Turbo Koder - Implementacja i Testy

Jędrzej Stefańczyk, Klemens Zalewski

20.03.2024

1 Wstęp

Torbo kody to klasa wysokiej jakości kodów korekcyjnych, charakteryzujących się sprzężeniem zwrotnym dwóch dekoderów podczas dekodowania wiadomości.
By zaimplementować taki kod stworzymy koder 2 i niezbędny do jego pracy przeplatacz (*interleaver*) 2.2. Opiszemy algorytm pojedynczego dekodera 3 oraz sposób ich połączenia 3.1. Do testów niezbędny będzie generator sygnału 4.1, generator błędów 4.2 i dekoder błędów 4.3.
Testowanie algorytmu polega na wygenerowaniu ciągu bitów przez generator, zakodowaniu ich przez koder, uszkodzeniu sekwencji przez generator błędów i próbę odkodowania przez dekoder. Dekoder błędów porówna sekwencję wysłaną od odebranej i obliczy BER.

2 Koder

Główną funkcją kodera jest wygenerowanie dodatkowej informacji która pozwoli na poprawienie błędów w czasie dekodowania. W tej implementacji będziemy używać rekursywnych systematycznych kodów splotowych (*RSC - recursive systematic convolution codes*). Są to kody z pamięcią (rejestrem przesuwającym) i w przeciwieństwie do nie-rekursywnych, bit który zostaje dodany do rejestru zależy od tych już zapamiętanych. Kodery charakteryzuje: ilość pamięci - v; ilość stanów - $S = 2^v$; wielomiany generujące - $G = A_8, B_8$ - ich reprezentacja binarna wskazuje które bity pamięci należy podłączyć do wejścia i wyjścia; stosunek bitów wejściowych do wyjściowych (*rate*) - $R = \frac{a}{b}$.

2.1 Pojedynczy koder

Użyjemy kodera, który dawał dobre efekty w implementacjach turbo kodów u Hagenauer et al. (1996) - o pamięci $v = 4$ i wielomianach $G = \{21_8, 37_8\}$. Schemat tego kodera przedstawiony jest w 1.
W celu zwiększenia informacji, których dekoder będzie mógł użyć, koder zawsze zaczyna w stanie $S = 0$ (wszystkie bity pamięci są równe 0) oraz w nim skończy. By zapewnić zakończenie pracy w konkretnym stanie, musimy dodać co najmniej v bitów. Koder oblicza te bity i również je koduje oraz wypisuje je na oddzielnym wyjściu.
W koderach których używaliśmy, wyżej wspomniane bity zerujące są równe bitom w pamięci w odwrotnej kolejności, aczkolwiek nie możemy zagwarantować że jest tak dla każdego kodera.

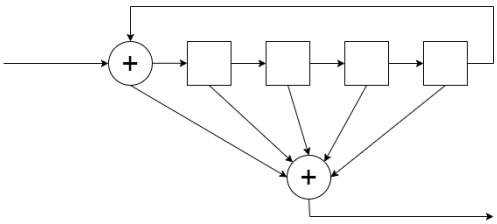

Figure 1: Schemat kodera $G = \{21_8, 37_8\}$

2.2 Przeplatacz

By zwiększyć ilość dodatkowych informacji turbo koder używa dwóch koderów. Zwykłe dodanie drugiego kodera niewiele pomaga, jako że da taki sam wynik. Użycie innego kodera jest trochę lepsze. By natomiast zmaksymalizować efekty drugiego kodera musi on operować na innych danych. Z tego powodu jego wejście przejdzie przez przeplatacz. Jest to element który zmienia kolejność bitów. Jego budowa jest bardzo ważna do maksymalizacji efektów dekodowania. Zdefiniowaliśmy dwa warunki dobrego przeplotu:

- Każde v sąsiednich bitów musi znaleźć się jak najdalej od siebie, a jeżeli po przeplocie którekolwiek z nich są w odległości mniej niż v od siebie to powinny być przynajmniej w innej kolejności. Kodery zasadniczo tworzą informacje o każdym v sąsiednich bitach, dlatego chcemy je jak najbardziej rozbić.
- Pierwszy i ostatni bit powinny znaleźć się jak najdalej od brzegów. Ponieważ nasze kodery zaczynają i kończą w znanym stanie, to mamy co do początkowych i końcowych bitów największą pewność w czasie dekodowania, a co za tym idzie chcemy by te bity były jak najdalej.

Stworzyliśmy następującą procedurę przeplotu:

1. Wejście jest ciągiem o długości podzielnej przez v.
2. Na koniec wyjścia dopisz co v-ty bit wejścia.
3. Jeżeli wyjdziesz poza długość wejścia to powtórz poprzedni krok zaczynając od bitu o 1 dalej niż ostatnio.
4. Jeżeli powtórzyłeś poprzedni krok v razy to przejdź do poniższego kroku.
5. Podziel stworzone wyjście na pół i odwróć obie połowy, po czym je sklej.

Diagram 2 przedstawia wynik tej procedury dla ciągu długości 8 i $v = 4$.

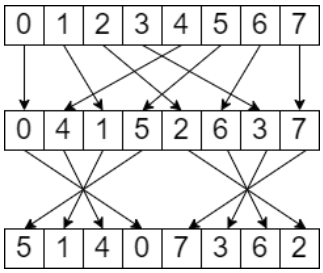

Figure 2: Diagram przeplotu dla ciągu długości 8 i $v = 4$

2.3 Turbo koder

Zdefiniujmy nasze Dane (*plaintext*) które chcemy przesłać i odczytać jako d, ich całkowita ilość to D, a d_k to k-ty bit danych.
Cały turbo koder składa się z dwóch koderów, które nie muszą być identyczne, ale w naszej implementacji są. Pierwszy koder koduje Dane i dokleja do nich swoje bity zerujące. Następnie ten ciąg przechodzi przez przeplatacz i zostaje zakodowany przez drugi koder. Przesłanie drugich bitów zerujących jest problematyczne. Udało nam się to zrobić, ale znacznie pogorszyło to działanie dekodera. Ostatecznie porzuciliśmy to rozwiązanie i w ogóle nie wysyłamy informacji zerującej drugiego kodera. Zamiast tego podczas dekodowania zakładamy, że kończy on w dowolnym stanie z równym prawdopodobieństwem. Wysłany pakiet składa się z Danych, pierwszego i drugiego zakodowanego ciągu. Na schemacie 3 przedstawiona jest ta procedura graficznie wraz z ilością bitów w każdym momencie. Jak widać ostateczny pakiet ma $3(D + v)$ bitów co daje stosunek całego kodera $R = \frac{D}{3(D+v)}$. Dla dużych D zbliża się on do $\frac{1}{3}$, acz niższy daje lepsze efekty.

3 Dekoder

Dekodowanie kodów splotowych zazwyczaj dokonuje się poprzez algorytmem Viterbiego. Wykorzystuje on prawdopodobieństwa przejść stanów w koderze by znaleźć najbardziej prawdopodobną sekwencję wejściową. W naszej implementacji skorzystamy ze zoptymalizowanej modyfikacji algorytmu BAHL, przez Pietrobon and Barbulescu (1994).
Algorytm przyjmuje x - odebrane Dane oraz y - odebrana zakodowaną sekwencję. Natomiast zwraca L - prawdopodobieństwa dla każdego bitu. Algorytm sprowadza się do obliczenia dwóch trójwymiarowych tablic prawdopodobieństw: $\alpha[i][k][m]$ i $\beta[i][k][m]$. α reprezentuje prawdopodobieństwa od początku do końca, natomiast β od końca do początku. Liczymy je w dwie strony ze względu na problem w oryginalnym algorytmie wynikający z kilku ścieżek o równym prawdopodobieństwie. W algorytmie definiujemy następujące oznaczenia: $i \in (0, 1)$ - przewidywany bit, $k \in (D + v)$ - który element rozpatrujemy, $m \in S$ - stan kodera który rozpatrujemy. Dla prostoty implementacji numer stanu odpowiada dziesiętnej reprezentacji bitów w pamięci kodera ($m = 3$ odpowiada pamięci 0011).
Pierwszym krokiem jest obliczenie ogólnych prawdopodobieństw dla każdego możliwego stanu:

$$\delta[i][k][m] = exp(\frac{1}{2}(x_k * i + y_k * Y(i, m)))$$

By równanie działało poprawnie, przed podstawieniem zamieniamy 0 i 1 na -1 i 1 wzorem $b = (2a-1)$. Funkcja Y zwraca bit jaki zakodował by koder gdyby znajdował się w stanie m i na wejściu otrzymał i. Wynik tej funkcji również zostaje zamieniony na -1 i 1. W oryginalnym równaniu, mnożymy przez wartość, zależną od błędu kanału, ale dla wartości innych niż $\frac{1}{2}$, otrzymywaliśmy znacznie gorsze efekty. Następnym krokiem jest wyznaczenie α i β ze wzorów:

$$\alpha[i][k][m] = \delta[i][k][m] * (\alpha[0][k-1][S_{\leftarrow}(0, m)] + \alpha[1][k-1][S_{\leftarrow}(1, m)])$$

$$\beta[i][k][m] = \beta[0][k+1][S_{\rightarrow}(i, m)] * \delta[0][k+1][S_{\rightarrow}(i, m)] + \beta[1][k+1][S_{\rightarrow}(i, m)] * \delta[1][k+1][S_{\rightarrow}(i, m)]$$

Funkcja $S_{\leftarrow}(i, m)$ zwraca stan w jakim był koder jeżeli znajduje się w stanie m i wcześniej otrzymał bit i. Natomiast funkcja $S_{\rightarrow}(i, m)$ zwraca stan w którym będzie koder jeżeli znajduje się w stanie m i otrzymuje bit i. Oba te wzory są rekurencyjne, dlatego zaczynamy je obliczać od drugiego (lub przedostatniego w wypadku β) elementu. Tablice początkowo całe wypełnione są zerami i wstępnie wypełniamy je w następujący sposób: $\alpha[0][1][0] = \delta[0][0][1], \alpha[1][1][0] = \delta[1][0][1], \beta[0][D + v][S_{\leftarrow}(0, 0)] = 1, \beta[1][D + v][S_{\leftarrow}(1, 0)] = 1$.

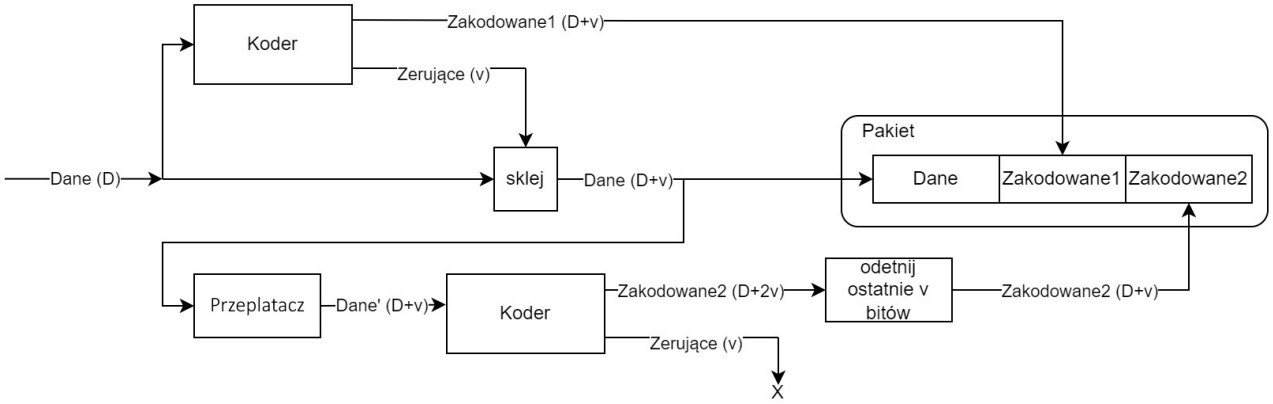

Figure 3: Diagram turbo kodera

Robimy tak ze względu na pewność zaczęcia/ zakończenia w stanie 0. Ponieważ drugi dekoder nie kończy w stanie 0, gdy dekodujemy jego dane, ustawiamy $\beta[i][D+v][m]$ dla każdego m i i na równie prawdopodobną. Ostatnim krokiem jest wyznaczenie prawdopodobieństwa dla każdego bitu ze wzoru:

$$L[k] = \frac{\sum_m \alpha[1][k][m] * \beta[1][k][m]}{\sum_m \alpha[0][k][m] * \beta[0][k][m]}$$

W ten sposób im bardziej $L[k]$ jest dodatnie tym większe prawdopodobieństwo, że $d_k = 1$, natomiast im bardziej jest ujemna tym większe prawdopodobieństwo, że $d_k = 0$. Z tego wynika, że w celu ostatniego zdekodowania należy użyć poniższego wzoru:

$$d'_k = \begin{cases} 1 & \text{dla} \quad L[k] \geq 0 \\ 0 & \text{dla} \quad L[k] < 0 \end{cases}$$

3.1 Sprzężenie zwrotne

Turbo koder bierze swoją nazwę ze sprzężenia zwrotnego dwóch dekoderów. By połączyć dwa dekodery chcemy by poprawnie wymieniały się informacjami. W tym celu definiujemy "z" jako dodatkową informację wygenerowaną przez koder, którą dekoder odzyskał. Oraz "W" czyli informacje odzyskane z oryginalnych Danych. Wynik dekodera możemy zapisać jako $L = W + z_1 + z_2$. Następnie będziemy wyniki działań obu dekoderów odejmować w taki sposób by j-ty z nich otrzymał jako x tylko $W + z_{!j}$. Schemat 4 przedstawia te operacje. $z_2 = 0$ w pierwszej iteracji.

4 Dodatkowe komponenty

4.1 Generator sygnału

By uniknąć w czasie testów ominięcia jakiegoś przypadku brzegowego, sekwencje do zakodowania są generowane pseudo-losowo. W tym celu zaimplementowaliśmy prosty rejestr przesuwający z liniowym sprzężeniem zwrotnym ($LSFR$) o wielomianie generującym $x^{15} + x^{14} + 1$. Jego okres to 32767, a ponieważ planujemy wykonywać 1000 testów dla wiadomości długości maksymalnie 16 bitów nie przejdziemy przez wszystkie stany.
Implementację zapożyczyliśmy z Wikipedia (2024). Do wybrania stanu początkowego używamy wbudowanej funkcji losowej języka.

4.2 Generator błędów

By mieć kontrolę nad BER-em kanału chcemy móc przekłamywać konkretną ilość bitów w każdej wiadomości. W tym celu tworzymy tablicę długości $3(D+v)$ i konkretną ilość elementów ustawiamy na 1, a pozostałe na 0. Następnie w każdej wiadomości odwracamy ten bit na którego miejscu pojawia się 1 w naszej tablicy. Przed każdym takim przekłamaniem wywołujemy na tablicy wbudowaną funkcję *shuffle()*, która w losowy sposób ustawia elementy tablicy.
By upewnić się o losowości funkcji z których korzystamy sprawdziliśmy ich implementację. Autorzy używają algorytmów z rodziny PCG, które dają dobrą równomierność rozkładu - Godot (2024).

4.3 Detektor błędów

Ponieważ prowadzimy testy w programie, a nie fizycznie możemy po prostu porównać wiadomość zdekodowaną przez dekoder, z tą wylosowaną na początku. Co za tym idzie sprawdzamy tylko pierwsze D bitów. Błędy na dodatkowych bitach nie wpływają na nasze dane, dlatego ich nie liczymy.
Detektor zlicza ilość wiadomości w których pojawił się jakikolwiek błąd oraz ilość przekłamanych bitów. Pozwala nam to na koniec testu obliczyć bitową stopę błędu (BER) oraz poziom błędów pakietów (PER).

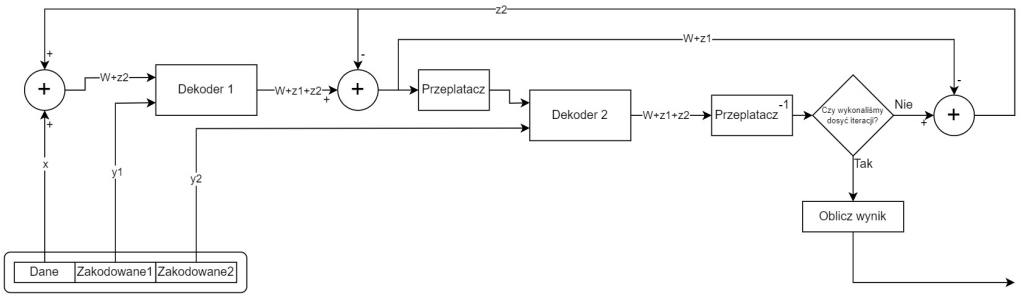

Figure 4: Diagram turbo dekodera

5 Testy

Wykonaliśmy testy dla wiadomości długości 8. Dla każdego ustawienia wykonaliśmy 1000 powtórzeń testu. Ilość przekłamanych bitów zwiększaliśmy o jeden od 0 do 12. Wykonywaliśmy generalnie małe ilości iteracji ze względu na długi czas pracy dla wielu iteracji. Wykres 5 przedstawia otrzymane przez nas wyniki.
Jak widać koder działa bardzo dobrze gdy poniżej $\frac{1}{5}$ wiadomości jest przekłamana, a gdy przekroczymy $\frac{1}{3}$ zaczyna pogarszać zamiast poprawiać. Ilość iteracji ma pewien wpływ na jakość dekodowania.

References

Godot (2024). Godot engine documentation - random number generation. Wersja: 4.11.

Hagenauer, J., E. Offer, and L. Papke (1996). Iterative decoding of binary block and convolutional codes. *IEEE Transactions on information theory 42*(2), 429–445.

Pietrobon, S. S. and A. S. Barbulescu (1994). A simplification of the modified bahl decoding algorithm for systematic convolutional codes. In *National Conferenc Publication-Institution of Engeneers Australia NCP*, pp. 1073–1077. Citeseer.

Wikipedia (2024). Linear-feedback shift register. Dostęp: 17.03.2024.

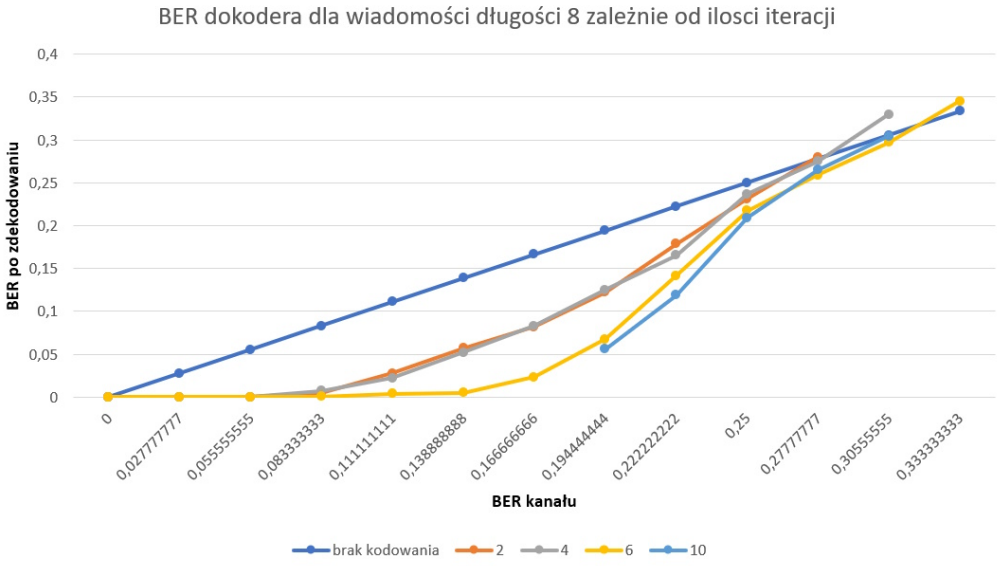

Figure 5: Wyniki testu