Niezawodność i Diagnostyka Układów Cyfrowych 2

Etap 3 - Sprawozdanie

Wrocław University
of Science and Technology

1. Wstęp

Projekt z kursu Niezawodność i Diagnostyka Układów Cyfrowych 2 podzielony został na 3 etapy:

Etap 1

Opracowanie symulatora :
W naszym przypadku opracowany został symulator FEC (Forward Error Correction) w języku Python z zaimprementowanymi algorytmami:

- Kodu Hamminga,
- CRC.

Etap 2

Analiza danych:
Do analizy danych wyjściowych symulatora wykorzystaliśmy:

- Średnią arytmetyczną,
- Statystykę pięciopunktową,
- Wykres pudełkowy (boxplot),
- Histogram.

Ze względu na przejrzystość oraz prostotę działania, a jednocześnie dużą ilość informacji dostarczanych z wykresów opartych na średniej arytmetycznej to właśnie ich użyjemy do realizacji Etapu 3.

Etap 3

Badanie parametrów:
Przy użyciu wykresów opartych na średniej arytmetycznej opracowujemy wyniki symulacji, która na wejściu przyjmuje różne wartości:

- Liczby bitów,
- Długości pakietów danych,
- Szansy na przekłamanie bitu.

Badanie realizowane jest zarówno dla algorytmu Kodu Hamminga jak również dla CRC.

2. Opis funkcjonowania symulatora

Symulator został napisany w języku Python 3
Działanie symulatora podzielone jest na:

1. **Generowanie danych**:
   Zaimplementowana została funkcja generująca tablicę 0/1 o zadanej w parametrach wejściowych długości oraz prawdopodobieństwa wystąpienia 0/1.
   W ramach testów dodana została również opcja wprowadzenia własnego ciągu 0/1.

2. **Podział na pakiety**:
   Funkcja dzieli wygenerowane lub wprowadzone bity na n elementowe pakiety.
   Zmienna n jest parametrem wejściowym symulatora.
   Ze względu na specyfikę działania zaimplementowanego kodu Hamminga, aby możliwe było jego użycie długość pakietu musi być potęgą dwójki (2^x).

W kolejnych etapach procedura różni się w zależności od użytego algorytmu:

3. **Kodowanie pakietów**:
   - **Kody Hamminga**:
     Algorytm generuje ciąg kodu Hamminga i wstawia odpowiednie elementy na pozycji 0 i na pozycjach kolejnych potęg 2 ($2^x \leq n$).
   - **CRC**:
     Dla kazdego pakietu obliczana jest suma kontrolna CRC ktora nastepnie dodawana jest na najmlodsze bity pakietu. Powstaje ona poprzez dzielenie ciągu bitów danych przez ustawiony dzielnik CRC. Ilosc bitow sumy kontrolnej determinuje wielomian CRC. W naszym symulatorze używamy wielomianu 0x12 ($x^5 + x^2 + 1$) zatem suma kontrolna ma 5 bitów.

4. **Symulacja przesyłu**:
   Funkcja w zależności od zadanej w parametrach wejściowych szany na przekłamanie neguje losowe bity w zakodowanym pakiecie.

5. **Dekodowanie pakietów oraz ich ewentualna naprawa**:

- **Kody Hamminga**:
  Każdy przesłany pakiet przed usunięciem kodu Hamminga jest przepuszczany przez ten sam algorytm liczący ciąg, jak w przypadku pakietu początkowego. Jeżeli:

  Kod Hamminga = 0
  Pakiet uznawany jest jako poprawny. Algorytm dodaje kod błędu = '1' oznaczający pakiet bez wykrytych błędów.

  Kod Hamminga != 0
  Przekłamany został jeden bit lub więcej bitów.
  Naprawa takiego pakietu następuje poprzez wykonanie operacji NOT bitu pakietu o indeksie równym dziesiętnemu zapisowi wyliczonego kodu. Algorytm dodaje kod błędu = '2' oznaczający, że pakiet przyszedł z błędem, lecz został on wykryty i naprawiony. W tej sytuacji wychodzi najwięcej błędów, ponieważ algorytm może naprawić maksymalnie 1 błąd, a w odpowiednim ułożeniu dwa i więcej przekłamań może symulować przekłamanie dobrze przesłanego bitu na innej pozycji.

  Kod Hamminga > długość pakietu
  Przekłamanie nastąpiło napewno na większej liczbie bitów.
  Algorytm dodaje kod błedu = '3' do pakietu, który oznacza wykrycie błędu i brak możliwości jego naprawy.

- **CRC**:
  Dla każdego pakietu (uwzględniając "dopisane"bity przy kodowaniu) ponownie liczona jest suma kontrolna. Na jej podstawie generowany jest kod błedu : 0,1 lub 2. Kod błędu uzależniony jest od ilości bitów o wartości 1 w sumie kontrolnej.

```
def check(packet):
    check_value = crc_maker(packet)
    unos = 0
    for i in range(len(check_value)):
        if check_value[i] == 1:
            unos += 1
    if unos == 0:
        return 0
    if unos == 1:
        return 1
    else:
        return 2
```

Jeżeli:

Kod błedu = 0
Pakiet uznawany jest jako poprawny.

Kod błedu = 1
Przekłamany został jeden bit w polu sumy kontrolnej.
Naprawa takiego pakietu następuje poprzez wykonanie operacji XOR bitu pakietu o indeksie jedynki znalezionej w sumie kontrolnej.

Kod błedu = 2
Przekłamany został bit w polu danych pakietu lub ilosc przeklamanych bitow przewyższa zdolność naprawczą wielomianu (w naszym przypadku zdolność naprawcza to 1 bit przy pakiecie do 26 bitów).
Naprawa realizowana jest poprzez cykliczne przesunięcie bitów w pakiecie aż do wystąpienia kodu błedu = 1. Jeżeli po powrocie do stanu wyjsciowego pakiet nadal zwraca kod błedu = 2, nie zostanie on naprawiony.

```
def repair (packet,check_code):
    if check_code == 1:
        check_value = crc_maker(packet)
        for j in range (len(check_value)):
            helper4 = len(packet) - (len(crc)-1) + j
            if check_value[j] == 1:
                packet[helper4] = packet[helper4] ^ 1
        return packet
    if check_code == 2:
        for a in range(1,len(packet)):
            packet = przesun1(packet)
            kontrolka = check(packet)
            if(kontrolka == 1):
                packet = repair(packet,1)
                while a > 0:
                    packet = przesun2(packet)
                    a -=1
                return packet
```

3. Opis danych wyjściowych symulatora

W strukturze symulatora opracowaliśmy funkcję mająca na celu zliczanie wystąpień czterech sytuacji:

Sytuacja 0 Pakiet dotarł bez zakłóceń.

Sytuacja 1 Pakiet został poprawnie naprawiony.

Sytuacja 2 Wykryto błąd w pakiecie jednak go nie naprawiono.

Sytuacja 3 Pakiet został błędnie naprawiony.

Podczas Etapu 2 dodaliśmy do symulatora narzędzia potrzebne do dokładniejszej analizy zachowania modelu przekształcające surowe wartości wyjściowe programu w odpowiednie dane oraz wykresy.
Dzięki opracowaniu takowych narzędzi umożliwiona została realizacja Etapu 3.

4. Opis zaimplementowanych struktur analizy danych(todo)

Poniższe struktury analizy danych oparte zostały na przykładowej symulacji dla podanych w Tablicy 1 danych wejściowych symulatora. Wszystkie wykresy oraz dane wygenerowane zostały za pomocą biblioteki matplotlib w języku Python.

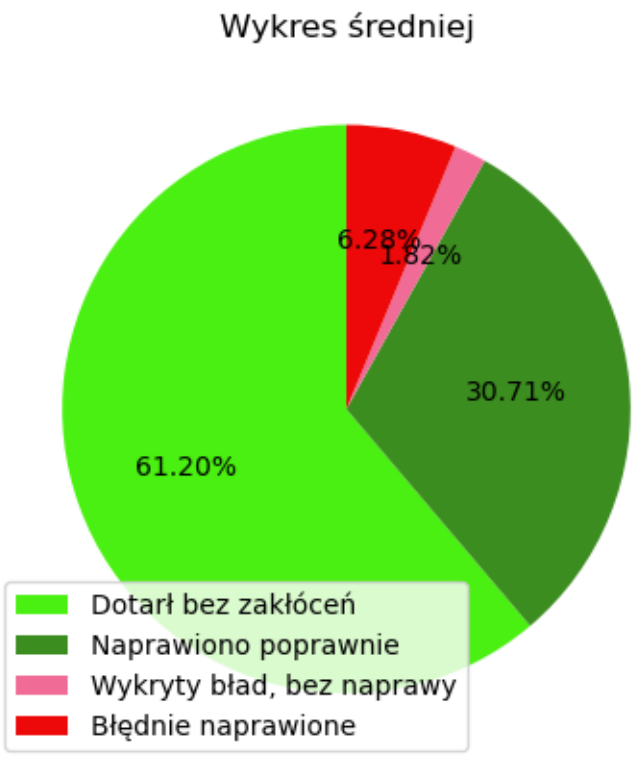
Tablica 1: Parametry wejściowe symulacji

Symulacja przykładowa	
Algorytm	*Kod Hamminga*
Liczba bitów	10000
Rozmiar pakietu	8
Prawdopodobieństwo zakłamania	4%
Ilosc symulacji	1000

Średnia arytmetyczna

Jest jedną z najbardziej intuicyjnych miar oceny. Jednocześnie daje sporo informacji pozwalających na określenie podstawowych parametrów mierzonej cechy.
W przypadku tej symulacji średnia liczona jest poprzez dodanie liczby pakietów odpowiedniej sytuacji, a następnie podzielenie tej sumy przez łączną liczbę symulowanych pakietów. Dla większej czytelności przedstawiliśmy średnią za pomocą wykresu kołowego i wartości procentowych.
Ogólnie średnią arytmetyczną definiuje się następującym wzorem

$$\bar{x} = 1N \sum_{i=1}^{N} x_i = \frac{x_1 + x_2 + x_3 + ... + x_n}{N}.$$



Rysunek 1: Wykres średniej arytmetycznej dla symulacjii przykładowej.

Statystyka pięciopunktowa

Zapewnia więcej szczegółow na temat rozkładu danych.
Wyznaczane są liczby - tak zwane **kwartyle** rozkładu:

$\mathbf{Q}_0$ - wartość minimalna,

$\mathbf{Q}_1$ - wartość większa niż 25% danych w zbiorze,

$\mathbf{Q}_2$ - mediana zbioru,

$\mathbf{Q}_3$ - wartość większa niż 75% danych w zbiorze,

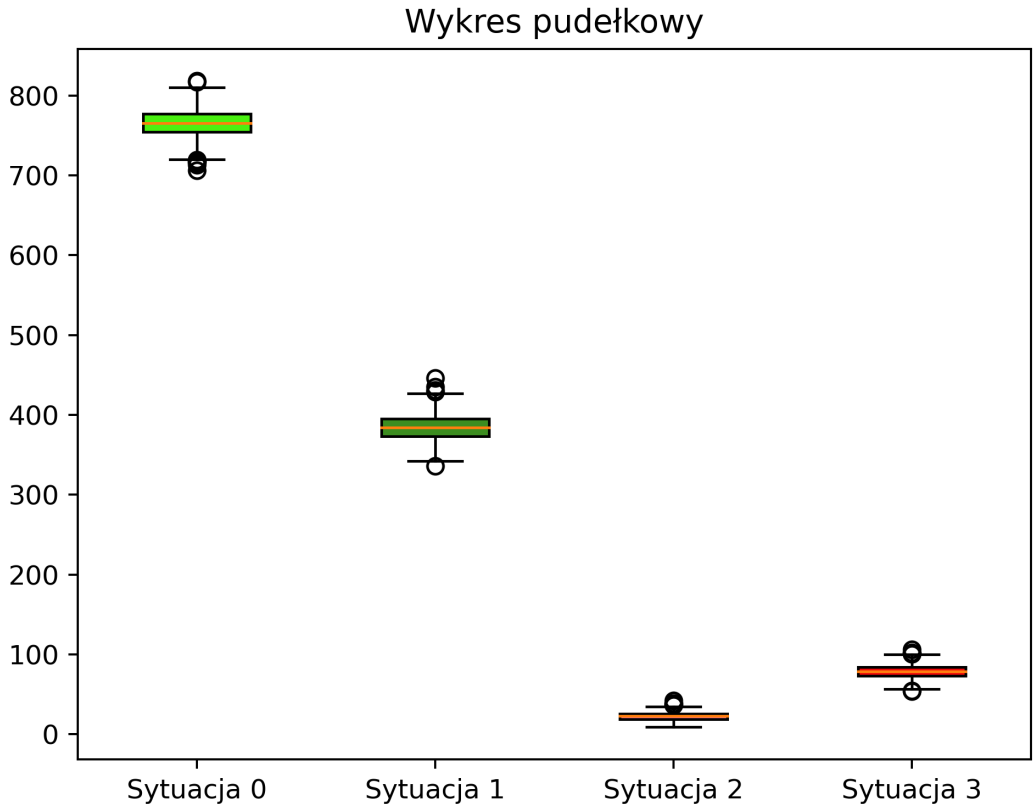
$\mathbf{Q}_4$ - wartość maksymalna.

Tablica 2: Wyniki statystyki pięciopunktowej dla symulacji przykładowej.

	Sytuacja 0	**Sytuacja 1**	**Sytuacja 2**	**Sytuacja 3**
Q_0	706.0	336.0	9.0	54.0
Q_1	754.0	373.0	19.0	73.0
Q_2	765.0	384.0	23.0	78.5
Q_3	777.0	395.0	26.0	84.0
Q_4	818.0	446.0	42.0	106.0

Wykres pudełkowy (boxplot)

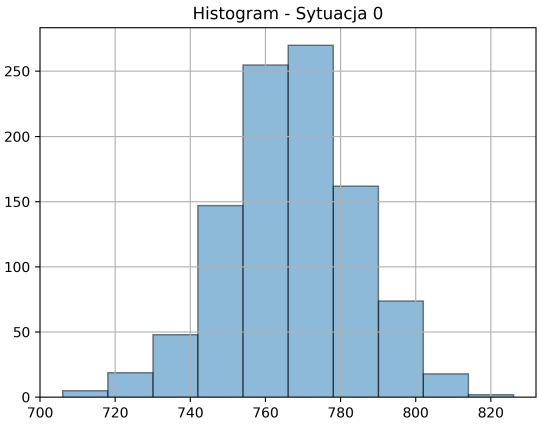
Wykorzystywany jest do graficznej wizualizacji rozkładu wartości w zbiorze danych.
Opiera się na kwartylach rozkładu opisanych statystyką pięciopunktową.
W wariancie złożonym wykresu (używanym w naszym symulatorze) stosuje się jedynie wartości Q_1, Q_2 i Q_3.
Dodatkowo definiuje się:
$min = Q_1 - 1.5 * IQR$
$max = Q_3 + 1.5 * IQR$
gdzie $IQR(przedzial\ midzykwartylowy) = Q_3 - Q_1$.



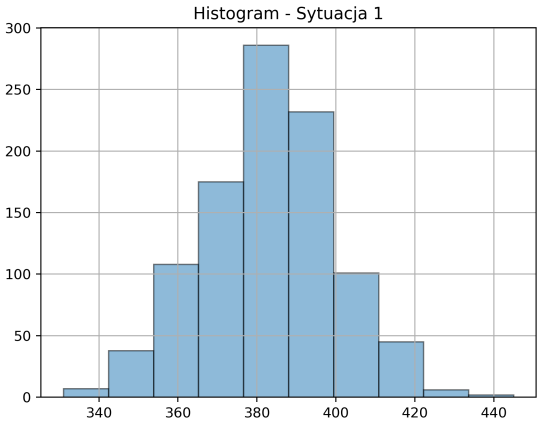
Rysunek 2: Wykres pudełkowy dla symulacjii przykładowej.

Histogram

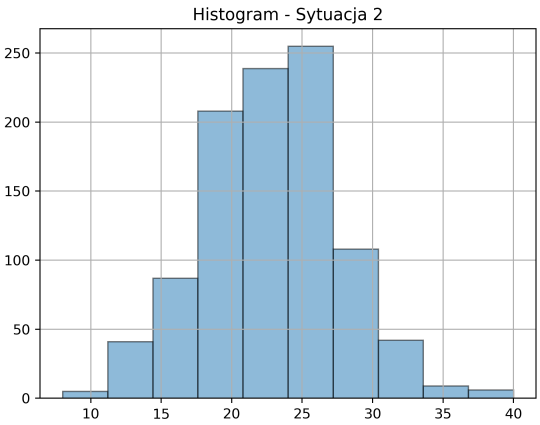
Jeden z graficznych sposobów pokazania rozkładu empirycznego.
Histogram składa się z szeregu prostokątów, a każdy z nich daje nam informację na temat liczebności wyników znajdujących się w pewnym zakresie.
W naszym przypadku histogram na osi X ma zakresy liczby wystąpienia danej sytuacji (dla każdej sytuacji generowany jest osobny histogram), a na osi Y pokazana jest liczebności sytuacji zawartych w tym zakresie.



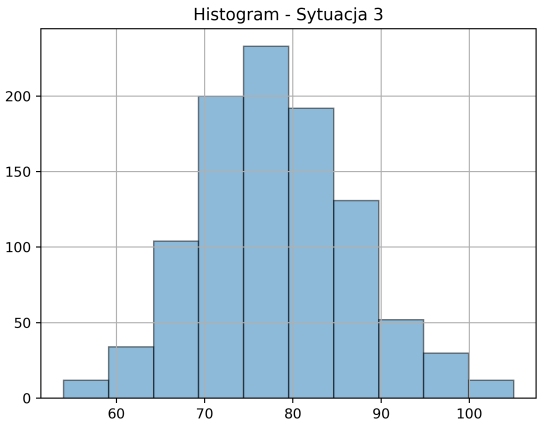
(a) Sytuacja 0



(b) Sytuacja 1



(c) Sytuacja 2



(d) Sytuacja 3

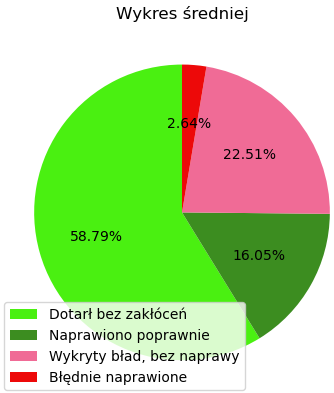
Rysunek 3: Histogramy poszczególnych sytuacji dla symulacji przykładowej.

5. Analiza wpływu parametrów wejściowych modelu(todo)

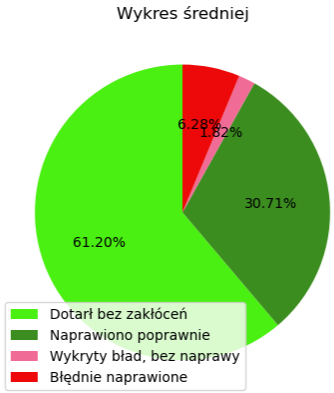
Jako parametry Testu 0 uznawanego przez nas za punkt odniesienia używamy tej samej konfiguracji jak przy prezentacji struktur analizy danych.

Tablica 3: Parametry wejściowe Testu 0.

Test 0	
Liczba bitów	10000
Rozmiar pakietu	8
Prawdopodobieństwo zakłamania	4%
Liczba symulacji	1000



(a) CRC



(b) Kod Hamminga

Rysunek 4: Wyniki Testu 0.

Analizując wpływ parametrów manipulowana będzie:

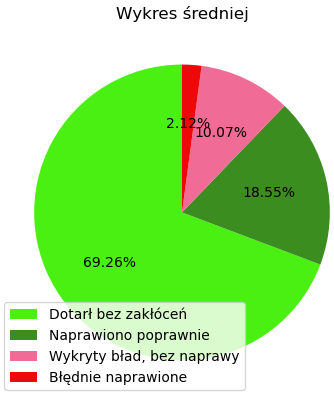
Długość pakietów

Aby zachować stałą liczbę pakietów, zmieniamy również liczbę bitów.

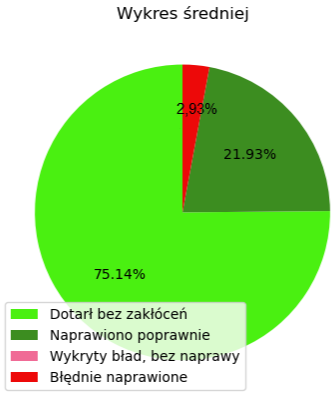
Test 1

Tablica 4: Parametry wejściowe Testu 1.

Test 1	
Liczba bitów	5000
Rozmiar pakietu	4
Prawdopodobieństwo zakłamania	4%
Liczba symulacji	1000



(a) CRC



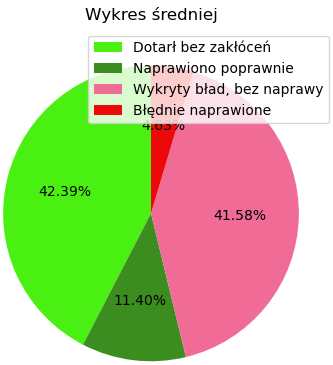
(b) Kod Hamminga

Rysunek 5: Wyniki Testu 1.

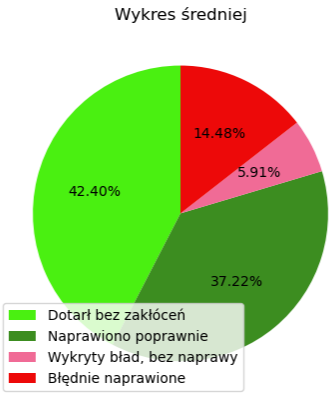
Test 2

Tablica 5: Parametry wejściowe Testu 2.

Test 2	
Liczba bitów	20000
Rozmiar pakietu	16
Prawdopodobieństwo zakłamania	4%
Liczba symulacji	1000



(a) CRC



(b) Kod Hamminga

Rysunek 6: Wyniki Testu 2.

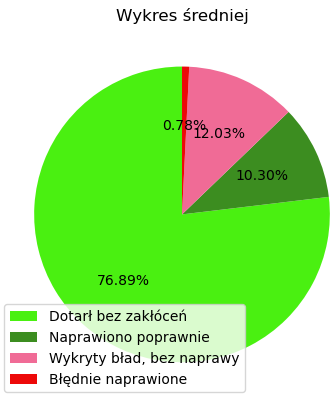
Wpływ zmiany długości pakietu na wyniki prób
Zmiana rozmiaru pakietu w algorytmie CRC sprawia, że znacząco zwiększa się liczba wykrytych, lecz niemożliwych do naprawy błędów. Przy czym liczba błędnie naprawionych pakietów nie zwiększa się aż tak drastycznie jak w przypadku algorytmu Hamminga.
Algorytm Hamminga lepiej radzi sobie z pakietami o mniejszej wielkości. Przy małych rozmiarach ryzyko błędu jest znikome, przy zwiększaniu rozmiaru pakietów rośnie o wiele bardziej niż w przypadku CRC, lecz wartym zauważenia jest fakt, że liczba poprawnie przyjętych pakietów (rozumiana tu jako suma sytuacji 0 oraz 1) jest znacząco większa od drugiego algorytmu.

Szansa zakłamania

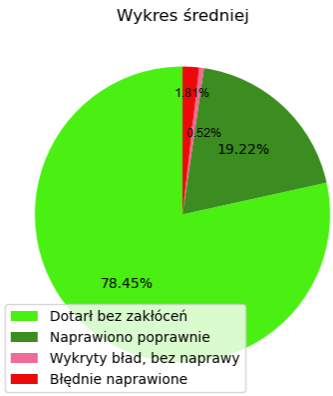
Test 3

Tablica 6: Parametry wejściowe Testu 3.

Test 3	
Liczba bitów	10000
Rozmiar pakietu	8
Prawdopodobieństwo zakłamania	2%
Liczba symulacji	1000



(a) CRC



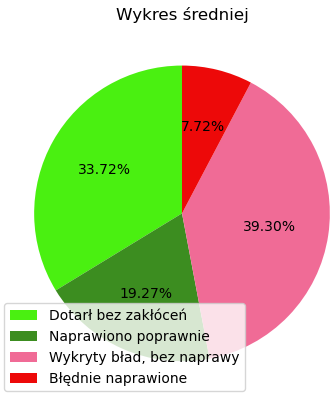
(b) Kod Hamminga

Rysunek 7: Wyniki Testu 3.

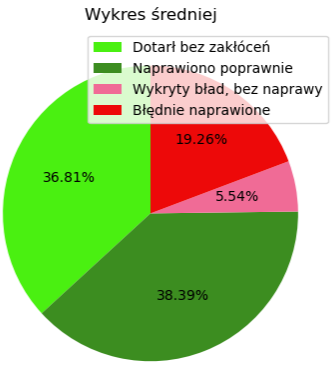
Test 4

Tablica 7: Parametry wejściowe Testu 4.

Test 4	
Liczba bitów	10000
Rozmiar pakietu	8
Prawdopodobieństwo zakłamania	8%
Liczba symulacji	1000



(a) CRC



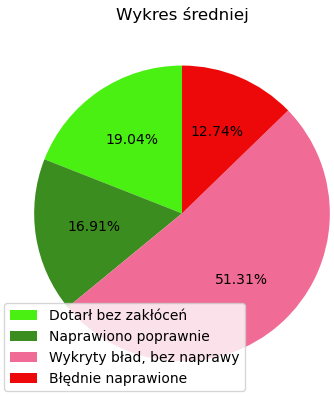
(b) Kod Hamminga

Rysunek 8: Wyniki Testu 4.

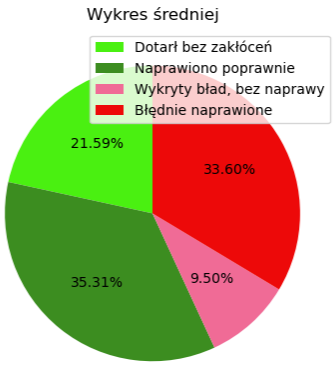
Test 5

Tablica 8: Parametry wejściowe Testu 5.

Test 5	
Liczba bitów	10000
Rozmiar pakietu	8
Prawdopodobieństwo zakłamania	12%
Liczba symulacji	1000



(a) CRC



(b) Kod Hamminga

Rysunek 9: Wyniki Testu 5.

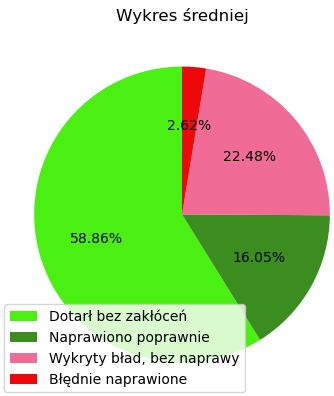
Wpływ zmiany szansy zakłamania na wyniki prób
Algorytm CRC dużo częściej poprawnie wykrywa błędy w pakiecie czyniąc ten algorytm bardziej odpornym na wysokie ryzyko zaszumienia sygnału.
Algorytm Hamminga jest mniej odporny w warunkach częstych błędów, lecz wciąż zachowuje wysoką ilość poprawnie naprawionych pakietów względem drugiego algorytmu.

Liczba bitów

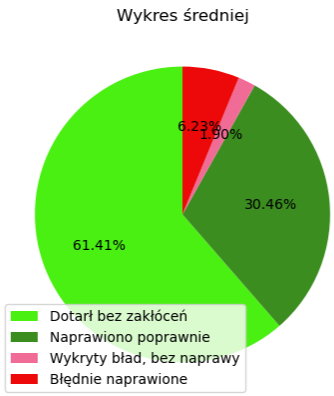
Test 6

Tablica 9: Parametry wejściowe Testu 6.

Test 6	
Liczba bitów	1024
Rozmiar pakietu	8
Prawdopodobieństwo zakłamania	4%
Liczba symulacji	1000



(a) CRC



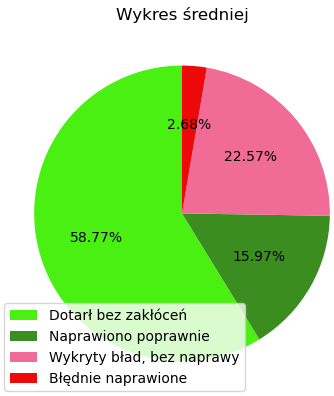
(b) Kod Hamminga

Rysunek 10: Wyniki Testu 6.

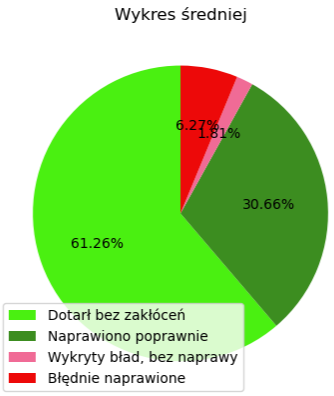
Test 7

Tablica 10: Parametry wejściowe Testu 7.

Test 7	
Liczba bitów	4096
Rozmiar pakietu	8
Prawdopodobieństwo zakłamania	4%
Liczba symulacji	1000



(a) CRC



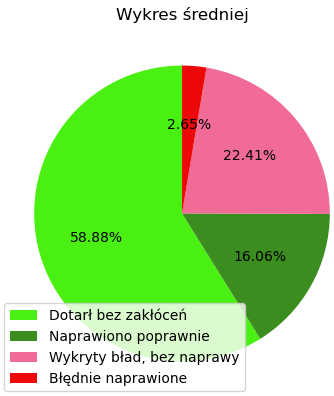
(b) Kod Hamminga

Rysunek 11: Wyniki Testu 7.

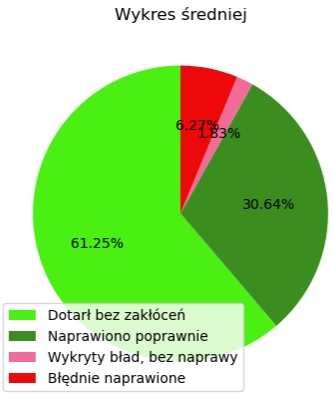
Test 8

Tablica 11: Parametry wejściowe Testu 8.

Test 8	
Liczba bitów	16384
Rozmiar pakietu	8
Prawdopodobieństwo zakłamania	4%
Liczba symulacji	1000



(a) CRC



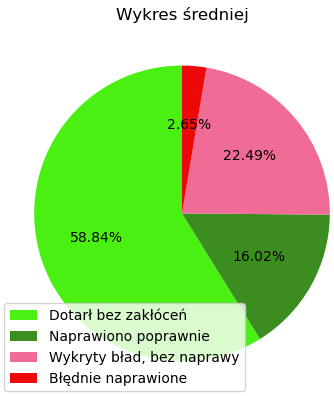
(b) Kod Hamminga

Rysunek 12: Wyniki Testu 8.

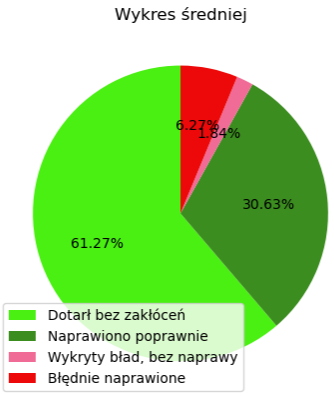
Test 9

Tablica 12: Parametry wejściowe Testu 9.

Test 9	
Liczba bitów	65536
Rozmiar pakietu	8
Prawdopodobieństwo zakłamania	4%
Liczba symulacji	1000



(a) CRC



(b) Kod Hamminga

Rysunek 13: Wyniki Testu 9.

Wpływ zmiany ilości przesyłanych bitow na wynik prób
W przypadku zmiany ilości bitów przesyłanych w trakcie symulacji nie widać znaczących różnic między średnimi wartościami każdego z testu.

6. Wnioski

Algorytm CRC wykorzystany w naszym prostym symulatorze wraz z odpowiednio dobranym dzielnikiem CRC posiada zdolność naprawczą do 1 zakłamanego bitu. Nie jest ona jednak zbyt efektywna. W wielu przypadkach symulator jest w stanie wykryć błąd w przekłamanym pakiecie, jednak nie udaje mu się go naprawić. Algorytm zaimplementowany w projekcie wykazuje najlepsze właściwości naprawcze przy podziale danych na małe pakiety (Test 1). Ze względu na częstą dysproporcję wykrycia błedu bez jego naprawy nad poprawnie wykonaną procedurą, algorytm CRC mógłby znaleźć lepsze zastosowanie w systemie ARQ (Automatic Reapeat Request) niż w FEC (Forward Error Correction).

Kod Hamminga zaimplemenotwany w projekcie jest w stanie naprawić pakiet odebrany z maksymalnie jednym błędem. Ze względu na charakterystykę algorytmu problem pojawia się, gdy błędów jest więcej niż jeden. W takiej sytuacji możliwe jest wykrycie błędów bez próby ich naprawy, lecz bardziej prawdopodobną opcją jest błędnie wykonana naprawa i uznanie przez algorytm pakietu za poprawny. Widać to szczególnie przy testach 2 oraz 5, gdzie sytuacji 3 jest wyraźnie więcej niż przy innych testach.
Dla kodu Hamminga badanego w projekcie optymalne są krótkie pakiety, aby zmniejszyć szanse niekorzystnego dla odbiorcy źle wykrytego błędu.

Porównanie obu algorytmów

Oba algorytmy wykorzystane w projekcie radzą sobie lepiej w innych sytuacjach.
Bez głębszej analizy można stwierdzić, że CRC dobrze radzi sobie z wykryciem błędów bez próby ich naprawy, lecz już porównanie go względem sytuacji 1 stawia go daleko w tyle za kodem Hamminga.
Można zatem sformuować następujący wniosek. Jeśli przesył danych odbywa się w otoczeniu narażonym na zakłócenia to bezpieczniej będzie użyć algorytmu CRC, lecz może to spowodować opóźnienie komunikacji ze względu na większą ilość zapytań o ponowne przesłanie pakietu, który dotarł do celu uszkodzony. Natomiast w przypadku, gdy przesyłanie danych ma miejsce w kontrolowanym otoczeniu, a użytkownikowi zależy na jednorazowym przesłaniu danych to kod Hamminga będzie lepszym rozwiązaniem ze względu na dużo bardziej efektywny algorytm naprawiania błędów w dostarczonym pakiecie mając w pamięci większe ryzyko uszkodzonego pakietu.

7. Bibliografia

- https://eduinf.waw.pl/inf/alg/002_struct/0010.php
- https://pl.wikipedia.org/wiki/%C5%9Arednia_arytmetyczna
- https://pl.wikipedia.org/wiki/Histogram
- https://datko.pl/NiDUC2
- https://en.wikipedia.org/wiki/Cyclic_redundancy_check
- https://mfiles.pl/pl/index.php/Wykres_pude%C5%82kowy
- https://en.wikipedia.org/wiki/Five-number_summary