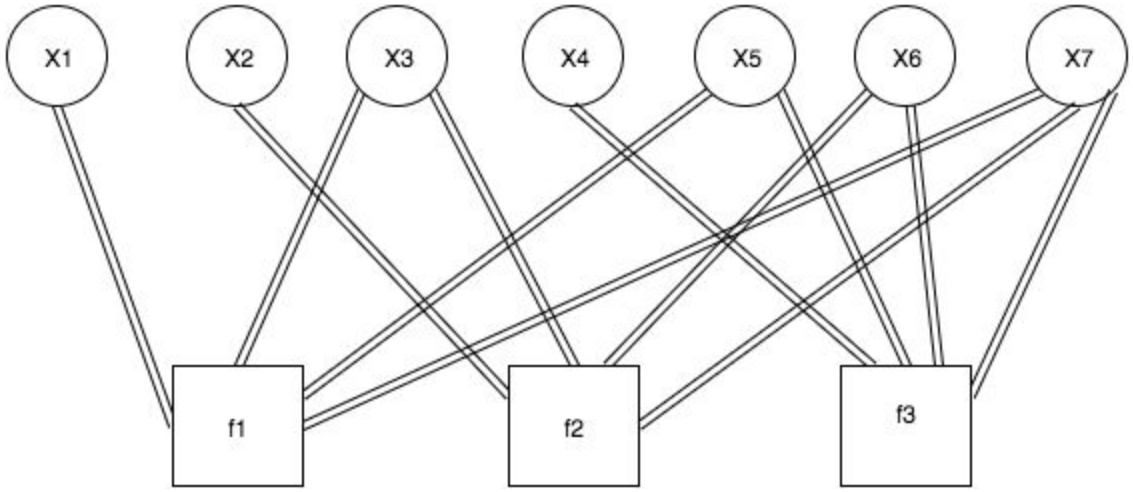
CSI5149 Project

Shaughn Finnerty, 6300433

In this project we use two algorithms, sum-product and max-product on a factor graphs with cycles to represent the specific (7,4) Hamming Code specified.

The resulting factor graph of this error correction code looks as follows:



The Global Function is $F(X_V) = 1$ when the codeword X_v is in the list of valid codewords and 0 otherwise. That is our global membership indicator function for the projects error code looks as follows:

$$F(X_V) = f_1(x_1,x_3,x_5,x_7) * f_2(x_2,x_3,x_6,x_7) * f_3(x_4,x_5,x_6,x_7)$$

With factors defined as follows:

$$f_1(x_1,x_3,x_5,x_7) = \delta(x_1 \oplus x_3 \oplus x_5 \oplus x_7)$$

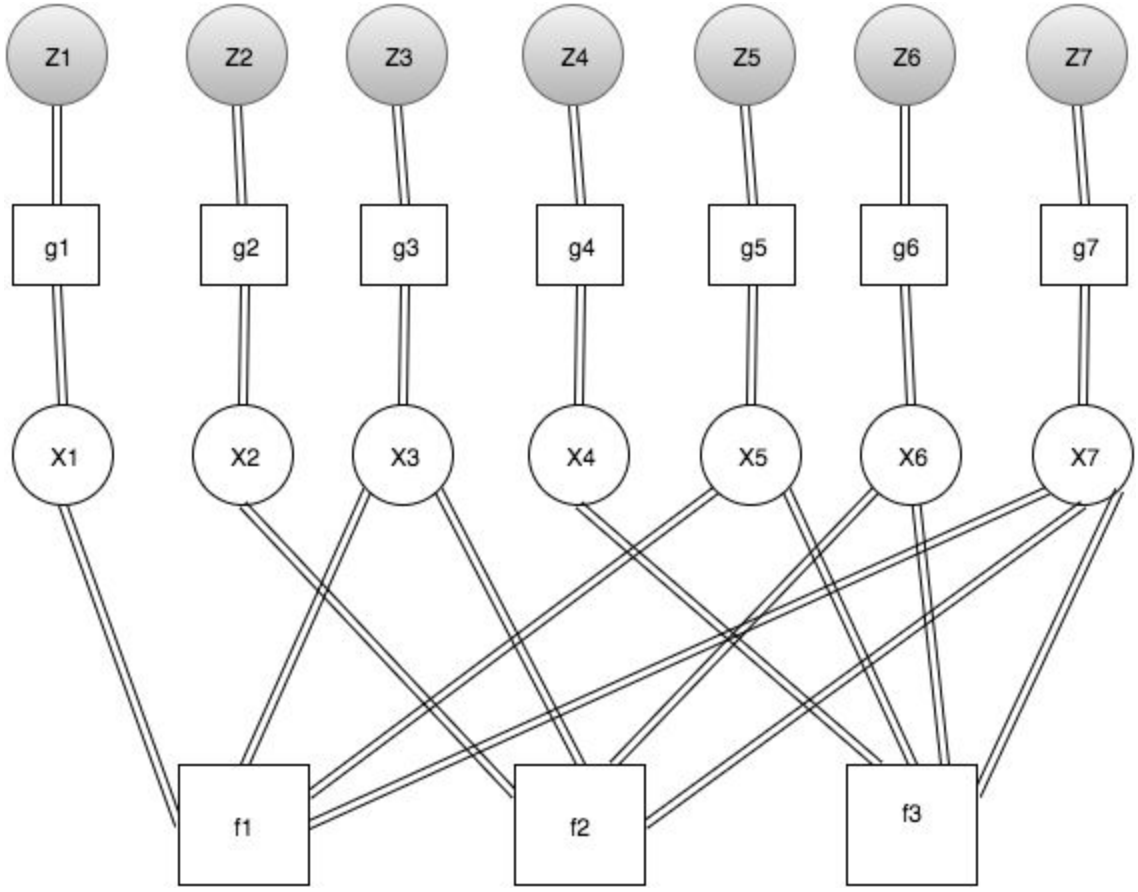
Where $\delta(x)$ is the Dirac Delta function (1 when x=0, 0 everywhere else for x). Similar definitions for factors $f_2(x_2,x_3,x_6,x_7)$ and $f_3(x_4,x_5,x_6,x_7)$ are assumed.

As a result, the following codewords are those which belong in this specific 16 word code satisfying the system of equations defined in this project:

```
0000000
```

```
0001111
0010110
0011001
0100101
0101010
0110011
0111100
1000011
1001100
1010101
1011010
1100110
1101001
1110000
1111111
```

Using the the received (observed) codewords $(z_1, z_2, z_3, z_4, z_5, z_6, z_7)$ at the decoder, the graphical model for this then becomes.



Important assumption/note: Since the factor nodes $g_1..g_7$ have only degree one after being evaluated using function `Decoder.compute_prior_prob(x, z)` which returns the value at z of the CDF of the normal distribution with the variance of the channel (we assume that channel noise is known to the decoder) and are purely a univariate function of x_a respectively. Essentially, we eliminate observed nodes by absorbing them (with their observed constant values) into the corresponding factors. **Since the factor node g_a becomes a leaf node (degree one) after being evaluated then their message $u_{g_a-->x_a}(x_a)$ will never change and so we include this message in the variable nodes implicitly and use it when computing messages from variable nodes throughout the iterations of the algorithm.**

Implementation Assumptions/Notes:

Some important assumptions about this implementation:

- Each undirected edge is represented by two directed edges. This was decided to allow for better identification/separation of **messages** and their directions. Together they form the undirected edge as required by factor graphs.
- Since we are operating on a factor graph with cycles, we initiate all messages from variable nodes to factor nodes with values 1 for every value in the message domain (e.g. 0 and 1 in this case).
- In this cyclic graphical model, we use a flooding iterative schedule to pass messages. The algorithm **terminates** when messages are passed that allow a final belief to be computed that produces a decoded codeword that exists in the 16 possibles codewords OR if a max number of iterations has occurred (e.g. we have this max set to 20 iterations).
- When computing bit error probability, we compare decoded codewords with the original codewords that are sent (assume the decoder knows of them). There is a boolean static Flag `USE_0_COMPUTATION` in `simulator.py` that can be changed to `True` to assume that the original codeword is (0,0,0,0,0,0,0) as suggested in the project details.

Running Instructions & Dependencies

To run this project, you will need the following

- **Python 2.7**

- **numpy** for math operations (i.e. computing codewords from parity check matrix)
  - `pip install numpy`
  - http://docs.scipy.org/doc/numpy-1.10.1/user/install.html
- **matplotlib** for plotting variance vs. bit error probability (in logarithmic domain)
  - `pip install matplotlib`
  - http://matplotlib.org/users/installing.html

As seen the pip package manager for python is the best tool to install these dependencies. This package manager can be installed by following the instructions here @ https://pip.pypa.io/en/stable/installing/ (downloading a python file and executing it)

Running Instructions

To run the project and generate a plot containing variance vs. bit error probability for both algorithms run the following command:

`python run.py <num_codewords>`

where `<num_codewords>` is the number of codewords to test at each variance level for each algorithm. In our results, we used 2000. With 2000 codewords, running both algorithms at 4 different variance levels each, this takes approximately 10 minutes on a MBP 2011 2.2 GHz Intel Core i7.

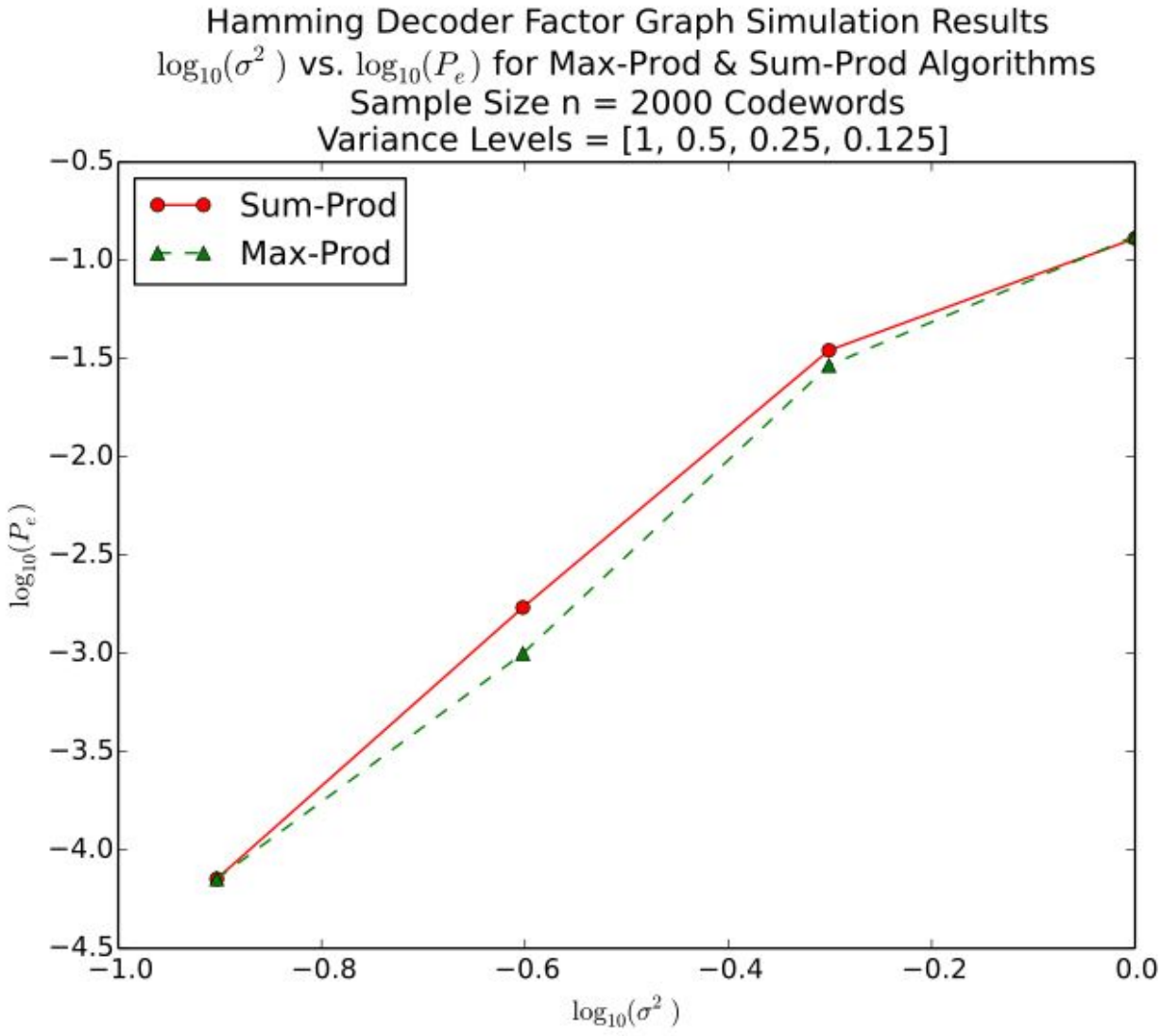
This will show a plot at the end and also save the file in the **graphs** directory as an SVG with the epoch timestamp in the filename.

Alternatively, if you would like to get an individual graph for each algorithm, some more specific simulations can be run with the `-advanced` flag. This will decode 2000 codewords at the variance levels required for the project, followed by decoding 1000 codewords at variance levels `[0.1, 0.2,...,0.9,1.0]` for each algorithm (That is, ~24000 codewords are processed so this will take ~½ hour). As before, the graphs will be printed into the graphs directory, and the variance vs. bit_error_probability will be saved in a csv file under the `stats` directory. Also, the `stats` directory will contain csv files for each variance level and each algorithm that hold the original codeword, its transmission values, and their resulting decoding.

`python run.py -advanced`

Results & Discussion

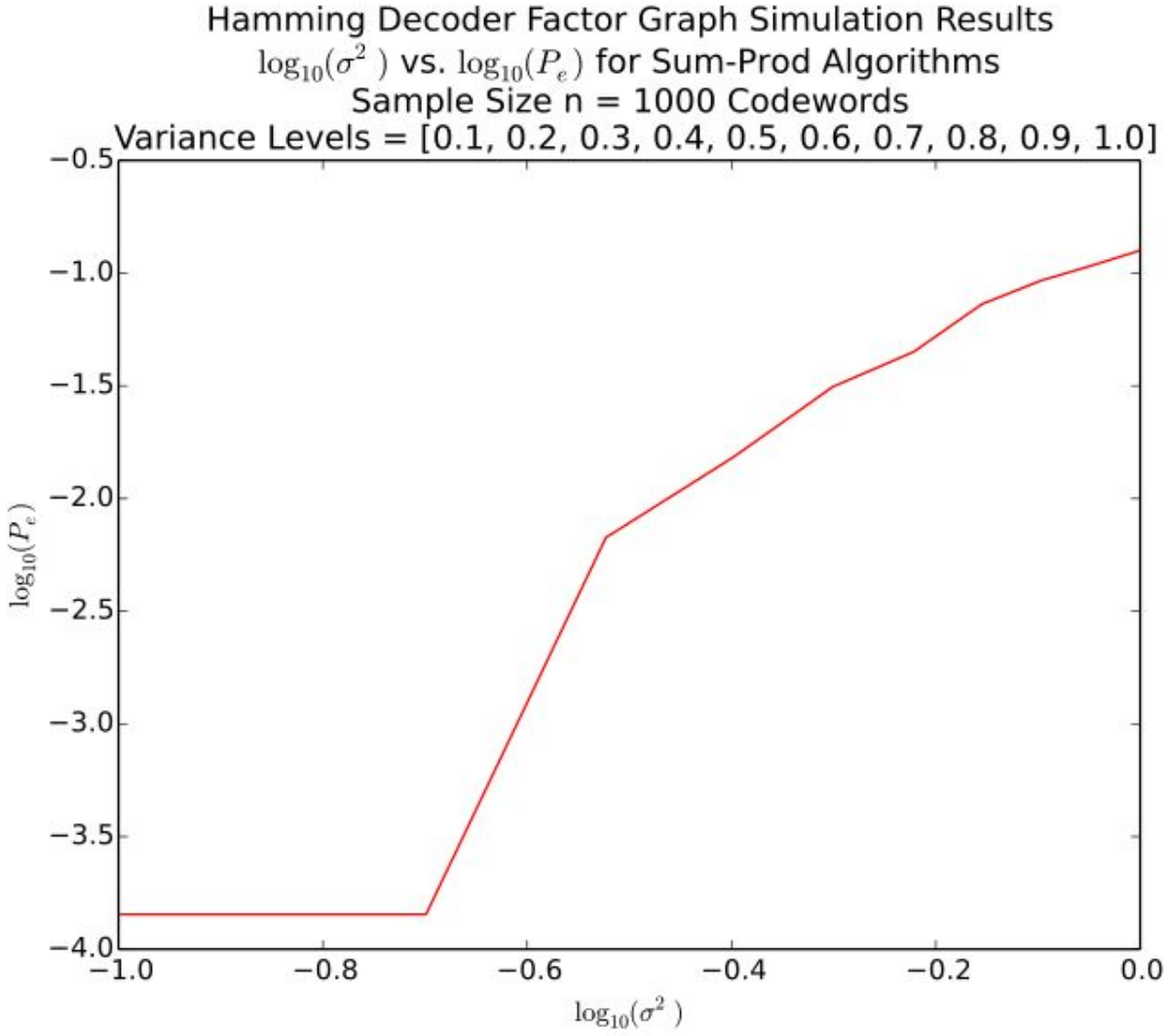
The result from running `python run.py 2000` generated the following plot on a logarithmic scale for σ^2 vs. P_e using 2000 codewords in each run of the algorithm at each variance level.

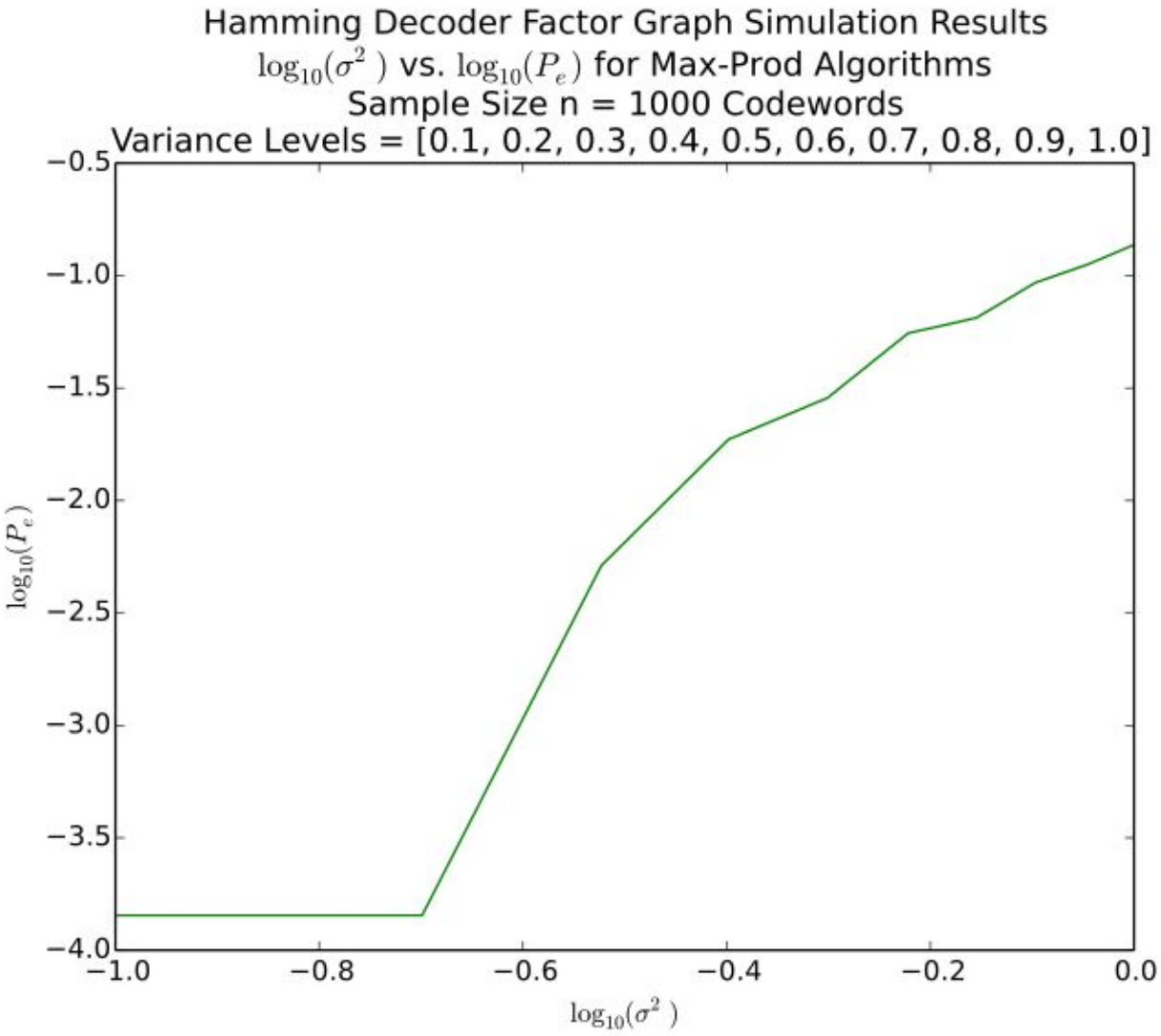


As you can see, both algorithms perform better when the variance in noise is smaller and the accuracy (ability to accurately decode a transmitted codeword) decreases when there is more noise added to the channel. The sum-prod seems to produce slightly better results at variance ¼ and ½ , although it is likely that the different

transmissions between algorithms may have accounted for this slight difference. Generally speaking though, both algorithms perform similarly at all evaluated levels of variance in channel noise.

More tests were run using 1000 codewords, but at more levels of variance. The results produced a more logarithmic graph for each algorithm. This again proved the idea that the decoder works very well on channels with little noise, and performance decreases as channel noise increases.





The flat lines indicate 0 bit errors at these variance levels. They are flat and the same because we always add 1 bit error when computing the error so that when converting the probability to logarithmic domain, we do not get a domain error (i.e. if the error rate is 0).

Conclusion

The use of Sum-prod and Max-prod algorithms to decode Hamming codes transmitted over channels with noise can be very effective. As one would guess, the less noise, the better these algorithms perform.

Using a graphical model like a factor graph to represent the Global function for membership in this code allows us perform inference on the original codewords based on the observations of the received codeword after transmission through a channel with noise. That is we are able to compute the posterior distribution of the hidden variables given the observed variables.