**1. Алгоритм Хаффмена**

Идея данного алгоритма состоит в том, чтобы минимизировать код для символа встречающегося чаще. Так самые частые символы займут меньше всего места, а редко встречаемые символы займут чуть больше места.

Чтобы получить код для каждого символа будем строить бинарное дерево, в котором символы будут находиться в листьях. Также в узлах будем хранить частоту появления. Строить начинаем с листьев, объединяя на каждом шаге 2 листа с наименьшими частотами в один узел, частота которого будет равняться сумме частот. Так мы получим дерево, в котором символы с большей частотой будут находиться ближе к корню. Далее пройдем по дереву и создадим код для каждого символа.

Реализация.

А) Кодирование Код для запуска Haffmen \Encode.java

1. Считываем текст из файла.

2. Проходим по всему тексту и определяем набор символов и частоту использования каждого символа. Так мы создадим оптимальный алфавит.

3. Используем класс TreeNode, у которого есть 4 параметра: символ, частота, левый лист, правый лист. Для каждого символа создаем TreeNode, это будут листья нашего будущего дерева.

4. Т.к. дерево строим, объединяя листы с наименьшей частотой, то для этого алгоритма нам необходимо будет постоянно сортировать листы. Чтобы это избежать, воспользуемся очередью с приоритетами. Приоритет будет равен числу употребления данного символа в тексте.

Берем два первых узла из очереди, составляем новый узел, у которого левым будет первый узел, правым – второй, символа не будет, так как это не лист, а частота будет равна суммам частот двух взятых узлов. Убираем из очереди два узла которые мы вытащили и добавляем туда только что созданный узел. Повторяем это до тех пор, пока не дойдем до 1 узла – корня.

5. Для каждого символа находим его код, полученный в результате кодирования, путем прохождения по дереву.

6. Каждый символ текста заменяем на его код.

7. Сохраняем. Записываем количество символов в алфавите исходного текста, далее передаем столько раз (символ частота), передаем длину закодированного текста, закодированный текст преобразуем в байты, для того чтобы единицы и нули хранились не как отдельные символы, а как биты информации, занимая меньшее количество памяти. Записываем байтовую версию в файл.

Для эксперимента был взят сгенерированный текст, вес текста 136кб. По итогам кодирования вес файла стал 44 кб

Б) Декодирование. Код для запуска Haffmen\Decode.java

1. Считываем данные из файла. А именно: количество символов в алфавите, символ-частота, длину закодированного сообщения, байты текста. Байты текста переводим в биты.

2. Генерируем бинарное дерево на основе значений символ-частота.

3.Так как в бинарном дереве все символы являются листьями, то этот код однозначно декодируется.

Идем по всем битам строки. Каждый бит – это переход вправо или влево по дереву, построенному на шаге 2. Дерево строилось так, что значение в узле есть только у листов. Поэтому мы опускаемся в глубину дерева до тех пор, пока в узле не найдем значение символа. Как нашли символ, записываем его в результирующую строку и вновь поднимаемся к корню дерева.

4. Записываем полученный результат в файл.

Для эксперимента был взят файл, полученный при кодировании текста на предыдущем шаге. Декодирование прошло успешно, содержание текста не исказилось.

**2. Арифметическое кодирование**

Арифметическое кодирование – это алгоритм сжатия без потерь.

А) Кодирование Изначально алгоритму выдается текст. Предполагается, что кодировщик знает алфавит и частоту употребления каждого символа.

Изначально берется отрезок [0,1).

1. Весь отрезок разбиваем на отрезки, равные частоте появления каждого символа. Для каждого символа запоминаем координаты начала и конца его отрезка.
2. Считываем символ. Выбираем отрезок, соответствующий данному символу. И возвращаемся на 1 шаг.

Делаем это до тех пор, пока не закончатся символы кодируемого сообщения.

В результате получаем некоторый отрезок. Любое число входящее в этот отрезок будет являться кодом данного сообщения.

Для того, чтобы вес сообщения уменьшился, необходимо взять из отрезка наименьшее число, которому будет соответствовать битовое представление.

\*Для разбиения отрезка на другие отрезки предлагаем пользоваться формулами:

Right = BaseLeft + (BaseRight- BaseLeft)\*DefaulRight

Left = BaseLeft + (BaseRight- BaseLeft)\*DefaulLeft,

где DefaulRight и DefaulLeft границы отрезка, соответствующего символу на отрезке [0,1],

BaseRight и BaseLeft это границы отрезка, с которым работаем в данный момент,

Right и Left – границы отрезка, соответствующего символу на отрезке, с которым работаем в данный момент.

Реализация. Код для запуска Arithmetic\Encode.java

1. Программа запрашивает пусть к файлу, и директорию, в которой необходимо сохранить закодированное сообщение.
2. Проходим по всему сообщению, составляем карту символ-количество упоминаний в тексте.
3. Генерируем карту символ – отрезок. Отрезок – левая и правая границы на отрезке [0;1]
4. Задаем изначальный отрезок [0;1].
5. Проходим по всем буквам сообщения и выполняем алгоритм, описанный выше.
6. Сохраняем данные. На выходе два файла. 1- setting.arithmetic, в него записываем размер алфавита, карту символ- количество упоминаний, и размер кодируемого сообщения в символах. 2-compressed.arithmetic, файл с непосредственным содержанием результата кодирования.

По тз перевод в бинарный вид был необязательной частью задания. Поэтому сохраняем строку в десятичном представлении. Это влияет только на размер выходного файла (значительное изменение веса файла достигается при переводе), алгоритм работает исправно.

Важная заметка. Так как мы работаем изначально с отрезком от нуля до единицы, то изначально границы имеют вещественный вид. С каждым шагом в этих значениях количество знаков после запятой увеличивается. Однако основные примитивные типы данных имеют ограничение точности. Поэтому предлагаю воспользоваться типом данных BigDecimal, который не имеет ограничений на размер.

Б) Декодирование

На вход получаем число р. Предполагается, что алфавит и частота символов известны, а также известно количество символов в начальном сообщении.

Отрезок [0;1] разбиваем на отрезки, равные частоте появления каждого символа.

Определяем интервал, для которого left <=p <right

Далее продолжаем работать на найденном отрезке. Также разбиваем этот отрезок на части, выбираем необходимый отрезок. Сохраняем символ. Повторяем это до тех пор, пока не декодируем нужное количество символов.

Для облегчения вычислений после выбора отрезка представим его в исходном интервале [0;1], для этого начальное число р необходимо представить на новом интервале.

p= (p -Left)/ (Right - Left), где Left и Right- границы отрезка, которого мы ходим представить в начальном варианте.

Реализация Код для запуска Arithmetic\Decode.java

1. Программа запрашивает начальную директорию, в которой хранится два файла setting. arithmetic и compressed. arithmetic, а также конечную директорию, куда будет записан результат декодирования.
2. Из настроечного файла считываем словарь с частотами для каждого символа и количество символов в исходном сообщении

Из основного файла считываем результат кодирования.

1. Восстанавливаем словарь символ-отрезок
2. Выполняем алгоритм декодирования, описанный выше, столько раз сколько символов в сообщении.

На выходе получаем исходный текст.

Единственный минус этой реализации алгоритма – время выполнения вычислений.

**3. Алгоритм LZW**

Алгоритм Лемпеля-Зива-Уэлча (Lempel—Ziv—Welch) предполагает кодирование сообщения в поточном режиме (по мере поступления данных) и формирование словаря (кодируемая последовательность - код) одновременно с кодированием, при этом учитывая уже закодированную часть текста.

Алгоритм не требует передачи декодеру полного словаря, необходим только базовый словарь, составленный из кодировок начального словаря. В процессе декодирования алгоритм сам восстановит кодовый словарь.

Коротко об алгоритме данного сжатия. Изначально есть некоторое сообщение (текст) с известным алфавитом (всевозможные символы текста).

На первом шаге кодируются все слова алфавита.

Будем называть K- кодовой строкой- некоторый буфер, в котором будем собирать элементы конечного словаря. А считанный символ Х.

1.В кодовую строку заносим 1 символ строки, а в результирующую строку код первого символа.

2. Далее считываем следующий символ строки Х, заносим его в кодовую строку, получаем КХ.

- Если такое кодовое слово имеется в словаре, то в результирующую строку пишем код КХ.

- Если такого кодового слова нет, то в результирующую строку записываем код Х, добавляем в словарь слово КХ, очищаем буфер и кладем туда Х.

3. Повторяем шаг 2 до тех пор, пока не дойдем до конца.

Результирующая строка и будет кодом сообщения.

Теперь об алгоритме декодирования.

Полагается, что декодировщик знает изначальный словарь из букв алфавита. Благодаря этому декодируем некоторый начальный отрезок закодированного текста. После того, как декодировали первые два числа в буквы, формируем слово из этих букв и помещаем в словарь. Декодировав следующее число, образуем новое слово, присоединив этот символ к предыдущему декодированному. На некотором этапе окажется, что считанное число не входит в базовый алфавит, но уже есть в промежуточном алфавите. Тогда в словарь заносим слово, начало - строка из предыдущего шага, а конец – первая буква последнего декодированного слова. Очередное слово должно иметь окончание, совпадающее с приставкой следующего слова, но эта общая часть должна состоять только из одной буквы.

Перейдем к реализации.

А) Кодирование

Код для запуска LZW\Encode.java. Программа запрашивает текстовый файл, содержащий сообщение, которое будем кодировать, и директорию, в которую необходимо сохранить последовательность битов.

1. Проходим по всему тексту для того, чтобы определить алфавит сообщения
2. Создаем HashMap, будем хранить и добавлять кодовые слова. Ключ-значение в данном случае будет слово-кодовое слово. Заносим туда алфавит и кодируем его.
3. Переводим текст в массив char и начинаем наш алгоритм. Считываем символ, добавляем к предыдущему считанному, проверяем есть ли оно в словаре. Если нет, тогда добавляем в словарь эту последовательность, в результат добавляем код слова, полученный на предыдущем шаге, а в буфер кладем последний считанный символ.
4. Повторяем пункт 3 до тех пор, пока не дойдем до конца.
5. В выходной файл записываем число n (размер словаря, состоящего из алфавита), далее передаем n раз (символ и его код), далее количество кодов и сами коды.

Для эксперимента был сгенерирован файл объемом 136кб, после сжатия этим методом размер файла стал 43кб.

Б) Декодирование

Код для запуска LZW\Decode.java. Программа запрашивает текстовый файл, содержащий закодированное сообщение, которое будем декодировать, и директорию, в которую необходимо сохранить последовательность битов.

1. Считываем данные.

Количество символов в словаре, n раз символ код, количество кодов, сами коды.

1. Создаем промежуточный словарь и помещаем туда полученный из файла словарь, только ключ значение будет код-символ, а не символ-код.
2. На первом шаге в некоторый буфер кладем символ 1 считанного кода. Добавляем символ в результирующую строку.
3. Проходим по всей последовательности кодов. Считываем код:
4. Добавляем в результирующую строку, в словарь добавляем буфер+1 только что считанный символ, в буфер переводим считанную строку.
5. Записываем в файл

Для эксперимента был взят закодированный документ, полученный на предыдущем шаге. Декодирование произошло успешно.

**(7,4) - Код Хэмминга**

Код Хэмминга используется для обнаружения ошибок. Создаются наборы битов четности, которые могут определить и исправить ошибку при передаче данных. В случае если ошибок больше, чем одна, код Хэмминга не сможет восстановить исходные данные.

В нашем случае рассмотрим код Хэмминга (7,4). Почему (7,4)? К основным 4 битам информации мы будем добавлять 3 бита четности. Общая длина блока станет 7.

Пусть есть некоторая последовательность 4 битов c1c2c3c4. Представим коды четности как b1, b2, b3. Где

b1 = с1 **XOR** c2 **XOR** c4

b2= с1 **XOR** c3 **XOR** c4

b3= с2 **XOR** c3 **XOR** c4

Для простоты вычисления заменим операцию XOR, тогда

b1 = (с1 **+** c2 **+** c4) %2

b2= (с1 **+** c3 **+** c4) %2

b3= (с2 **+** c3 **+**c4) %2

Результатом кодирования будет строка b1 b2 c1 b3 c2 c3 c4.

Как определить ошибку и исправить ее?

Допустим мы получили строку из 7 символов а1а2а3а4а5а6а7.

1 шагом извлечем биты передаваемой информации. Это будет слово а3а5а6а7.

Для этого слова вычислим коды четности.

b1 = (а3 **+** а5 **+** а7) %2

b2= (а3 **+** а6 **+** а7) %2

b3= (а5 **+** а6 **+**а7) %2

2 шагом проверим совпадают ли вычисленные коды с полученными кодами. Для определения бита с ошибкой, учитываем, что ошибка всего одна.

1. Если b1!=а1 И b2!=а2 И b3!=а4

То ошибка была в 7 передаваемом бите, т.е в 4 исходном бите.

Следовательно, ожидаемое слово а3а5а6(1-а7)

1. Если b1=а1 И b2!=а2 И b3!=а4

Тогда ошибка в 6 передаваемом бите, т.е в 3 исходном

Следовательно, ожидаемое слово а3а5(1-а6)а7

1. Если b1!=а1 И b2=а2 И b3!=а4

Тогда ошибка в 6 передаваемом бите, т.е в 2 исходном

Следовательно, ожидаемое слово а3(1-а5)а6а7

1. Если b1!=а1 И b2!=а2 И b3=а4

Тогда ошибка в 6 передаваемом бите, т.е в 1 исходном

Следовательно, ожидаемое слово (1-а3)а5а6а7

Если учитывать, что ошибка одна, то для других вариантов окажется, что ожидаемая строка совпадет с а3а5а6а7, ошибка будет либо в битах четности, либо ее совсем не будет.

Реализация

А) Кодирование

Код для запуска Hamming\Encode.java. Программа запрашивает текстовый файл, содержащий сообщение, которое будем кодировать, и директорию, в которую необходимо сохранить последовательность битов.

1. Считываем текст из файла и преобразовываем его в последовательность байтов, а затем битов. Один символ занимает 2 байта, т.е на один символ приходится 16 битов.
2. Делим получившуюся последовательность на блоки по 4 бита. Т.к символ = 16 битов, то наша последовательность всегда будет делиться на 4.
3. Для каждого блока генерируем коды четности, формируем новую строку по принципу, описанному выше. Полученные символы сохраняем в строку.
4. Полученную последовательность записываем в файл. Для чистоты эксперимента было принято решение не оптимизировать полученную строку и записать в исходном виде.

Б) Декодирование (перед этим этапом можно взять файл, полученный на этапе кодирования и изменить некоторые биты, но не чаще чем 1 раз на 7 битов, т.к. иначе ошибка не будет распознана и исправлена)

Код для запуска Hamming\Decode.java. Программа запрашивает текстовый файл, содержащий сообщение в битовом виде, которое будем декодировать, и директорию, в которую необходимо сохранить последовательность битов.

1. Считываем текст из файла
2. Делим строку на блоки по 7 битов. Для каждого блока проверяем коды четности и воссоздаем первоначальную последовательность битов.
3. Переводим последовательность битов в байты, далее байты в символы и записываем результат в файл.