В этом документе я кратко описала основную теорию (+ комментарии с теорией к файлам и функциям программы). Также в тексте самих файлов присутствуют комментарии-описания их параметров и работы.

Передо мной стояла задача изучить Коды Рида-Соломона и реализовать декодер, на основе алгоритма Берлекемпа-Месси. Для этого я начала с изучения теории вопроса и реализации кодировщика.

Табличные коды -> Линейно-блочные коды -> Циклические коды -> Коды Рида-Соломона

В кодах Рида-Соломона сообщение представляется в виде набора символов некоторого алфавита. Собственно говоря, в качестве алфавита используется поле Галуа.

То есть, если мы хотим закодировать сообщение, представленное двоичным кодом, то мы разбиваем его (в моем случае, используем поле Галуа из 256 элементов) на группы по 8 битов и дальше работаем с каждой группой как с числом из этого поля Галуа.

При построении кода Рида-Соломона задаётся пара чисел N, K, где N – общее количество символов, а K – «полезное» количество символов, остальные N-K символов представляют собой избыточный код, предназначенный для восстановления ошибок.

Такой код Галуа будет иметь так называемое «расстояние Хэмминга» D = N – K +1; Расстояние Хэмминга является параметром кода и определяется как минимальное число различий между двумя различными кодовыми словами. В соответствии с теорией кодирования, код, имеющий расстояние Хемминга D = 2t+1, позволяет восстанавливать t ошибок. Таким образом, если в наше кодовое слово случайно внести t = (N-K)/2 ошибок (т.е. просто произвольно заменить значения t символов любыми значениями), то окажется возможным обнаружить и исправить эти ошибки.

Сообщения при кодировании Рида-Соломона представляются полиномами. Исходное сообщение представляется как коэффициенты полинома p(x) степени K-1, имеющего (разумеется!) k коэффициентов. Важную роль играет порождающий многочлен Рида-Соломона, g(x) (порождающие полиномы мы берем из разложения $x^n - 1$).

Общий план:

В целом, решение поставленной задачи можно разбить на 3 основных пункта:

1) Нахождение поля Галуа и определение математических операций (я выбрала полином 285, $GF(2^8)$)
2) Кодирование сообщения
3) Декодирование сообщения

Поле Галуа (файл **help_gf.h**)

Конечные поля существуют не при любом числе элементов, а только в случае, если число элементов является простым числом p или степенью q=p^m простого числа. Делается это не просто так, ведь если мы будем вычислять поле используя многочлен из диапазона 0..255 (и др. числа) то при (xor 256) у нас обязательно появятся дубликаты.

Поэтому, прежде, чем кодировать и декодировать сообщения, используя поле Галуа, нам необходимо найти неприводимый многочлен, чтобы в нашем исходном поле у нас было 0,1,2,..., p-1 различных уникальных значений.

Функция find_prime_polys(). Мы генерируем все возможные простые многочлены (т.е. все целые числа от 256..512), строим поле Галуа и проверяем, чтобы не было дубликатов. Также используется функция умножения gf_mult() на основе алгоритма «Русских крестьян». (Я реализовала не самый быстрый алгоритм, пока не придумала как его улучшить). В результате функция выведет все возможные простые многочлены (у меня получились следующие: 285, 299, 301, 333, 351, 355, 357, 361, 369, 391, 397, 425, 451, 463, 487, 501). Я выбрала 285.

Математические операции над элементами поля (файл **gf.h**)

Интуитивно хочется использовать классические определения для математических операций (сложение, вычитание, умножение, деление) и затем просто сделать mod 256, для избегания переполнения (но это работает для всего кроме деления). Поскольку база поля = 2, логично, что для сложения и вычитания используется xor.

Интереснее дела обстоят с умножением и делением.

Для того, чтобы программа работала корректно, при умножении и делении 8 битных чисел мы всегда должны получать 8 битные числа в результате. Именно поэтому нам необходимо «приводить» результат с помощью деления на некоторое выбранное (не случайно!) число. А точнее, с помощью деления на неприводимый многочлен. (см. поиск неприводимого многочлена).

Умножение в классическом понимании занимает очень много времени (сколько итераций нужно проделать!?), поэтому почти во всех статьях приводится алгоритм «умножения с логарифмами». Находим в таблице «логарифм», то есть «в какую степень нужно возвести 2 чтобы получить нужное число» - для множителей, после этого складываем их и находим значение в таблице «экспонента» (мы ее заранее делаем размера x2, чтобы не делать %255).

Деление происходит также, только вместо сложения мы используем разность.

Построение самих таблиц для выбранного не приведенного полинома (285 или 100011101 или $x^8 + x^4 + x^3 + x^2 + 1$) можно посмотреть в файле **help_gf.h** – функция **init_tables().** (Для наглядности в программе таблицы «логарифм» и «экспонента» явно проинициализированы).

Также нам необходимо определить математические операции (сложение, умножение и т.д.) для самих полиномов (я представила их векторами), использующихся для кодирования/декодирования, коэффициенты которых как раз являются элементами поля Галуа. Для деления был использован метод синтетического деления.

Кодирование Рида-Соломона(файл **rs_encoding.h**)

Если коротко, то смысл кодирования заключается в том, чтобы к исходной информации добавить избыточную, по которой с помощью определенных математических правил мы могли бы восстановить исходное сообщение.

Существует две разновидности кодирования: систематический и несистематический код.

В несистематическом коде закодированное сообщение не содержит в явном виде исходного сообщения: закодированное сообщение получается как произведение исходного сообщения (а сообщения представляются в виде многочленов!) на порождающий многочлен g(x) : C(x) = p(x) * g(x).

Порождающий многочлен строится следующим образом:

$\prod_{(i=0..D-1)}(x - \alpha^i) = (x - \alpha^0)* (x - \alpha^1)*\dots (x + \alpha^{D-1})$ (функция **rs_generator_poly())**

Систематический код строится по-другому: Сначала полином сдвигается на K коэффициентов влево p'(x) = p(x) * $x^{(N-k)}$

а потом вычисляется его остаток от деления на порождающий полином и прибавляется к p'(x): C(x) = p'(x) + p'(x) mod g(x)

Другими словами, сообщение «сдвигается» на N-K символов - так, что его полином имеет такие коэффициенты: mk, .. m2, m1, m0, 0,0,0,0,0,0 (m0.. mk - символы сообщения)

Далее этот полином делится с остатком на порождающий полином g(x), в результате чего в остатке получается полином степени (N-K-1) с N-k коэффициентами. (Поскольку полином g(x) имеет степень N-k, что следует из принципа его построения). Этот полином прибавляется к исходному полиному, сдвинутому на N-K символов (т.е. коэффициенты остатка как раз занимают место нулей) (функция **rs_encode_msg())**

Закодированное сообщение C(x) обладает очень важным свойством: оно без остатка делится на порождающий многочлен g(x)! (это не так очевидно, как в несистематическом)

Для систематического кода очевидно, что K старших коэффициентов полученного кода C(x) содержат исходное сообщение. Это очень удобно при декодировании и в дальнейшем я рассматривала именно систематический вариант.

Обобщая, можно сказать, что наш генераторный полином – словарь кодирования, а полиномиальное деление = оператор преобразования нашего сообщения в код Рида Соломона.

Программно это выглядит очень просто: генерируем полином, вызываем метод деления (определенный для полиномов, алгоритм синтетического деления) и возвращаем исходное сообщение + остаток. (мне показалось интересным реализовать метод синтетического деления, к тому же оно должно работать быстрее, в теории).

Декодирование Рида-Соломона(файл **rs_decoding.h**)

Самая интересная и объёмная часть – это декодирование. Очевидно, что первым шагом необходимо выполнить деление полинома на порождающий полином g(x). Если остаток равен нулю, то сообщение не искажено и декодирование (для систематического кода) тривиально: следует просто выделить из сообщения коэффициенты с N-k+1 до N-1 – это и будет основное сообщение. В случае же присутствия ошибки (т.е. e(x) = C(x) mod g(x) != 0), придётся выполнить следующие действия. (Программно я не стала делать эту проверку, чтобы избежать лишнего деления, мы сразу строим полином синдрома ошибки и смотрим, есть ли в нем не 0 элементы)

Декодирование основано на построении многочлена синдрома ошибки S(x) и отыскании соответствующего ему многочлена локаторов L(x). Локаторы ошибок – это элементы поля Галуа, степень которых совпадает с позицией ошибки. Так, если искажён коэффициент при x^4 , то локатор этой ошибки равен a^4.

Многочлен локаторов L(x) – это многочлен, корни которого обратны локаторам ошибок. Таким образом, многочлен L(x) должен иметь вид L(x) = (1+xX1)(1+xX2)…(1+xXi), где X1, X2, Xi – локаторы ошибок. (1+xXi обращается в ноль при x=Xi -1 : $Xi Xi^{-1}$ = 1, 1+1 =0.) Ясно, что если этот многочлен будет найден, то мы легко сможем определить локаторы ошибок – для этого потребуется только определить его корни, что легко сделать обычным перебором.

Для определения этого полинома сначала получают вспомогательный полином S(x), так называемый синдром ошибки. Коэффициенты синдрома ошибки получаются подстановкой степеней примитивного члена в сам многочлен сообщения C(x).

Нетрудно убедиться, что если бы сообщение не было искажено, то все коэффициенты Si оказались бы равны нулю: ведь неискажённое сообщение C(x) кратно порождающему многочлену g(x), для которого числа a1 , a2 … aN-K являются корнями

Между L(x) и S(x) существует соотношение L(x)*S(x) = W(x) mod xN-k. W(x) называется многочленом ошибок. Степень многочлена W(x) не может превышать u-1, где u – количество ошибок. А максимальное количество ошибок, которые может исправить код Рида-Соломона, это (N-K)/2. С учётом этого обстоятельства, а также учитывая, что свободный член L(x) L0=1 (ведь L(x) = (1+xX1)(1+xX2)…(1+xXi)) можно составить систему линейных уравнений.

Коротко шаги декодирования (функция rs_decode_msg()):

1. Вычислить полином синдрома Si = C(ai+1) (функция **rs_calc_syndromes()**).
2. Если все элементы нулевые – значит исходное сообщение не искажено. Вывести его.
3. Иначе вычислить L(x) (при условии что мы не знаем в каком месте находятся ошибки), используя алгоритм Берлекемпа-Месси (функция **rs_find_error_locator()**).
4. Определить локаторы ошибок (функция **rs_find_errors())**
5. Декодировать сообщение (функция **rs_correct_errata())**
   5.1 Найти корни L(x) – они будут обратны к локаторам ошибок.
   5.2 Находим L2(x), для ошибок, чье местоположение мы знаем. (функция **rs_find_errarta_locator()).**
   5.3 Находим W(x) для L2(x) (функция **rs_find_error_evaluator()).**
   5.4 Используем алгоритм Форни для вычисления значения ошибки. Для этого находим формальную производную многочлена и значения ошибок ei по формулам.

$$f'(z)=\sum_{i-1} a_{2i+1} z^{2i} = a_1 + a_3 z^2 + \ldots + a_{2\psi+1} z^{2\psi},\ \psi = int\,0.5(n-k), \quad e_i = \frac{\omega(\alpha^{-i})}{\sigma_z'(\alpha^{-i})},$$

5.5 Сформировать многочлен ошибок E(X) на основе локаторов и значений ошибок. Деление W на производную локатора ошибок дает нам величину ошибки (т. е. значение для восстановления) i-го символа.

5.6 Скорректировать C(x) = C(x) + E(x).

6. Вычислить полином синдрома Si = C(ai+1) (функция **rs_calc_syndromes()**).
7. Проверить, чтобы все значения равнялись 0.

Алгоритм Берлекемпа-Месси

Алгоритм Берлекемпа-Месси является наиболее эффективным для решения множества линейных уравнений и нахождения L(x). Я поняла основную идею следующим образом – алгоритм итеративно вычисляет полином локатора ошибок. Для этого он вычисляет дельта-расхождение, по которому мы можем определить, нужно ли нам обновлять локатор ошибок или нет.