Универзитет у Београду
Електротехнички факултет

Извештај домаћег задатка из Основа Телекомуникација

Симулација телекомуникационог канала са LZW кодером/декодером и заштитним кодером са понављањем уз алгоритам већинског одлучивања

Ментор:
доц. др Срђан Бркић
доцент

Студент:
Владимир Јанковић
2018/0121

Београд, Април 2021.

Садржај

1. Увод

Телекомуникациони канал представља медијум преко кога се преносе информације са једног краја на други крај канала. Циљ телекомуникационог канала је да обезбеди поуздан пренос порука. На жалост, не постоји идеалан комуникациони канал, већ ће увек постојати неки вид сметње у каналу било да је у виду спољашњег или унутрашњег фактора. Из тог разлога се напорно ради на томе да се поруке заштите на што бољи начин од сметњи у каналу и да се осигура што поузданији могући пренос.

Један начин да се заштите поруке је коришћење блок кодера. Блок кодери прихватају ***k*** бита и кодирају их у ***n*** бита уношењем редундансе ради заштите и откривање грешака. Један тип блок кодера су блок кодери са понављањем који један бит понављају ***n*** пута и њихова ознака је ***(n, 1)***.

У симулацији се разматрају два кодера са понављањем и то ***(3, 1)*** и ***(5, 1)*** који један бит понављају 3 и 5 пута, респективно. Пошто канал може да унесе грешке у одређеним битима, са друге стране канала вршимо исправљање бита помоћу већинског одлучивања. Ако у групи од 3, односно 5 бита има више јединица него нула, сматрамо да је вредност поновљеног бита једнака јединици, у супротном је вредност једнака нули. На тај начин се реконструише секвенца бита која је прошла кроз канал.

Извор генерише секвенцу симбола за коју треба прво извршити компресију пре него што се пошаље у заштитни кодер па у канал. За компресију поруке коју генерише извор, а касније и за њену декомпресију, користи се Lempel-Zivov алгоритам, који је погодан за поруке које су велике дужине или имају велику корелацију у времену.

На крају сваке симулације се пореде на колико симбола се разликују оригинална и примљена секвенца симбола за различите вредности вероватноће грешке коју канал уноси. Резултат се приказује на графику који показује однос вероватноће грешке по симболу и вероватноћа грешке у каналу, у случају да се користи заштитни кодер са понављањем (3, 1) и у случају да се користи понављање (5, 1).

2. Рад програма

Извор који генерише оригиналну секвенцу симбола је у виду текстуалног фајла, чији се садржај учитава у одређену променљиву ради лакше обраде. Из те променљиве се чита симбол по симбол и смешта у један низ симбола који ће касније послужити за поређење са секвенцом која је пристигла на другу страну. Паралелно са читањем и смештањем у низ симбола, формира се иницијални речник тако што се у речник смешта свака појава новог симбола. Сматра се да извор генерише четири различита симбола S = {‘A’, ‘B’, ‘C’, ‘D’} и да је дужина поруке N = 1000 симбола.

Следећи корак је да се изврши компресија низа симбола ради ефикаснијег преноса поруке кроз канал. Компресиони LZW алгоритам кодира низ симбола у нумерички низ који је лакше претворити у бинарну секвенцу него саме симболе. Алгоритам узима иницијални речник и током кодирања га проширује са новим комбинацијама симбола. Рад алгоритма је једноставан и може се приказати на следећи начин. Променљива K служи за смештање следећег пристиглог карактера. Променљива W је иницијално празна. У првом пролазу, симбол у променљивој K се мора наћи у речнику (јер је то први симбол који је речник записао), па се променљивој W додели симбол из K, а променљива K прима следећи симбол.

Нова променљива WK служи за конкатенацију симбола из W и K у један нови симбол. Ако се симбол из WK не налази у речнику, тада се он уписује у речник, алгоритам избацује нумеричку вредност из речника за симбол у променљивој W, а W добија симбол из K. У случају да се симбол из WK налази у речнику, тада W добија симбол из WK. У променљивој K се смешта следећи симбол из низа. Алгоритам ради све док се не заврши унос свих симбола из низа. На крају, који год симбол да је остао у W, избацује се нумеричка вредност тог симбола из речника. Тако се од низа симбола добио нумерички низ.

Нумерички низ се сада може представити у бинарном запису. Програм тражи вредност највећег елемента у низу и на основу њега гледа колико бита је потребно за запис свих бројева у низу. Циљ је уклањање што више пратећих нула у бинарном запису, тј. тражи се најмањи број бита за представљање највећег броја у низу. Ако је то број 7, довољно је записати га са 3 бита (111), а ако је то број 19, довољно је 5 бита (10011).

Таква бинарна секвенца улази у заштитни кодер са понављањем. Програм симулира рад понављања (3, 1) и (5, 1) коришћењем петљи и помоћног низа. Од овог тренутка се независно разматрају два паралелна тока извршавања и две бинарне секвенце чији је сваки бит поновљен 3 пута код једне и 5 пута код друге. Повећањем оригиналне бинарне секвенце повећава се и дужина трајања преноса целе поруке кроз канал, али да ли се тиме и повећава заштита оригиналне поруке?

Претпоставља се да канал поседује вероватноћу грешке по биту означену са *p*. Дакле сваки бит има вероватноћу *p* да промени своју вредност, тј. да се инвертује. Пошто су грешке у каналу неизбежне, пожељно је имати добар алгоритам за уклањање грешака које су се јавиле током проласка кроз канал.

Алгоритам већинског одлучивања је имплементиран у симулацији тако што се броји колико јединица је пристигло у групи од *n* бита где *n* означава који број понављања је коришћен. Тако се формира један нови бинарни низ, који се претвори у нумерички и шаље на улаз у алгоритам за декомпресију поруке.

Декомпресиони LZW алгоритам, је мало компликованији од компресионог LZW-а. Користи нешто више променљивих, али користи исти иницијални речник као и компресиони LZW. Алгоритам поседује две мане, од којих је за једну извршена оптимизација, а у случају друге мане је алгоритам немоћан и пропада. Прва мана је у случају да се иста нумеричка секвенца која је компресована са LZW, доведе на LZW декомпресију и дође до губљења одређеног симбола у поруци. То је решено на начин да се кодна вредност из речника за прво карактер који се одмах исписује памти у помоћну променљиву C која се у коду користи за конкатенацију.

Друга и озбиљнија мана се јавља ако у декомпресију уђе вредност која се не налази у речнику и алгоритам то не може да разреши. Једина опција је да се врши ретрансмисија оригиналне поруке. У случају да се десила грешка у декомпресији, програм враћа негативну вредност кроз низ и та вредност се испитује да ли је потребно ново слање поруке кроз канал или је декомпресија успешно извршена.

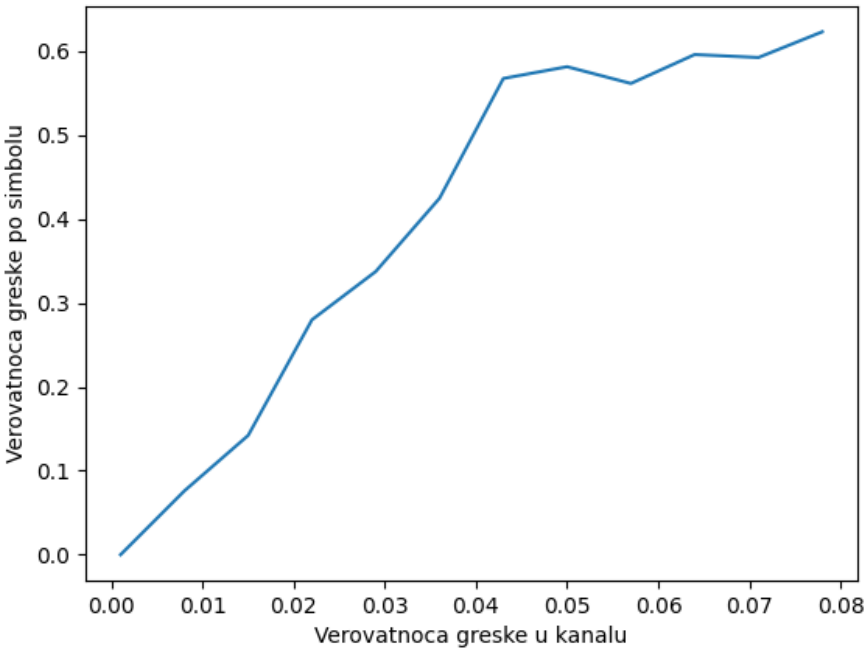
Међутим, то што се декомпресија успешно извршила не значи да је и порука исправно реконструисана. Последњи корак у програму је да се пореди низ карактера оригиналне секвенце симбола и примљене. Да би се та функција исправно извршила, потребно је да се одреди да ли је настала промена у дужини низа. Ако су оба низа исте дужне, програм иертра кроз оба низа и пореди симболе на истим позицијама на једнакост. Бројач који прати број промењених симбола се на крају подели са дужином оригиналног низа и тиме се добије вероватноћа грешке по симболу. Ако су низови различитих дужина, итерира се по дужини краћег низа.

Излаз програма представља график зависности вероватноћа грешака по симболу и по биту. Програм ради са низом вероватноћа грешака по биту у каналу у опсегу од 10^{-3} до 10^{-1} и за сваку од тих вероватноћа се кроз већинско одлучивање, декомпресију и проверу грешака, добијају две вредности вероватноћа грешака по симболу, једна за понављање (3, 1) и друга за понављање (5, 1).

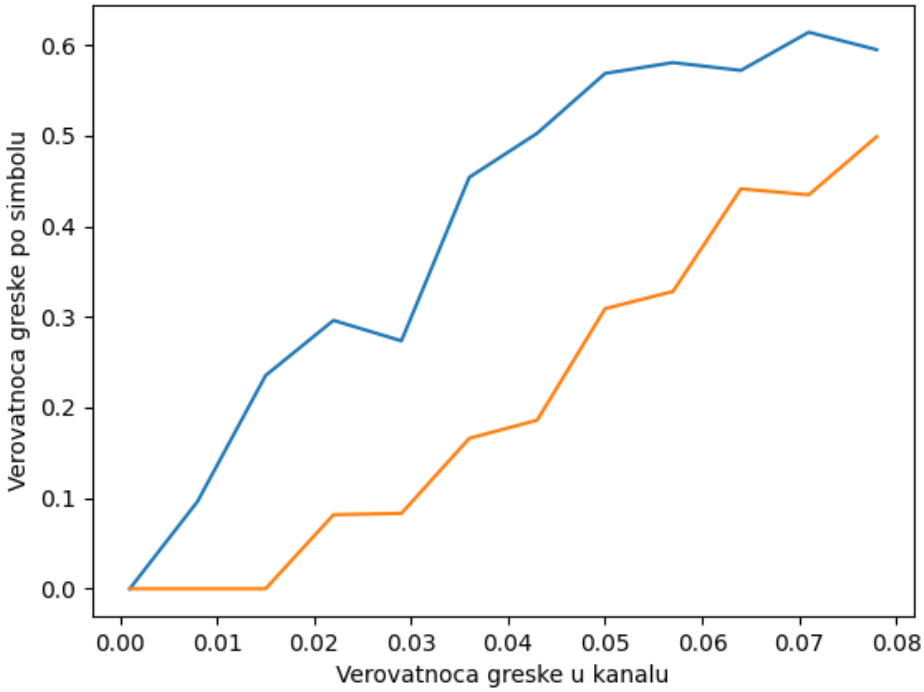
За сваку вероватноћу грешке у каналу, ради прецизнијег анализирања се радило по 10 симулација за (3, 1) понављање, а као резултат се узима средња вредност вероватноће грешке по симболу и тај резултат се убацује у нови низ у ком се смештају просечне вредности вероватноћа грешака по симболу за дату вероватноћу грешке у каналу. То важи и за понављање (5, 1). Након што се формирају два низа просечних вероватноћа грешака по симболу, цртају се два графика у језику Python.

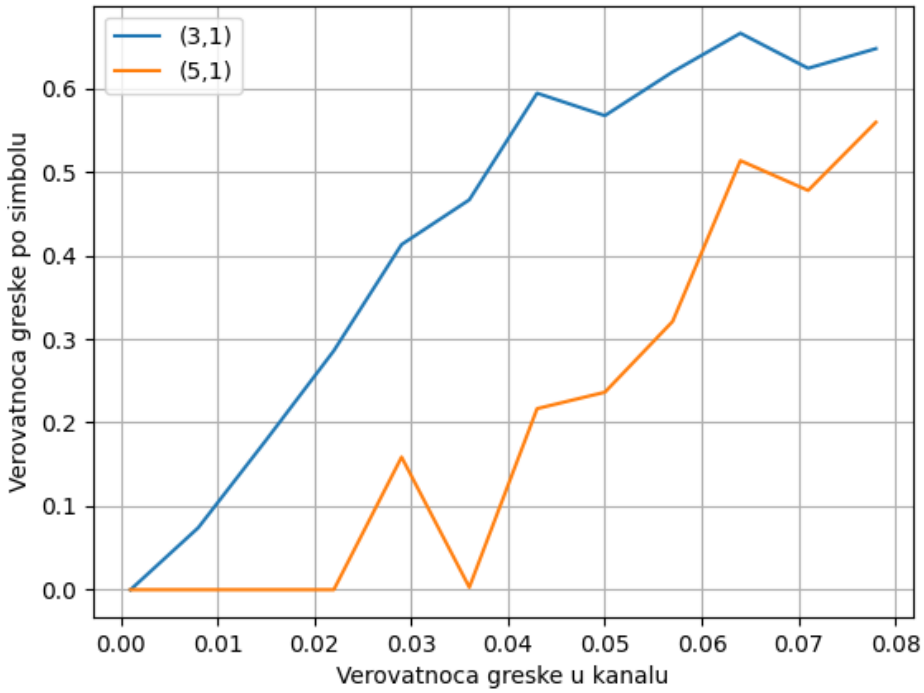
У наставку су дати изгледи графика који представљају излаз програма, као и низ симбола које је генерисао извор.

```
ABBAAABDACCABCCBAAACABCABBAACCAACCABCAABACCCABACCAAAAADAACABAA
ACCABBAAACACAADABABACACCACBACCBAAACABACAABAACACBACAABABACACBAB
CABBACCCBABABABAAACABAACABAABACABACCABBABACDCABABABADAAABCABAC
ABBACAABACAABDDACACABABABCAABACDBADBDBACDBCCDDCABACCDBABAAAACD
ACCABCAABACCCABACCAAAAAAACABACAACCBAAACAABACACCCCCCCACACBACACB
AAACABDCAABAACACBACDAABACACABABCABBACCBAAABBBBAAAAAAACABACABAC
ABACCABBABACACABABAAAACDDACABACACABACDCDBACDDCCCABABABACABDCAD
ABBAAABBACCABCCBAACABCABBAACCABACCABCAABACCCDBACDAADCBAACABACA
ACCABBAAACACAABBABACACCACBACACBAAACABACAABAACDBACAABABACACABAB
BACDCABDCDDCACACABACABAACDBACACABACCABCCCCCABACABBACACAAACABAA
ABBACAABACABACACDCABABABACAABACABBAAABBACABCCBAAACABCABBAACCAB
ACCBAAACABACACCCCCCCACCACBACACBAAACABACABAADACBACAABABACACABAB
ACABDCABDCCABBBACCDCABBBBACDDADABACDABADCACDBADBABAAACABACABAC
ABBACABACAABABBACACACCABABAAACAABBAABBACCDBCCBAADCABCABBAACCAB
ACCBCADBACCCABACCAAAAABAACABACADBABDBACDBABBACCCABCABBCACCDBDC
ABABDBABDCCABCCABCCABBBCDCABBAACABBADCCBABABABAAACABAACDDCABAC
```



Слика 1: Симулација за (3,1) понављање





Слике 2 и 3: Симулације за (3, 1) и (5, 1) понављања

3. Програмски код

```
from math import *
from random import *
import matplotlib.pyplot as plt


####################----------FUNCTIONS----------##############################

def lzw_code(characters1, dictionary1):
    w = ""
    output = []
    for index0 in range(0, len(characters1), 1):
        k = characters1[index0]
        wk = w + k
        ind2 = False

        tmp = dictionary1.values()
        if wk in tmp:
            ind2 = True

        if ind2:
            w = wk
        else:
            for index1 in range(0, len(dictionary1), 1):
                if w == dictionary1[index1]:
                    output.append(index1)
                    break
            dictionary1[len(dictionary1)] = wk
            w = k

    for index0 in range(0, len(dictionary1), 1):
        if w == dictionary1[index0]:
            output.append(index0)
            break
    return output


def binary_repetition_code(n, array):
    output = []
    for nums in array:
        for bit in nums:
            for a in range(0, n, 1):
                output.append(bit)
    return output



def channel(p, signal):
    for bit in range(0, len(signal), 1):
        r = random()
        if r < p:
            signal[bit] = '1' if signal[bit] == '0' else '0'
    return signal
```

```
def lzw_decode(numbers, dictionary2):
    output = []
    old = numbers[0]
    ################################################

    tmp = dictionary2.keys()
    if old not in tmp:
        # print("NUMBER NOT IN DICTIONARY! CRITICAL ERROR!")
        t = [-1]
        return t
    s = dictionary2[old]
    output.append(dictionary2[old])  #
    c = s[0]

    ################################################
    for index3 in range(1, len(numbers), 1):
        new = numbers[index3]
        tmp = dictionary2.keys()
        if new in tmp:
            s = dictionary2[new]
        else:
        ############################################################################

            if old not in tmp:
                # print("NUMBER NOT IN DICTIONARY! CRITICAL ERROR!")
                t = [-1]
                return t

        ############################################################################
            s = dictionary2[old]
            s = s + c
        output.append(s)
        c = s[0]
        ############################################################################

        if old not in tmp:
            # print("NUMBER NOT IN DICTIONARY! CRITICAL ERROR!")
            t = [-1]
            return t
        dictionary2[len(dictionary2)] = dictionary2[old] + c

        ############################################################################
        old = new
    tmp1 = []
    for word in output:
        for index2 in range(0, len(word), 1):
            tmp1.append(word[index2])
    return tmp1
```

```
def majority_decision(n, array, bitgroup):
    output = []
    ones = 0
    count = 0
    for bit in array:
        if bit == '1':
            ones += 1
        count += 1
        if count == n:
            out = '1' if ones > n / 2 else '0'
            output.append(out)
            ones = 0
            count = 0
    g = 0
    st = ""
    tmp = []
    for out in output:
        st = st + out
        g += 1
        if g == bitgroup:
            tmp.append(st)
            st = ""
            g = 0
    return tmp


def check_for_errors(array1, array2):
    min1 = len(array2) if len(array1) >= len(array2) else len(array1)
    cnt = len(array1) - len(array2) if len(array1) - len(array2) > 0 else 0
    for index2 in range(0, min1, 1):
        if array1[index2] != array2[index2]:
            cnt += 1
    return cnt / len(array1)


####################----------MAIN----------####################################


text = open("test.txt", 'r')
text = text.read()
print("\nSent message:")
print(text, end='\n\n')


characters = []  # Niz karakteri iz teksta
dictionary = {}  # recnik za LZW
init_dictionary = {}
seed()



for ch in range(0, len(text), 1):  # pravljenje recnika i niza simbola
    if text[ch] == '\n':
        continue
    characters.append(text[ch])
```

```
    if len(dictionary) == 0:
        dictionary[len(dictionary)] = text[ch]
    else:
        ind1 = 0
        for index in range(0, len(dictionary), 1):
            if text[ch] == dictionary[index]:
                break
            ind1 = ind1 + 1
        if ind1 == len(dictionary):
            dictionary[len(dictionary)] = text[ch]

init_dictionary = dictionary.copy()  # Cuvanje inicijalnog recnika

lzw_out = lzw_code(characters, dictionary)  # izlazna sekvenca LZW algoritma
print("Sequence after LZW: ")
print(lzw_out, end='\n\n')

maxnumber = max(lzw_out)  # Broj bita za kodovanje odgovara broju bita porebnih
                          # za kodovanje najveceg broja
bits_to_format = log2(maxnumber) + 1 \
    if log2(maxnumber) - floor(log2(maxnumber)) == 0 \
    else ceil(log2(maxnumber))
bits = "0" + str(bits_to_format) + "b"  # Formatiranje u binarnu sekvencu
for index in range(0, len(lzw_out), 1):
    lzw_out[index] = format(lzw_out[index], bits)
print("Binary representation: ")
print(lzw_out, end='\n\n')

channel_in_3_1 = binary_repetition_code(3, lzw_out)  # Izlaz iz zastitnog kodera
                                                     # sa ponavljanjem (3,1)
print("Channel input sequence (3,1) coding: ")
print(channel_in_3_1, end='\n\n')

channel_in_5_1 = binary_repetition_code(5, lzw_out)  # Izlaz iz zastitnog kodera
                                                     # sa ponavljanjem (5,1)
print("Channel input sequence (5,1) coding: ")
print(channel_in_5_1, end='\n\n')


channel_error = []              # Niz verovatnoca gresaka u kanalu
for it in range(1, 80, 7):
    channel_error.append(it / 1000)
pe_3_1 = []                # Niz prosecnih verovatnoca gresaka po simbolu (3,1)
pe_5_1 = []                # Niz prosecnih verovatnoca gresaka po simbolu (5,1)

repeat = 10



for index in range(0, len(channel_error), 1):
    print("#################################################", end='\n\n')
    print("        Simulation ", end='')
    print(index + 1, end='\n\n')
    print("Channel error: ", end='')
    print(channel_error[index])
    avg_3_1 = 0
    avg_5_1 = 0
```

```
    for o1 in range(0, repeat, 1):
        while True:
            channel_out_3_1 = channel(channel_error[index],
channel_in_3_1.copy())  # Simulacija Kanala
            # print("Channel output sequence: ", end='')
            # print(channel_out, end='\n')

            channel_out_3_1 = majority_decision(3, channel_out_3_1,
bits_to_format)  # Vecinsko odlucivanje
            # print("Majority decision: ", end='')
            # print(channel_out, end='\n')

            for index1 in range(0, len(channel_out_3_1), 1):  # Formatiranje u
cele brojeve
                channel_out_3_1[index1] = int(channel_out_3_1[index1], 2)
            # print("LZW decompression input: ", end='')
            # print(channel_out, end='\n')

            channel_out_3_1 = lzw_decode(channel_out_3_1,
init_dictionary.copy())  # Dekodovanje poruke
            if channel_out_3_1[0] == -1:
                dummy = 1
                # print("LZW DECOMPOSITION FAILURE! RETRANSMISSION REQUESTED!")
                # print("RETRANSMITTING . . .", end='\n\n')
            else:
                break
        avg_3_1 += check_for_errors(characters, channel_out_3_1)

    for o1 in range(0, repeat, 1):
        while True:
            channel_out_5_1 = channel(channel_error[index],
channel_in_5_1.copy())  # Simulacija Kanala
            # print("Channel output sequence: ", end='')
            # print(channel_out, end='\n')

            channel_out_5_1 = majority_decision(5, channel_out_5_1,
bits_to_format)  # Vecinsko odlucivanje
            # print("Majority decision: ", end='')
            # print(channel_out, end='\n')

            for index1 in range(0, len(channel_out_5_1), 1):  # Formatiranje u
cele brojeve
                channel_out_5_1[index1] = int(channel_out_5_1[index1], 2)
            # print("LZW decompression input: ", end='')
            # print(channel_out, end='\n')

            channel_out_5_1 = lzw_decode(channel_out_5_1,
init_dictionary.copy())  # Dekodovanje poruke
            if channel_out_5_1[0] == -1:  # Pri neuspesnom dekodovanju
                dummy = 1
                # print("LZW DECOMPOSITION FAILURE! RETRANSMISSION REQUESTED!")
                # print("RETRANSMITTING . . .", end='\n\n')
            else:
                break
        avg_5_1 += check_for_errors(characters, channel_out_5_1)
```

```
    pe_3_1.append(avg_3_1 / repeat)
    print("Symbol error 3-1: ", end='')
    print(pe_3_1[index])
    pe_5_1.append(avg_5_1 / repeat)
    print("Symbol error 5-1: ", end='')
    print(pe_5_1[index])

print("\nError probability (3,1): ")
print(pe_3_1)
print("\nError probability (5,1): ")
print(pe_5_1)

plt.plot(channel_error, pe_3_1, label="(3,1)")
plt.plot(channel_error, pe_5_1, label="(5,1)")
plt.xlabel("Verovatnoca greske u kanalu")
plt.ylabel("Verovatnoca greske po simbolu")
plt.grid()
plt.legend()
plt.show()
```

4. Закључак

На основу резултата симулација долазимо до основног закључка да са повећањем вероватноће грешке у каналу, расте и вероватноћа грешке по симболу. Ако се пореде резултати симулација за понављање (3,1) и (5,1), може се приметити да се боље показала заштита са понављањем (5,1) што је и за очекивати. Канал мора да инвертује више бита код понављања (5,1) да би променио вредност једног оригиналног бита него код понављања (3,1). Ако се прати та логика, са повећањем *n* се смањује вероватноћа инвертовања оригиналног бита који се понављао. Цена тога је повећан број бита за пренос, па за велико *n* постаје непрактично, због дугог трајања преноса поруке.

Још један битан закључак који се може уочити је релативно велика вероватноћа грешке по симболу при порасту вероватноће грешке у каналу. Кад LZW изврши успешну декомпресију погрешне нумеричке секвенце, таква грешка може изазвати велике промене у реконструкцији поруке. Са порастом величине поруке, грешке у каналима могу при декомпресији формирати другачији речник од оригиналног што додатно компликује ситуацију.

Једна оптимизација би била да уместо већинског одлучивања, прихватамо само оне вредности које имају све бите једнаке (111 и 000 за (3,1); 11111 и 00000 за (5,1)), а за остале случаје сматрамо да је постојала грешка.

Дакле, иако су грешке и сметње у каналу неизбежне, коришћењем алгоритма заштите се повећава шанса да порука буде поуздано пренета кроз медијум на другу страну и та поузданост расте све више, коришћењем бољих кодова за заштиту као што су Хемингов код и циклични кодови као и савременијих и много напреднијих кодова (Турбо код, LDPC, …).