Studying Single Parity Check Codes and its Error Correction

Name:	**Aditya Mishra**
Roll Number:	21013
University:	IISER Bhopal
Department:	EECS
Course:	Principles of Communications (ECS301)
Instructor:	Prof. Ankur Raina
Date of Submission:	April 17, 2024

1 Introduction

Single Parity Check is a widely used method for identifying a single-bit error within transmitted data. It involves ensuring that the parity of a specific bit, such as '1', remains either even or odd during transmission, and is subsequently verified upon receipt of the data. Detection of an error occurs if the parity of the designated bit differs. However, this technique is limited to detecting single-bit errors and is ineffective in detecting burst errors.

1.1 Error

When data is sent from one device to another, there is no guarantee that the transmitted data will match the received data. In instances where the received data differs from the transmitted data, it is classified as an error. Errors can be categorized into two types:

1. **Single-Parity Error** - It involves an error in only one bit of the data
2. **Burst Error** - It entails errors in multiple bits of the data.

This report will exclusively address Single-Parity Errors, focusing on their detection and correction methods.

1.2 Single-Bit Error

A single-bit error refers to a situation where only one bit within the given data is altered, transitioning from 0 to 1 or from 1 to 0. Such errors are among the least common in data transmission, as the majority of errors tend to be burst errors.

1.3 Parity

Parity means the total count of any bit (0 or 1) is either even or odd. They are of two types:

1. **Even Parity** - When number of 1's (or 0's) in the data is even, we call it even parity of the bit 1 (or 0).
2. **Odd Parity** - When number of 1's (or 0's) in the data is odd, we call it odd parity of the bit 1 (or 0).

2 Methods

There are wide range of available methods for detecting and correcting single-parity errors. In this report we will be only focusing on simple parity checks and Hamming codes.

2.1 Simple-Parity Check

Simple-parity check is a straightforward approach to detecting parity errors. Initially, the count of 1's (or 0's) in the data to be transmitted is tallied, and then its parity is examined. A specific parity (even or odd) is designated for a bit agreed upon by both the sender and the receiver. In this instance, the selected bit is 1, and its parity is set as even, although the same principle can be applied to 0 and odd parity.

Upon counting the number of 1's, if it turns out to be odd, 1 is appended to the data to ensure that the parity of 1 is even. This additional bit is termed the parity bit. Conversely, if the count of 1's in the data is even, the parity bit is set to 0 and included in the data.

For example, consider the data to be transmitted as **100101**. Here, the count of 1's is 3, which is an odd number. Therefore, the parity bit will be set to 1, resulting in the final data being **1001011**.

At the receiver's end, the parity of 1 is verified. If the parity remains 1, indicating an even parity, no single-bit error is present in the data. However, if the parity is odd, an error is detected in the data, prompting re-transmission until the error is rectified. Given below is the python implementation of this method including error during transmission:

```python
# Aditya Mishra - 21013

import random

binaryNumber = '' # Global variable for binary number

# Initially, the even parity of 1 is set to true, assuming that the initial count of
    1s is even.
# Sender's end
while True: # Loop to check wrong binary input
    binaryNumber = input("Enter binary number ")
    correctInput = True
    for num in binaryNumber:
        if (num == '0'):
            # if current bit is 0 do nothing
            continue
        elif (num == '1'):
            # if current bit is 1, change parity from even to odd and odd to even
            evenParity = not evenParity
        else:
            # if input is not 0 or 1, ask for input again
            correctInput = False
            break
    if (correctInput == True):
        break
    print("Wrong Input! Try again")
parityBit = '' # global variable for the bit to be added to the binary string
# Keeping the parity of 1 even
if (evenParity):
    parityBit = '0'
else:
    parityBit = '1'

# Adding the parity bit to binary number
binaryNumber += parityBit
print("Transmitted binary string is: " + binaryNumber)

# Randomly adding noise to the string
i = random.randint(0, len(binaryNumber) - 1)
charList = list(binaryNumber)
charList[i] = str(random.randint(0,1))
```

```python
binaryNumber = ''.join(charList)

# Receiver's end
checkParity = True
for bit in binaryNumber:
    if (num == '0'):
        continue
    else:
        checkParity = not checkParity
print("Received binary string is: " + binaryNumber)
if (checkParity):
    print("No errors in parity of 1")
else:
    print("Error found in parity of 1")

#Aditya Mishra - 21013
```

Listing 1: Implementation of Single-Parity Check with transmission error in Python.

The code is segmented into three parts, each preceded by a comment. Initially, the `random` library is imported to introduce noise in the subsequent section. Two global variables are initialized: `binaryNumber` and `evenParity`. `binaryNumber` stores the data to be transmitted as a string, while `evenParity` serves to verify the parity of 1 in the data as a boolean. It is set to *True*, reflecting that the initial count of 1's is zero, which is an even number.

The code is divided into the following segments:

1. **Sender's End** - This section contains a while loop responsible for receiving input from the user to obtain the binary string intended for transmission. Within this loop, the count of 1's in the data is calculated, subsequently determining the value of the `evenParity` variable. Additionally, it validates whether the input data is in binary format; if not, the loop iterates again to prompt the user for another input. Afterward, the code determines the parity bit based on the `evenParity` variable and appends it to the original data.

2. **Noise Addition** - In this segment, a random index within the length of the binary data is chosen, and its value is randomly altered to either 0 or 1. This step introduces random noise into the data, ensuring that even correct data might be affected during transmission.

3. **Receiver's End** - In this part, the parity of the received data is reevaluated. If the parity of 1 in the received data is even, it indicates that there are no errors. Conversely, if the parity of 1 is odd, it suggests that an error has occurred in the received data.

   This code simulation mimics real-world scenarios where if such an error is detected, the data can be re-transmitted until no error is detected.

2.1.1 Output

To evaluate the efficiency of the code, another *Python* script was employed to execute it 'n' times. The objective was to determine the number of steps required to successfully transmit the correct data across binary numbers ranging from 1 to 10000.

For instance, consider the binary representation of the decimal number 1021, which is **1111111101**. In this evaluation, the decimal number 1021 ($i = 1021$) was taken as input, transmitted with random noise, and observed. If it took, say, 3 steps for the program to send the correct data, then it was recorded as $k_{1021} = 3$. This process was repeated for each decimal number from 1 to 10000. Finally, the average number of steps required for transmission was calculated using the formula given below:

$$\frac{\sum_{i=1}^{n} k_i}{n} \tag{1}$$

After performing the average calculation across 10,000 numbers for 10 iterations, the results were as follows: **0.9837, 0.9952, 0.9982, 1.0267, 1.0072, 0.9978, 0.9804, 1.0118, 1.0030, 1.0047**.

2.1.2 Observations

The average value around 1 indicates that, on average, only one single-parity error occurs during transmission among the 10,000 numbers. Hence, single-parity errors have a very low probability of occurrence, and correcting them would not require a significant amount of time.

2.2 Hamming Codes

Hamming codes offer both detection and correction of errors in transmitted data, eliminating the need for multiple transmissions. They employ multiple parity bits, positioned at powers of **2**, instead of a single parity bit. The calculation of the number of redundant parity bits r is determined by the inequality:

$$2^r \geq n + r + 1 \tag{2}$$

where n represents the number of bits in the transmitted data. The choice of r is typically selected as the minimum value that satisfies the relation for optimal space utilization, although any value that meets the condition can be chosen.

In a Hamming code representation $b_{n+k}b_{n+k-1}...b_3b_2b_1$ with parity bits $p_k, p_{k-1}, ..., p_1$, the parity bits are placed only at positions corresponding to powers of **2**. The remaining bit positions are allocated for the n bits of the binary code. It's important to maintain consistency in using either even or odd parity when constructing a Hamming code, as the same parity technique must be applied to detect errors in the received data.

The detailed procedure for finding parity bits and check bits is mentioned below:

1. **Parity Bits** - To determine the parity bits for error detection and correction, follow these steps:

   - Calculate the value of p_1 based on the number of ones present in bit positions b_3, b_5, b_7, and so on, where these bit positions correspond to binary representations with a **1** in the place value of 2^0.
   - Compute the value of p_2 based on the number of ones present in bit positions b_3, b_6, b_7, and so on, where these bit positions correspond to binary representations with a **1** in the place value of 2^1.
   - Determine the value of p_3 based on the number of ones present in bit positions b_5, b_6, b_7, and so on, where these bit positions correspond to binary representations with a **1** in the place value of 2^2.
   - Repeat this process for additional parity bits as needed.

2. **Check Bits** - To determine the check bits for error detection and correction, follow these steps:

   - Calculate the value of c_1 based on the number of ones present in bit positions b_1, b_3, b_5, b_7, and so on, where these bit positions correspond to binary representations with a **1** in the place value of 2^0.
   - Compute the value of c_2 based on the number of ones present in bit positions b_2, b_3, b_6, b_7, and so on, where these bit positions correspond to binary representations with a **1** in the place value of 2^1.
   - Determine the value of c_3 based on the number of ones present in bit positions b_4, b_5, b_6, b_7, and so on, where these bit positions correspond to binary representations with a **1** in the place value of 2^2.
   - Repeat this process for additional check bits as needed.

The code to run the Hamming Codes algorithm is as follows:

```python
#Aditya Mishra - 21013

import random
from typing import List

# Using Hamming codes to detect and correct errors

# Function for calculating the number of redundant bits in the given binary string
# using the formula 2^r >= m + r + 1
def RedundantBits(m:int):
    # Loop through possible values of r to find the smallest value satisfying the formula
    for r in range(m):
        if (2**r >= m + r + 1):
            return r  # Returns the number of redundant bits

# Finding a modified string with the number of bits = n + r + 1
def PositionRedundantBits(givenString , r):
    j = 0
    k = 1
    m = len(givenString)
    par =
    # Iterate through the positions to determine where to place the parity bits
    for i in range(1, m + r + 1):
        if (2**j == i):
            par = par +    0
            j += 1
        else:
            # Append the data backwards because the 0th position in binary is the last in the string
            par = par + givenString[-1 * k]
            k += 1
    # Reverse the modified string to get the correct order of parity bits
    return par[::-1]

# Calculating and adding parity bits to their specified positions
def CalculateParityBits(modifiedString , r):
    n = len(modifiedString)
    # Iterate through each parity bit position
    for i in range(r):
        binVal = 0
        # Iterate through each bit position
        for j in range(1, n + 1):
            if (j & (2**i) == ((2**i))):
                # XOR the bits in significant positions
                binVal = binVal ^ int(modifiedString[-1 * j])
        # Update the modified string with the correct parity bit at its specified position
        modifiedString = modifiedString[:n-(2**i)] + str(binVal) + modifiedString[n - (2**i) + 1:]
    return modifiedString

# Detecting errors in the received string
def DetectError(modifiedString , numberOfRedBit):
    n = len(modifiedString)
    res = 0
    # Iterate through each parity bit position
    for i in range(numberOfRedBit):
        val = 0
        # Iterate through each bit position
        for j in range(1, n + 1):
            if (j & (2**i) == (2**i)):
                # XOR the bits in significant positions
                val = val ^ int(modifiedString[-1 * j])
        res = res + val * (10**i)
    # Convert the binary result to decimal to get the position of the error bit
    return int(str(res),2)

# Function for adding single-bit errors
def AddRandomNoise(binaryNumber):
    # Select a random index and change its value randomly to introduce noise
    i = random.randint(0, len(binaryNumber) - 1)
    charList = list(binaryNumber)
    charList[i] = random.randint(0,1)
    binaryNumber =       .join(str(e) for e in charList)
    return binaryNumber

##########################################################
# Main Code
##########################################################

data =
while True: # Loop to check if there is a wrong binary input
    data = input("Enter binary number ")
    correctInput = True
    # Validate the input to ensure it contains only binary digits (0 or 1)
    for num in data:
        if (num ==    0     or num ==     1    ):
            # Continue if the current bit is valid
            continue
        else:
            # Prompt for input again if it is not binary
            correctInput = False
            break
    if (correctInput == False):
        print("Wrong Input! Try again")
    else:
        break

m = len(data)
r = RedundantBits(m)
# Get the modified string with parity bits
modifiedString = PositionRedundantBits(data, r)
modifiedString = CalculateParityBits(modifiedString , r)
print("Transmitted data is: " + modifiedString)

# Add random noise to the transmitted data
modifiedString = AddRandomNoise(modifiedString)
# Detect and correct errors in the received data
correct = DetectError(modifiedString , r)
if correct == 0:
    pass
else:
    listString = list(modifiedString)
    # Toggle the bit at the detected error position to correct the error
    if (listString[len(listString) - correct] == 0):
        listString[len(listString) - correct] =    1
    else:
        listString[len(listString) - correct] =    0
    modifiedString = "".join(listString)

print("Received data is: " + modifiedString)

#Aditya Mishra - 21013
```

Listing 2: Implementation of Hamming Codes in Python.

2.2.1 Output

Upon entering a binary number each time, the code generates and returns the same modified string. This modified string can be reverted back to the original binary number using the same method applied for modification. Running another script that converts numbers from 1 to 10,000 into binary and then processes them through this code demonstrates that there are no errors in the transmitted and received data.

2.2.2 Observations

While this method appears to be highly reliable for detecting and correcting single-bit errors, it's evident that for large binary strings, the number of redundant bits will increase significantly. Consequently, calculating these redundant bits will consume more time and space, which is not efficient. As demonstrated earlier with the simple parity check code, on average, data only needs to be transmitted twice (once initially and once again if an error occurs). Therefore, unless there is a requirement to transmit small data with low latency, this method should be avoided.

3 Applications

Single-parity error detection and correction codes find applications across various domains in computer science and networking, either directly or as the basis for multi-bit error correcting codes. Some of the fields where they are utilized include:

1. **Data Storage** - Error detection and correction codes play a crucial role in enhancing the reliability of data storage media. The concept of utilizing a parity track to detect single-bit errors dates back to the first magnetic tape data storage in 1951. Today, a variety of error detecting and correcting codes are employed to bolster the reliability of data storage systems.

   Single-bit errors often stem from natural phenomena such as thermal noise, power noise, cross talk, attenuation, and other forms of electromagnetic interference. An intriguing example of a single-bit error occurred during a Super Mario 64 speedrun race in 2013. DOTA Teabag encountered a glitch that seemed impossible within the game: an up-warp without any ceiling to grab onto. This glitch attracted the attention of prominent Super Mario 64 player pannenkoek12, who offered a 1000 USD bounty for replicating it. However, despite numerous attempts, no one could reproduce the glitch.

   Subsequently, it was discovered that during the race, an ionizing particle from outer space collided with DOTA Teabag's Nintendo 64 console, causing a single-bit error in Mario's height data. Specifically, the particle flipped the eighth bit of Mario's first height byte, resulting in a change from "C5" to "C4". Remarkably, this alteration in height happened to precisely align with the conditions needed to warp Mario to the higher floor at that exact moment. This suspicion of a bit flip was later confirmed by pannenkoek12 himself using a script that manually flipped the bit at the right time. Interested readers may refer to the link for more information about the above example.

2. **Satellite Broadcasting** - Similar to deep-space telecommunication, error detecting and correcting codes can be applied in this context to mitigate noise and errors that occur over long distances.

3. **Deep-Space Telecommunications** - Error detection and correction codes play a crucial role in deep-space telecommunications, as demonstrated by their utilization in missions such as Voyager 1 and Voyager 2. In the vast expanse of interplanetary space, signals experience significant attenuation, and power resources are limited. Moreover, the transmission over long distances introduces considerable noise. Error detecting and correcting codes serve to address these challenges by effectively identifying and rectifying errors in the transmitted data.

4. **Internet** - In a standard TCP/IP stack, various error detecting and correcting codes are implemented across different layers. These codes serve to ensure data integrity from the Ethernet frame and IPv4 header to protocols like UDP and TCP, where some form of error detection or correction code, including single-parity check codes, is employed.

4 Conclusion

While single-parity check codes may not be as prevalent in communication and networking, they remain significant as the foundation for more complex error detection and correction techniques handling multiple-bit errors. Despite the availability of advanced methods for detecting and correcting errors, single-parity check codes are still utilized in devices to efficiently manage and correct small errors without the need for more sophisticated approaches. The references for the report have been taken from [1, 2, 3] and the codes are available on the Github repository.

References

[1] `https://www.geeksforgeeks.org/hamming-code-implementation-in-python/`.

[2] `https://www.javatpoint.com/computer-network-error-detection`.

[3] `https://en.wikipedia.org/wiki/Error_detection_and_correction`.