Analysis of Noisy Gradient-Descent Bit Flipping (NGDBF

Using MATLAB/Octave and the PRISM Model Checking Tool

Eric Reiss
Utah State University

Overview

- LDPC Codes and Trappins Sets

Overview

- LDPC Codes and Trappins Sets
- Algorithm Overview

Overview

- LDPC Codes and Trappins Sets
- Algorithm Overview
- Model Construction

Overview

- LDPC Codes and Trappins Sets
- Algorithm Overview
- Model Construction
- MATLAB/Octave Tool Overview

Overview

- LDPC Codes and Trappins Sets
- Algorithm Overview
- Model Construction
- MATLAB/Octave Tool Overview
- Sample Generation

Overview

- LDPC Codes and Trappins Sets
- Algorithm Overview
- Model Construction
- MATLAB/Octave Tool Overview
- Sample Generation
- Energy Calculation

Overview

- LDPC Codes and Trappins Sets
- Algorithm Overview
- Model Construction
- MATLAB/Octave Tool Overview
- Sample Generation
- Energy Calculation
- Transisition Probabilities

Overview

- LDPC Codes and Trappins Sets
- Algorithm Overview
- Model Construction
- MATLAB/Octave Tool Overview
- Sample Generation
- Energy Calculation
- Transisition Probabilities
- Write Files and Process Output

Overview

- LDPC Codes and Trappins Sets
- Algorithm Overview
- Model Construction
- MATLAB/Octave Tool Overview
- Sample Generation
- Energy Calculation
- Transisition Probabilities
- Write Files and Process Output
- Sources

LDPC Codes and Trapping Sets

- Low-Density Parity Check (LDPC) codes were introduced by Gallager in 1963

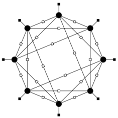

Figure 1: (8,8) Absorbing set that is dominant in the 802.3an 10GBASE-T LDPC Code [2].

LDPC Codes and Trapping Sets

- Low-Density Parity Check (LDPC) codes were introduced by Gallager in 1963
- LDPC codes are commonly represented as sparse Tanner graph

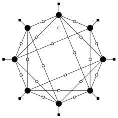

Figure 1: (8,8) Absorbing set that is dominant in the 802.3an 10GBASE-T LDPC Code [2].

LDPC Codes and Trapping Sets

- Low-Density Parity Check (LDPC) codes were introduced by Gallager in 1963
- LDPC codes are commonly represented as sparse Tanner graph
- Trapping sets are a sub-set of the graph that limit the performance of decoding algorithms, creating an error-floor

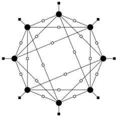

Figure 1: (8,8) Absorbing set that is dominant in the 802.3an 10GBASE-T LDPC Code [2].

LDPC Codes and Trapping Sets

- Low-Density Parity Check (LDPC) codes were introduced by Gallager in 1963
- LDPC codes are commonly represented as sparse Tanner graph
- Trapping sets are a sub-set of the graph that limit the performance of decoding algorithms, creating an error-floor
- Absorbing sets are a special case of a trapping sets that are stable in a bit flipping decoder [1]

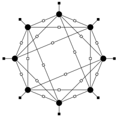

Figure 1: (8,8) Absorbing set that is dominant in the 802.3an 10GBASE-T LDPC Code [2].

Algorithm Overview

- NGDBF is part of a family of bit flipping decoding algorithms

Algorithm Overview

- NGDBF is part of a family of bit flipping decoding algorithms
- Improves upon the Gradient-Descent Bit Flipping (GDBF) Decoding Algorithm proposed by Wadayama et al. [3]

Algorithm Overview

- NGDBF is part of a family of bit flipping decoding algorithms
- Improves upon the Gradient-Descent Bit Flipping (GDBF) Decoding Algorithm proposed by Wadayama et al. [3]

  - GDBF gets “stuck” in local minima while decoding

Algorithm Overview

- NGDBF is part of a family of bit flipping decoding algorithms
- Improves upon the Gradient-Descent Bit Flipping (GDBF) Decoding Algorithm proposed by Wadayama et al. [3]

- GDBF gets “stuck” in local minima while decoding

- NGDBF introduces psuedo-random noise to escape local minima

Algorithm Overview

- NGDBF is part of a family of bit flipping decoding algorithms
- Improves upon the Gradient-Descent Bit Flipping (GDBF) Decoding Algorithm proposed by Wadayama et al. [3]

  - GDBF gets “stuck” in local minima while decoding

- NGDBF introduces psuedo-random noise to escape local minima
- Algorithm steps [4]:

Algorithm Overview

- NGDBF is part of a family of bit flipping decoding algorithms
- Improves upon the Gradient-Descent Bit Flipping (GDBF) Decoding Algorithm proposed by Wadayama et al. [3]

  - GDBF gets "stuck" in local minima while decoding

- NGDBF introduces psuedo-random noise to escape local minima
- Algorithm steps [4]:

  - Let H be an $n \times m$ parity check matrix, N_i be the adjacency for bit i, M_j be the adjacency for parity check j

Algorithm Overview

- NGDBF is part of a family of bit flipping decoding algorithms
- Improves upon the Gradient-Descent Bit Flipping (GDBF) Decoding Algorithm proposed by Wadayama et al. [3]

  - GDBF gets "stuck" in local minima while decoding

- NGDBF introduces psuedo-random noise to escape local minima
- Algorithm steps [4]:

  - Let H be an $n \times m$ parity check matrix, N_i be the adjacency for bit i, M_j be the adjacency for parity check j
  - Given channel samples, $\vec{y}$, initialize $\vec{x}$ to be the $sign(\vec{y})$

Algorithm Overview

- NGDBF is part of a family of bit flipping decoding algorithms
- Improves upon the Gradient-Descent Bit Flipping (GDBF) Decoding Algorithm proposed by Wadayama et al. [3]

  - GDBF gets "stuck" in local minima while decoding

- NGDBF introduces psuedo-random noise to escape local minima
- Algorithm steps [4]:

  - Let H be an $n \times m$ parity check matrix, N_i be the adjacency for bit i, M_j be the adjacency for parity check j
  - Given channel samples, $\vec{y}$, initialize $\vec{x}$ to be the $sign(\vec{y})$
  - Compute the syndrome, $s_j = \prod_{i \in M_j} x_i$

Algorithm Overview

- NGDBF is part of a family of bit flipping decoding algorithms
- Improves upon the Gradient-Descent Bit Flipping (GDBF) Decoding Algorithm proposed by Wadayama et al. [3]

  - GDBF gets “stuck” in local minima while decoding

- NGDBF introduces psuedo-random noise to escape local minima
- Algorithm steps [4]:

  - Let H be an $n \times m$ parity check matrix, N_i be the adjacency for bit i, M_j be the adjacency for parity check j
  - Given channel samples, $\vec{y}$, initialize $\vec{x}$ to be the $sign(\vec{y})$
  - Compute the syndrome, $s_j = \prod_{i \in M_j} x_i$
  - Calculate the energy for bit i, $E_i = y_i x_i + w \sum_{j \in N_i} s_j + z_i$, where w is i.i.d white noise and z_i is a zero mean noise pertubation

Algorithm Overview

- NGDBF is part of a family of bit flipping decoding algorithms
- Improves upon the Gradient-Descent Bit Flipping (GDBF) Decoding Algorithm proposed by Wadayama et al. [3]

  - GDBF gets "stuck" in local minima while decoding

- NGDBF introduces psuedo-random noise to escape local minima
- Algorithm steps [4]:

  - Let H be an $n \times m$ parity check matrix, N_i be the adjacency for bit i, M_j be the adjacency for parity check j
  - Given channel samples, $\vec{y}$, initialize $\vec{x}$ to be the $sign(\vec{y})$
  - Compute the syndrome, $s_j = \prod_{i \in M_j} x_i$
  - Calculate the energy for bit i, $E_i = y_i x_i + w \sum_{j \in N_i} s_j + z_i$, where w is i.i.d white noise and z_i is a zero mean noise pertubation
  - Given a threshold θ flip bit i if $E_i < \theta$

Model Construction

- The Markov Chain structure used in the tool was proposed in [1]

Model Construction

- The Markov Chain structure used in the tool was proposed in [1]
- For a given (a,b) trapping set, the state space is described by the a-bit binary representation of the numbers from 0 to $2^a - 1$

Model Construction

- The Markov Chain structure used in the tool was proposed in [1]
- For a given (a,b) trapping set, the state space is described by the a-bit binary representation of the numbers from 0 to $2^a - 1$

  - For example, the (3,3) trapping set would have 8 states, 000 - 111

Model Construction

- The Markov Chain structure used in the tool was proposed in [1]
- For a given (a,b) trapping set, the state space is described by the a-bit binary representation of the numbers from 0 to $2^a - 1$

  - For example, the (3,3) trapping set would have 8 states, 000 - 111
  - A transition from 010 to 000 implies that the middle bit flipped

Model Construction

- The Markov Chain structure used in the tool was proposed in [1]
- For a given (a,b) trapping set, the state space is described by the a-bit binary representation of the numbers from 0 to $2^a - 1$

  - For example, the (3,3) trapping set would have 8 states, 000 - 111
  - A transition from 010 to 000 implies that the middle bit flipped

- Edge probabilities are calculated by first calculating the probability that a bit flips using the cdf centered on the energy function with the noise pertubation subtracted

Model Construction

- The Markov Chain structure used in the tool was proposed in [1]
- For a given (a,b) trapping set, the state space is described by the a-bit binary representation of the numbers from 0 to $2^a - 1$

  - For example, the (3,3) trapping set would have 8 states, 000 - 111
  - A transition from 010 to 000 implies that the middle bit flipped

- Edge probabilities are calculated by first calculating the probability that a bit flips using the cdf centered on the energy function with the noise pertubation subtracted

  - $P_{flip} = normcdf(\theta, E_i, \sigma)$

Model Construction

- The Markov Chain structure used in the tool was proposed in [1]
- For a given (a,b) trapping set, the state space is described by the a-bit binary representation of the numbers from 0 to $2^a - 1$

  - For example, the (3,3) trapping set would have 8 states, 000 - 111
  - A transition from 010 to 000 implies that the middle bit flipped

- Edge probabilities are calculated by first calculating the probability that a bit flips using the cdf centered on the energy function with the noise pertubation subtracted

  - $P_{flip} = normcdf(\theta, E_i, \sigma)$

- The prbability of a transistion is then calculated by multiplying the flip probabilities

Model Construction

- The Markov Chain structure used in the tool was proposed in [1]
- For a given (a,b) trapping set, the state space is described by the a-bit binary representation of the numbers from 0 to $2^a - 1$

  - For example, the (3,3) trapping set would have 8 states, 000 - 111
  - A transition from 010 to 000 implies that the middle bit flipped

- Edge probabilities are calculated by first calculating the probability that a bit flips using the cdf centered on the energy function with the noise pertubation subtracted

  - $P_{flip} = normcdf(\theta, E_i, \sigma)$

- The prbability of a transistion is then calculated by multiplying the flip probabilities

MATLAB/Octave Tool Overview

- Located in the USU_stochastic_case_studies repository in the ngdbf_models folders, contains the following files

MATLAB/Octave Tool Overiview

- Located in the USU_stochastic_case_studies repository in the ngdbf_models folders, contains the following files
    - run_ngdbf.m: Driver function that contains most of the relevant code

MATLAB/Octave Tool Overview

- Located in the USU_stochastic_case_studies repository in the ngdbf_models folders, contains the following files
  - run_ngdbf.m: Driver function that contains most of the relevant code
  - load_trapping_sets.m: Script containing some of the trapping sets in University of Arizona's Trapping Set Ontology [5].

MATLAB/Octave Tool Overview

- Located in the USU_stochastic_case_studies repository in the ngdbf_models folders, contains the following files
  - run_ngdbf.m: Driver function that contains most of the relevant code
  - load_trapping_sets.m: Script containing some of the trapping sets in University of Arizona's Trapping Set Ontology [5].
  - write_model.m: Helper function that takes data from run_ngdbf to write a PRISM model

MATLAB/Octave Tool Overview

- Located in the USU_stochastic_case_studies repository in the ngdbf_models folders, contains the following files
  - run_ngdbf.m: Driver function that contains most of the relevant code
  - load_trapping_sets.m: Script containing some of the trapping sets in University of Arizona's Trapping Set Ontology [5].
  - write_model.m: Helper function that takes data from run_ngdbf to write a PRISM model
  - write_explicit_model.m: Helper function that creates the necessary files to generate an explicit PRISM model

MATLAB/Octave Tool Overview

- Located in the USU_stochastic_case_studies repository in the ngdbf_models folders, contains the following files
  - run_ngdbf.m: Driver function that contains most of the relevant code
  - load_trapping_sets.m: Script containing some of the trapping sets in University of Arizona's Trapping Set Ontology [5].
  - write_model.m: Helper function that takes data from run_ngdbf to write a PRISM model
  - write_explicit_model.m: Helper function that creates the necessary files to generate an explicit PRISM model
  - isOctave.m: Helper function to determine if the script is running on Octave or MATLAB

MATLAB/Octave Tool Overview

- Located in the USU_stochastic_case_studies repository in the ngdbf_models folders, contains the following files
  - run_ngdbf.m: Driver function that contains most of the relevant code
  - load_trapping_sets.m: Script containing some of the trapping sets in University of Arizona's Trapping Set Ontology [5].
  - write_model.m: Helper function that takes data from run_ngdbf to write a PRISM model
  - write_explicit_model.m: Helper function that creates the necessary files to generate an explicit PRISM model
  - isOctave.m: Helper function to determine if the script is running on Octave or MATLAB
  - get_error_sample_size.m: Helper function to calculate the number of error samples needed

MATLAB/Octave Tool Overview

- Located in the USU_stochastic_case_studies repository in the ngdbf_models folders, contains the following files
  - run_ngdbf.m: Driver function that contains most of the relevant code
  - load_trapping_sets.m: Script containing some of the trapping sets in University of Arizona's Trapping Set Ontology [5].
  - write_model.m: Helper function that takes data from run_ngdbf to write a PRISM model
  - write_explicit_model.m: Helper function that creates the necessary files to generate an explicit PRISM model
  - isOctave.m: Helper function to determine if the script is running on Octave or MATLAB
  - get_error_sample_size.m: Helper function to calculate the number of error samples needed
- The tool uses channel information to programmatically generate a model of the algorithm for each possible initial state

MATLAB/Octave Tool Overview

- Located in the USU_stochastic_case_studies repository in the ngdbf_models folders, contains the following files
  - run_ngdbf.m: Driver function that contains most of the relevant code
  - load_trapping_sets.m: Script containing some of the trapping sets in University of Arizona's Trapping Set Ontology [5].
  - write_model.m: Helper function that takes data from run_ngdbf to write a PRISM model
  - write_explicit_model.m: Helper function that creates the necessary files to generate an explicit PRISM model
  - isOctave.m: Helper function to determine if the script is running on Octave or MATLAB
  - get_error_sample_size.m: Helper function to calculate the number of error samples needed
- The tool uses channel information to programmatically generate a model of the algorithm for each possible initial state
- Three main sections of the code are

MATLAB/Octave Tool Overview

- Located in the USU_stochastic_case_studies repository in the ngdbf_models folders, contains the following files
  - run_ngdbf.m: Driver function that contains most of the relevant code
  - load_trapping_sets.m: Script containing some of the trapping sets in University of Arizona's Trapping Set Ontology [5].
  - write_model.m: Helper function that takes data from run_ngdbf to write a PRISM model
  - write_explicit_model.m: Helper function that creates the necessary files to generate an explicit PRISM model
  - isOctave.m: Helper function to determine if the script is running on Octave or MATLAB
  - get_error_sample_size.m: Helper function to calculate the number of error samples needed
- The tool uses channel information to programmatically generate a model of the algorithm for each possible initial state
- Three main sections of the code are

MATLAB/Octave Tool Overview

- Located in the USU_stochastic_case_studies repository in the ngdbf_models folders, contains the following files
  - run_ngdbf.m: Driver function that contains most of the relevant code
  - load_trapping_sets.m: Script containing some of the trapping sets in University of Arizona's Trapping Set Ontology [5].
  - write_model.m: Helper function that takes data from run_ngdbf to write a PRISM model
  - write_explicit_model.m: Helper function that creates the necessary files to generate an explicit PRISM model
  - isOctave.m: Helper function to determine if the script is running on Octave or MATLAB
  - get_error_sample_size.m: Helper function to calculate the number of error samples needed
- The tool uses channel information to programmatically generate a model of the algorithm for each possible initial state
- Three main sections of the code are

MATLAB/Octave Tool Overview

- Located in the USU_stochastic_case_studies repository in the ngdbf_models folders, contains the following files

  - run_ngdbf.m: Driver function that contains most of the relevant code
  - load_trapping_sets.m: Script containing some of the trapping sets in University of Arizona's Trapping Set Ontology [5].
  - write_model.m: Helper function that takes data from run_ngdbf to write a PRISM model
  - write_explicit_model.m: Helper function that creates the necessary files to generate an explicit PRISM model
  - isOctave.m: Helper function to determine if the script is running on Octave or MATLAB
  - get_error_sample_size.m: Helper function to calculate the number of error samples needed

- The tool uses channel information to programmatically generate a model of the algorithm for each possible initial state
- Three main sections of the code are

MATLAB/Octave Tool Overview

- Located in the USU_stochastic_case_studies repository in the ngdbf_models folders, contains the following files
    - run_ngdbf.m: Driver function that contains most of the relevant code
    - load_trapping_sets.m: Script containing some of the trapping sets in University of Arizona's Trapping Set Ontology [5].
    - write_model.m: Helper function that takes data from run_ngdbf to write a PRISM model
    - write_explicit_model.m: Helper function that creates the necessary files to generate an explicit PRISM model
    - isOctave.m: Helper function to determine if the script is running on Octave or MATLAB
    - get_error_sample_size.m: Helper function to calculate the number of error samples needed
- The tool uses channel information to programmatically generate a model of the algorithm for each possible initial state
- Three main sections of the code are

Sample Generation

- Samples are pulled from Gaussian distribution with a mean of 1 and a standard deviation of $\sigma = \sqrt{\frac{1}{R*10^{SNR/10}}}$, where R is the code rate

```
loop_check = get_error_sample_size(sym_size);
valid_samples = zeros(1,loop_check); % initialize valid samples
valid_idx = 1;
error_samples = zeros(1,loop_check); % initialize error samples
error_idx = 1;
while valid_idx <= loop_check || error_idx <= loop_check
   temp = normrnd(1,sigma); % generate samples
   if temp > 0 % sort valid samples
      valid_samples(valid_idx) = temp;
      valid_idx = valid_idx +1;
```

Sample Generation

- Samples are pulled from Gaussian distribution with a mean of 1 and a standard deviation of $\sigma = \sqrt{\frac{1}{R*10^{SNR/10}}}$, where R is the code rate
- Currently the tool generates a list of samples and sorts into valid and error sample bins

```
loop_check = get_error_sample_size(sym_size);
valid_samples = zeros(1,loop_check); % initialize valid samples
valid_idx = 1;
error_samples = zeros(1,loop_check); % initialize error samples
error_idx = 1;
while valid_idx <= loop_check || error_idx <= loop_check
  temp = normrnd(1,sigma); % generate samples
  if temp > 0 % sort valid samples
    valid_samples(valid_idx) = temp;
    valid_idx = valid_idx +1;
```

Sample Generation

- Samples are pulled from Gaussian distribution with a mean of 1 and a standard deviation of $\sigma = \sqrt{\frac{1}{R*10^{SNR/10}}}$, where R is the code rate
- Currently the tool generates a list of samples and sorts into valid and error sample bins
- An error sample is one that comes from the negative tail of the Gaussian distribution

```
loop_check = get_error_sample_size(sym_size);
valid_samples = zeros(1,loop_check); % initialize valid samples
valid_idx = 1;
error_samples = zeros(1,loop_check); % initialize error samples
error_idx = 1;
while valid_idx <= loop_check || error_idx <= loop_check
   temp = normrnd(1,sigma); % generate samples
   if temp > 0 % sort valid samples
      valid_samples(valid_idx) = temp;
      valid_idx = valid_idx +1;
```

Sample Generation

- Samples are pulled from Gaussian distribution with a mean of 1 and a standard deviation of $\sigma = \sqrt{\frac{1}{R*10^{SNR/10}}}$, where R is the code rate
- Currently the tool generates a list of samples and sorts into valid and error sample bins
- An error sample is one that comes from the negative tail of the Gaussian distribution
- There is probably a better way to do this, and finding that is on the to-do list

```
loop_check = get_error_sample_size(sym_size);
valid_samples = zeros(1,loop_check); % initialize valid samples
valid_idx = 1;
error_samples = zeros(1,loop_check); % initialize error samples
error_idx = 1;
while valid_idx <= loop_check || error_idx <= loop_check
   temp = normrnd(1,sigma); % generate samples
   if temp > 0 % sort valid samples
      valid_samples(valid_idx) = temp;
      valid_idx = valid_idx +1;
```

Energy Calculation

```
%initialize Energy and check node matrices
E = zeros(2^sym_size,sym_size);
chk_nodes = ones(1, check_size);
chk_sum = zeros(1,sym_size);
% Calculate all possible energy values for each state
for row = 1:2^sym_size
   % Calculate all check nodes
   for adj_row = 1:check_size
         for adj_col = 1:sym_size
            if adj_mat(adj_row,adj_col) == 1
               chk_nodes(adj_row) = chk_nodes(adj_row)*x(row,adj
               chk_sum(adj_col) = chk_sum(adj_col)+chk_nodes(adj
            end
         end
   end
   % Calculate energy values
   for E_idx = 1:sym_size
         E(row,E_idx) = y(E_idx)*x(row,E_idx)+w*chk_sum(E_idx);
   end
end
```

Transition Probabilities

```
p = ones(2^sym_size,2^sym_size);
        % Flip probabilities calculated according to Eq 3.13 in
        % dissertation (pg. 26)
        for row = 1:2^sym_size
            px = zeros(1,sym_size);
            for p_idx = 1:sym_size
                px(p_idx) = normcdf(theta,E(row,p_idx),sigma);
            end
            rowbin = dec2bin(row-1,sym_size);
            for col = 1:2^sym_size
                colbin = dec2bin(col-1,sym_size);
                for p_idx = 1:sym_size
                    if rowbin(p_idx) == colbin(p_idx)
                        p(row,col) = p(row,col)*(1-px(p_idx));
                    else
                        p(row,col) = p(row,col)*px(p_idx);
                    end
                end
            end
        end
        % Sanity check
```

Write Files and Process Outputs

```
% Process Output for transient and steady state
if (tag(3) == 't' && tag(4) == 'r') || (tag(3) == 's' && tag(4)
      str_idx = regexp(output,regexptranslate('wildcard','0:\(*\
      output = substr(output,str_idx);
      split_output = strsplit(output,"\n");
      for out_idx = 1:2^sym_size
         str_to_parse = char(split_output(out_idx));
         if (str_to_parse(1) >= "0") && (str_to_parse(1) <= "9")
            temp = textscan(str_to_parse,"%d:(%d)=%f");
            state_temp = temp{1,2};
            p_out(idx,state_temp+1) = temp{1,3};
         else
            break;
         end
      end
elseif tag(3) == 'p'
      str_idx = strfind(output,"Result");
      output = substr(output,str_idx);
      p_temp = textscan(output,"Result: %f (exact floating point
      p_out(idx,1) = p_temp{1,1};
else
```

Sources

- [1] Tasnuva Dissertation

Sources

- [1] Tasnuva Dissertation
- [2] T. Tithi, C. Winstead, and G. Sundararajan, Gopalakrishnan, "Decoding LDPC codes via Noisy Gradient Descent Bit-Flipping with Re-Decoding", 2015.

Sources

- [1] Tasnuva Dissertation
- [2] T. Tithi, C. Winstead, and G. Sundararajan, Gopalakrishnan, "Decoding LDPC codes via Noisy Gradient Descent Bit-Flipping with Re-Decoding", 2015.
- [3] T. wadayama, K. Nakamura, M. Yagita, Y. Funahashi, S. Usami, and I. Takumi, "Gradient descent bit flipping algorithms for decoding LDPC codes", *Communications, IEEE Transactions on*, vol. 58, no. 6, pp. 1610-1614, 2010.

Sources

- [1] Tasnuva Dissertation
- [2] T. Tithi, C. Winstead, and G. Sundararajan, Gopalakrishnan, “Decoding LDPC codes via Noisy Gradient Descent Bit-Flipping with Re-Decoding”, 2015.
- [3] T. wadayama, K. Nakamura, M. Yagita, Y. Funahashi, S. Usami, and I. Takumi, “Gradient descent bit flipping algorithms for decoding LDPC codes”, *Communications, IEEE Transactions on*, vol. 58, no. 6, pp. 1610-1614, 2010.
- [4] NGDBF demo

Sources

- [1] Tasnuva Dissertation
- [2] T. Tithi, C. Winstead, and G. Sundararajan, Gopalakrishnan, "Decoding LDPC codes via Noisy Gradient Descent Bit-Flipping with Re-Decoding", 2015.
- [3] T. wadayama, K. Nakamura, M. Yagita, Y. Funahashi, S. Usami, and I. Takumi, "Gradient descent bit flipping algorithms for decoding LDPC codes", *Communications, IEEE Transactions on*, vol. 58, no. 6, pp. 1610-1614, 2010.
- [4] NGDBF demo
- [5] Trapping set ontology