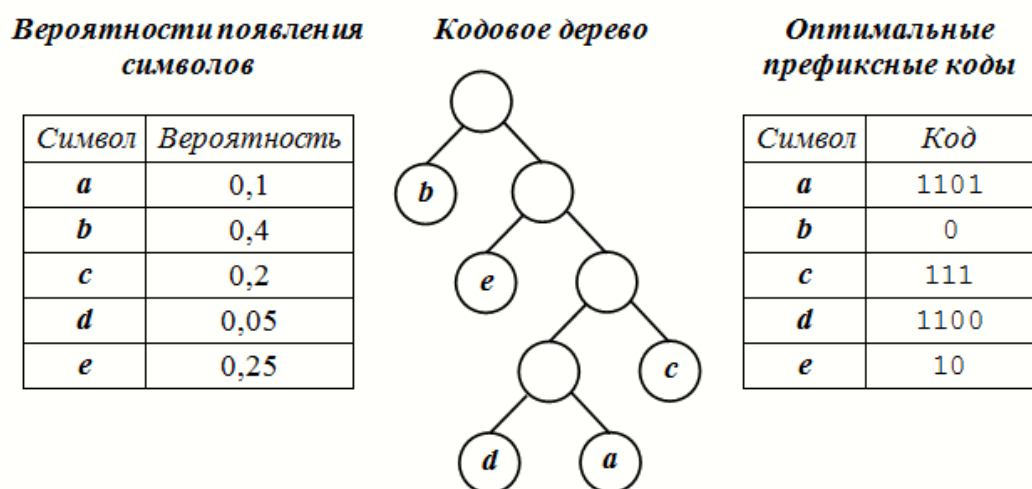


Рисунок 1 – Пример сжатия методом Хаффмана

1.2 Принципы сжатия данных

В основе любого способа сжатия лежит модель источника данных, или, точнее, модель избыточности. Иными словами, для сжатия данных используются некоторые априорные сведения о том, какого рода данные сжимаются. Не обладая такими сведениями об источнике, невозможно сделать никаких предположений о преобразовании, которое позволило бы уменьшить объём сообщения. Модель избыточности может быть статической, неизменной для всего сжимаемого сообщения, либо строиться или параметризоваться на этапе сжатия (и восстановления). Методы, позволяющие на основе входных данных изменять модель избыточности информации, называются адаптивными. Неадаптивными являются обычно узкоспециализированные алгоритмы, применяемые для работы с данными, обладающими хорошо определёнными и неизменными характеристиками. Подавляющая часть достаточно универсальных алгоритмов является в той или иной мере адаптивной.

Все методы сжатия данных делятся на два основных класса:

- *Сжатие без потерь*
- *Сжатие с потерями*

Сжатие данных – это процесс, обеспечивающий уменьшение объема данных путем сокращения их избыточности. Сжатие данных связано с компактным расположением порций данных стандартного размера. Сжатие данных можно разделить на два основных типа:

Сжатие без потерь (полностью обратимое) – это метод сжатия данных, при котором ранее закодированная порция данных восстанавливается после их распаковки полностью без внесения изменений. Для каждого типа данных, как правило, существуют свои оптимальные алгоритмы сжатия без потерь.

Сжатие с потерями – это метод сжатия данных, при котором для обеспечения максимальной степени сжатия исходного массива данных часть содержащихся в нем данных отбрасывается.

Для текстовых, числовых и табличных данных использование программ, реализующих подобные методы сжатия, является неприемлемыми. В основном такие алгоритмы применяются для сжатия аудио- и видеоданных, статических изображений. Алгоритм сжатия данных (алгоритм архивации) – это алгоритм, который устраняет избыточность записи данных.

При использовании сжатия без потерь возможно полное восстановление исходных данных, сжатие с потерями позволяет восстановить данные с искажениями, обычно несущественными с точки зрения дальнейшего использования восстановленных данных. Сжатие без потерь обычно используется для передачи и хранения текстовых данных, компьютерных программ.

1.3 Коэффициент сжатия

Коэффициент сжатия — основная характеристика алгоритма сжатия. Она определяется как отношение объёма исходных несжатых данных к объёму сжатых данных.

Коэффициент сжатия = размер входного потока/размер выходного потока

Значения больше 1 обозначают сжатие, а значения меньше 1 — расширение.

Так же существует такое понятие как *отношение сжатия*.

Отношение сжатия — одна из наиболее часто используемых величин для обозначения эффективности метода сжатия.

Отношение сжатия = размер выходного потока/размер входного потока

Значение 0,6 означает, что данные занимают 60% от первоначального объёма. Значения больше 1 означают, что выходной *поток* больше входного (отрицательное сжатие, или расширение).

Следует отметить:

- если $k = 1$, то алгоритм не производит сжатия, то есть выходное сообщение оказывается по объёму равным входному
- если $k < 1$, то алгоритм порождает сообщение большего размера, нежели несжатое, то есть, совершает «вредную» работу.

Ситуация с $k < 1$ вполне возможна при сжатии. Принципиально невозможно получить алгоритм сжатия без потерь, который при любых данных образовывал бы на выходе данные меньшей или равной длины.

1.4 Допустимость потерь

Основным критерием различия между алгоритмами сжатия является описанное выше наличие или отсутствие потерь. В общем случае алгоритмы сжатия без потерь универсальны в том смысле, что их применение безусловно возможно для данных любого типа, в то время как возможность применения сжатия с потерями должна быть обоснована. Для некоторых типов данных искажения не допустимы в принципе.

2. ОПИСАНИЕ ПРОГРАММЫ

В качестве быстрой и удобной среды для разработки приложения была выбрана **Visual Studio 2022**. Microsoft Visual Studio — линейка продуктов компании Microsoft, включающих интегрированную среду разработки программного обеспечения и ряд других инструментальных средств. Данные продукты позволяют разрабатывать как консольные приложения, так и игры и приложения с графическим интерфейсом.

В Visual Studio Вы можете использовать следующие технологии и языки программирования: .NET, Node.js, C, C#, C++, Python, Visual Basic, JavaScript. В качестве платформы выбран Windows Forms, наиболее удобная и часто используемая платформа для графических приложений. На данной платформе существует множество компонентов, при помощи которых легко осуществляется вывод информации и работа с пользователем. Соответственно, язык программирования – **C#**, был выбран для написания курсовой работы, как наиболее подходящий.

Для демонстрации реализации метода Хаффмана сжатия информации с помощью бинарного дерева была реализована программа, в которой пользователь может визуально ознакомиться с возможностями сжатия информации на основе метода Хаффмана. Пользователю предлагается ввести строку, на основе которой выстраиваются данные двух таблиц: таблица частот (вероятностей появления символов) и таблица оптимальных префиксных кодов (вкладка кодирование). Таблица оптимальных префиксных кодов составляется на основе дерева Хаффмана. соответственно дерево Хаффмана основывается на частоте символов исходной строки:

Алгоритм построения дерева Хаффмана.

Шаг 1. Символы входного алфавита образуют список свободных узлов. Каждый лист имеет вес, который может быть равен либо вероятности, либо количеству вхождений символа в сжимаемый текст.

Шаг 2. Выбираются два свободных узла дерева с наименьшими весами.

Шаг 3. Создается их родитель с весом, равным их суммарному весу.

Шаг 4. Родитель добавляется в список свободных узлов, а двое его детей удаляются из этого списка.

Шаг 5. Одной дуге, выходящей из родителя, ставится в соответствие бит 1, другой — бит 0.

Шаг 6. Повторяем шаги, начиная со второго, до тех пор, пока в списке свободных узлов не останется только один свободный узел. Он и будет считаться корнем дерева.

Существует два подхода к построению кодового дерева: от корня к листьям и от листьев к корню.

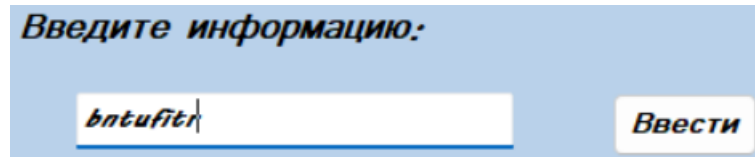
После построения дерева Хаффмана, получаем вторую таблицу с оптимальными префиксными кодами. Префиксные коды получаются при проходе от корня дерева к его листьям, присваивая всем левым веткам 0 и всем правым 1 соответственно.

Классический алгоритм Хаффмана имеет один существенный недостаток. Для восстановления содержимого сжатого текста при *декодировании* необходимо знать таблицу частот, которую использовали при кодировании. Следовательно, *длина* сжатого текста увеличивается на длину таблицы частот, которая должна посылаться впереди данных, что может свести на нет все усилия по сжатию данных. Кроме того, необходимость наличия полной частотной статистики перед началом собственно кодирования требует двух проходов по тексту: одного для построения модели текста (таблицы частот и дерева Хаффмана), другого для собственно кодирования.

Для осуществления *декодирования* необходимо иметь кодовое *дерево*, которое приходится хранить вместе со сжатыми данными. Это приводит к некоторому незначительному увеличению объема сжатых данных. Используются самые различные форматы, в которых хранят это *дерево*. Обратим внимание на то, что узлы кодового дерева являются пустыми. Иногда хранят не само *дерево*, а исходные данные для его формирования, то есть сведения о вероятностях появления символов или их количествах. При этом процесс *декодирования* предваряется построением нового кодового дерева, которое будет таким же, как и при кодировании.

3. ТЕСТИРОВАНИЕ ПРОГРАММЫ

При запуске программы пользователь должен ввести исходную строку для сжатия в элемент ввода информации. Пример ввода информации представлен на рисунке 2.



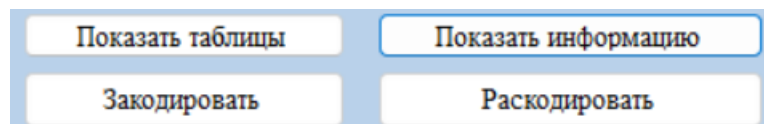
Введите информацию:

bntufitr

Ввести

Рисунок 2 – Пример введенной информации

После ввода информации можем нажать кнопку “Ввести” для построения дерева Хаффмана и заполнения таблиц вероятностей появления символов, которая получается в результате конвертирования исходной строки и записи в словарь, просмотр осуществляется при нажатии клавиши “Показать таблицы”. Пример кнопок представлен на рисунке 3. Таблица вероятности появления символов представлена на рисунке 4.



Показать таблицы

Показать информацию

Закодировать

Раскодировать

Рисунок 3 – Отображение кнопок.



| | Key | Value |
|---|-----|-------|
| ▶ | b | 1 |
| | n | 1 |
| | t | 2 |
| | u | 1 |
| | f | 1 |
| | i | 1 |

Рисунок 4 – Получение вероятностей появления символов из исходной строке

После данного действия программа строит дерево Хаффмана, листьями которого должны быть символы и их частоты исходной строки. Пример построения дерева Хаффмана представлен на рисунке 5.

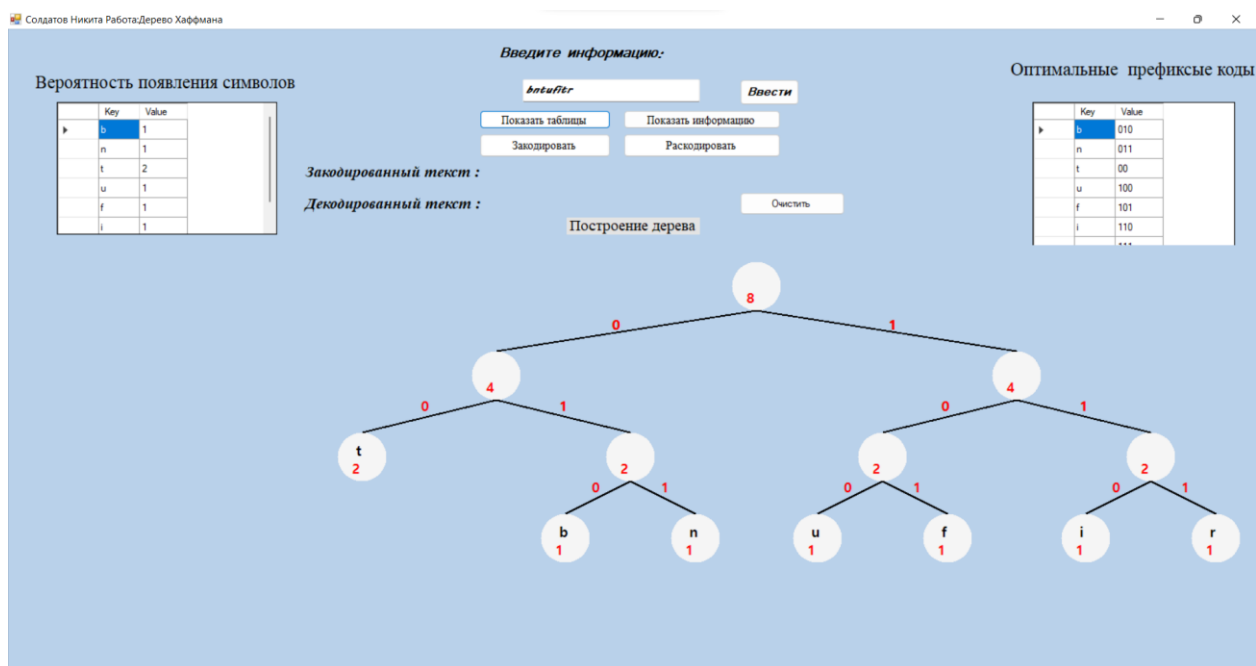


Рисунок 5 – Построение дерева Хаффмана на основе вероятностей появления символов из исходной строке

Далее при построении дерева приложение высчитывает префиксные коды каждого из символов, и записывает в таблица оптимальных префиксных кодов. Пример таблицы оптимальные префиксные коды представлени на рисунке 5.

Оптимальные префиксные коды

| | Key | Value |
|---|-----|-------|
| ▶ | b | 010 |
| | n | 011 |
| | t | 00 |
| | u | 100 |
| | f | 101 |
| | i | 110 |
| | | 111 |

Рисунок 6 – Отображение таблицы оптимальных префиксных кодов на основе дерева Хаффмана

После данных действий информацию можно закодировать и раскодировать при нажатии на кнопки “Закодировать” и “Раскодировать” соответственно. Пример закодированной и декодированной информации показан на рисунке 7.

Закодированный текст : 0100110010010111000111

Декодированный текст : bntufitr

Рисунок 7 – Отображение закодированного и декодированного текста

Также при нажатии на кнопку “Показать информацию” получим статистику о сжатой информации. Пример информации о файле показан на рисунке 7. Пример полной работы программы представлен на рисунке 8.

| | |
|-----------------------------------|----------------|
| Размер файла до сжатия: | 64 |
| Размер файла после сжатия: | 22 |
| Количество символов: | 8 |
| Коэффициент сжатия: | 2,90909 |

Рисунок 8 – Отображение статистики сжатой информации

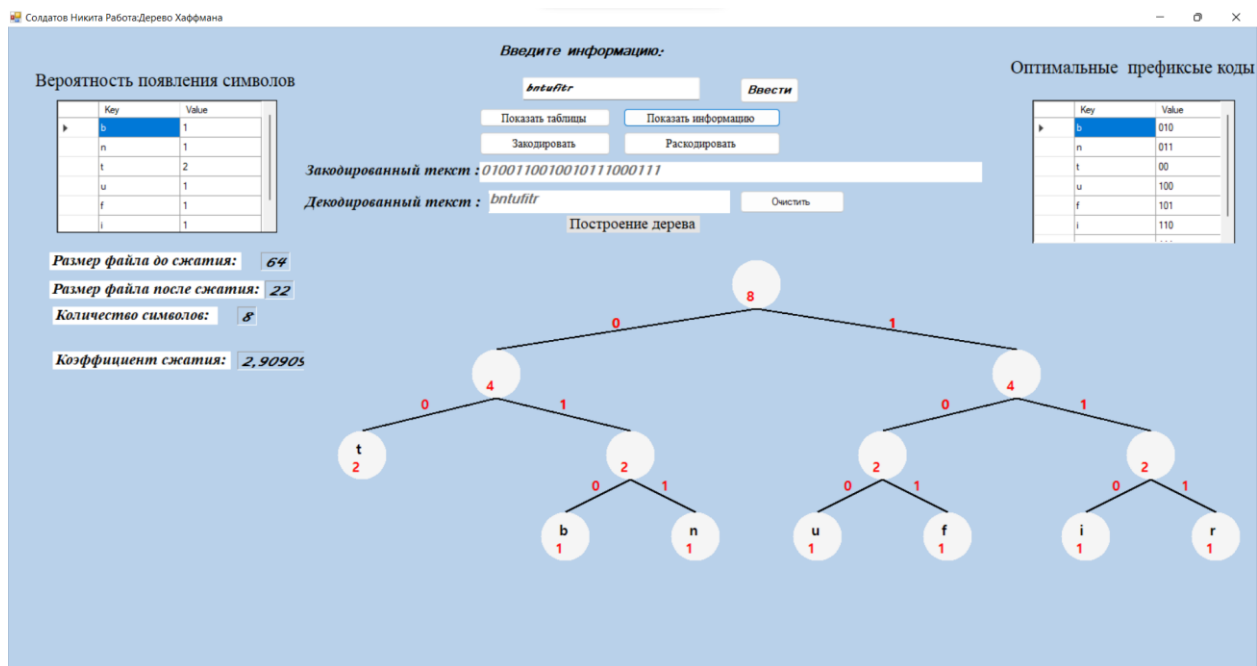


Рисунок 9 – Отображение полной формы при завершении работы приложения

4. ЗАКЛЮЧЕНИЕ

В ходе выполнения курсовой работы было выполнено глубокое понимание и изучения алгоритма Хаффмана, который является одним из наиболее важных и широко применяемых методов сжатия данных. Для успешной реализации этой задачи были использованы разнообразные алгоритмы, начиная от методов анализа частоты символов в исходных данных, заканчивая алгоритмами построения оптимальных префиксных кодов.

Особое внимание уделено визуализации процесса построения кодового дерева Хаффмана при помощи библиотеки System.Drawing. Эта визуализация становится неотъемлемой частью обучающего процесса, предоставляя пользователям и студентам наглядное представление о том, как формируются оптимальные коды для символов и как дерево Хаффмана эффективно представляет структуру данных.

Алгоритм Хаффмана остается актуальным в современных условиях, и его применение находит в различных областях, включая передачу данных в сетях, хранение информации, архивацию и т. д. Этот метод сжатия данных обеспечивает отсутствие потерь, что является важным фактором, особенно при работе с чувствительной к качеству информацией.

Однако, следует отметить, что несмотря на эффективность алгоритма Хаффмана, ожидание слишком высокого коэффициента сжатия может быть нереалистичным. Это обстоятельство связано с особенностями данных и тем, что на данный момент нет универсального метода, который бы идеально подходил для всех видов данных.

Использование алгоритма Хаффмана является распространенным в современных архиваторах, и его применение оправдано в контексте обеспечения эффективности сжатия, а также удобства использования. Приложение, разработанное в ходе работы, представляет собой полезный инструмент для обучения и демонстрации принципов работы этого алгоритма.

Приложение, созданное в рамках данной работы, представляет собой ценный инструмент для обучения и наглядной демонстрации работы алгоритма Хаффмана. Оно оказывает помощь как студентам, так и профессионалам в понимании и применении этого важного метода сжатия данных.

5. СПИСОК ЛИТЕРАТУРЫ.

1. "Introduction to Algorithms" by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein.
2. "Algorithms" by Robert Sedgewick and Kevin Wayne.
3. "C# 9.0 in a Nutshell" by Joseph Albahari and Ben Albahari.
4. Online Resources (Ресурсы в Интернете):
 - Жадные алгоритмы на GeeksforGeeks.
 - Хаффманово кодирование на GeeksforGeeks.
5. WinForms и C# — полное руководство (электронный ресурс). — URL: <https://metanit.com/sharp/winforms/> (дата обращения: 21.10.2023).

ПРИЛОЖЕНИЕ А

Листинг исходного кода Drawing.cs

Этот код реализует рисование дерева Хаффмана с использованием графической библиотеки Windows Forms. Он отвечает за визуализацию структуры дерева, отображая узлы и линии связей между ними.

Программа создает объект класса «Drawing», который содержит методы для отрисовки узлов дерева и связей между ними. Он использует класс «Graphics» для рисования на форме.

Основные методы:

- «DrawCircle»: Рисует круглый узел дерева.
- «DrawFirstNode»: Рисует первый узел дерева.
- «DrawNode»: Рисует узел дерева с определенными символами и их частотами.

- «DrawLeftLine» и «DrawRightLine»: Рисуют линии, соединяющие узлы слева и справа.

- «Draw»: Основной метод, который принимает закодированную последовательность (предположительно, битовую строку) и использует ее для определения пути в дереве Хаффмана для каждого символа. После этого он отрисовывает соответствующую последовательность узлов и связей.

Основная идея - построение дерева Хаффмана по закодированной последовательности и последующая его визуализация на форме, отображая символы и их частоты на узлах дерева.

По всей видимости, программа предназначена для визуализации процесса работы кодирования Хаффмана.

```
using System;
using System.Collections.Generic;
using System.Drawing;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;

namespace Cursac.temerev.model
{
    class Drawing
    {
        private const int radius = 30;
        private string symbols = "";
        private int temp = 1;
        private List<string> listOneSym = new List<string>();

        private readonly Font font = new Font("Malgun Gothic", 13,
FontStyle.Bold);
        private Pen nodePen = new Pen(Color.Black, 3);
        private Pen linePen = new Pen(Color.Black, 2);

        private Point center = new Point(550, 50);
        private Point dynamicPoint = new Point(550, 50);

        public void DrawCircle(Graphics g, Pen pen, float centerX, float centerY,
float radius)
```

```

        {
            g.FillEllipse(Brushes.Black, centerX - radius, centerY - radius,
radius * 2, radius * 2);
            g.FillEllipse(Brushes.WhiteSmoke, centerX - radius, centerY - radius,
radius * 2, radius * 2);
        }

        public void DrawFirstNode(Graphics g, Point center)
        {
            DrawCircle(g, nodePen, center.X, center.Y, radius);
        }

        public void DrawNode(Graphics g, Point center, Node nodes, string
strSymbol)
        {
            DrawCircle(g, nodePen, center.X, center.Y, radius);

            try
            {
                if (temp == 1)
                {
                    if (strSymbol.ToString() == "1")
                    {
                        g.DrawString(nodes.Right.Frequency.ToString(), font,
Brushes.Red, center.X - 15, center.Y + 2);
                        listOneSym.Add("1");
                        temp++;
                    }
                    else if (strSymbol.ToString() == "0")
                    {
                        g.DrawString(nodes.Left.Frequency.ToString(), font,
Brushes.Red, center.X - 15, center.Y + 2);
                        listOneSym.Add("0");
                        temp++;
                    }
                }

                else if (temp == 2)
                {
                    if (listOneSym[0] == "1")
                    {
                        if (strSymbol.ToString() == "1")
                        {
                            g.DrawString(nodes.Right.Right.Frequency.ToString(),
font, Brushes.Red, center.X - 15, center.Y + 2);
                            listOneSym.Add("1");
                            temp++;
                        }

                        else if (strSymbol.ToString() == "0")
                        {
                            g.DrawString(nodes.Right.Left.Frequency.ToString(),
font, Brushes.Red, center.X - 15, center.Y + 2);
                            listOneSym.Add("0");
                            temp++;
                        }
                    }

                    else if (listOneSym[0] == "0")
                    {
                        if (strSymbol.ToString() == "1")
                        {
                            g.DrawString(nodes.Left.Right.Frequency.ToString(),
font, Brushes.Red, center.X - 15, center.Y + 2);
                            listOneSym.Add("1");
                            temp++;
                        }
                    }
                }
            }
        }
    }

```

```

        }
        else if (strSymbol.ToString() == "0")
        {
            g.DrawString(nodes.Left.Left.Frequency.ToString(),
font, Brushes.Red, center.X - 15, center.Y + 2);
            listOneSym.Add("0");
            temp++;
        }
    }

    else if (temp == 3)
    {
        if (listOneSym[0] == "1")
        {
            if (listOneSym[1] == "1")
            {
                if (strSymbol.ToString() == "1")
                {
                    g.DrawString(nodes.Right.Right.Right.Frequency.ToString(), font, Brushes.Red,
center.X - 15, center.Y + 2);
                    listOneSym.Add("1");
                    temp++;
                }

                else if (strSymbol.ToString() == "0")
                {
                    g.DrawString(nodes.Right.Right.Left.Frequency.ToString(), font, Brushes.Red,
center.X - 15, center.Y + 2);
                    listOneSym.Add("0");
                    temp++;
                }
            }

            else if (listOneSym[1] == "0")
            {
                if (strSymbol.ToString() == "1")
                {
                    g.DrawString(nodes.Right.Left.Right.Frequency.ToString(), font, Brushes.Red,
center.X - 15, center.Y + 2);
                    listOneSym.Add("1");
                    temp++;
                }

                else if (strSymbol.ToString() == "0")
                {
                    g.DrawString(nodes.Right.Left.Left.Frequency.ToString(), font, Brushes.Red,
center.X - 15, center.Y + 2);
                    listOneSym.Add("0");
                    temp++;
                }
            }
        }

        else if (listOneSym[0] == "0")
        {
            if (listOneSym[1] == "1")
            {
                if (strSymbol.ToString() == "1")
                {

```

```

g.DrawString(nodes.Left.Right.Right.Frequency.ToString(), font, Brushes.Red,
center.X - 15, center.Y + 2);
        listOneSym.Add("1");
        temp++;
    }

    else if (strSymbol.ToString() == "0")
    {

g.DrawString(nodes.Left.Right.Left.Frequency.ToString(), font, Brushes.Red,
center.X - 15, center.Y + 2);
        listOneSym.Add("0");
        temp++;
    }

    }

    else if (listOneSym[1] == "0")
    {
        if (strSymbol.ToString() == "1")
        {

g.DrawString(nodes.Left.Left.Right.Frequency.ToString(), font, Brushes.Red,
center.X - 15, center.Y + 2);
            listOneSym.Add("1");
            temp++;
        }

        else if (strSymbol.ToString() == "0")
        {

g.DrawString(nodes.Left.Left.Left.Frequency.ToString(), font, Brushes.Red,
center.X - 15, center.Y + 2);
            listOneSym.Add("0");
            temp++;
        }
    }

    }

    else if (temp == 4)
    {
        if (listOneSym[0] == "1")
        {
            if (listOneSym[1] == "1")
            {
                if (listOneSym[2] == "1")
                {
                    if (strSymbol.ToString() == "1")
                    {

g.DrawString(nodes.Right.Right.Right.Right.Frequency.ToString(), font,
Brushes.Red, center.X - 15, center.Y + 2);
                        listOneSym.Add("1");
                        temp++;
                    }

                    else if (strSymbol.ToString() == "0")
                    {

g.DrawString(nodes.Right.Right.Right.Left.Frequency.ToString(), font, Brushes.Red,
center.X - 15, center.Y + 2);
                        listOneSym.Add("0");
                        temp++;
                    }
                }
            }
        }
    }

```

```

    }

    else if (listOneSym[2] == "0")
    {
        if (strSymbol.ToString() == "1")
        {
g.DrawString(nodes.Right.Right.Left.Right.Frequency.ToString(), font, Brushes.Red,
center.X - 15, center.Y + 2);

            listOneSym.Add("1");
            temp++;
        }

        else if (strSymbol.ToString() == "0")
        {
g.DrawString(nodes.Right.Right.Left.Left.Frequency.ToString(), font, Brushes.Red,
center.X - 15, center.Y + 2);

            listOneSym.Add("0");
            temp++;
        }
    }

}

else if (listOneSym[1] == "0")
{
    if (listOneSym[2] == "1")
    {
        if (strSymbol.ToString() == "1")
        {
g.DrawString(nodes.Right.Left.Right.Right.Frequency.ToString(), font, Brushes.Red,
center.X - 15, center.Y + 2);

            listOneSym.Add("1");
            temp++;
        }

        else if (strSymbol.ToString() == "0")
        {
g.DrawString(nodes.Right.Left.Right.Left.Frequency.ToString(), font, Brushes.Red,
center.X - 15, center.Y + 2);

            listOneSym.Add("0");
            temp++;
        }
    }

    else if (listOneSym[2] == "0")
    {
        if (strSymbol.ToString() == "1")
        {
g.DrawString(nodes.Right.Left.Left.Right.Frequency.ToString(), font, Brushes.Red,
center.X - 15, center.Y + 2);

            listOneSym.Add("1");
            temp++;
        }

        else if (strSymbol.ToString() == "0")
        {
g.DrawString(nodes.Right.Left.Left.Left.Frequency.ToString(), font, Brushes.Red,
center.X - 15, center.Y + 2);

            listOneSym.Add("0");
            temp++;
        }
    }
}

```

```

        }
    }
}

else if (listOneSym[0] == "0")
{
    if (listOneSym[1] == "1")
    {
        if (listOneSym[2] == "1")
        {
            if (strSymbol.ToString() == "1")
            {
g.DrawString(nodes.Left.Right.Right.Right.Frequency.ToString(), font, Brushes.Red,
center.X - 15, center.Y + 2);

                listOneSym.Add("1");
                temp++;
            }

            else if (strSymbol.ToString() == "0")
            {
g.DrawString(nodes.Left.Right.Right.Left.Frequency.ToString(), font, Brushes.Red,
center.X - 15, center.Y + 2);

                listOneSym.Add("0");
                temp++;
            }
        }

        if (listOneSym[2] == "0")
        {
            if (strSymbol.ToString() == "1")
            {
g.DrawString(nodes.Left.Right.Left.Right.Frequency.ToString(), font, Brushes.Red,
center.X - 15, center.Y + 2);

                listOneSym.Add("1");
                temp++;
            }

            else if (strSymbol.ToString() == "0")
            {
g.DrawString(nodes.Left.Right.Left.Left.Frequency.ToString(), font, Brushes.Red,
center.X - 15, center.Y + 2);

                listOneSym.Add("0");
                temp++;
            }
        }
    }

    else if (listOneSym[1] == "0")
    {
        if (listOneSym[2] == "1")
        {
            if (strSymbol.ToString() == "1")
            {
g.DrawString(nodes.Left.Left.Right.Right.Frequency.ToString(), font, Brushes.Red,
center.X - 15, center.Y + 2);

                listOneSym.Add("1");
                temp++;
            }

            else if (strSymbol.ToString() == "0")
            {

```

```

g.DrawString(nodes.Left.Left.Right.Left.Frequency.ToString(), font, Brushes.Red,
center.X - 15, center.Y + 2);

        listOneSym.Add("0");
        temp++;
    }
}

if (listOneSym[2] == "0")
{
    if (strSymbol.ToString() == "1")
    {

g.DrawString(nodes.Left.Left.Left.Right.Frequency.ToString(), font, Brushes.Red,
center.X - 15, center.Y + 2);

        listOneSym.Add("1");
        temp++;
    }

    else if (strSymbol.ToString() == "0")
    {

g.DrawString(nodes.Left.Left.Left.Left.Frequency.ToString(), font, Brushes.Red,
center.X - 15, center.Y + 2);

        listOneSym.Add("0");
        temp++;
    }
}
}
}
}

catch (NullReferenceException)
{
}
}

public void DrawNode(Graphics g, Point center, char symb)
{
    g.DrawString(symb.ToString(), font, Brushes.Black, center.X + 30,
center.Y - 160);
}

public void DrawLeftLine(Graphics g, Point point, int j)
{
    Point firstPoint = new Point(point.X, point.Y + radius);

    if (j == 0)
    {
        Point secondPoint = new Point(point.X - 320, point.Y + 80);
        g.DrawLine(linePen, firstPoint, secondPoint);
        g.DrawString("0", font, Brushes.Red, secondPoint.X + 140,
secondPoint.Y - 45);
    }

    else if (j == 1)
    {
        Point secondPoint = new Point(point.X - 165, point.Y + 70);
        g.DrawLine(linePen, firstPoint, secondPoint);
        g.DrawString("0", font, Brushes.Red, secondPoint.X + 70,
secondPoint.Y - 45);
    }

    else if (j == 2)

```



```

        {
            Point secondPoint = new Point(point.X - 80, point.Y + 70);
            g.DrawLine(linePen, firstPoint, secondPoint);
            g.DrawString("0", font, Brushes.Red, secondPoint.X + 30,
secondPoint.Y - 45);
        }

        else
        {
            Point secondPoint = new Point(point.X - 40, point.Y + 110);
            g.DrawLine(linePen, firstPoint, secondPoint);
            g.DrawString("0", font, Brushes.Red, secondPoint.X + 12,
secondPoint.Y - 70);
        }
    }

    public void DrawRightLine(Graphics g, Point point, int j)
    {
        Point firstPoint = new Point(point.X, point.Y + radius);

        if (j == 0)
        {
            Point secondPoint = new Point(point.X + 320, point.Y + 80);
            g.DrawLine(linePen, firstPoint, secondPoint);
            g.DrawString("1", font, Brushes.Red, secondPoint.X - 160,
secondPoint.Y - 45);
        }

        else if (j == 1)
        {
            Point secondPoint = new Point(point.X + 165, point.Y + 70);
            g.DrawLine(linePen, firstPoint, secondPoint);
            g.DrawString("1", font, Brushes.Red, secondPoint.X - 90,
secondPoint.Y - 45);
        }

        else if (j == 2)
        {
            Point secondPoint = new Point(point.X + 80, point.Y + 70);
            g.DrawLine(linePen, firstPoint, secondPoint);
            g.DrawString("1", font, Brushes.Red, secondPoint.X - 45,
secondPoint.Y - 45);
        }

        else
        {
            Point secondPoint = new Point(point.X + 40, point.Y + 110);
            g.DrawLine(linePen, firstPoint, secondPoint);
            g.DrawString("1", font, Brushes.Red, secondPoint.X - 25,
secondPoint.Y - 70);
        }
    }

    public void Draw(Graphics g, string[] encode, Node nodes, Dictionary<char,
int> dict)
    {
        DrawFirstNode(g, center);
        g.DrawString(nodes.Frequency.ToString(), font, Brushes.Red, center.X -
15, center.Y + 2);

        for (int i = 0; i < encode.Length; i++)
        {
            string str = encode[i];

            for (int j = 0; j < str.Length; j++)

```

```

        {
            if (str[j] == '1')
            {
                DrawRightLine(g, new Point(center.X, center.Y), j);

                if (j == 0)
                {
                    DrawNode(g, new Point(center.X + 320, center.Y + 110),
nodes, str[j].ToString());
                    center.X += 320;
                    center.Y += 110;
                }

                else if (j == 1)
                {
                    DrawNode(g, new Point(center.X + 165, center.Y + 100),
nodes, str[j].ToString());
                    center.X += 165;
                    center.Y += 100;
                }

                else if (j == 2)
                {
                    DrawNode(g, new Point(center.X + 80, center.Y + 100),
nodes, str[j].ToString());
                    center.X += 80;
                    center.Y += 100;
                }

                else
                {
                    DrawNode(g, new Point(center.X + 40, center.Y + 140),
nodes, str[j].ToString());
                    center.X += 40;
                    center.Y += 140;
                }
            }

            else
            {
                DrawLeftLine(g, new Point(center.X, center.Y), j);

                if (j == 0)
                {
                    DrawNode(g, new Point(center.X - 320, center.Y + 110),
nodes, str[j].ToString());
                    center.X -= 320;
                    center.Y += 110;
                }

                else if (j == 1)
                {
                    DrawNode(g, new Point(center.X - 165, center.Y + 100),
nodes, str[j].ToString());
                    center.X -= 165;
                    center.Y += 100;
                }

                else if (j == 2)
                {
                    DrawNode(g, new Point(center.X - 80, center.Y + 100),
nodes, str[j].ToString());
                    center.X -= 80;
                    center.Y += 100;
                }

                else

```

```

        {
            DrawNode(g, new Point(center.X - 40, center.Y + 140),
nodes, str[j].ToString());
            center.X -= 40;
            center.Y += 140;
        }
    }

    if (j == (str.Length - 1))
    {
        char Symb = dict.ElementAt(i).Key;
        DrawNode(g, new Point(center.X - 40, center.Y + 140),
Symb);
    }
}

center.X = dynamicPoint.X;
center.Y = dynamicPoint.Y;

temp = 1;
    }
}
}

```

ПРИЛОЖЕНИЕ Б

HuffmanTree.cs

Этот код реализует структуру дерева Хаффмана и связанные с ним методы для кодирования и декодирования символов.

Класс «MyHuffmanTree» представляет собой дерево Хаффмана и содержит следующие методы:

- «Root»: Свойство, представляющее корень дерева.
- «nodes»: Список узлов дерева.
- «Frequencies»: Словарь, хранящий символы и их частоты.

Методы:

- «setFrequencies»: Устанавливает частоты символов для дерева Хаффмана.

- «Build»: Строит дерево Хаффмана на основе переданных частот символов.

- «Encode»: Кодировать входную строку с использованием построенного дерева Хаффмана и возвращает массив строк, представляющих закодированные символы.

- «FullCode»: Кодировать полную входную строку в битовое представление с использованием дерева Хаффмана и возвращает «BitArray».

- «Decode»: Декодирует битовую последовательность обратно в строку с использованием дерева Хаффмана.

- «IsLeaf»: Проверяет, является ли узел листом в дереве Хаффмана.

Краткое описание метода «Build»:

1. На основе переданных частот символов создаются узлы дерева «nodes».

2. Пока в списке «nodes» есть более чем один узел:

- Упорядочиваются узлы по частоте.
- Берутся два узла с наименьшей частотой.
- Создается родительский узел, объединяющий эти два узла.
- Удаляются взятые узлы из списка и добавляется новый родительский узел.

3. Устанавливается корень дерева.

Основная идея - построение дерева Хаффмана на основе частот символов и последующее использование этого дерева для кодирования и декодирования данных.

```
using System;
using System.Collections.Generic;
using System.Collections;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows.Forms;
using Cursac.temerev.util;
```

```
namespace Cursac.temerev.model
{
    class MyHuffmanTree
    {
```

```

public Node Root { get; set; }
private List<Node> nodes = new List<Node>();
public Dictionary<char, int> Frequencies;

public MyHuffmanTree()
{

}

public void setFrequencies(Dictionary<char, int> frequencies)
{
    Frequencies = frequencies;
}

public void Build()
{
    foreach (KeyValuePair<char, int> symbol in Frequencies)
    {
        nodes.Add(new Node() { Symbol = symbol.Key, Frequency =
symbol.Value });
    }

    while (nodes.Count > 1)
    {
        List<Node> orderedNodes = nodes.OrderBy(node =>
node.Frequency).ToList<Node>();

        if (orderedNodes.Count >= 2)
        {
            List<Node> taken = orderedNodes.Take(2).ToList<Node>();

            Node parent = new Node()
            {
                Symbols = taken[0].Symbol.ToString() +
taken[1].Symbol.ToString(),
                Frequency = taken[0].Frequency + taken[1].Frequency,
                Left = taken[0],
                Right = taken[1]
            };

            nodes.Remove(taken[0]);
            nodes.Remove(taken[1]);
            nodes.Add(parent);
        }
        Root = nodes.FirstOrDefault();
    }
}

public Node GetRoot()
{
    return Root;
}

public string[] Encode(string source, Dictionary<char, int> dict)
{
    int countt = WorkerWithDictionary.CountSymbols(dict);
    List<bool> encodedSource = new List<bool>();

    string str = "";
    string[] massEachSymbol = new string[countt];

    int j = 0;
    int a = 0;

```

```

        foreach (KeyValuePair<char, int> keyValue in dict)
        {
            List<bool> encodedSymbol = Root.Traverse(keyValue.Key, new
List<bool>());

            while (a != (encodedSymbol.Count))
            {
                str += (encodedSymbol[a] ? "1" : "0");
                a++;
            }

            massEachSymbol[j] = str;
            str = "";
            j++;
            a = 0;

            if (j > dict.Count) { break; };
        }
        return massEachSymbol;
    }

    public BitArray FullCode(string source)
    {
        List<bool> encodedSource = new List<bool>();

        for (int i = 0; i < source.Length; i++)
        {
            List<bool> encodedSymbol = Root.Traverse(source[i], new
List<bool>());
            encodedSource.AddRange(encodedSymbol);
        }

        BitArray bits = new BitArray(encodedSource.ToArray());
        return bits;
    }

    public string Decode(BitArray bits)
    {
        Node current = Root;
        string decoded = "";

        foreach (bool bit in bits)
        {
            if (bit)
            {
                if (current.Right != null)
                {
                    current = current.Right;
                }
            }
            else
            {
                if (current.Left != null)
                {
                    current = current.Left;
                }
            }

            if (IsLeaf(current))
            {
                decoded += current.Symbol;
                current = this.Root;
            }
        }
    }

```

```
        return decoded;
    }

    public bool IsLeaf(Node node)
    {
        return (node.Left == null && node.Right == null);
    }
}
```

ПРИЛОЖЕНИЕ В

MyDictionary.cs

Этот код создает класс «MyDictionary», который представляет собой обертку для двух словарей: одного для хранения символов и их целочисленных значений, а второго для хранения символов и их строковых представлений.

В классе «MyDictionary» есть:

- «dictionary»: Словарь, хранящий символы в качестве ключей и целочисленные значения в качестве соответствующих значений.

- «dictionarySecond»: Второй словарь, где символы хранятся в качестве ключей, а строки в качестве соответствующих значений.

Методы:

- «setDictionary»: Устанавливает словарь «dictionary» с помощью переданного словаря.

- «getDictionary»: Возвращает словарь «dictionary».

- «getDictionarySecond»: Возвращает словарь «dictionarySecond».

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace Cursac.temerev.model
{
    class MyDictionary
    {
        private static Dictionary<char, int> dictionary;
        private static Dictionary<char, string> dictionarySecond;

        public MyDictionary()
        {
            dictionary = new Dictionary<char, int>();
            dictionarySecond = new Dictionary<char, string>();
        }

        public void setDictionary(Dictionary<char, int> dict)
        {
            dictionary = dict;
        }

        public Dictionary<char, int> getDictionary()
        {
            return dictionary;
        }

        public Dictionary<char, string> getDictionarySecond()
        {
            return dictionarySecond;
        }
    }
}
```


ПРИЛОЖЕНИЕ Г

Form1.cs

Этот код представляет собой часть программы, вероятно, связанную с пользовательским интерфейсом для работы с методами кодирования и декодирования данных с использованием алгоритма Хаффмана.

Взглянем на основные элементы этого кода:

1. Инициализация переменных:

- Создание экземпляров «MyDictionary» и «MyHuffmanTree».
- Установка обработчика для кнопки «button1_Click», который выполняет операции по сжатию данных, а также кодированию и декодированию данных.

2. Обработчик «button1_Click»:

- Получение текста из «textBox1».
- Проверка длины введенной строки.
- Запись данных в файл и их сжатие.
- Использование «WorkerWithDictionary» для заполнения и отображения таблиц данных о вероятностях символов и префиксах.
- Построение дерева Хаффмана, кодирование и декодирование данных, отображение результатов в «textBox2_encode» и «textBox2_Decode».
- Визуализация дерева на «panel3».
- Расчет и отображение коэффициента сжатия и размера закодированных данных.

3. Другие методы «button2_Click», «button3_Click», «button4_Click», «button5_Click»:

- «button2_Click» и «button3_Click» записывают данные в файлы и отображают их на форме.
- «button4_Click» и «button5_Click» делают видимыми таблицы и панель.

В целом, код управляет интерфейсом для выполнения операций сжатия и кодирования данных с использованием алгоритма Хаффмана. Он обрабатывает ввод пользователя, осуществляет операции с файлами и отображает результаты на форме.

```
using Cursac.temerev.model;  
using Cursac.temerev.util;  
using System;  
using System.Drawing;  
using System.Windows.Forms;  
using Cursac.temerev.model;  
using Cursac.temerev.util;
```

```

using static System.Windows.Forms.VisualStyles.VisualStyleElement;
using Brushes = System.Drawing.Brushes;
using Color = System.Drawing.Color;
using Pen = System.Drawing.Pen;

namespace Cursac
{
    public partial class Form1 : Form
    {
        private MyDictionary dictionaryProb = new MyDictionary();
        private MyDictionary dictionaryPrefix = new MyDictionary();

        private MyHuffmanTree tree = new MyHuffmanTree();

        public Form1()
        {
            InitializeComponent();
        }

        private void button1_Click(object sender, EventArgs e)
        {
            string dataString = textBox1.Text;
            int Maxlen = 18;

            if (dataString.Length == 0)
            {
                return;
            }

            if (dataString.Length > Maxlen)
            {
                MessageBox.Show("Было введено больше 20 символов!", "Ошибка!");
                return;
            }

            WorkerWithFiles.WriteToFile(dataString);

            // Сжатие файла после записи данных
            WorkerWithFiles.CompressFile();

            label_amountOfCharacters.Text = Convert.ToString(dataString.Length);
            label_sizeOfFile.Text =
            Convert.ToString(WorkerWithFiles.GetSizeOfFile());

            // Остальной код остается без изменений
            WorkerWithDictionary.parseOfStingToDictionary(dataString,
            dictionaryProb.getDictionary());

            dataGridView1_probability.DataSource = new
            BindingSource(dictionaryProb.getDictionary(), null);
            dataGridView1_probability.AutoSizeColumns();

            tree.setFrequencies(dictionaryProb.getDictionary());
            tree.Build();

            string encode = "";
            string[] mass = tree.Encode(dataString,
            dictionaryProb.getDictionary());
            string decode = "";

            WorkerWithDictionary.FillingPrefixToDictionary(dataString, mass,
            dictionaryPrefix.getDictionarySecond());

            dataGridView2_prefix.DataSource = new
            BindingSource(dictionaryPrefix.getDictionarySecond(), null);
            dataGridView2_prefix.AutoSizeColumns();
        }
    }
}

```

```

        decode = tree.Decode(tree.FullCode(dataString));

        foreach (bool b in tree.FullCode(dataString))
        {
            encode += (b ? "1" : "0");
        }

        textBox2_Decode.Visible = false;
        textBox2_Decode.Text = decode;

        textBox2_encode.Visible = false;
        textBox2_encode.Text = encode;

        Graphics g = panel3.CreateGraphics();
        var obj = new Drawing();
        obj.Draw(g, mass, tree.GetRoot(), dictionaryProb.getDictionary());

        float encodeInt = encode.Length;
        float decodeInt = decode.Length;
        float kf = (decodeInt * 8) / encodeInt;

        label_after.Text = (encode.Length).ToString();
        label_coefficient.Text = kf.ToString();

        button1.Enabled = false;
    }

    private void button2_Click(object sender, EventArgs e)
    {
        WorkerWithFiles.WriteToFile(textBox2_encode.Text);
        textBox2_encode.Visible = true;
    }

    private void button3_Click(object sender, EventArgs e)
    {
        WorkerWithFiles.WriteToFile(textBox2_Decode.Text);
        textBox2_Decode.Visible = true;
    }

    private void button4_Click(object sender, EventArgs e)
    {
        dataGridView1_probability.Visible = true;
        dataGridView2_prefix.Visible = true;
    }

    private void button5_Click(object sender, EventArgs e)
    {
        panell1.Visible = true;
    }

    private void Form1_Load(object sender, EventArgs e)
    {
    }
}

```

ПРИЛОЖЕНИЕ Д

WorkerWithFiles.cs

Этот код представляет класс «WorkerWithFiles», который содержит методы для работы с файлами, включая запись данных в файл, получение размера файла и сжатие файла с использованием алгоритма сжатия GZip.

1. Метод «WriteToFile»:

- Записывает переданную строку «str» в файл с использованием «StreamWriter» и «FileStream».
- Создает файл по указанному пути «pathFile».

2. Метод «GetSizeOfFile»:

- Возвращает размер файла, умноженный на 8. Он предполагает, что каждый символ занимает 8 битов. Однако, важно помнить, что если файл содержит текстовые данные, размер будет отличаться в зависимости от кодировки.

3. Метод «CompressFile»:

- Сжимает файл, используя алгоритм GZip.
- Открывает файл для чтения, создает файл для записи сжатых данных.
- Использует «GZipStream» для сжатия данных из исходного файла в файл сжатых данных.

Примечание:

- Пути к файлам «pathFile» и «compressedFilePath» жестко закодированы в классе. Это может привести к проблемам при перемещении программы или при развертывании на других устройствах или операционных системах. Было бы лучше использовать относительные пути или передавать пути в качестве параметров при необходимости.

- Размер файла вычисляется как размер в байтах, умноженный на 8, предполагая, что каждый символ занимает 8 битов. Это может быть не совсем точно для текстовых файлов, особенно если используется кодировка Unicode или другие кодировки с переменной длиной символов.

```
using System;
using System.Collections.Generic;
using System.IO;
using System.IO.Compression;
using System.Text;

namespace Cursac.temerev.util
{
    class WorkerWithFiles
```

```

{
    const string pathFile = @"C:\Users\user\Desktop\Учёба\Семестр 3\Курсовая
работа c++\курсач1\CourseWorkDPSA\CourseWorkDPSA\DataFile.txt";
    const string compressedFilePath = @"C:\Users\user\Desktop\Учёба\Семестр
3\Курсовая работа
c++\курсач1\CourseWorkDPSA\CourseWorkDPSA\CompressedDataFile.gz";

    public static void WriteToFile(string str)
    {
        using (FileStream objFile = new FileStream(pathFile, FileMode.Create))
        {
            using (StreamWriter objWriter = new StreamWriter(objFile))
            {
                objWriter.Write(str);
            }
        }

        public static long GetSizeOfFile()
        {
            return new FileInfo(pathFile).Length * 8;
        }

        public static void CompressFile()
        {
            using (FileStream originalFileStream = new FileStream(pathFile,
FileMode.Open))
            {
                using (FileStream compressedFileStream =
File.Create(compressedFilePath))
                {
                    using (GZipStream compressionStream = new
GZipStream(compressedFileStream, CompressionMode.Compress))
                    {
                        originalFileStream.CopyTo(compressionStream);
                    }
                }
            }
        }
    }
}

```

ПРИЛОЖЕНИЕ Е

WorkerWithDictionary.cs

Этот код содержит класс «WorkerWithDictionary», предназначенный для работы со словарями, основанными на символах.

1. Метод «parseOfStingToDictionary»:

- Принимает строку «str» и словарь «dict».
- Перебирает каждый символ в строке и обновляет словарь: если символ уже есть в словаре, увеличивает его частоту, иначе добавляет символ в словарь с частотой 1.

2. Метод «FillingPrefixToDictionary»:

- Принимает строку «str», массив строк «strPrefix» и словарь «dict».
- Используется для заполнения словаря «dict» парами ключ-значение, где символы из строки «str» являются ключами, а соответствующие им строки из массива «strPrefix» - значениями.

3. Метод «CountSymbols»:

- Возвращает количество символов в словаре.

4. Метод «changeFromAmountToPersent»:

- Преобразует частоты символов из словаря «dict» в проценты, создавая новый словарь «withPersent», где каждая частота символа делится на общее количество символов.

5. Приватный метод «getTotalAmount»:

- Возвращает общее количество символов в словаре.

Основной функционал класса - работа с частотами символов, их обновление и конвертация в процентное соотношение от общего количества символов. Однако, метод «changeFromAmountToPersent» должен использовать новый словарь «withPersent» для сохранения процентных значений, вместо изменения исходного словаря «dict».

```
using System;
using System.Collections.Generic;
using System.Linq;

namespace Cursac.temerev.util
{
    class WorkerWithDictionary
    {
        public static void parseOfStingToDictionary(string str, Dictionary<char,
int> dict)
        {
            foreach (char c in str)
```

```

        {
            if (dict.ContainsKey(c)) //содержится ли указанный ключ в словаре
            {
                dict[c]++;
            }
            else // в ином случае добавляем объект
            {
                dict.Add(c, 1);
            }
        }
    }

    public static void FillingPrefixToDictionary(string str, string []
    strPrefix, Dictionary<char, string> dict)
    {
        int i = 0;

        foreach (char c in str)
        {
            if (!dict.ContainsKey(c))
            {
                dict.Add(c, strPrefix[i].ToString());
                i++;
            }
        }
    }

    public static int CountSymbols(Dictionary<char, int> dict)
    {
        return dict.Count;
    }

    public static Dictionary<char, double>
    changeFromAmountToPersent(Dictionary<char, int> dict)
    {
        Dictionary<char, double> withPersent = new Dictionary<char, double>();
        int total_amount = getTotalAmount(dict);

        foreach (char key in dict.Keys.ToList())
        {
            dict[key] = dict[key] / total_amount;
        }

        return withPersent;
    }

    private static int getTotalAmount(Dictionary<char, int> dict) // метод
    возвращает количество символов
    {
        int totalAmount = 0;

        foreach (int amount in dict.Values)
        {
            totalAmount += amount;
        }

        return totalAmount;
    }
}

```

ПРИЛОЖЕНИЕ Ж

Node.cs

Этот код содержит класс «Node», который представляет узел для дерева Хаффмана.

1. Свойства класса «Node»:

- «Symbol»: символ.
- «Frequency»: частота символа.
- «Right» и «Left»: ссылки на правый и левый потомок узла.
- «Symbols»: строка, содержащая символы (используется при построении дерева).

2. Метод «Traverse»:

- Рекурсивно обходит дерево Хаффмана, чтобы найти код символа.
- Принимает символ «symbol» и список «data» (путь кодирования).
- Если узел - лист (не имеет дочерних узлов), проверяет, соответствует ли символ узлу. Если да, возвращает путь кодирования для этого символа, если нет - «null».
- В противном случае, выполняется рекурсивный обход по левому и правому потомкам, формируя путь кодирования. Если путь найден в левом поддереве, возвращается путь из левого поддерева, иначе из правого. Если ни один путь не найден, возвращается «null».

Этот класс представляет узел для дерева Хаффмана и реализует метод для поиска кода символа в дереве.

```
using System.Collections.Generic;

namespace Cursac.temerev.model
{
    class Node
    {
        public char Symbol { get; set; }
        public int Frequency { get; set; }
        public Node Right { get; set; }
        public Node Left { get; set; }

        public string Symbols { get; set; }
        public List<bool> Traverse(char symbol, List<bool> data)
        {
            // Leaf
            if (Right == null && Left == null)
            {
                if (symbol.Equals(this.Symbol))
                {
                    return data;
                }
            }
        }
    }
}
```



```

        else
        {
            return null;
        }
    }

    List<bool> left = null;
    List<bool> right = null;

    if (Left != null)
    {
        List<bool> leftPath = new List<bool>();
        leftPath.AddRange(data);
        leftPath.Add(false);

        left = Left.Traverse(symbol, leftPath);
    }

    if (Right != null)
    {
        List<bool> rightPath = new List<bool>();
        rightPath.AddRange(data);
        rightPath.Add(true);
        right = Right.Traverse(symbol, rightPath);
    }

    if (left != null)
    {
        return left;
    }
    else
    {
        return right;
    }
}
}
}

```