Krzysztof Nowakowski 235053

Piotr Borowski 234996

Grzegorz Pietrucha 235052

Mgr inż. Piotr Semberecki

Termin: pn 11:15 TP

Zadanie projektowe:
Niezawodność i diagnostyka układów cyfrowych

Temat: „Zadania HARQ”

Spis treści

Opis zadania projektowego

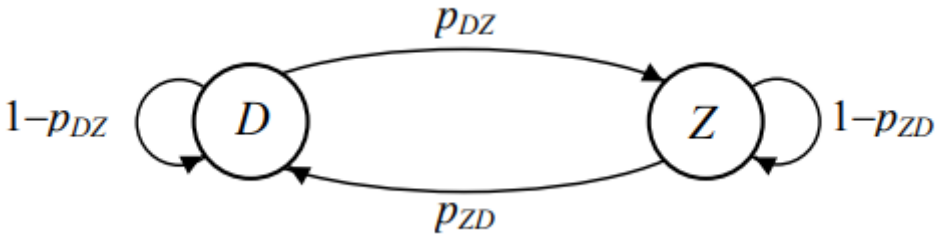
Projekt polegał na stworzeniu programu do symulacji przesyłania danych w postaci pakietów, zakłócaniu ich, detekcji błędów, prób napraw, dekodowaniu itp. Naszymi zadaniami było:

- Implementacja protokołu STOP-AND-WAIT + SELECTIVE REPEAT,
- Implementacja systemu hybrydowego (HARQ) Hybrid Automatic Repeat Request typ I,
- Implementacja modeli BSC + protokół SAW + bit parzystości + TMR (najlepiej 5 bitów powtórzonych),
- Implementacja modelu Gilberta, protokół SR,
- Implementacja kodu Hamminga (7,4), sumy CRC32,
- Optymalizacja protokołów, wizualizacja działania symulacji.

Opis protokołów, modeli oraz ich implementacja

Model Gilberta

Model Gilberta kanału jest przykładem przyczynowego modelu dwustanowego, opierający się na teorii łańcuchów Markowa. Zakłada się, że błędy mogą występować w każdym ze stanów, w D – dobrym oraz Z – złym. W stanie dobrym prawdopodobieństwo wystąpienia błędów jest bliskie zera, jednocześnie w stanie dobrym z prawdopodobieństwem dużo większym od prawdopodobieństwa w stanie D.

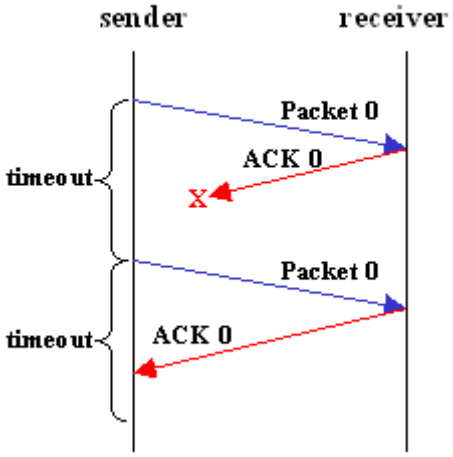


Dodatkowo przejścia ze stanu D do Z oraz ze stanu Z do D mają swoje prawdopodobieństwa, które przy niewielkich wartościach gwarantują występowanie błędów seryjnych o średniej długości 1 / P_{DZ} .

Implementacja modelu Gilberta opiera się na zmiennych posiadających wartości prawdopodobieństwa odpowiednich przeskoków i zakłóceń w stanach D i Z oraz zmienna binarna określająca stan modelu (Dobry czy Zły). Samo zakłócanie odbywa się w sposób analogiczny do modelu BSC.

Stop And Wait

Stop And Wait (SAW) protokół ten wstrzymuje transmisje po każdej wysłanej ramce i czeka na potwierdzenie (ACK) od odbiornika. Jeżeli ramka zostanie odrzucona (NAK) to zostaje ona retransmitowana dopóki nadajnik nie otrzyma ACK. Jeżeli pakiet został odrzucony więcej niż wyznaczona wartość, najczęściej od 5 do 10 razy, to transmisja zostaje przerwana ze względu na złą jakość połączenia.



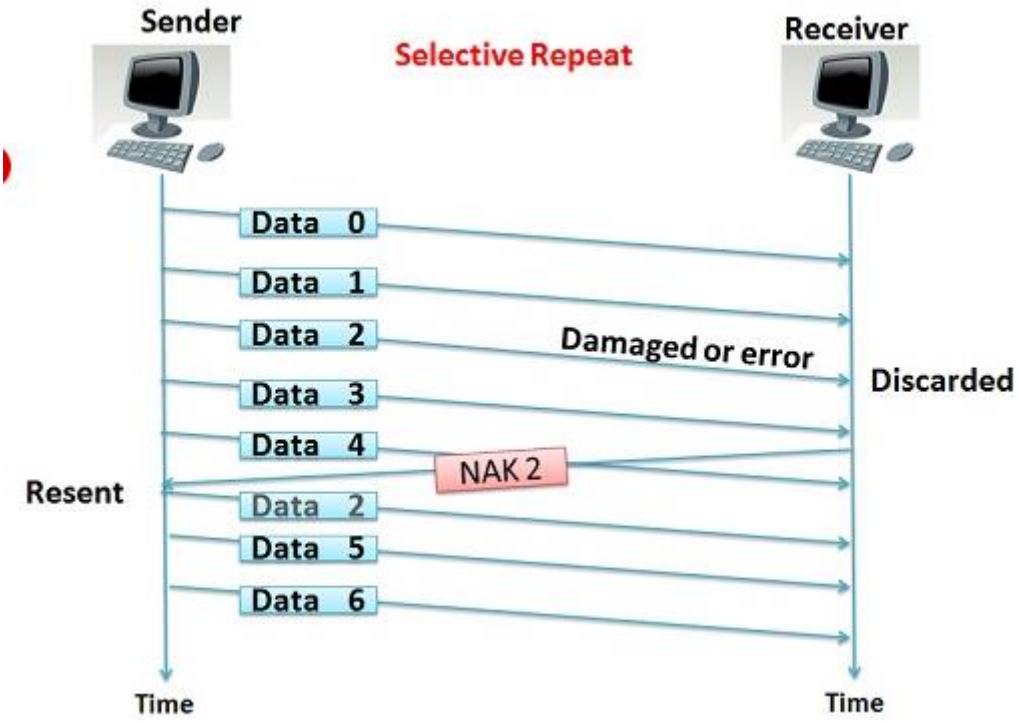
Kod SAW

```
1.  while(sended < packets)
2.          if(self.isBSC):
3.              packet = self.channelModel.addBSCNoise(self.sourcePackages[sended])
4.          else:
5.              packet = self.channelModel.addGilbertNoise(self.sourcePackages[sended])
6.
7.          while (self.protocol.isValid(packet) == False):
8.            if (self.isBSC):
9.               packet = self.channelModel.addBSCNoise(self.sourcePackages[sended])
10.           else:
11.             packet = self.channelModel.addGilbertNoise(self.sourcePackages[sended])
12.
13.             self.destPackages[sended] = packet
14.             sended += 1
15.
```

Nie jest to pełny kod wykorzystany w projekcie, chodzi tutaj o proste przedstawienie protokołu. Kod wykonuje się w pętli dla wszystkich pakietów znajdujących się w pliku. Każdy pakiet zostaje zakłócony przez funkcje addBSCNoise lub addGilbertNoise, które są odpowiednimi modelami kanałów. Funkcja isValid imituje odbiornik, który sprawdza odpowiednim protokołem poprawność pakietu. Jeżeli pakiet jest wykryty jako niepoprawny to zostaje on znowu przesyłany przez kanał, stąd ponowne wywołanie funkcji zakłócającej.

Selective Repeat

Selective Repeat (SR) jest to protokół pozwalający na ciągłe wysyłanie pewnej ilości ramek (liczba ta jest określona przez szerokość okna) bez potrzeby czekania na pojedyncze sygnały ACK od każdej ramki. Jeżeli jakaś ramka będzie potrzebowała retransmisji, wtedy odbiornik wysyła sygnał NAK do nadajnika. Zostanie ona retransmitowana niezależnie od innych ramek. W naszym symulatorze dzieje się to w następnym oknie, zaraz po odebraniu sygnału NAK.



Kod SR

```
1.          while (len(errorBuf) > 0):
2.              packet = errorBuf.pop()
3.              index = errorIndexes.pop()
4.              if (self.protocol.isValid(packet)):
5.                  self.destPackages[index] = packet
6.              else:
7.                  errors.append(index)
8.          while (len(errors) > 0):
9.              index = errors.pop()
10.             if (self.isBSC):
11.                 errorBuf.append(self.channelModel.addBSCNoise(self.sourcePackages[index]))
12.             else:
13.                 errorBuf.append(self.channelModel.addGilbertNoise(self.sourcePackages[index]))
14.             errorIndexes.append(index)
```

Przedstawiony powyżej kawałek kodu dotyczy bufora błędów w protokole Selective Repeat. Pakiety są przesyłane na początku analogicznie do protokołu SAW, lecz po zakłóceniu i sprawdzeniu poprawności pakietu, pakiet ,który był niepoprawny zostaje oznaczony jako błędny (errors). Następnie następuje próba retransmisji pakietów znajdujących się w buforze błędów (errorBuf). Po

próbie retransmisji następuje ponowne zakłócenie pakietów oznaczonych jako błędne (errors) i dodanie ich do bufora błędów, dzięki czemu zostaną ona retransmitowane w następnym oknie.

Parity bit – kontrola parzystości

Metoda ta polega na tym, że do wysyłanej wiadomości, jest dodawany dodatkowy bit, ustawiany w taki sposób, aby liczba jedynek była zawsze parzysta. Umożliwia wykrywanie pojedynczych przekłamań. Kontrola parzystości opiera się na sprawdzeniu parzystości sumy bitów wiadomości. Bitem parzystości nazywa się bit kontrolny, który przyjmuje wartość 1, gdy liczba jedynek w przesyłanej wiadomości jest nieparzysta, lub 0, gdy parzysta.

Przy zamianie 2 bitów, metoda już nie wykryje problemu, jednak jest to bardzo proste i szybkie rozwiązanie w implementacji – można to nawet zrobić sprzętowo.

Rozwiązanie w kodzie:

Dodawanie bitu parzystości polega na prostym zliczeniu wystąpień jedynek w danej paczce binarnej, następnie sprawdzenie jest to parzysta liczba, lub nie. Dopisanie 0 lub 1 powoduje, że kod będzie miał parzystą liczbę jedynek.

```
1.  def addParityBit(self, bitList):
2.      counter = 0
3.      for bit in bitList:
4.          if(bit == '1'):
5.              counter += 1
6.
7.      # sprawia ze liczba jedynek bedzie parzysta
8.      if(counter % 2 == 0):
9.          bitList.append('0')     # 0 jesli ilosc jedynek jest parzysta
10.     else:
11.         bitList.append('1')     # 1 jesli ilosc jedynek jest parzysta
12.
13.     return bitList
```

Sprawdzenie poprawności, działa podobnie jak dodawanie bitu parzystości. Zliczane są wszystkie wystąpienia „1”, wraz z ostatnim dodatkowym bitem.

```
1.  def isValid(self, bitList):
2.      counter = 0
3.      for bit in bitList:
4.          if (bit == '1'):
5.              counter += 1
6.
7.      if (counter % 2 == 0):
8.          return True
9.      else:
10.         return False
```

TMR – Potrójna redundancja

Model, polegający na trzykrotnym zwielokrotnieniu każdego występującego bitu w wiadomości. Przykładowo dla pakietu „101”, wynikowym pakietem będzie: „111000111”. Wprowadza to

oczywiście dodatkową nadmiarowość danych (która dodatkowo jest skłonna do zakłóceń), jednak TMR ma możliwości samo naprawcze. Tabela przedstawia, jak działa technika odkodowywania:

111 Wszystkie poprawne	➔	1
110 Jeden zamieniony	➔	1
001 Dwa zamienione	➔	***0***
110 Dwa zamienione	➔	***1***
001 Jeden zamieniony	➔	0
000 Wszystkie poprawne	➔	0

Jak widać, przy zamianie 2 bitów, odkodowana wiadomość, będzie z błędem.

W naszej implementacji zastosowaliśmy nie potrójną redundancję, a pięciokrotną. Czyli występuje pięciokrotne zwielokrotnienie każdego bitu. Co zwiększa szansę na naprawę uszkodzonych danych.

Rozwiązanie w kodzie:

Kodowanie polega na odczytywaniu bitu po bicie i następnie tworzeniu nowej listy z pięciokrotnie powielonym danym bitem.

```
1.  def codeTMR(self, bitList):
2.      codedBitList = []
3.      for bit in bitList:
4.          for i in range(0, 5):
5.              codedBitList.append(bit)
6.
7.      return codedBitList
```

Dekodowanie już jest trochę dłuższe. Należy policzyć ilość jedynek, następnie, jeśli ich będzie więcej niż 2 (2 bity mogą być błędne – które da się naprawić), to znaczy, że jest szansa, że przed zakodowaniem i poddaniem ich procesu zakłócania, była tam jedynka, a jeśli nie, to zero.

```
1.  def decodeTMR(self, bitList):
2.      decodedBitList = []
3.      count = 0
4.      amount = 0
5.      for bit in bitList:
6.          if(bit == '1'): # Liczy ilosc jedynek
7.              amount += 1
8.
9.          count += 1
10.         if(count == 5): # Po każdych 5 zliczonych bitach, sprawdza co powinno zostac
    odczytane
11.             if(amount >= 3):
12.                 decodedBitList.append('1')
13.             else:
14.                 decodedBitList.append('0')
15.
16.             amount = 0
17.             count = 0
18.
19.     return decodedBitList   # zwraca odkodowaną liste
```

BER – Bit Error Rate – współczynnik błędów

BER jest to współczynnik ilości otrzymanych błędnych bitów do całkowitej liczby otrzymanych bitów, wysłanych podczas ustalonego interwału czasowego. Interwał czasowy może być zastąpiony, nie jako

stała liczba, tylko np. do otrzymania wszystkich wysłanych pakietów. BER często jest wyrażany jako procent.

Przykład:

Wysłano: „0 1 1 0 0 0 1 0 1 1” i odebrano „0 0 1 0 1 0 1 0 0 1” sekwencję. Jak widać, w tym przypadku, zostały zamienione 3 bity (3 errory). BER jest stosunkiem niepoprawnych danych do wszystkich wysłanych danych. Aby to obliczyć, wystarczy podzielić ilość błędnych bitów przez ilość wszystkich bitów: 3/10. Wynik można przedstawić jako 0.3 lub jako 30%.

Rozwiązanie w kodzie:

Rozwiązanie jest bardzo proste. Tylko wystarczy sprawdzać bit po bicie wiadomość oryginalną oraz tą odebraną (po podaniu jej zakłóceniom).

```
1.  counterError = 0
2.  ind = 0
3.  for bit in bitListFinal:
4.      if (bit != bitList[ind]):
5.          counterError += 1
6.      ind += 1
7.
8.  print("Ilosc blednych bitow wynikowych")
9.  print(counterError)
10. print("Ilosc pakietow ogolem")
11. print(len(bitListFinal))
12. print("BER")
13. print(counterError / len(bitListFinal) * 100)
```

BSC

Model zakłócający dane. Działa na bardzo prostej zasadzie: każdy bit, ma dane prawdopodobieństwo zamiany z 0 na 1 lub z 1 na 0. Generuje to błędy losowe, z zadaną szansą na pojedynczy błąd.

Rozwiązanie w kodzie:

```
1.  def addBSCNoiseBit(self, bit, prop):  #zmiana na przeciwny bit z danym prawdopodobie
    nstwem
2.      if (self.draw(prop)):
3.          if (bit == '0'):
4.              bit = '1'
5.          else:
6.              bit = '0'
7.
8.      return bit
9.
10. def addBSCNoise(self, packet):# BSC DLA PAKIETU
11.     noised = []
12.     for bit in packet:
13.         noised.append(self.addBSCNoiseBit(bit,self.__prop))
14.
15.     return noised
16.
17. def draw(self, propability):  #losowanie czy wystapi zdarzenie z okreslonym prawdopo
    dobienstwem seed -> 0.0 - 1.0
18.     seed = random.random()
19.     if (seed <= propability):
20.         return True
21.     else:
22.         return False
```

CRC – Cykliczny kod nadmiarowy

Metoda polega na tym, ze do ciągu danych dodajemy 3 wyzerowane bity, tworzymy i dopisujemy 4-bitowy dzielnik CRC. Jeżeli mamy 0 nad najstarszą pozycją dzielnika, to przesuwamy dzielnik w prawo o jedną pozycję, aż do napotkania 1, wykonujemy operację XOR pomiędzy bitami dzielnika i odpowiednimi bitami ciągu danych, uwzględniając dopisane 3 bity, wynik zapisujemy w nowej linii poniżej, jeżeli liczba bitów danych jest większa lub równa 4, przechodzimy do kroku 2, 3 najmłodsze bity stanowią szukane CRC, czyli cykliczny kod nadmiarowy.

Rozwiązanie w kodzie:

Na początku dodajemy 3 wyzerowane bity do naszego ciągu danych

```
1.  def addCRC(self, bitList): # wrzucenie trzech 0 na koniec naszej paczki
2.          for i in range(0,3):
3.              bitList.append('0')
4.          return bitList
```

Następnie tworzymy nasz 4-bitowy dzielnik CRC i wykonujemy wszystkie obliczenia potrzebne do znalezienia szukanego CRC, jest to CRC początkowe, które zostaje wykorzystane do ponownego wyliczenia CRC.

```
def computeInitialCRC(self, bitList): #polynomial x^3 + x + 1 --
> 1011, liczymy remainder -> tak jak na wiki
        tmp = [] # robimy kopie bo inaczej mamy wyzeorwany pakiecik
        for bit in bitList:
            tmp.append(bit)
        polynomial = ['1','0','1','1']
        lenInput = len(tmp)
        while '1' in tmp[:lenInput]:
            moveRight = tmp.index('1')
            for i in range(len(polynomial) - 1):
                if polynomial[i] == tmp[moveRight + i]:
                    tmp[moveRight + i] = '0'
                else:
                    tmp[moveRight + i] = '1'
        return tmp[lenInput:]
```

Sprawdzenie poprawności, wykorzystanie wszystkich innych metod CRC, i na koniec sprawdzenie czy w ostatecznym CRC znajdują się jedynki. Jeśli nie, to pakiet jest poprawny, inaczej, wystąpił błąd.

```
def isValid(self, bitList): # metoda isValid
        initialCRC = self.computeInitialCRC(bitList) # liczymy sobie ten pierwszy re
mainder
        self.deleteCRC(bitList)
        tmp = self.computeCRC(bitList, initialCRC) # uzywamy pierwszego remaindera d
o sprawdzenia bledu
        counter = 0
        for bit in tmp:
            if bit == '1':
                counter += 1
        if counter == 0: # jezeli ostatecznie lista bedzie skladala sie z samych 0 l
ub bedzie pusta
            return True # oznacza ze nie ma bledu
        else:
            return False # jest blad
```

Na koniec, po odkodowaniu, usuwamy 3 już niepotrzebne bity z naszego pakietu danych, aby wysłać nasz pakiecik do ostatecznego wyniku.

```
def deleteCRC(self, bitList): # wyrzucenie tych 3 bitow dorzuconych na koncu
        bitList = bitList[:-3]
        return bitList
```

Hamming – liniowy kod korekcyjny, single error correction

Kod Hamminga wykrywa i koryguje błędy polegające na przekłamaniu jednego bitu (ang.) single error correction. Kod Hamminga (7,4) polega na tym, że koduje 4 pozycje informacyjne jako słowo 7-bitowe, dodając 3 bity parzystości. Używając 3 macierzy: macierzy G,H oraz R, potrafimy nasz ciąg danych następująco: zakodować, sprawdzić czy wystąpiły błędy podczas przesyłania, odkodować.

Jeżeli został zmieniony tylko 1 bit, to Hamming jeszcze znajdzie, który bit oraz spróbuje go naprawić. Natomiast, jeśli pojawi się już więcej niż 1 błąd, to Hamming może naprawić zły bit lub kompletnie nie działać poprawnie.

Rozwiązanie w kodzie:

Przez to, że najczęściej wysyłamy paczki o różnych wielkościach, musimy się upewnić, ze do Hamminga lądują paczki o wielkości 4.

```
def createPacket4(self, pack): # robimy z naszych pakiecikow po 8, pakieciki po 4 do Hamminga
        packet = []
        newPacket = []
        counter = 0
        for bit in pack:
            if bit == '1':
                packet.append(1)
                counter += 1
            if bit == '0':
                packet.append(0)
                counter += 1
            if counter == 4:
                newPacket.append(packet)
                counter = 0
                packet = []

        matrix = numpy.asarray(newPacket)
        return matrix
```

W naszym projekcie, paczkami są listy znaków, dlatego musimy upewniać się, że do obliczeń na macierzach używamy liczb. Stworzyliśmy metodę numpyToChar, która jak sama nazwa wskazuje, zmienna ze zmiennej typu numpy, staje się zmienna typu char (string). CharToNumpy, jest to metoda działająca w odwrotna stronę.

```
def numpyToChar(self, enc):
        chA = []
        for row in enc:
            for col in row:
                if col == 1:
                    chA.append('1')
                if col == 0:
                    chA.append('0')
        return chA
```

CharToNumpy:

```
def charToNumpy(self, list):
        row = []
        col = []
        i = 0
        for char in list:
            if(char == '1'):
                col.append(1)
            else:
                col.append(0)

            i += 1

            if(i == 7):
                i = 0
                row.append(col)
                col = []
```

```
17.
18.         return numpy.array(row)
```

Na początku musimy nasz pakiecik zakodować za pomocą macierzy G, czyli z 4-bitowego pakietu, tworzymy 7-bitowy pakiet, dodając 3 bity parzystości do naszych danych. Metoda zwraca listę stringów, jest nam to potrzebne do poprawnego działania reszty kodu.

```
1.  def codeHamming(self, pack):   #uzywajac macierzy G kodujemy nasz pakiecik
2.          newPacket = self.createPacket4(pack)
3.          g = numpy.array([
4.              [1, 1, 1, 0, 0, 0, 0],
5.              [1, 0, 0, 1, 1, 0, 0],
6.              [0, 1, 0, 1, 0, 1, 0],
7.              [1, 1, 0, 1, 0, 0, 1]
8.          ]) #(4,7)
9.          enc = numpy.dot(newPacket, g)%2
10.         charEnc = self.numpyToChar(enc)
11.         return charEnc
```

Po zakodowaniu, musimy sprawdzić, czy podczas przesyłania nasze dane się nie zakłóciły, do tego używamy macierzy H.

```
1.  def parityCheck(self, enc): #uzywamy macierzy H aby stworzyc macierz na ktorej spraw
    dzimy czy sa bledy
2.          enc = self.charToNumpy(enc)
3.          h = numpy.array([
4.              [1, 0, 1, 0, 1, 0, 1],
5.              [0, 1, 1, 0, 0, 1, 1],
6.              [0, 0, 0, 1, 1, 1, 1]
7.          ]) #(3,7)
8.          parch = numpy.dot(h, enc.T)%2
9.          parch = parch.T
10.         charParch = self.numpyToChar(parch)
11.         return charParch
```

Sprawdzenie poprawności, jeśli w podczas parityCheck, macierz zwrócona będzie złożona z samym 0, to wtedy pakiet jest w pełni w porządku. Jeśli wystąpi kilka jedynek, lecz liczba binarna z nich policzona jest mniejsza od wielkości całego pakietu danych, to wtedy mamy kilka możliwości. Wystąpił 1 błąd i Hamming bez problemu go naprawi lub wystąpiło kilka błędów i Hamming błędnie naprawi pakiet, wtedy do pomocy wkracza CRC. Jeżeli liczba binarna jest większa od długości naszego ciągu danych, wtedy wiadomo, że ilość błędów jest nie do ogarnięcia przez Hamminga.

```
1.  def isValid(self, enc):
2.          parch = self.parityCheck(enc)
3.          dlugosc = 0
4.          suma = 0
5.          for bit in parch: # liczymy od 0
6.              if bit == '1':
7.                  number = 1*pow(2, dlugosc) # liczymy z binarki od razu na system dzi
    esietny
8.                  dlugosc += 1
9.                  suma += number
10.             else:
11.                 dlugosc += 1
12.         if suma > len(enc):
13.             return False #sa minimum 2 bledy, hamming juz tego nie ogarnie
14.         else:
15.             return True #nie ma bledu lub jest tylko 1 wiec mozna naprawic
```

Przed odkodowaniem sprawdzana jest możliwość naprawy pakietu i jego naprawa, jeśli jest to możliwe. Odkodowanie w kodzie wygląda następująco.

```
def decodeHamming(self, enc): #uzywamy macierz R do odkodowania
        print("enc przed zmiana: ")
        print(enc)
        print("enc po zmianie:")
        enc[0] = '1'
        #enc[3]='1'
        print(enc)
        r = numpy.array([
            [0, 0, 1, 0, 0, 0, 0],
            [0, 0, 0, 0, 1, 0, 0],
            [0, 0, 0, 0, 0, 1, 0],
            [0, 0, 0, 0, 0, 0, 1]
        ])
        parch = self.parityCheck(enc)
        print("parch po zmianach: ")
        print(parch)
        dlugosc = 0
        suma = 0
        for bit in parch: # liczymy od 0
            if bit == '1':
                number = 1*pow(2, dlugosc) # liczymy z binarki od razu na system dziesietny
                dlugosc += 1
                suma += number
            else:
                dlugosc += 1
        print(suma)
        if suma < len(enc):
            for index, bit in enumerate(enc): # jezeli bylo 0 robimy 1, a z 1 robimy 0
                if index == suma & suma != 0:
                    if enc[index-1] == '1':
                        enc[index-1] = '0'
                    else:
                        enc[index-1] = '1'
        print("enc po poprawie: ")
        print(enc)
        enc = self.charToNumpy(enc)
        dec = numpy.dot(r, enc.T)
        charDec = self.numpyToChar(dec)

        return charDec
```

Metoda testowania

Metoda została przedstawiona poniżej w liście krok po kroku, jako założenia

1. Wczytanie pliku, na którym będą wykonywane operacje symulacji transmisji danych.
2. Wygenerowanie listy pakietów do wysłania (różna ilość bitów).
3. Kodowanie pakietów: 5-krotna redundancja + parity bit, Hamming + CRC, sam Hamming.
4. Ustawienie wartości parametrów prawdopodobieństw, dla modelu BSC / Gilbert.
5. Transmisja danych przez protokół SR (tu z możliwością ustawienia wielkości okna) lub SAW.
6. Podanie liczby retransmisji (spowodowane wykryciem błędów przez parity bit oraz CRC).
7. Jeśli retransmisja danych pakietów wynosi więcej niż 5-10 razy, należy przerwać transmisję i zgłosić błąd – za duża ilość generowanych błędów.
8. Odebrane pakiety, należy odkodować w odwrotnej kolejności co do kodowania (usunięcie parity bit i redundantnych bitów, usunięcie CRC oraz odkodowanie Hammingiem – z ewentualną naprawą pojedynczych błędów).
9. Obliczenie BER
10. Powrót do kroku 2 i powtórzenie testów 100 razy, dla tych samych danych, żeby móc następnie policzyć średni wynik.

Wyniki przedstawić w tabelach, dla różnych ustawień błędów i na ich podstawie następnie wyciągnąć wnioski.

Testy – Wyniki oraz przebieg

Wczytywany plik „test.jpg” zawiera w sobie 535680 bitów i wygląda następująco:

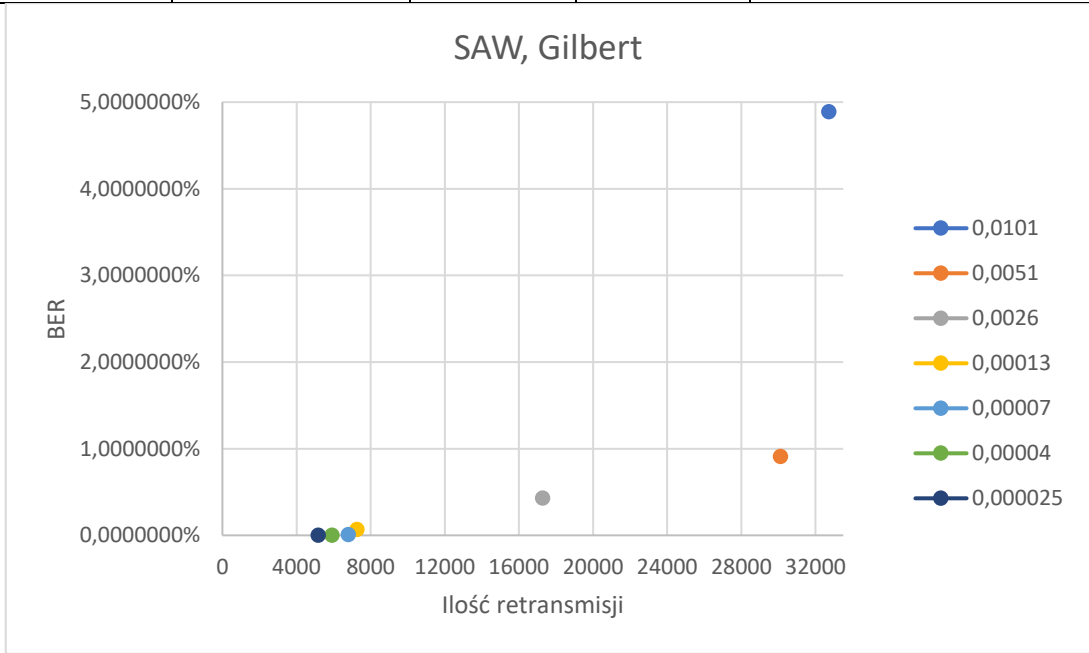
Po każdych testach, tworzymy plik „wynik.jpg”, dzięki któremu możemy zobaczyć jak błędy podczas przesyłania danych zmieniają ostateczny wygląd obrazka. Przykład dla BER = 0,016:

Mała legenda:

- PD – prawdopodobieństwo wystąpienia błędu
- PZ – prawdopodobieństwo wystąpienia błędu
- PDZ – prawdopodobieństwo przejścia stanu z dobrego na zły
- PZD – prawdopodobieństwo przejścia stanu ze złego na dobry

Na początku zajęliśmy się testowaniem przy użyciu protokołu STOP AND WAIT, modelu Gilberta, TMR + Parity Bit.

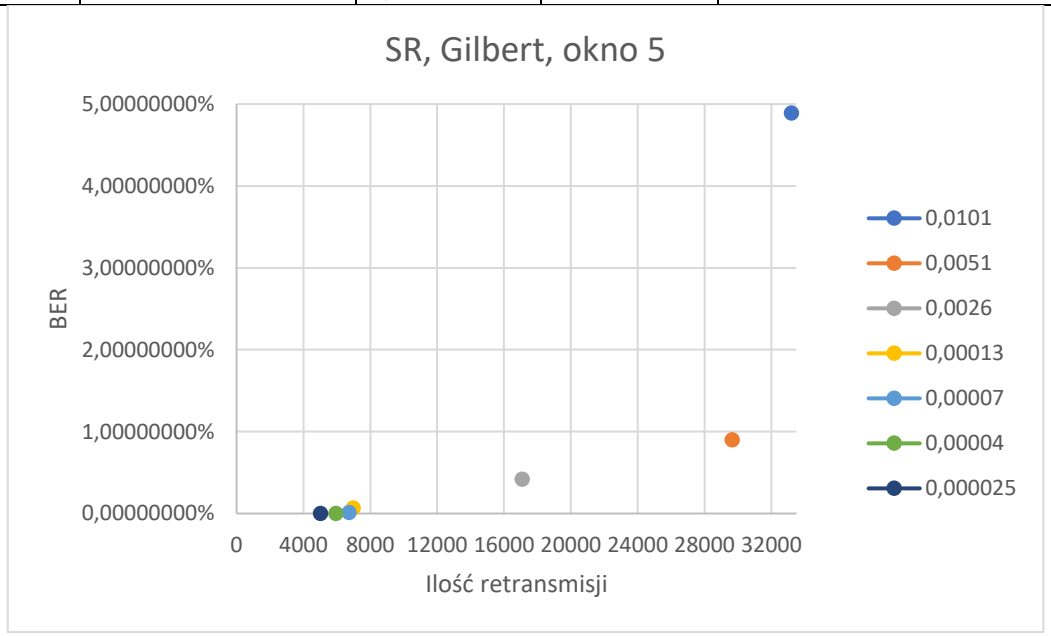
Model Gilberta, protokół STOP AND WAIT									
PD	PZ	PDZ	PZD	Gilbert Stopa Błędu	Plik można otworzyć	BER	Retransmisje	Retr. Powyżej 8 (granica na jeden pakiet)	Całkowita liczba bitów
0,001	0,5	0,02	0,1	0,0101	NIE	4,8900000%	32713	TAK	535680
0,001	0,25	0,02	0,1	0,0051	NIE	0,9100000%	30110	TAK	
0,001	0,25	0,01	0,1	0,0026	NIE	0,4300000%	17281	NIE	
0,001	0,12	0,001	0,01	0,00013	NIE	0,0690000%	7254	TAK	
0,001	0,06	0,001	0,01	0,00007	TAK	0,0082000%	6789	TAK	
0,001	0,03	0,001	0,01	0,00004	TAK	0,0013000%	5911	TAK	
0,001	0,015	0,001	0,01	0,000025	TAK	0,0001200%	5168	NIE	



Zwiększając prawdopodobieństwa wystąpień błędów, mocno zwiększa się BER oraz liczba retransmisji danych. W większości przypadków pliku wynikowego nie można było otworzyć. Otwarte pliki zawierały przekłamania danych, co skutkowało błędnym wyglądem obrazu.

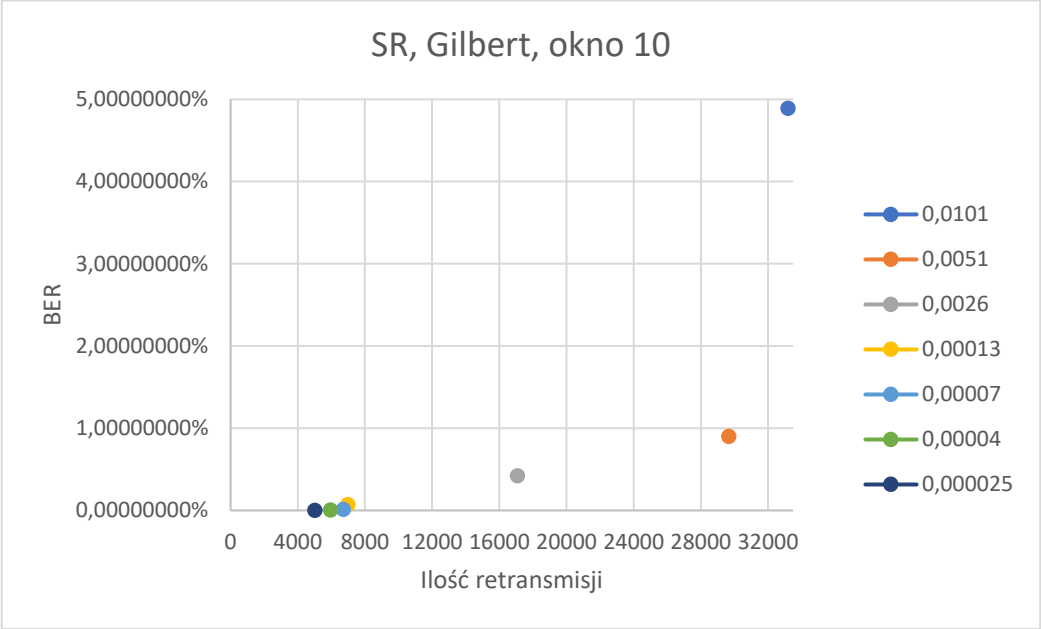
Następnie skupiliśmy się na protokole SELECTIVE REPEAT, modelu Gilberta, TMR + Parity Bit i zmiennym oknie -> 5,10,16,32

SR, Gilbert, okno 5									
PD	PZ	PDZ	PZD	Gilbert Stopa Błędu	Plik można otworzyc	BER	Retransmisje	Retr. Powyżej 8 (granica na jeden pakiet)	Calkowita liczba bitow
0,001	0,5	0,02	0,1	0,0101	NIE	4,89000000%	33194	TAK	535680
0,001	0,25	0,02	0,1	0,0051	NIE	0,90000000%	29657	TAK	
0,001	0,25	0,01	0,1	0,0026	NIE	0,42000000%	17083	NIE	
0,001	0,12	0,001	0,01	0,00013	NIE	0,06800000%	6977	NIE	
0,001	0,06	0,001	0,01	0,00007	TAK	0,01100000%	6717	NIE	
0,001	0,03	0,001	0,01	0,00004	TAK	0,00140000%	5944	NIE	
0,001	0,015	0,001	0,01	0,000025	CZYSTY	0,00006200%	5027	NIE	



Znów zwiększenie prawdopodobieństwa błędów powoduje, że BER rośnie bardzo szybko, lecz w tym przypadku już mamy moment, kiedy otwarty obraz jest już obrazem czystym, w pełni czytelnym przy wielkości okna równym 5.

SR, Gilbert, okno 10									
PD	PZ	PDZ	PZD	Gilbert Stopa Błędu	Plik można otworzyc	BER	Retransmisje	Retr. Powyżej 8 (granica na jeden pakiet)	Calkowita liczba bitow
0,001	0,5	0,02	0,1	0,0101	NIE	4,89000000%	32644	TAK	535680
0,001	0,25	0,02	0,1	0,0051	NIE	0,92000000%	29838	TAK	
0,001	0,25	0,01	0,1	0,0026	NIE	0,42000000%	17090	NIE	
0,001	0,12	0,001	0,01	0,00013	TAK	0,07100000%	7143	NIE	
0,001	0,06	0,001	0,01	0,00007	TAK	0,00850000%	6521	NIE	
0,001	0,03	0,001	0,01	0,00004	TAK	0,00099000%	6108	NIE	
0,001	0,015	0,001	0,01	0,000025	CZYSTY	0,00024000%	5003	NIE	

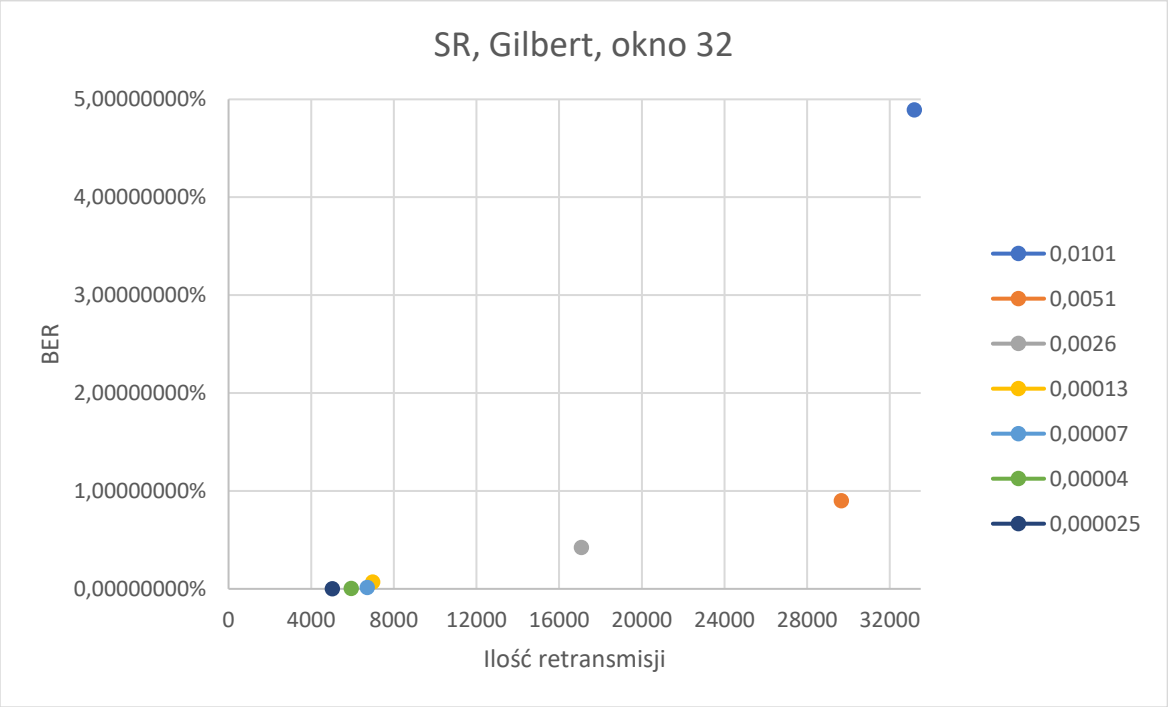


Podobnie jak przy oknie równym 5, lecz tym razem plik można otworzyć w większej ilości przypadków oraz ogólna liczba retransmisji wydaje się mniejsza.

SR, Gilbert, okno 16									
PD	PZ	PDZ	PZD	Gilbert Stopa Błędu	Plik można otworzyc	BER	Retransmisje	Retr. Powyżej 8 (granica na jeden pakiet)	Calkowita liczba bitow
0,001	0,5	0,02	0,1	0,0101	NIE	4,89000000%	32588	TAK	535680
0,001	0,25	0,02	0,1	0,0051	NIE	0,91000000%	29868	TAK	
0,001	0,25	0,01	0,1	0,0026	NIE	0,43000000%	17502	NIE	
0,001	0,12	0,001	0,01	0,00013	NIE	0,05000000%	7007	NIE	
0,001	0,06	0,001	0,01	0,00007	TAK	0,00560000%	6015	NIE	
0,001	0,03	0,001	0,01	0,00004	CZYSTY	0,00049000%	5981	NIE	
0,001	0,015	0,001	0,01	0,000025	CZYSTY	0,00012000%	5186	NIE	

W tym przypadku wielkość okna zmniejszyła potrzebna liczbę retransmisji i teraz już w 2 przypadkach obraz jest w pełni czytelny.

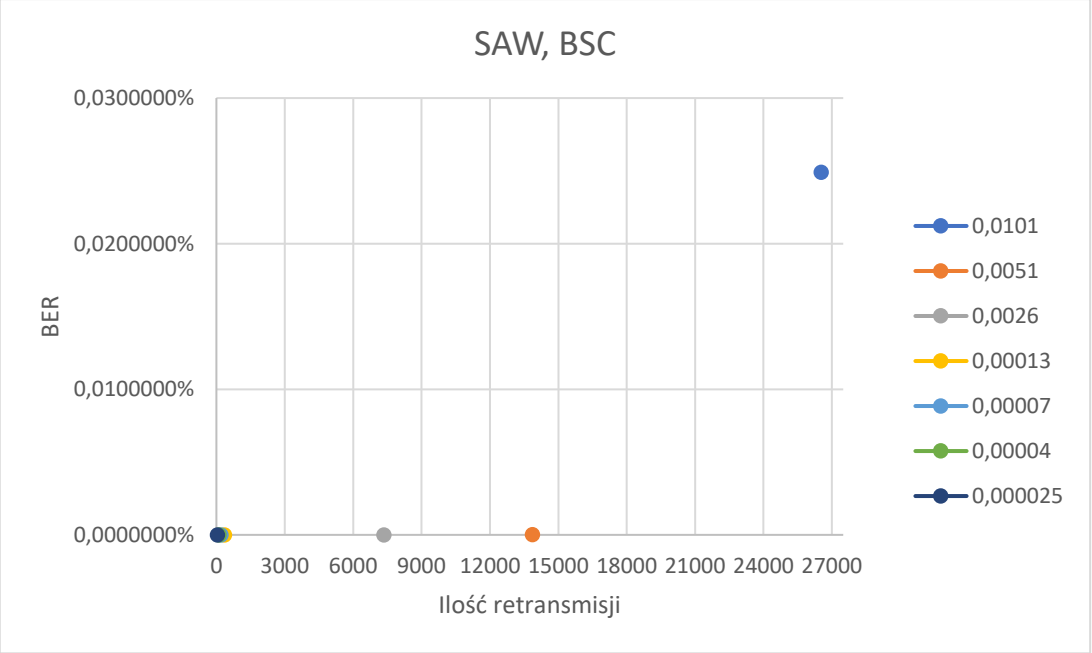
SR, Gilbert, okno 32									
PD	PZ	PDZ	PZD	Gilbert Stopa Błędu	Plik można otworzyc	BER	Retransmisje	Retr. Powyżej 8 (granica na jeden pakiet)	Calkowita liczba bitow
0,001	0,5	0,02	0,1	0,0101	NIE	4,83000000%	32682	TAK	535680
0,001	0,25	0,02	0,1	0,0051	NIE	0,92000000%	29813	TAK	
0,001	0,25	0,01	0,1	0,0026	NIE	0,42000000%	17511	TAK	
0,001	0,12	0,001	0,01	0,00013	TAK	0,05200000%	11261	NIE	
0,001	0,06	0,001	0,01	0,00007	NIE	0,00590000%	9244	NIE	
0,001	0,03	0,001	0,01	0,00004	TAK	0,00064000%	7913	NIE	
0,001	0,015	0,001	0,01	0,000025	TAK	0,00012000%	5639	NIE	



Okno o wielkości 32 już zamiast pomoc, tylko przeszkodziło, obraz nigdy nie był w pełni czytelny, liczba retransmisji podobna.

Następnie robiliśmy te same testy, z tymi samymi danymi tylko teraz dla modelu BSC, protokół STOP AND WAIT, TMR + parity bit

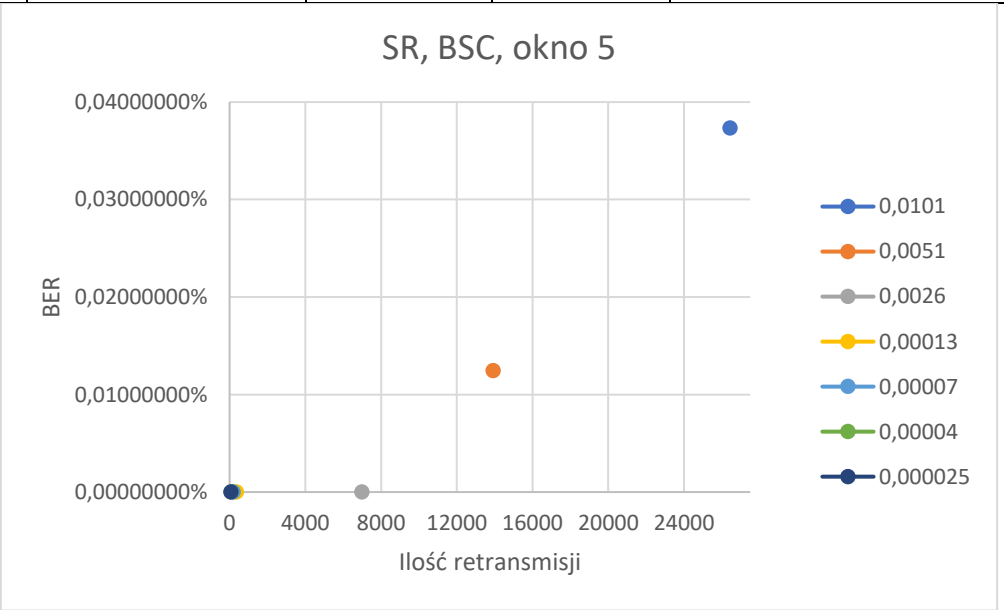
Model BSC, protokół STOP AND WAIT									
PD	PZ	PDZ	PZD	BSC Stopa Błędu	Plik można otworzyć	BER	Retransmisje	Retr. Powyżej 8 (granica na jeden pakiet)	Calkowita liczba bitow
0,001	0,5	0,02	0,1	0,0101	TAK	0,0248905%	26536	NIE	535680
0,001	0,25	0,02	0,1	0,0051	TAK	0,0000005%	13865	NIE	
0,001	0,25	0,01	0,1	0,0026	TAK	0,0000003%	7350	NIE	
0,001	0,12	0,001	0,01	0,00013	TAK	0,0000003%	362	NIE	
0,001	0,06	0,001	0,01	0,00007	TAK	0,0000002%	206	NIE	
0,001	0,03	0,001	0,01	0,00004	TAK	0,0000001%	117	NIE	
0,001	0,015	0,001	0,01	0,000025	TAK	0,0000000%	55	NIE	



Tym razem wyniki są bardzo ciekawe, z taka stopa błędu, model BSC prawie nic nie robi w porównaniu do modelu Gilberta, plik za każdym razem można było otworzyć i był albo prawie czysty lub dosłownie bez błędny. Błędy się pojawiały, lecz były bardzo szybko eliminowane.

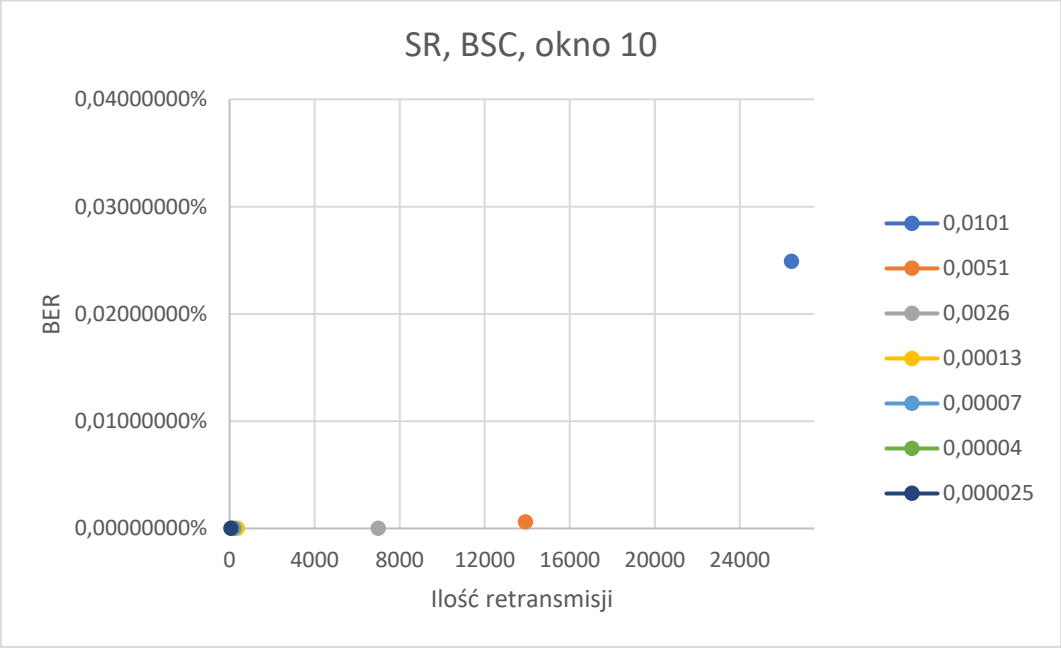
Model BSC, protokół SELECTIVE REPEAT, TMR + parity bit, okna -> 5,10,16,32

SR, BSC, okno 5									
PD	PZ	PDZ	PZD	BSC Stopa Błędu	Plik można otworzyc	BER	Retransmisje	Retr. Powyżej 8 (granica na jeden pakiet)	Calkowita liczba bitow
0,001	0,5	0,02	0,1	0,0101	TAK	0,03733572%	26431	NIE	535680
0,001	0,25	0,02	0,1	0,0051	TAK	0,01244524%	13926	NIE	
0,001	0,25	0,01	0,1	0,0026	TAK	0,00000010%	7000	NIE	
0,001	0,12	0,001	0,01	0,00013	TAK	0,00000006%	367	NIE	
0,001	0,06	0,001	0,01	0,00007	CZYSTY	0,00000003%	197	NIE	
0,001	0,03	0,001	0,01	0,00004	CZYSTY	0,00000001%	98	NIE	
0,001	0,015	0,001	0,01	0,000025	CZYSTY	0,00000000%	71	NIE	



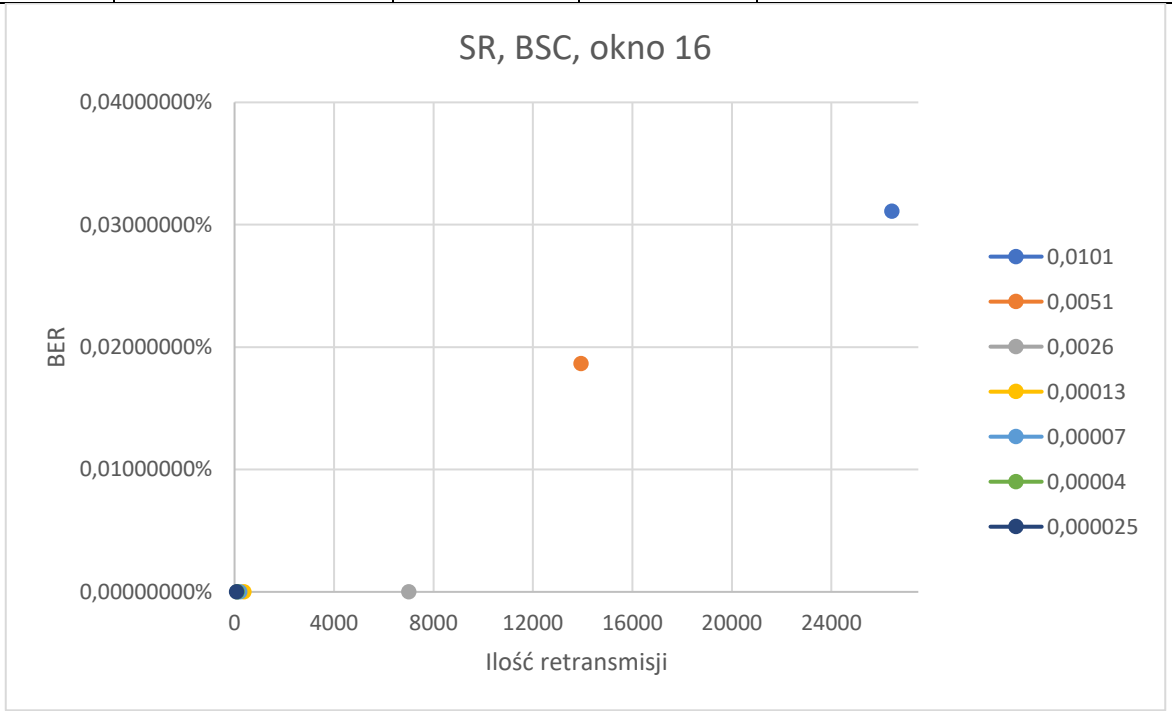
Plik w każdym przypadku mógł zostać otworzony, na początku BER cos jeszcze znaczył, lecz później był już prawie niezauważalny.

SR, BSC, okno 10									
PD	PZ	PDZ	PZD	BSC Stopa Błędu	Plik można otworzyc	BER	Retransmisje	Retr. Powyżej 8 (granica na jeden pakiet)	Calkowita liczba bitow
0,001	0,5	0,02	0,1	0,0101	TAK	0,02489048%	26606	NIE	535680
0,001	0,25	0,02	0,1	0,0051	TAK	0,00059583%	13877	NIE	
0,001	0,25	0,01	0,1	0,0026	TAK	0,00000090%	7114	NIE	
0,001	0,12	0,001	0,01	0,00013	CZYSTY	0,00000006%	372	NIE	
0,001	0,06	0,001	0,01	0,00007	CZYSTY	0,00000005%	199	NIE	
0,001	0,03	0,001	0,01	0,00004	CZYSTY	0,00000001%	95	NIE	
0,001	0,015	0,001	0,01	0,000025	CZYSTY	0,00000000%	55	NIE	



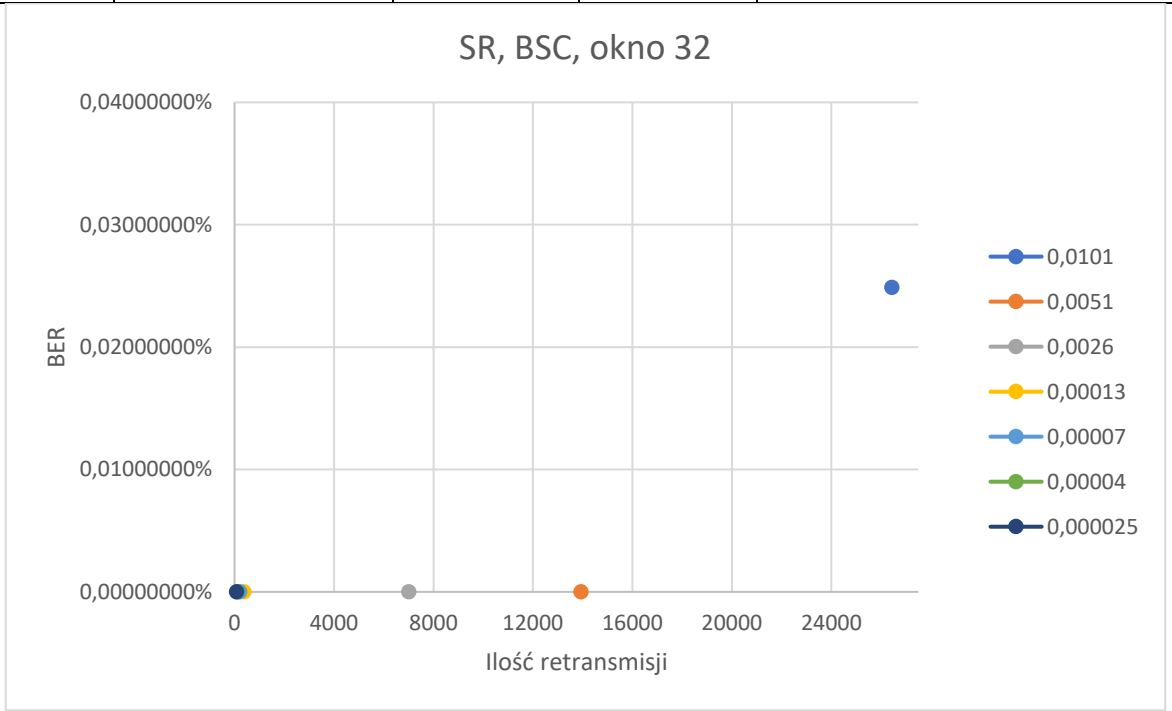
Okno o wielkości 10, pozwoliło na zmniejszenie ilości retransmisji, BER oraz plik 4 razy, a nie 3 był w pełni czytelny.

SR, BSC, okno 16									
PD	PZ	PDZ	PZD	BSC Stopa Błędu	Plik można otworzyc	BER	Retransmisje	Retr. Powyżej 8 (granica na jeden pakiet)	Calkowita liczba bitow
0,001	0,5	0,02	0,1	0,0101	TAK	0,03111310%	26426	NIE	535680
0,001	0,25	0,02	0,1	0,0051	TAK	0,01866786%	13624	NIE	
0,001	0,25	0,01	0,1	0,0026	TAK	0,00000080%	7120	NIE	
0,001	0,12	0,001	0,01	0,00013	TAK	0,00000003%	352	NIE	
0,001	0,06	0,001	0,01	0,00007	CZYSTY	0,00000002%	182	NIE	
0,001	0,03	0,001	0,01	0,00004	CZYSTY	0,00000001%	91	NIE	
0,001	0,015	0,001	0,01	0,000025	CZYSTY	0,00000000%	63	NIE	



Okno o wielkości 16, sytuacja została lekko pogorszona względem okna o wielkości 10, BER większy, ilość retransmisji tez.

SR, BSC, okno 32									
PD	PZ	PDZ	PZD	BSC Stopa Błędu	Plik można otworzyc	BER	Retransmisje	Retr. Powyżej 8 (granica na jeden pakiet)	Calkowita liczba bitow
0,001	0,5	0,02	0,1	0,0101	TAK	0,02489048%	26688	NIE	535680
0,001	0,25	0,02	0,1	0,0051	TAK	0,00000190%	13978	NIE	
0,001	0,25	0,01	0,1	0,0026	TAK	0,00000090%	7200	NIE	
0,001	0,12	0,001	0,01	0,00013	CZYSTY	0,00000004%	357	NIE	
0,001	0,06	0,001	0,01	0,00007	CZYSTY	0,00000002%	202	NIE	
0,001	0,03	0,001	0,01	0,00004	CZYSTY	0,00000001%	101	NIE	
0,001	0,015	0,001	0,01	0,000025	CZYSTY	0,00000000%	48	NIE	



Okno o wielkości 32, zadziałało podobnie co okno o wielkości 10. Plik był tyle samo razy w pełni czytelny, BER jest najmniejszy ze wszystkich okien, w każdym możliwym przypadku.

Hamming + CRC

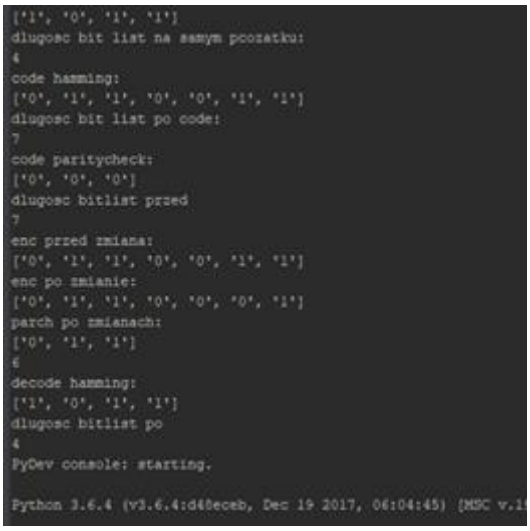
Testy dla Hamminga wykonywały się w okresie ok. 40 minut, każdy pojedynczy. Prawdopodobnie było to spowodowane za słabym procesorem w komputerze lub źle zoptymalizowanymi działaniami matematycznymi (liczeniem macierzy itp.) Dlatego tez nie pokazujemy wyników testów, gdyż po prostu trzeba czekać zbyt długo aby jakikolwiek otrzymać. Wykonaliśmy 1 test, dla modelu BSC oraz protokołu STOP AND WAIT, wyniki są następujące:

PD	PZ	PDZ	PZD	BSC Stopa Błędu	Plik można otworzyć	BER	Retransmisje	Retr. Powyżej 8 (granica na jeden pakiet)	Całkowita liczba bitów
0,001	0,5	0,02	0,1	0,0101	NIE	31,1962366%	78559292	TAK	535680

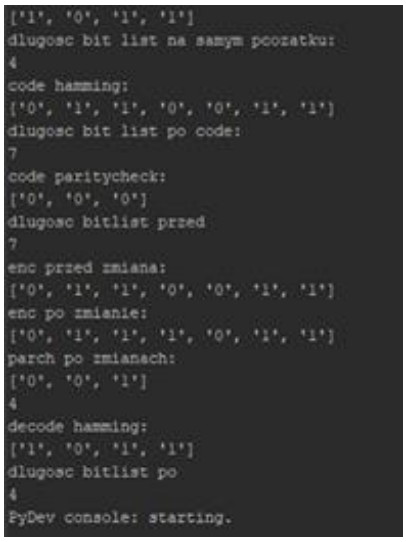
Duże prawdopodobieństwo błędów, czyli w dużej ilości pakietów występowało więcej niż 1 błąd, dlatego Hamming tyle razy musiał to odsyłać, bo nie wiedział co z tym zrobić, udało się poprawić polowe pakietów, lecz ostatecznie Hamming w takim przypadku zawiódł. Ilość błędów była po prostu przytłaczająca dla Hamminga (7,4). Jesteśmy pewni ze Hamming działa, co zaraz pokażemy poniżej. Następujący kod został wykorzystany do testów manualnych:

```
1.  bitList = []
2.  #bitList = ['0','0','0','0','0','0','0','0','0','0','0','1']
3.  #bitList = ['1','0','1','1','0']  # PRZY MALYCH SIE W CHUJ PIERDOLI
4.  bitList = ['1','0','1','1']
5.  fileOperator = FileOperator()
6.  #bitList = fileOperator.readFile("test.txt")
7.  print(bitList)
8.  print("dlugosc bit list na samym pcozatku: ")
9.  print(len(bitList))
10. hamming = Hamming()
11. bitList = hamming.codeHamming(bitList)
12. print("code hamming: ")
13. print(bitList)
14. print("dlugosc bit list po code: ")
15. print(len(bitList))
16. print("code paritycheck: ")
17. print(hamming.parityCheck(bitList))
18. print("dlugosc bitlist przed")
19. print(len(bitList))
20. bitList = hamming.decodeHamming(bitList)
21. print("decode hamming: ")
22. print(bitList)
23. print("dlugosc bitlist po")
24. print(len(bitList))
```

Wyniki testów manualnych na bitList = ['1','0','1','1']:



W tym przypadku błąd został dodany ręcznie na pozycje enc[5] = ‘0’, czyli w parityCheck wychodzi, ze na 6 pozycji bo liczy od 1, a nie od 0, jak widać, kod zostaje poprawiony i w decode mamy to co na samym początku.



W tym przypadku błąd został dodany na pozycji enc[3], czyli dla parityCheck to 4, poprawiony i decode pokazuje pakiet jak na początku.

```
['1', '0', '1', '1']
dlugosc bit list na samym pcozatku:
4
code hamming:
['0', '1', '1', '0', '0', '1', '1']
dlugosc bit list po code:
7
code paritycheck:
['0', '0', '0']
dlugosc bitlist przed
7
enc przed zmiana:
['0', '1', '1', '0', '0', '1', '1']
enc po zmianie:
['1', '1', '1', '0', '0', '1', '1']
parch po zmianach:
['1', '0', '0']
1
decode hamming:
['1', '0', '1', '1']
dlugosc bitlist po
4
```

W tym przypadku błąd na pozycji enc[0], czyli dla parityCheck to 1, znowu decode dobry.

```
['1', '0', '1', '1']
dlugosc bit list na samym pcozatku:
4
code hamming:
['0', '1', '1', '0', '0', '1', '1']
dlugosc bit list po code:
7
code paritycheck:
['0', '0', '0']
dlugosc bitlist przed
7
enc przed zmiana:
['0', '1', '1', '0', '0', '1', '1']
enc po zmianie:
['1', '1', '1', '1', '0', '1', '1']
parch po zmianach:
['1', '0', '1']
5
enc po poprawie:
['1', '1', '1', '1', '1', '1', '1']
decode hamming:
['1', '1', '1', '1']
dlugosc bitlist po
4
```

W tym przypadku błąd na pozycji enc[0] oraz enc[3], mamy 2 błędy wiec parityCheck wyliczony blednie, Hamming blednie poprawił błąd, decode błędny, problem, który jest wyłapywany później przez CRC.

```
['1', '0', '1', '1']
dlugosc bit list na samym pcozatku:
4
code hamming:
['0', '1', '1', '0', '0', '1', '1']
dlugosc bit list po code:
7
code paritycheck:
['0', '0', '0']
dlugosc bitlist przed
7
enc przed zmiana:
['0', '1', '1', '0', '0', '1', '1']
enc po zmianie:
['0', '1', '1', '0', '0', '1', '1']
parch po zmianach:
['0', '0', '0']
0
enc po poprawie:
['0', '1', '1', '0', '0', '1', '1']
decode hamming:
['1', '0', '1', '1']
dlugosc bitlist po
4
```

Przypadek, kiedy błędów w ogóle nie było, wszystko działa poprawnie

Wnioski i przemyślenia

Hamming + CRC

Podsumowując, Hamming działał u nas bez problemowo, jedyne co to czas wykonywania operacji na nim, przy użyciu już protokołów, pętli był już bardzo długi. Przykłady i dokładniejszy opis jest wyżej, podczas opisu testów na Hammingu.

TMR + parity Bit, Gilbert, SAW

Z wykresu i tabel, widać że im większa ilość błędów, tym potrzebne jest więcej retransmisji oraz BER rośnie. Plik z obrazem można było otworzyć tylko w 3 przypadkach, z szansa na błąd już mała, lecz nie były to pliki czyste, nadal zawierały przekłamania bitów.

TMR + parity Bit, Gilbert, SR

W tym przypadku, już było dużo lepiej w porównaniu do SAW. Dla okna równego 5,10,16, występował już 1 przypadek, kiedy plik był plikiem czystym. Dla okna 32 już takiego przypadku nie odnotowaliśmy. Przy takich zmiennych i takich protokołach, najlepiej wypadło SR z oknem 16. 2 przypadki pliku czystego, ilość retransmisji najmniejsza, BER tez.

TMR + parity Bit, BSC, SAW

Porównując do Gilberta, ilość potrzebnych retransmisji dużo mniejsza, BER tez. W każdym przypadku, plik mógł zostać otwarty. Występowały w nich nadal małe zmiany, lecz były bardzo bliskie do bycia plikami czystymi.

TMR + parity Bit, BSC, SR

Przy zmianie protokołu na SR zamiast SAW, zauważyliśmy zmiany w BER, oraz potrzebnych retransmisji, kiedy błędy miały największe prawdopodobieństwo, reszta bardzo podobna do SAW. Dla okien o rozmiarach 5, 16 -> 3 krotnie plik czysty przy małych prawdopodobieństwach. Okna o rozmiarach 10 oraz 32, 4 razy plik był czysty. Ilości retransmisji w każdym przypadku podobna. Najlepszym rozmiarach okna okazało się okno 32, BER najmniejszy, plik największą ilość razy był czysty.