Niezawodność i diagnostyka układów cyfrowych

–

Sprawozdanie z projektu

Kinga Banasiak – 281094

Iuliia Kapustinskaia – 282341

Anna Kuras - 280945

Temat projektu	
Koder i dekoder kodu BCH	
Prowadzący kursu	Dr. Inż. Maciej Nikodem
Termin zajęć	Wtorki Nieparzyste 9:15-10:45
Termin oddania	

1. Podział pracy

Kinga Banasiak – dekoder uproszczony, wstępny dekoder pełny

Iuliia Kapustinskaia – implementacja ciała Gallois, pisanie testów, dekoder pełny (obliczanie syndromów, algorytm Berlekampa-Masseya, algorytm Chiena)

Anna Kuras –wielomian generujący, koder, opracowanie liczby i wyników testów, sprawozdanie, wyświetlanie wyników dekodera uproszczonego i pełnego

GitHub: https://github.com/drQuintilis/NIDUC_projekt.git

2. Wprowadzenie

Kody BCH (Bose-Chaudhuri-Hocquenghem) należą do klasy cyklicznych kodów korekcyjnych, znajdujących zastosowania w systemach komunikacyjnych i pamięciach cyfrowych. Zostały one opracowane w latach 60. XX wieku.

Kody BCH działają w oparciu o algebrę wielomianów nad ciałem skończonym GF(2^m).

Parametry naszego kodu:

t = 11 (zdolność korekcyjna)
n = 255 (długość słowa kodowego)
k = 171 (liczba bitów informacyjnych)

Oznacza to, że wiadomość o 171 bitach kodowana jest w naszym kodzie na 255 bitach, z czego 84 bity, to bity kontrolne. Nasz kod jest w stanie teoretycznie poprawiać do 11 błędów.

3. Przebieg realizacji projektu

1. Implementacja ciała skończonego

Cały projekt został napisany w języku Python. Zaczęłyśmy realizację projektu od implementacji ciała skończonego GF(2^8) – stworzeniu wszystkich metod potrzebnych do pracy na tym ciele. Wielomianem pierwotnym, którym się posługiwałyśmy, był:
$x^8+x^4+x^3+x+1$ – 0x11d

2. Tworzenie tablic potęg i logarytmów:

Wybieramy prymitywny pierwiastek α=0x02.

Generujemy kolejne potęgi α^i, wykonując mnożenie w GF(2^8) z redukcją modulo wielomian pierwotny (0x11d)

Tworzymy tablicę antylogarytmów, zawierającą wartości α^i oraz tablicę logarytmów, która mapuje każdy element x do potęgi i dla której $x=\alpha^i$:

```
def build_tables(self, prim=0x11d):
    # Budowanie tabel wykładników i logarytmów
    # α = 0x02 w wybranej reprezentacji (można zmieniać w zależności od
konwencji)
    alpha = 0x02
    log_table = [0] * 256
    antilog_table = [0] * 512
    x = 1
    for i in range(255):
```

```
        antilog_table[i] = x
        log_table[x] = i
        x = gf_mul(x, alpha, prim=prim)
    for i in range(255, 512):
        antilog_table[i] = antilog_table[i - 255]
    return log_table, antilog_table

def gf_pow(self, alpha_power):
    """
    Zwraca element pola odpowiadający α^(alpha_power).
    Ponieważ α^255 = 1 i cykl się powtarza.
    """
    return self.antilog_table[alpha_power % 255]
```

Zamiana wielomianów na liczby typu int i liczby typu int na wielomiany:

```
def poly_to_int(poly):
    """Konwertuje wielomian z listy na liczbę całkowitą."""
    return int(''.join(map(str, poly)), 2)

def int_to_poly(num, length):
    """Konwertuje liczbę całkowitą na wielomian w postaci listy zer i
jedynek."""
    return list(map(int, bin(num)[2:].zfill(length)))
```

Elementy odwrotne:

```
def gf_inv(self, a):
    """
    Zwraca element multiplikatywnie odwrotny do 'a' w GF(2^8),
    używając tabel logarytmów i antilogarytmów.
    """
    if a == 0:
        raise ZeroDivisionError(" nie ma odwrotności 0 w  GF(2^8).")

    # Obliczamy indeks w tabeli logarytmów
    a_log = self.log_table[a]  # log(a)

    # Wzór a^-1 = α^(255 - log(a)), ale trzeba wziąć modulo 255
    exponent = 255 - a_log

    # Jeśli wykładnik == 255, to wykładnik % 255 = 0
    exponent %= 255

    # Zwracamy odpowiedni element z tabeli antilogarytmów
    return self.antilog_table[exponent]
```

Mnożenie liczb:

Wykonujemy iteracyjne przesunięcia i redukcje. Jeśli najmłodszy bit liczby y jest równy 1, dodajemy x do wyniku (XOR-em). Przesuwamy x w lewo (mnożenie przez 2) i sprawdzamy, czy najstarszy bit wynosi 1. Jeśli tak, wykonujemy redukcję modulo 0x11d. Wynik pozostaje ograniczony do 8 bitów poprzez AND 0xFF. Powtarzamy proces dla wszystkich bitów y.

```
def gf_mul(x, y, p
rim=0x11d, field_size=8):
    """
    Mnożenie w GF(2^m) przy m=8 i prymitywnym wielomianie 0x11d dla operacji
podobnych do AES.
    0x11d odpowiada x^8 + x^4 + x^3 + x + 1.
```

```
    """
    r = 0
    for i in range(field_size):
        if y & 1:
            r ^= x
        hbs = x & 0x80
        x <<= 1
        if hbs:
            x ^= prim
        x &= 0xFF
        y >>= 1
    return r
```

Mnożenie wielomianów:

Odbywa się w pętli, każdy współczynnik pierwszego wielomianu jest mnożony przez każdy współczynnik drugiego, następnie wartości są XOR-owane.

```
def multiply_polynomials(self, poly1, poly2):
    result = [0] * (len(poly1) + len(poly2) - 1)
    for i, coef1 in enumerate(poly1):
        for j, coef2 in enumerate(poly2):
            result[i + j] ^= coef1 & coef2  # XOR (mod 2)
    return result
```

Obliczanie reszty z dzielenia:

Wykonujemy XOR z dzielnikiem, dopóki dzielna jest większa lub równa dzielnikowi, jeśli najstarszy bit dzielnej jest równy 1 i przesuwamy dzielną, usuwając najstarszy bit.

```
def compute_remainder(self, dividend, divisor):
    dividend = dividend[:]
    while len(dividend) >= len(divisor):
        if dividend[0] == 1:
            for i in range(len(divisor)):
                dividend[i] ^= divisor[i]
        dividend.pop(0)
    return dividend
```

3. Wielomian generujący:

Wielomian generujący uzyskano poprzez przemnożenie przez siebie wielomianów minimalnych dla warstw cyklotymicznych od m_1 do m_{21} (do 2t – 1). Warstwa cyklotymiczna dla liczby i to zbiór potęg α, które można uzyskać, w wyniku kolejnego potęgowania α^i, z uwzględnieniem mod 255. Na przykład warstwa cyklotymiczna dla i =1 to {1, 2, 4, 8, 16, …}, natomiast warstwa cyklotymiczna dla i = 3 to {3, 6, 12, 24, …}.

Wielomian minimalny m_i dla elementu ciała α^i to wielomian najniższego stopnia taki, że α^i jest pierwiastkiem tego wielomianu. Wielomian minimalny dla danej warstwy cyklotymicznej to wielomian minimalny, którego pierwiastkami są elementy tej warstwy.

Wielomiany minimalne dla ciała GF (2^8) w “Kodach korekcyjnych i kryptografii” W. Mochnackiego (str. 39)

```
minimal_polynomials = {
    1: [1, 0, 0, 0, 1, 1, 1, 0, 1],   # m1
    3: [1, 0, 1, 1, 1, 0, 1, 1, 1],   # m3
    5: [1, 1, 1, 1, 1, 0, 0, 1, 1],   # m5
    7: [1, 0, 1, 1, 0, 1, 0, 0, 1],   # m7
    9: [1, 1, 0, 1, 1, 1, 1, 0, 1],   # m9
```

```
    11: [1, 1, 1, 1, 0, 0, 1, 1, 1],  # m11
    13: [1, 0, 0, 1, 0, 1, 0, 1, 1],
    15: [1, 1, 1, 0, 1, 0, 1, 1, 1],
    17: [1, 0, 0, 1, 1],
    19: [1, 0, 1, 1, 0, 0, 1, 0, 1],
    21: [1, 1, 0, 0, 0, 1, 0, 1, 1],
}
```

Wykorzystałyśmy wspomniane wyżej wielomiany minimalne do utworzenia wielomianu generującego poprzez mnożenie przez siebie wielomianów minimalnych od m1 do m21.

Listy te należy odczytywać następująco: pozycja po lewej stronie listy decyduje o najwyższej potędze x, a pozycja po prawej stronie listy o najniższej potędze.

1 lub 0 to indeks przy danym x – na przykład wielomian zapisany na liście jako [1, 0, 0, 1, 1] to inaczej x^4+x+1

```
def generate_generator_polynomial(self):
    g = [1]
    for i in range(1, 2 * self.t + 1, 2):
        if i in self.minimal_polynomials:
            m = self.minimal_polynomials[i]
            g = self.multiply_polynomials(g, m)
    return g
```

4. Implementacja kodera:

```
def encode(self, message):
    if len(message) != self.k:
        raise ValueError(f"Message must be exactly {self.k} bits.")
    padded_message = message + [0] * (self.n - self.k)
    remainder = self.compute_remainder(padded_message,
self.generator_polynomial)
    encoded_message = message + remainder
    return encoded_message
```

Pomnożenie wiadomości razy x^{84} (84 to stopień wielomianu generującego) – w kodzie jest to realizowane poprzez dodanie 84 zer na końcu. Obliczenie reszty z dzielenia wyniku mnożenia opisanego wyżej przez wielomian generujący Dodanie reszty z dzielenia do wiadomości pomnożonej przez x^{84}, ponieważ na końcu wiadomości są same zera, to na miejsca o odpowiednim stopniu we wiadomości albo zostaną dodane jedynki – jeśli znajdują się w tym samym stopniu w reszcie z dzielenia, albo zostanie pozostawione zero – jeśli na odpowiadającej pozycji (liczonej od najmłodszego bitu) w reszcie z dzielenia występuje zero. Funkcja zwraca zakodowaną wiadomość.

5. Implementacja dekodera prostego

Dekoder oblicza syndrom (reszta z dzielenia słowa kodowego przez wielomian generujący), następnie obliczana jest jego waga Hamminga w(s):

- Jeśli waga syndromu (liczba jedynek) jest równa zeru, to oznacza, że odebrany wektor jest poprawny, tzn. nie występują żadne błędy.
- Jeśli waga syndromu jest większa od zdolności korekcyjnej kodu (w(s)>t), oznacza to, że błędy znajdują się w części informacyjnej wektora kodowego, a aby je skorygować należy przesunąć cyklicznie wektor odebrany o jedną pozycję w dowolnym kierunku, obliczyć syndrom oraz jego

wagę Hamminga a następnie sprawdzić, czy w(s)>t. Jeśli tak to kontynuujemy powyższe czynności do momentu, gdy w(s)≤t lub gdy wykonamy k+1 przesunięć cyklicznych (wtedy błędy nie są korygowalne).

– Jeśli waga syndromu jest mniejsza lub równa zdolności korekcyjnej (w(s)≤t), oznacza to, że błędy znajdują się w części kontrolnej wektora kodowego oraz jest możliwe poprawienie wektora odebranego poprzez wykonanie operacji XOR na wektorze odebranym i syndromie oraz przesunięcie cykliczne w przeciwnym kierunku niż w przypadku, gdy w(s)<t.

```
 def decode_with_error_correction(self, received_codeword):
    shifts = 0
    max_shifts = len(received_codeword)

    while shifts < max_shifts:
        # Obliczamy syndrom
        syndrome = self.compute_remainder(received_codeword,
self.generator_polynomial)
        weight = sum(syndrome)  # Waga syndromu (liczba jedynek)

        # Jeśli waga syndromu <= t, dokonaj korekcji
        if weight <= self.t:
            for i in range(len(syndrome)):
                received_codeword[-len(syndrome) + i] ^= syndrome[i]  #
Odejmowanie syndromu (XOR)

            # Przywracamy pierwotną postać przez przesunięcie w lewo
            for _ in range(shifts):
                received_codeword = received_codeword[1:] +
[received_codeword[0]]
            return received_codeword

        # Waga syndromu > t, przesuwamy w prawo
        received_codeword = [received_codeword[-1]] + received_codeword[:-1]
        shifts += 1

    # Jeśli nie można poprawić, zgłoszamy błąd
    raise MessageUnfixableError("Błędy są niekorygowalne.")


def recover_original_message(self, decoded_codeword):
       if len(decoded_codeword) != self.n:
        raise ValueError(f"Poprawiony kod musi mieć dokładnie {self.n}
bitów.")

    # Podziel kod przez wielomian generujący
    remainder = self.compute_remainder(decoded_codeword,
self.generator_polynomial)
    if any(remainder):
        raise ValueError("Kod nie jest wielokrotnością wielomianu
generującego.")

    # Wyciągnij pierwsze k bitów jako oryginalną wiadomość
    original_message = decoded_codeword[:self.k]
    return original_message
```

6. Implementacja dekodera pełnego

- Obliczanie syndromów:

```
def poly_evaluate(self, poly, alpha_power):
    value = 0
    for i, coef in enumerate(poly):
        if coef == 1:
            power = (len(poly) - 1 - i) * alpha_power % 255
            value ^= self.gf_pow(power)
    return value
```

Żeby obliczyć syndromy do dekodera pełnego, najpierw obliczamy resztę z dzielenia słowa kodowego przez wielomian generujący, a następnie traktujemy wynik jako wielomian, i podstawiamy do niego jako wartości α^i, dla i od 1 do 22. Dla przykładu: jeśli mamy wielomian 1011 – tj. x^3+x+1, i mamy podstawić do niego α^2, to obliczamy: $(\alpha^2)^3+\alpha^2+1$. Obliczamy po kolei potęgi (wykonujemy mnożenie potęg), a następnie dodajemy uzyskane elementy syndromu do siebie (operacja XOR). W ten sposób otrzymujemy 22 syndromy.

```
def syndrome_test(bch, errors_amount, error_generator,
error_type=error_flip):
    message = [random.randint(0, 1) for _ in range(k)]
    encoded_message = bch.encode(message)
    if not bch.validate_codeword(encoded_message):
        raise EncodingError("Niepoprawny kod.")

    received_message = encoded_message[:]
    errors_array = error_generator(n, errors_amount)
    for error_position in errors_array:
        error_type(received_message, error_position)

    syndromes = [bch.poly_evaluate(received_message, i) for i in range(1, 2 *
bch.t)]
    return syndromes
```

- Algorytm Berlekampa-Masseya

Algorytm wyznacza wielomian lokatorów błędów, który opisuje pozycje błędów w zakodowanej wiadomości. Iteracyjnie oblicza niespójności oraz aktualizuje wielomiany Lambda i pomocniczy B.

```
 def berlekamp_massey(self, syndromes):
    """
    syndromy: lista liczb całkowitych (0..255), długości 2t (na przykład,
2*11=22),
    każdy element to wartość syndromu S_i w GF(2^8).
    Zwraca (Lambda, L), gdzie Lambda to lista współczynników wielomianu
    lokatorów błędów (od najmniejszego do największego), L to jego stopień.
    """
    L = 0
    m = 1
    b = 1  # ostatnia niezerowa niespójność
    Lambda = [1] + [0] * (len(syndromes))  # Wielomian Lambda(x), długość z
zapasem
    B = [1] + [0] * (len(syndromes))  # Pomocniczy wielomian B(x)

    for i in range(len(syndromes)):
        # 1) Obliczamy niespójność discrepancy = S_i + sum_{j=1..L} (Lambda_j
* S_{i-j})
        # W GF(2^8) dodawanie = XOR
        delta = syndromes[i]
        for j in range(1, L + 1):
            if Lambda[j] != 0 and (i - j) >= 0:
```

```
                delta = gf_add(delta, gf_mul(Lambda[j], syndromes[i - j]))

        # 2) Jeśli discrepancy == 0, nic nie robimy, po prostu m++
        if delta != 0:
            # 3) Tymczasowa kopia Lambda, aby zaktualizować B w razie
potrzeby
            T = Lambda[:]

            # Lambda = Lambda + delta/b * x^m * B
            inv_b = self.gf_inv(b)  # b^-1
            factor = gf_mul(delta, inv_b)
            # Przesuwamy B(x) o m pozycji
            for k in range(len(syndromes) - m):
                if B[k] != 0:
                    # Dodajemy factor * B[k] na pozycję k+m
                    Lambda[k + m] = gf_add(Lambda[k + m],
                                           gf_mul(factor, B[k]))

            if 2 * L <= i:
                L_new = i + 1 - L
                L = L_new
                B = T
                b = delta
                m = 1
            else:
                m += 1
        else:
            m += 1

    # Teraz mamy wielomian Lambda o wymaganej długości L+1
    # Przycinamy "ogon" niepotrzebnych zer
    Lambda = Lambda[:L + 1]
    return Lambda, L
```

Wielomian lokatorów błędów Lambda jest początkowo ustawiony jako [1] (brak błędów). Pomocniczy wielomian B oraz zmienne L (stopień Lambda), b (ostatnia niespójność) i m (licznik przesunięć) są inicjalizowane wartościami początkowymi.
Dla każdego syndromu, obliczana jest niespójność delta jako suma:
$delta = syndromes[i] + \sum_{j=1}^{L}(Lambda[j] * syndromes[i-j])$,
gdzie dodawanie i mnożenie odbywają się w GF(2^8).

- Algorytm Chiena

Algorytm przeszukuje elementy pola GF(2^8), aby znaleźć korzenie wielomianu Lambda. Każdy korzeń odpowiada pozycji błędu w kodzie.

```
def chien_search(self, Lambda):
    """
    Szukamy korzeni wielomianu lokatorów błędów Lambda(x) w GF(2^8).
    Lambda: lista współczynników (Lambda[0], Lambda[1], ..., Lambda[L])
    gdzie Lambda[j] jest elementem GF(2^8) w zakresie [0..255].
    Zwraca listę indeksów pozycji, w których wykryto błąd.
    """
    # stopień wielomianu
    error_positions = []
    # Przeglądamy i od 0 do n-1 (n=255 dla BCH(255, k))

    for i in range(self.n):
        # Obliczamy Lambda(alpha^i):
        # val = sum_{j=0..L} [ Lambda[j] * alpha^(i*j) ]
        val = 0
```

```
        for j in range(len(Lambda)):
            if Lambda[j] != 0:  # jeśli 0, mnożenie i tak da 0
            # (i*j) % 255 ponieważ alpha^255 = 1
                power = (i * j) % 255
                val = gf_add(val, gf_mul(Lambda[j], self.gf_pow(power)))
        if val == 0:
            error_positions.append(i-1)
    return error_positions
```

Przygotowywana jest lista error_positions, która będzie przechowywać indeksy wykrytych błędów. Iteracja odbywa się przez wszystkie możliwe pozycje i w zakodowanym słowie (od 0 do n−1), gdzie n to długość słowa kodowego). Dla każdej pozycji obliczana jest wartość wielomianu lokatorów błędów: $val = \sum_{j=0}^{L}(Lambda[j] * a^{ij})$,
gdzie mnożenie i dodawanie odbywają się w GF(2^8), a potęgi są redukowane modulo 255 (bo α^{255}=1). Jeśli val == 0, oznacza to, że na pozycji i wystąpił błąd. Pozycja ta jest dodawana do error_positions.

- Korekcja błędów

Po zidentyfikowaniu pozycji błędów następuje ich korekcja poprzez zmianę odpowiednich bitów w kodzie na ich przeciwną wartość (operacja XOR).

```
def decode_with_full_correction(self, received_codeword):
    syndromes = self.calculate_syndromes(received_codeword)
    Lambda, L = self.berlekamp_massey(syndromes)
    error_positions = self.chien_search(Lambda)
    if len(error_positions) > self.t:
        raise MessageUnfixableError("Błędy są niekorygowalne.")
    corrected_codeword = received_codeword[:]
    for error_position in error_positions:
        corrected_codeword[error_position] ^= 1
    syndromes = self.calculate_syndromes(corrected_codeword)
    if any(syndromes):
        raise MessageUnfixableError("Błędy są niekorygowalne.")
    # Wyciągnij pierwsze k bitów jako oryginalną wiadomość
    original_message = corrected_codeword[:self.k]
    return original_message
```

Najpierw wywoływana jest funkcja calculate_syndromes, która sprawdza, czy w odebranym słowie kodowym występują błędy. Na podstawie syndromów, algorytm Berlekampa-Masseya wyznacza wielomian Lambda. Algorytm Chiena przeszukuje wszystkie możliwe pozycje w kodzie, aby określić, gdzie występują błędy. Odebrane słowo kodowe received_codeword jest korygowane przez odwrócenie (operacja XOR) bitów na pozycjach wskazanych przez error_positions. Po korekcji ponownie obliczane są syndromy. Jeśli wszystkie syndromy wynoszą zero, oznacza to, że wszystkie błędy zostały poprawione. W przeciwnym razie zgłaszany jest błąd, że wiadomość nie jest możliwa do naprawienia.

4. Testy i generowanie błędów w słowie kodowym

Wykonałyśmy testy dla czterech typów błędów:

- błędy losowe – występujące na losowych, niepołączonych ze sobą miejscach
- błędy typu wiązka:
  - wiązka wysoka – wszystkie bity w danym przedziale są zamieniane na 1
  - wiązka niska – wszystkie bity w danym przedziale są zamieniane na 0
  - wiązka “flip” – wszystkie bity w danym przedziale są odwracane.

Warto wspomnieć już w tym miejscu o tym, jak będzie to wpływało na wyniki testów – nasz kod teoretycznie ma zdolność korekcyjną t = 11, to znaczy, że nie powinien być w stanie poprawiać więcej błędów. Natomiast w naszych wynikach testów da się zobaczyć przypadki, gdzie poprawiana jest na przykład 12-krotna albo 30-krotna wiązka wysoka lub niska. Wiąże się to z charakterystyką tego typu błędu – wiązka wysoka 30-krotna wcale nie oznacza, że 30 bitów będzie przekształconych, oznacza to zaś, że na obszarze 30 bitów, wszystkie z nich będą teraz jedynkami, jeśli jeszcze nie były. To może oznaczać, że zostanie przekształcone na przykład tylko 15 bitów, lub tylko 5, bo pozostałe już były jedynkami, więc wcale nie są zmieniane. Dlatego też w niektórych przypadkach testy wychodziły poprawnie dla wiązek (niskich i wysokich) o większej krotności niż nasza zdolność korekcyjna.

Poniżej pokazano kod odpowiedzialny za generowanie błędów w słowie kodowym oraz kod odpowiedzialny za sprawdzanie czy błędy zostały odpowiednio poprawione.

```
 def error_generator_random(n, errors_amount):
    errors_array = []
    for i in range(errors_amount):
        while len(errors_array) < i + 1:
            error_position = random.randint(0, n - 1)
            if error_position in errors_array:
                continue
            errors_array.append(error_position)
    return errors_array

def error_generator_burst(n, errors_amount):
    errors_array = []
    error_position = random.randint(0, n - 1)
    for i in range(errors_amount):
        errors_array.append((error_position+i) % n)
    return errors_array


def error_flip(message, position):
    message[position] ^= 1

def error_to_high(message, position):
    message[position] = 1

def error_to_low(message, position):
    message[position] = 0
```

Jak opisano wcześniej, błędy tworzone są na losowych pozycjach albo we wiązkach, gdzie wszystkie pozycje albo są zamieniane na jedynki, albo na zera, albo odwracane (operacja xor w funkcji error_flip).

```
def decoder_test(bch, errors_amount, error_generator, error_type=error_flip):
    message = [random.randint(0, 1) for _ in range(k)]
    encoded_message = bch.encode(message)
    if not bch.validate_codeword(encoded_message):
        raise EncodingError("Niepoprawny kod.")

    received_message = encoded_message[:]
    errors_array = error_generator(n, errors_amount)
    for error_position in errors_array:
        error_type(received_message, error_position)
    corrected_codeword = bch.decode_with_error_correction(received_message)
    original_message = bch.recover_original_message(corrected_codeword)
    if message != original_message:
        raise MessagesNotMatchError("Odzyskana wiadomość nie zgadza się z
oryginalną.")
```

Wyniki testów załączone są w poniższej tabeli. Korzystałyśmy z dwóch sposobów na obliczenie liczby testów dla danego typu i krotności błędu:

- jeśli liczba kombinacji wynosiła poniżej 1000 dla danej krotności i typu to została ona obliczana bezpośrednio i tyle właśnie testów wykonywałyśmy, np. istnieje 255 kombinacji na wprowadzenie błędu pojedynczego losowego oraz 253 kombinacje na wprowadzenie wiązki 3-krotnej (255-3+1).
- jeśli liczba kombinacji była wysoka, to robiono testy, aż wyniki kolejnych testów będą do siebie zbliżone. Algorytm wyglądał następująco: zaczynając od 100 testów, przy każdej kolejnej próbie dodawano 100 do tej liczby. Jeśli spośród wyników trzech następujących bezpośrednio po sobie testach różnice między dowolnymi dwoma wynikami testów (procentowe) różnią się od siebie o mniej niż jeden procent, to uznałyśmy to za wystarczającą liczbę testów. W ten sposób minimalną liczbą testów jest 300 dla jeśli wynik testów cały czas wychodzi taki sam – na przykład kiedy mamy 100% skuteczności lub 0% skuteczności – zaczynamy od 100 testów, z nich uzyskujemy na przykład wynik 0%, potem próbujemy kolejne 100 testów, znowu otrzymujemy wynik 0% - dwa wyniki nie różnią się od siebie o więcej niż jeden procent, wykonujemy testy trzeci raz, jeśli również otrzymamy ten sam wynik, to kryterium jest spełnione, i wystarczy nam 300 testów żeby otrzymać wiarygodny wynik.

5. Wyniki testów:

1. Błędy losowe

Krotność błędu	Dekoder prosty			Dekoder pełny		
	Liczba testów	Liczba poprawnie odzyskanych przypadków	Współczynnik poprawnie odzyskanych przypadków	Liczba testów	Liczba poprawionych odpowiednio przypadków	Współczynnik poprawnie odzyskanych przypadków
1	255	255	100%	300	300	100%
2	600	393	65,5%	300	300	100%
3	900	286	31,8%	300	300	100%
4	600	93	15,5%	300	300	100%
5	400	28	7%	300	300	100%
6	600	10	1,7%	300	300	100%
7	300	1	0,3%	300	300	100%
8	300	1	0,3%	300	300	100%
9	300	0	0%	300	300	100%
10	300	0	0%	300	300	100%
11	300	0	0%	300	300	100%
12	300	0	0%	300	0	0%
30	300	0	0%	300	0	0%

2. Błędy typu wiązka: niska/wysoka

Wyniki testów dla wiązki niskiej były bardzo zbliżone do wyników testów dla wiązki wysokiej, dlatego wyniki te zostały przedstawione jako jedne.

Krotność błędu	Dekoder prosty			Dekoder pełny		
	Liczba testów	Liczba poprawnie odzyskanych przypadków	Współczynnik poprawnie odzyskanych przypadków	Liczba testów	Liczba poprawionych odpowiednio przypadków	Współczynnik poprawnie odzyskanych przypadków
1	255	255	100%	255	255	100%
2	254	254	100%	254	254	100%
3	253	253	100%	253	253	100%
4	252	252	100%	252	252	100%
4	251	251	100%	251	251	100%
6	250	250	100%	250	250	100%
7	249	249	100%	249	249	100%
8	248	248	100%	248	248	100%
9	247	247	100%	247	247	100%
10	246	246	100%	246	246	100%
11	245	245	100%	245	245	100%
12	244	244	100%	244	244	100%
30	226	21	9,7%	226	23	10,1%

3. Błędy typu wiązka: odwrócenie bitów

Krotność błędu	Dekoder prosty			Dekoder pełny		
	Liczba testów	Liczba poprawnie odzyskanych przypadków	Współczynnik poprawnie odzyskanych przypadków	Liczba testów	Liczba poprawionych odpowiednio przypadków	Współczynnik poprawnie odzyskanych przypadków
1	255	255	100%	255	255	100%
2	254	254	100%	254	254	100%
3	253	253	100%	253	253	100%
4	252	252	100%	252	252	100%
4	251	251	100%	251	251	100%
6	250	250	100%	250	250	100%
7	249	249	100%	249	249	100%
8	248	248	100%	248	248	100%
9	247	247	100%	247	247	100%
10	246	246	100%	246	246	100%
11	245	245	100%	245	245	100%
12	244	0	0%	244	0	0%
30	226	0	0%	226	0	0%

6. Wnioski

Można zauważyć, że wyniki testów dla dekodera prostego nie są pozytywne dla błędów losowych – dekoder ten nie naprawia wszystkich błędów nawet jeśli mieszczą się one w zdolności korekcyjnej kodu. Jest to spowodowane tym, że dekoder ten, ze względu na swoją charakterystykę, zawodzi wtedy, kiedy błędy są od siebie odsunięte o więcej niż n-k (84) bity, ponieważ mogą wystąpić wtedy przypadki, kiedy w części informacyjnej występuje błąd, którego nie można skorygować. Więcej jest takich kombinacji, im większa krotność błędu, dlatego też dekoder prosty nie naprawia poprawnie błędów wielokrotnych, nawet jeśli krotność błędu jest mniejsza niż zdolność korekcyjna kodu. Zdolność korekcyjna dekodera uproszczonego jest natomiast dobrze widoczna w błędach typu wiązka, w których wszystkie bity są odwrócone. Ze względu na brak odległości między poszczególnymi błędami, dekoder potrafi je naprawić w większych krotnościach niż gdy błędy są losowe. Natomiast gdy błędów jest więcej, niż zdolność korekcyjna kodu, to w ani jednym przypadku nie są one poprawnie naprawiane. Znacznie skuteczniejszy okazał się być dekoder pełny, do którego wykorzystano algorytm Berlekampa-Masseya i algorytm Chiena. Dekoder pełny poprawiał 100% przypadków każdej krotności błędów w swojej zdolności korekcyjnej. Można jednak zauważyć, że przy błędach typu “wiązka” obydwa dekodery mają bardzo zbliżoną do siebie skuteczność - obydwa poprawiają 100% przypadków błędów typu “wiązka” w swojej zdolności korekcyjnej.

7. Bibliografia

- Costello D.J., Lin S., Error Control Coding: Fundamentals and Applications, Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1983.
- Mochnacki W., Kody korekcyjne i kryptografia, Oficyna Wydawnicza Politechniki Wr