Overview

To develop an application capable of taking a string of bits, encoding it using Hamming code, introducing random errors into the encoded string, and then decoding it, correcting as many errors as possible.

Build Instructions

Run in project root folder: *cd out/artifacts/Practical2_jar*

java -jar Practical2.jar

Design and Implementation

The GUI is very simple. Two input areas are given for the user to enter the probability of errors occurring, and the amount of times each R value should be tested to build average statistics.

Once the “Go!” button is pressed, an action event is triggered. This will start a for loop that will iterate through each R value to be tested, which has been specified as 2 – 6 inclusively.

For each R value, a parity matrix is firstly generated. Information is then extrapolated from the parity matrix in order to build the generator matrix. A syndrome table is also built. All three are too scale of the R value currently being investigated.

The parity and generator matrices are stored as *Matrix* objects, a class containing the methods needed for manipulating matrices.

Once all three are successfully built, another for loop is instigated. At the beginning of each iteration, a random string of bits will be generated, with an equal probability of each bit being either a 0 or a 1. The length of this string is always 32,604 bits. 32,604 is the lowest common multiple of 1, 4, 11, 26 and 57, which are the lengths of the word blocks to be encoded. Since the string will always be divisible for any R value, the need for padding is negated.

This string is then split into blocks of length $2^r – r – 1$. Each block is turned into a matrix object, and then multiplied by the generator matrix to create the encoded string. This encoded string matrix is then converted to a string and appended to the encoded string.

Once the entire original string has been encoded, the encoded string is passed into a method to introduce errors. A random double between 0 and 1 is generated, and if it is less than the probability given by the user, the currently inspected bit is flipped. Every bit in the encoded string is put through this process.

This new erroneous string is then passed into the decoder. This will split the erroneous string into blocks of length $2^r – 1$. Each block is then turned into a matrix object and multiplied by the parity matrix. The resulting syndrome code is then looked up within the syndrome table. If the code is all 0’s, the first $2^r – r – 1$ bits are stripped from the encoded block and appended to the decoded string. However, if the syndrome code is anything else, the corresponding syndrome string will be XORed with the erroneous block, attempting to correct the error.

Once every block has been decoded, a check is run to determine just how accurate the error detection was. During the decoding, a check is also being run to count just how many times an error is corrected. This is later compared to the total amount of errors introduced into the original encoded string.

This process is done as many times as the user desired for each R value. Average statistics for decoding accuracy and error correction rates are displayed, as well as the information rate for each R value.

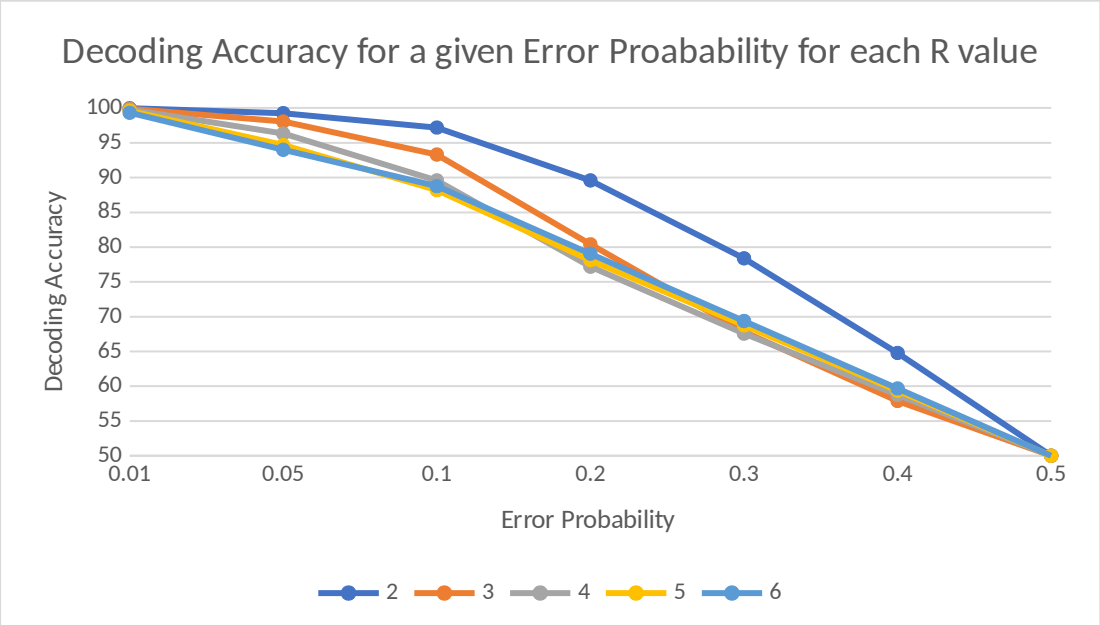
Statistics

Decoding Accuracy

Error Probability	R = 2	R = 3	R = 4	R = 5	R = 6
0.01	99.9699	99.9128	99.8019	99.6137	99.3305
0.05	99.272	98.0627	96.3324	94.6745	94.0023
0.1	97.2013	93.3198	89.6008	88.2006	88.7663
0.2	89.6017	80.4086	77.1991	78.1311	79.0679
0.3	78.399	67.8264	67.5701	68.7585	69.3865
0.4	64.7936	57.8917	58.738	59.3776	59.692
0.5	50.0103	50.0236	50.0114	50.0194	49.9936

Each R value had a random string encoded and decoded 500 times for each probability. The results were averaged after each probability run. You can clearly see that as the probability of an error occurring rises, the accuracy of the decoded message drops. Since a higher R value creates higher encoded string lengths, the amount of erroneous bits present will be greater than those of lower R values.

Surprisingly, an R value of 6 becomes more and more accurate as the probability of error increases (relative to the other R values – aside from 2). At a probability of 0.4, having an R value of 6 becomes your second most accurate value to encode with.



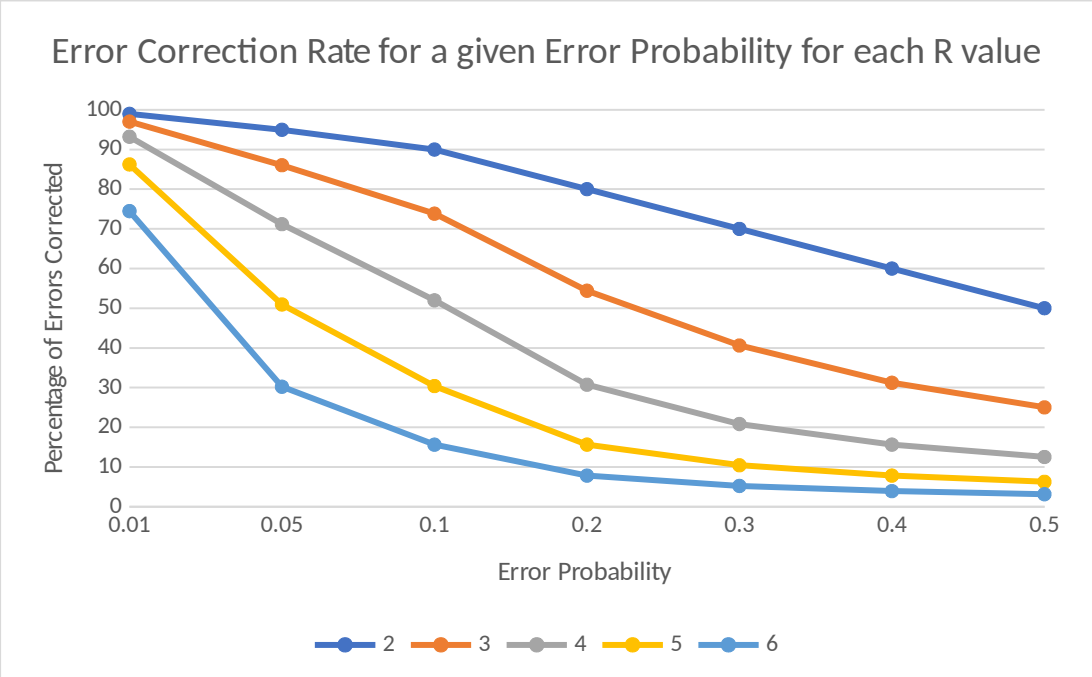
The graph above illustrates how diverse each R value is with low error probabilities, but as the possibility of error grows, the accuracy of all values of R deteriorates to become essentially equivalent.

Error Correction Rates

Error Probability	R = 2	R = 3	R = 4	R = 5	R = 6
0.01	98.993	97.0391	93.2167	86.2421	74.4752
0.05	94.9818	86.0426	71.1834	50.9467	30.2077
0.1	89.9999	73.829	51.9919	30.3824	15.6037
0.2	80.0154	54.4119	30.7226	15.6264	7.8113
0.3	69.9969	40.6053	20.8214	10.4119	5.2107
0.4	59.9983	31.2057	15.6185	7.8156	3.905
0.5	49.9991	25.005	12.5097	6.2518	3.1251

Much like before, each R value was tested 500 times on each error probability. You can clearly see the effect of a longer encoded string has on the chance of an error being corrected. Since an encoded block with an R value of 6 has 63 bits, even with a probability of 0.01, there's a substantial chance two or more bits within the block will be flipped. If that happens, only one of those bits will be corrected.

There is a clear trend stating that the higher your R value AND the higher your probability, the lower your error catch rate. This is simply because the higher an R value, the more parity bits you're adding, which each have their own chance of being flipped.

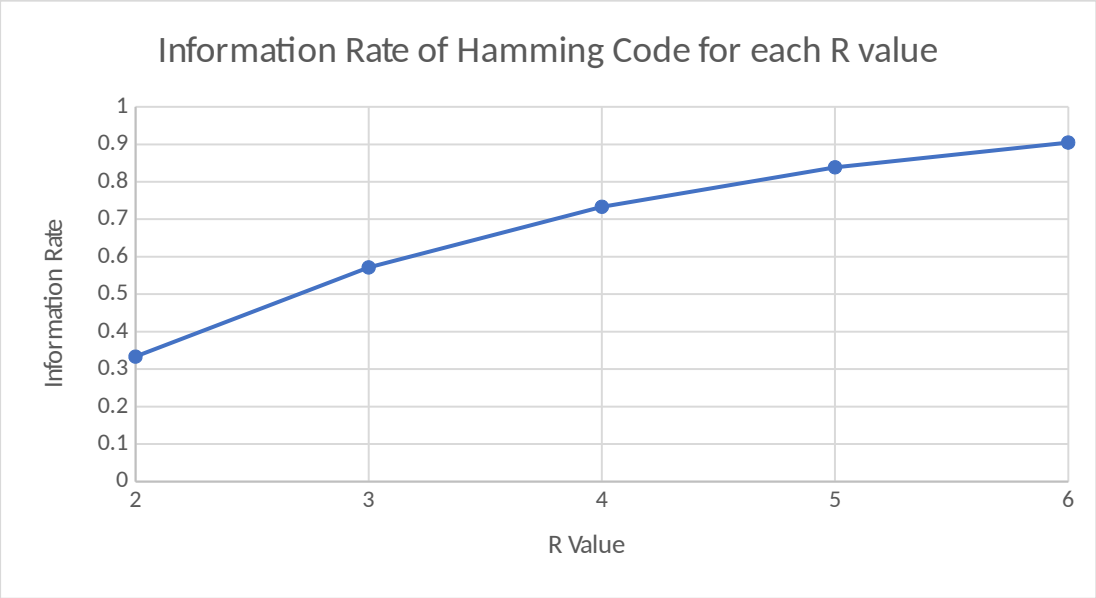


The graph above illustrates the rapid, almost logarithmic, decay for the R value of 6, whereas an R value of 2 has a much more linear decline.

Information Rates

R Value	Information Rate
2	0.3333
3	0.5714
4	0.7333
5	0.8387
6	0.9047

The information rate for each R value doesn't change as the length of the string to be encoded is constant, however the trend of the rate approaching 1 as R increases is still visible.



Conclusion

Having a small R value increases the accuracy of your decoding and error catch rate, but you heavily sacrifice on data compression.

I feel I now have a solid understanding of the Hamming code algorithms.