CS3302 Practical 2 - Error Correction

150000845

November 2016

1 Introduction

The aim of this practical was to implement Hamming encoding and decoding to see how it performs under a model of burst errors. The Hamming codes should perform with a given value of redundant bits r, such that :

- The Hamming codes have word length $2^r - 1$.
- The Hamming codes have dimension (i.e non-redundant bits) $2^r - r - 1$.

The different probabilities for which the burst error model should operate under should also be specified at the start. The model should also implement interleaving to try and distribute the errors and try to prevent all of the bit flips from occurring in the same word.

2 Design

I wanted my model to imitate communication between two separate entities as much as possible. Originally, I designed and even implemented a model which was similar to the first practical, in that I read in files in stream form, converted each byte into corresponding bits and then performed the encoding and decoding on those bits and stored them. However, I realised that using raw bits meant that presentation of my model would be difficult - I was still able to count errors and it was even possible to see where undetected errors or incorrectly changed bits occurred in the file afterwards when in text files for example a character changed, but this was still not ideal for final presentation, as it did not use a stream of bits which had equal probability of producing either a 1 or a 0, and with large files it made comparisons at the end difficult.

Therefore, I changed my model. It instead uses a Hamming manager, which takes encoded and interleaved values from the Hamming encoder, passes it through the burst error model, and then passes the bits with the errors in to the Hamming decoder.

The program takes in the following as an input from the user :

- r - the number of redundant bits i.e parity bits to use in the codeword. In the code this is sometimes referenced as *val*.
- $P(Error)$ - the probability that while the burst error model is in a bad state that a bit will be flipped.
- $P(GB)$ - the probability that whilst in a good state that the burst error model will flip to a bad state.
- $P(BG)$ - the probability that whilst in a bad state that the burst error model will flip to a good state.
- Interleave Height - the height of the interleaving table that will be used.

The arguments are passed to the Hamming manager, which creates a Hamming encoder object and decoder using the value of r and the interleaving table height, and creates the burst error model using the various probabilities. On creation, the encoders and decoders construct only the matrices that they need to simulate two separate entities in real life - for example, the encoder only generates the generator matrix without any need for the parity check matrix. The manager then retrieves a string of bits from the encoder, which was produced by the following process :

- Get a string the length of the dimension of bits, which have been generated with equal probability from the channel. Add this word into the array of words.
- Convert this word into a Hamming codeword using the generator matrix. Add this codeword into the array of codewords.
- If the number of codewords we have created is not yet equal to the height of the interleaving table then repeat the above process until this case has been fulfilled. Otherwise,
  - Fill the values in the interleaving table row by row with the values of the codewords.
  - Read all of the bits out into one string by reading the table column by column and return it to the encoder.
- Add the string to the encoding results object and return the encoding results object to the Hamming process manager.

The Hamming manager receives the string of bits and passes it through the burst error model with this process :

- Create an empty string that will be the result string.
- Take the next character in the string of bits.

- If the burst error model outputs true on its *flip()* method, which outputs whether or not to flip a bit depending on the current states of the error model and comparing random numbers within the probabilities passed to the the model to potentially change states or create errors, then add the flipped bit to the result string. Otherwise, just add the bit to the result string.

The result string created by the burst model is then passed by the interleaving manager over to the Hamming decoder, which carries out this process :

- Fill the interleaving table column by column with bits from the string. Return the string that is formed by reading out from the table row by row.
- Take the first *word length* sized chunk of bits from the front of the result string and store it into the array which stores all of the words which were received after the interleaving process.
- Perform the error checking/correcting process on the word we just extracted. This is done using the parity check matrix, which produces a syndrome table corresponding to which bit in an erroneous string of bits is erroneous. Add this corrected string into the array which holds the error corrected strings.
- Take this error corrected code word and decode it into its original word using the decoder matrix. Add this decoded word into the matrix of code words and also into the string of corrected data.
- Once all of the data has been corrected and decoded, add all of the data we collected to the results object and return it to the Hamming manager.

After these processes, the Hamming manager is able to determine how many bits the decoder correctly decoded by comparing the string that the encoder generated by adding all of the channel outputs together with the resulting string that the the decoder generated by putting together all of the error corrected and decoded results. From this, we can see the success rate of the decoder.

3 Testing

For testing, I tested the output of the matrix creation processes for when the r value = 3. I knew that I was generating the matrices correctly when I knew that they corresponded to the following :

- Generator Matrix

$$\begin{pmatrix} 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

- Parity Check Matrix

$$\begin{pmatrix} 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 1 & 1 & 1 \end{pmatrix}$$

- Decoding Matrix

$$\begin{pmatrix} 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

Another method of testing that I used was to carry out the entire process but without any burst errors in the middle (I did this whilst I had the file handling design, as it was easy to check if a two text files matched each other). This too was the case.

4 Implementation

I have included an ant build file for compilation and building processes. To build the project, change the current working directory into the project directory, which is the folder that contains my build.xml file, not the source code directory itself, and carry out the command :

```
$ ant clean-build
```

To then run the program, execute the jar in the same project directory that you are currently in with the following arguments :

- The flag "-o" so that you get the full output - if you do not include this flag the program will not run. This flag differentiates it from the "-t" flag which I used to build the results file (which took well over an hour, so I have included it so that you do not have to build it yourself.)

- The value of r.
- The probability of flipping a bit whilst the burst error model is in a bad state.
- The probability of transitioning to a bad state whilst in a good state.
- The probability of transitioning to a good state whilst in a bad state.
- The height of the interleaving table.

An example of running the program would be :

```
$ java -jar HammingCoding.jar -o 3 0.4 0.4 0.4 3
```

The output will demonstrate different words coming in from the channel and their conversion to Hamming code words followed by the words generated from the interleaving table during the encoding stage, and then this process in reverse during the decoding stage so that they can be compared. Three iterations are done automatically every time so that if small values of r and small interleaving table values are used then we still see a variety of different data being outputted. We also see a conclusion statement with the different statistics afterwards too.

5 Results and Analysis

I have a large set of results in results.txt - this is simply a collection of data where each possibly combination of values is tested in the ranges :

- r for everything in 3, 4, ..., 8
- $P(Error)$ for everything in 0.1, 0.2, ..., 0.9
- $P(GB)$ for everything in 0.1, 0.2, ..., 0.9
- $P(BG)$ for everything in 0.1, 0.2, ..., 0.9
- Interleave height for everything in 1, 2 ..., 10

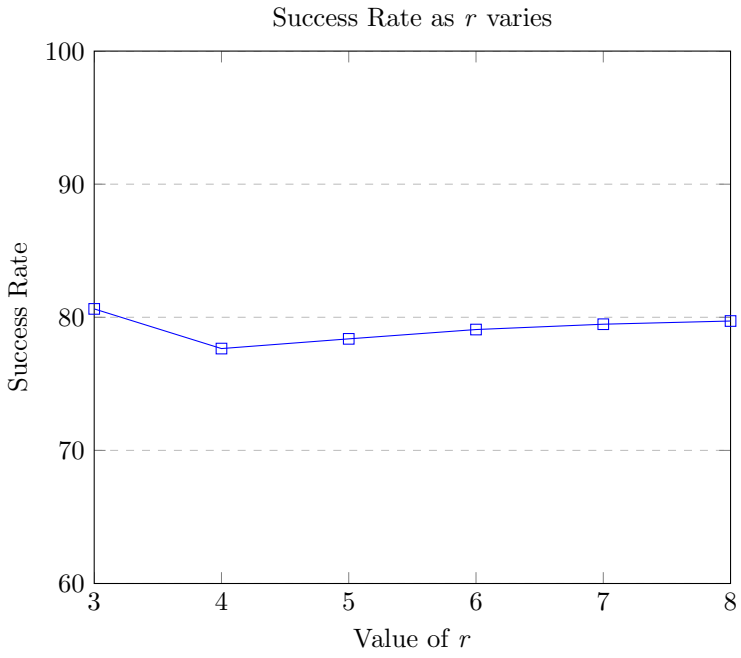
and the success rate (i.e the number of bits transferred and corrected successfully) is determined. I had to stop it before it was able to finish as it was taking too long, but there is still a large amount of data which can be compared.

5.1 Graphs

The data for the graphs below was gathered by using my program and varying 1 value at a time but fixing all other values. Each iteration is done 500 times and an average taken to produce the data point for the graph. For all of the graphs used below, if a value is not being changed then it is fixed at this value :

- $r = 3$
- $Interleave\ Height = 4$
- $P(Error) = 0.4$
- $P(GB) = 0.4$
- $P(BG) = 0.4$

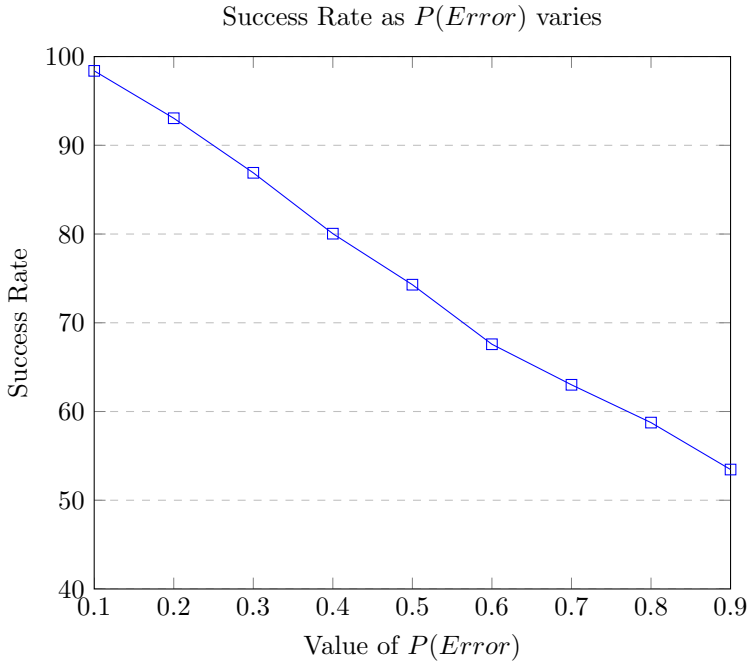
5.1.1 Varying r Value



r	3	4	5	6	7	8
Success Rate	80.63	77.65	78.38	79.08	79.48	79.72

We can see here that the value of r does not affect the results too much. The value of r affects the word length and the number of data bits transferred. However, the dimension does not increase linearly with r, meaning that the number of parity bits and the number of data bits do not increase linearly, which is obvious.

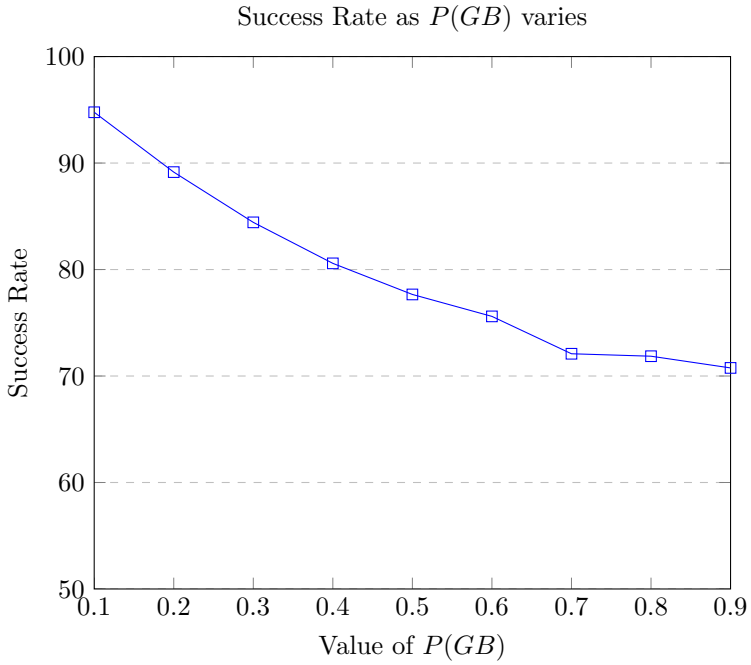
5.1.2 Varying $P(Error)$ Value



$P(Error)$	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
Success Rate	98.39	93.05	86.90	80.04	74.29	67.58	63.01	58.75	53.46

As we can see from the graph and the table data, as $P(Error)$ increases, the success rate decreases linearly. This makes sense, as this probability is what directly causes the errors - the rate at which you increase this probability is the same rate at which the number of errors occurring will increase.

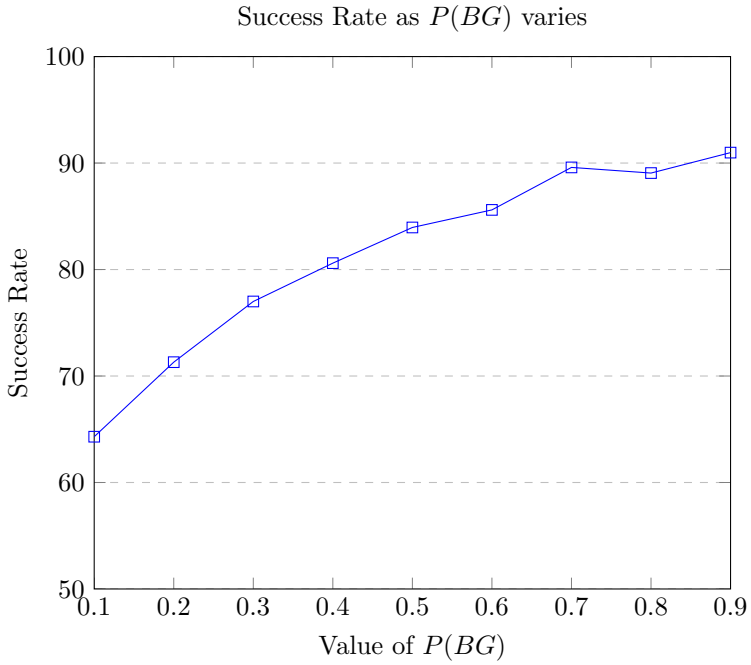
5.1.3 Varying $P(GB)$ Value



$P(GB)$	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
Success Rate	94.77	89.15	84.43	80.58	77.66	75.60	72.09	71.86	70.76

The graph trend shows that as $P(GB)$ increases, the amount of errors increases. The graph shape is similar to $-log(n)$, which is because the the probability of switching from good to bad does not directly increase the number of errors, but increases the chance that the errors will have a chance to materialise. The increase in errors behaves logarithmically because as we increase from 0.1 to 0.2 for example, we are doubling the chance for errors to occur, hence there is a large decrease in success, but this increase slows down as we increase $P(GB)$ more and more because that probability does not directly affect the number of errors.

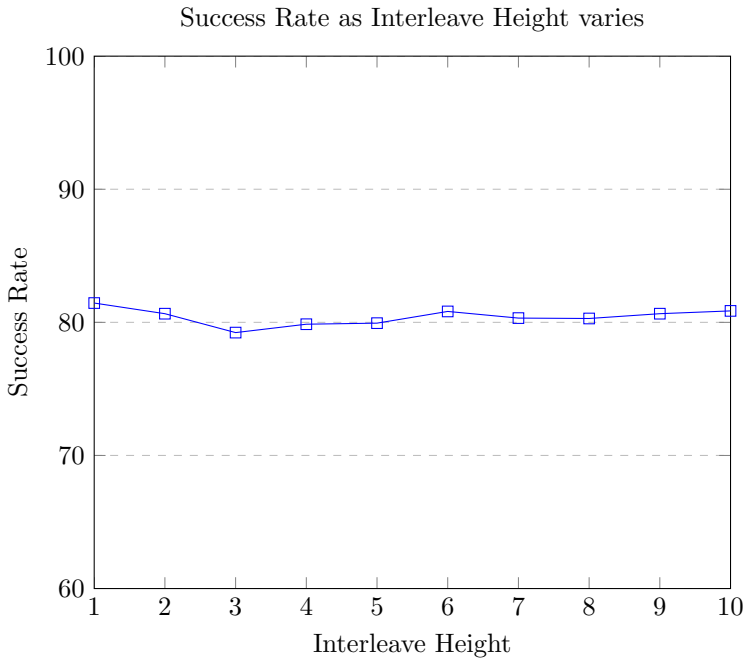
5.1.4 Varying $P(BG)$ Value



$P(BG)$	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9
Success Rate	64.30	71.30	77.00	80.60	83.95	85.60	89.59	89.06	90.98

Similarly to the behaviour of varying $P(GB)$, varying $P(BG)$ behaves logarithmically - as the probability increases, the success rate does too, since we are reducing the chance every time for an error to occur. This graph is logarithmic, since the affect at the start of changing from a small probability to a large one is massive - we go from being almost totally unable to move out of a state where we are subjected to errors, to doubling the chance of being able to move away from causing errors. As P(GB) gets large later however and is already large enough where it can alternate on a pretty regular basis back to a state where it is not subjected to errors, it makes very little difference to increase the probability of it being able to change back into a good state from the bad state.

5.1.5 Varying Interleave Height



Interleave Height	1	2	3	4	5	6	7	8	9	10
Success Rate	81.45	80.65	79.23	79.86	79.94	80.82	80.32	80.29	80.65	80.86

These are the results from the using the original fixed values. I thought that these results did not show us much, as it does make sense that when error probabilities are low that changing the value of the interleaving height makes little difference as there are very few errors to distribute among the different code words, especially when r is fixed at a low value too, since if r is low then it means that the code words are short and so there are fewer bits in the words that can be affected.

However, I did more and more tests, where I changed the r value and the different probabilities, and could not find anything much different from the results above.

6 Conclusion

I found this practical really useful in understanding and implementing Hamming codes, and also found it interesting to model different conditions that data channels can come under, along with ways to try and get around these problems. Its interesting to see just how important some factors are in affecting the suc-

cess rates of the data streams and also mathematically the way how different behaviours trend.