18.413 Final Project Report: Irregular Repeat Accumulate Codes FINAL DRAFT

Chun-Chieh Lin
May 14, 2004

This report describes the design and implementation of irregular repeat-accumulate codes given in the paper "Irregular Repeat-Accumulate Codes" written by Hui Jin, Aamod Khandekar, and Robert McEliece. This report also includes test results from my own implementation of the code and comparisons between my results and the results from the original paper.

TABLE OF CONTENTS

1. Introduction……………………………………………………………………4
2. Description of Irregular Repeat-Accumulate (IRA) Codes……………......4
    2.1 Overview………………………………………………………………..4
    2.2 Encoding Process……………………………………………………..6
    2.3 Communication Channel……………………………………………...6
    2.4 Decoding Process……………………………………………………..7
3. Description of My Implementation…………………………………...………9
    3.1 Overview……………………………………..…………………………9
    3.2 Repeat Generator……………………………………………….………9
    3.3 Permutation Generator………………………………………………10
    3.4 Encoder………………………………………………………………..10
    3.5 Channel Simulator……………………………………………………10
    3.6 Decoder………………………………………………………………..10
    3.7 Evaluator………………………………………………………………10
    3.8 Driver…………………………………………………………………...10
4. Reproducing Results from the Paper………………………………………11
5. Conclusion…………………………………………………………………..…13
APPENDIX: SPEADSHEETS AND EXTRA GRAPHS…………………..……14

FIGURES

Figure 1: Tanner graph of an IRA code copied from “Irregular Repeat-Accumulate Codes” by Jin, Khandekar, and McEliece……………………………..5
Figure 2: The repeating pattern used……………………………………………..11
Figure 3: My results compared to that of the paper’s (my results are +’s). The graph with the result from the paper is copied from “Irregular Repeat-Accumulate Codes” written by Hui Jin, Aamod Khandekar, and Robert McEliece………..….12

1. Introduction

Currently, error-correction codes can be divided into two main groups, the classical codes and iterative codes.

Classical codes focus on the maximum number of errors a code could correct or detect without making a single error in the decoded message. The classical codes are typically based on algebra, and are often time-consuming to decode when the block lengths are large.

Iterative codes typically focus on minimizing the bit error rate, which is the fraction of bit-errors in the decoded message. Their decoding algorithms are based on probability and are usually linear time algorithms. By using known relationships between bits of the codeword, each iteration of the decoding algorithm is able to refine probability estimates for each bit using the probability estimates from the previous iteration.

Since the discovery of Turbo codes, a class of simple iterative codes that performed very well, interest in iterative codes has increased, and several new classes of iterative codes have been devised that can out-perform Turbo codes. One of the new iterative coding schemes is irregular repeat-accumulate codes, which is the subject of this report.

2. Description of Irregular Repeat-Accumulate (IRA) Codes

2.1 Overview

The design of the IRA codes is best described using the Tanner graph representation. In figure 1 below, the Tanner graph of an IRA code is shown.

On the left of the graph are the message bits to be encoded, where each empty circle corresponds to a message bit. The empty circles on the right side of the graph correspond to the output of the encoder.

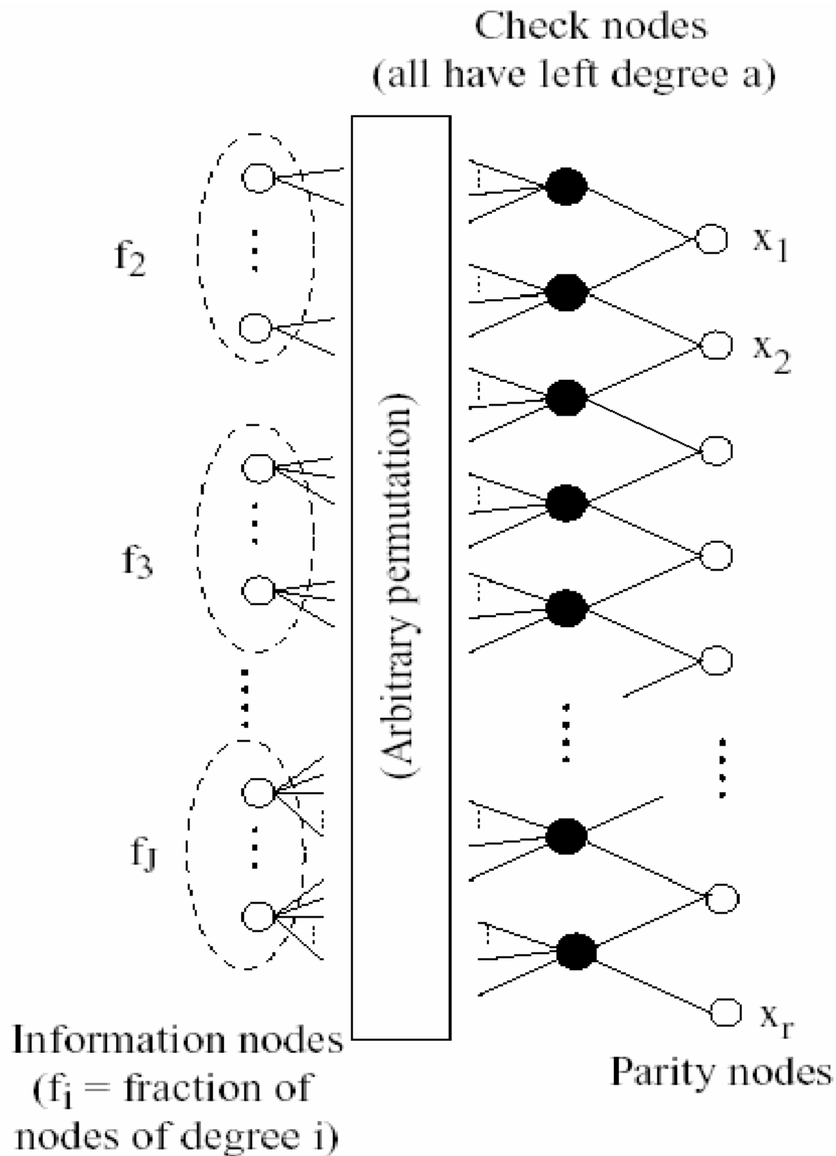

Figure 1: Tanner graph of an IRA code copied from “Irregular Repeat-Accumulate Codes” written by Hui Jin, Aamod Khandekar, and Robert McEliece

2.2 Encoding Process

The encoder works from the left side of the Tanner graph to the right. First of all, the message bits are repeated using a preset repeating pattern, and each bit may be repeated twice, three times, or more. The repeating patter is specified by a sequence of numbers lambda_2, lambda_3, lambda_4, and so on. Each lambda_i from the sequence gives the fraction of repeated bits that originated from bits repeated i times.

The repeated bits, which in the Tanner graph are represented by the edges leaving the message nodes, are then scrambled with a random permutation, which is represented by the rectangular block in the middle. The permutation, though randomly generated, is chosen beforehand and agreed on by the encoder and decoder, so the encoder does not generate a new permutation for each message. Note that the permutation block is simply a mapping from edges on the left to edges on the right. For each edge to the left of the block, there is a corresponding edge to the right that is considered the same edge and vice versa.

The permuted bits are then fed through an accumulator to generate the encoded bits, which correspond to the empty circles on the right of the Tanner graph. The encoded bits contain every a^{th} output of the accumulator, where a is a preset parameter chosen for the encoding scheme. The first bit of the output is then the parity (sum mod two) of the first a bits, the second bit of the output is the parity of the first 2a bits, and so on.

If the IRA code used is systematic, the message bits and the encoded bits together form the codeword. Otherwise, the codeword consists of only the encoded bits. The IRA code used in this report is systematic, so the message bits and the encoded bits are both part of the codeword.

2.3 Communication Channel

The codeword is passed through a communication channel, which introduces random errors into the codeword. Even though the errors are random, statistical properties of the channel are known, which enables the receiver to predict the original input to the channel.

For the testing done for this report, the Gaussian channel was used. For each bit in the codeword, the Gaussian channel outputs a Gaussian random variable of a certain known standard deviation. If the bit going in the channel is one, the output has mean one, and if the bit going in is zero, the output has mean negative one.

The receiver knows the standard deviation of the channel used. Therefore, for each bit received, the receiver is able to deduce a probability for the original bit to be one given the output from the channel for that bit.

2.4 Decoding Process

As explained above, for each of the code bits, the receiver provides the decoder with a probability for it to be one. The decoder's job is to refine the probability estimates using the relationship between the bits.

The way that the decoder accomplishes that is by passing probability messages back and forth on the edges of the Tanner graph. The message passing algorithm has several stages, which are listed below.

The "initial stage" is performed once after the decoder receives the codeword. The probabilities received for each bit, which all correspond to empty circles (also called bit nodes) on the Tanner graph, are passed on to each of the edges connected to it.

The initial stage is followed by a "check node stage," which happens around the filled circles, which are called check nodes, on the Tanner graph. A careful look

at the check nodes and a review of the encoding process reveals an important fact. Each check node is connected to at least one bit node, which is to the bottom right of the check node. The way that we encoded the bit corresponding to the bottom-right bit node was by taking the parity of all the other bits connected to the check node. Because of that, the bits connected to a check node must have total parity of zero, and each bit is equal to the sum mod two of the other bits.

That last fact is what we use to propagate the probability measures around the check nodes. The outgoing message on each of the edges connected to a check node is determined by all the other edges connected to that check node. After some computation, the formula for the outgoing message could be derived. Suppose the incoming probabilities to be taken into account are named in1, in2, …, ink, the value for the outgoing message is (1-(1-2*in1)*(1-2*in2)*…*(1-2*ink))/2.

After the check node stage, there is a “bit node stage,” which centers around the bit nodes. Each bit node corresponds to a single bit in the message bits or the encoded bits. Therefore, all the edges going in it are supposed to indicate the same value (one or zero), if the channel outputs has no errors.

Once again, the outgoing message on each edge is determined by all the probability messages on the edges coming into that node except the one that will carry the outgoing message. The only difference here is that the original channel outputs are added in as an extra input edge, which does not carry an output. After some probability computation, formulas to compute the outgoing messages here could be derived as well. If the incoming probabilities are once again in1, in2, …, ink, and the channel input is inc, the value of the outgoing message is (in1*in2*…*ink*inc) / (in1*in2*…*ink*inc+(1-in1)*(1-in2)*…*(1-ink)*(1-inc)).

After the bit node stage, the decoder performs another check node stage, and the cycle of bit and check node stages continue for a number of iterations. I used one hundred iterations for my code, which I thought provided a good balance between efficiency and performance.

When the values passing on the edges stabilize, the "terminal stage," which happens after a check node stage, determines the final decoded message. The output of the terminal stage is the final estimate of the probability for each message bit to be one. The calculation is identical to that of a bit node stage, except all the incoming messages to each node are used in the calculation along with the original channel output.

To decide whether each bit should be 0 or 1, the decoder uses 0.5 as a threshold. If the probability estimate is above 0.5, the decoded bit is 1, and if the probability estimate is below 0.5, the decoded bit is 0.

3. Description of My Implementation

3.1 Overview

The implementation of the simulation program for the IRA code is written in C++, and contains several modules. The modules are the repeat generator, the permutation generator, the encoder, the channel simulator, the decoder, the evaluator, and the driver code.

3.2 Repeat Generator

The repeat generator takes the specification of the repeating pattern, which is an array I type in by hand, and generates a more computation-friendly version. The output of this module is an array that specifies how many times each bit in the message should be repeated.

3.3 Permutation Generator

The permutation generator simply generates a permutation using a pseudo-random number generator. The output is in the form of an array that holds the permutation.

3.4 Encoder

The encoder randomly generates a message using a pseudo-random number generator. Then it calculates the codeword exactly as described in the Encoding Process section of this report, by repeating, permuting, accumulating, and adding the original message bits.

3.5 Channel Simulator

The channel simulator simulates the behavior of the Gaussian channel and the channel receiver. Its output is in the form of an array of floating point numbers. The floating point numbers denote the probability for each bit to be one after the receiver receives the channel output.

3.6 Decoder

The decoder implements the exact algorithm described in the Decoding Process section of this report. Each of the two rows of edges (one to the left of the check nodes, and one to the right) is represented by two arrays, and the two arrays are used to hold the messages passed on the edges. One of the arrays holds the messages going left, and the other holds the messages going right.

3.7 Evaluator

The evaluator checks the decoded message against the original message to give a bit error rate measure in the form of a floating point number.

3.8 Driver

The driver is implemented in two parts, the lower driver and the main function. The lower driver simulates using a given channel and a given number of samples.

The main function of the program does the preparation work like calling repeat generator and permutation generator. It also calls the lower driver with a variety of channels and sampling numbers.

4. Reproducing Result from the Paper

The "Irregular Repeat-Accumulate Codes" specified the lambda values that the authors used in the testing. However, because the block length N of the message is finite, I had to round the numbers. The fact that the output of the repeater has to be a multiple of "a" (which is 8 in this case) complicates matters even more. To clarify what I used for the repeating pattern, my repeating patterns are included in Figure 2.

		The number of message bits repeated i times		
lambda_i	i	N=1000	N=10000	N=100000
0.252744	3	667	6679	66788
0	6	1	0	1
0	10	0	1	0
0.081476	11	59	587	5872
0.327162	12	216	2161	21614
0.184589	46	32	318	3181
0.154029	48	25	254	2544

Figure 2: The repeating pattern used

I entered the repeating pattern shown in the Figure 2 above, and I produced several data using simulation. The results are graphed in figure 3, with my data graphed against the data from the paper.

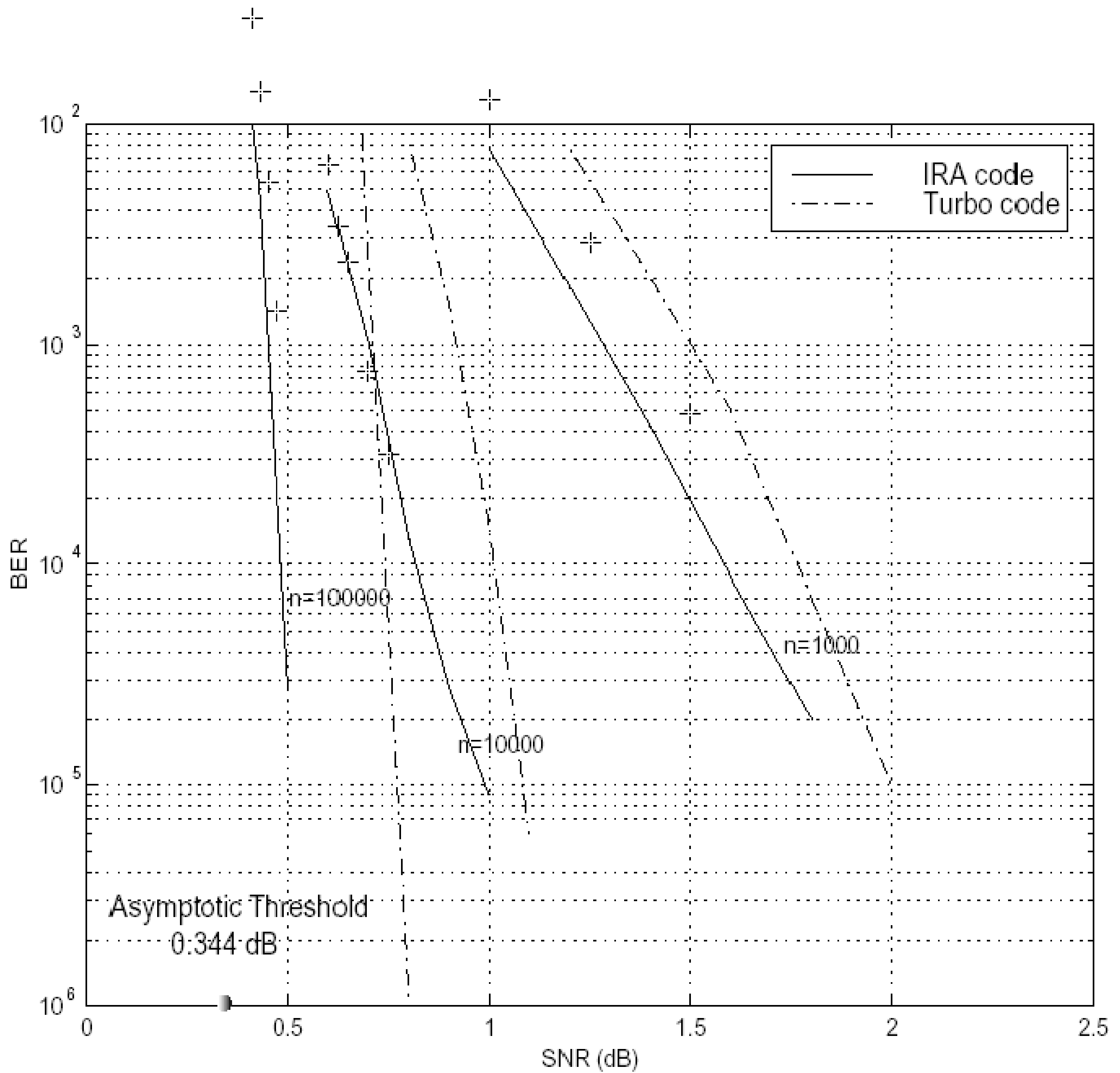

Figure 3: My results compared to that of the paper's (my results are +'s). The graph with the result from the paper is copied from "Irregular Repeat-Accumulate Codes" written by Hui Jin, Aamod Khandekar, and Robert McEliece

The graph shows that my data fits the result from the paper, especially for N=10000. However, my results for the N=1000 and N=100000 are consistently a little worse than that of the paper's.

For the N=100000 case, I believe the reason is that I did not cycle through the decoding stages enough times. My data shows that the bit error rate is still improving at a significant rate between the 90th round of the decoding cycle and the 100th round.

I think that the N=1000 simulations under performed either because I did not use the correct repeating pattern or because I did not test different permutations to find one that is good for the code. Since N=1000 is rather small, the result is more easily affected by little changes, such as rounding the numbers differently or using a bad permutation.

5. Conclusions

I was able to learn the encoding and decoding process of the Irregular Repeat Accumulate codes from "Irregular Repeat-Accumulate Codes" written by Hui Jin, Aamod Khandekar, and Robert McEliece. I wrote a program using the algorithm that was able to mostly reproduce the results given by the paper.

APPENDIX: SPREADSHEETS AND EXTRA GRAPHS