New LDPC code design scheme combining differential evolution and simplex algorithm

Min Kyu Song

The Graduate School
Yonsei University
Department of Electrical and Electronic Engineering

New LDPC code design scheme combining differential evolution and simplex algorithm

by

Min Kyu Song

A Master Thesis Submitted to the
Graduate School of Yonsei University
in Partial Fulfillment of the
Requirements for the Degree of
MASTER OF SCIENCE

Supervised by
Professor Hong-Yeop Song, Ph.D.

Department of Electrical and Electronic Engineering
The Graduate School
YONSEI University

December 2012

This certifies that the master thesis
of Min Kyu Song is approved.

____

Thesis Supervisor : Hong-Yeop Song

____

Sooyong Choi

____

Taewon Hwang

The Graduate School
Yonsei University

December 2012

Acknowledgments

At first, I would like to thank my advisor professor Hong-yoep Song who have generously given advice to me for my research. From his advice, I was able to learn a lot. Also, I am deeply thankful to professor Sooyong Choi and professor Taewon Hwang who gave me lots of advice about my Master’s thesis.

I have pleased studying with my colleagues in Coding and Crypto Lab. during the Master’s course. Especially, Jin Soo Park, Suc Min Jun, and Young-Tae Kin gave me lots of advice. I would like to thank them.

I would like to say thank you to my family and all my friends. They always tell me ‘you can do it.’ It seems to be a very powerful word.

Finally, I would like to thank my grandmother who goes to heaven fifth month ago. I want to say her ‘I miss you. And I love you.’

Min Kyu Song

January 2013

Contents

List of Figures

List of Tables

ABSTRACT

New LDPC code design scheme combining differential evolution and simplex algorithm

Song, Min Kyu
Dept. of Electronic Engineering
The Graduate School
Yonsei University

In this thesis, we propose a new LDPC code design scheme that combines differential evolution and iterative simplex algorithm. In the proposed scheme, we find good check and variable node degree distribution and threshold by using differential evolution, and then, we find check and variable node degree distribution that has enhanced code rate by using an iterative simplex algorithm. We also discuss some practical implementation issues of using differential evolution for threshold optimization and verify that it works well by comparing it with some well-known degree distributions. An iterative simplex algorithm consists of two simplex algorithms, optimizing $\rho(x)$ and $\lambda(x)$, respectively.

Some simulation results show that we can find degree distribution with proposed LDPC code design scheme better than some well-known degree distribution.

Key words : LDPC, code design, simplex algorithm, differential evolution, degee distribution optimization

Chapter 1

Introduction

1.1 Motivation

From the discovery of turbo codes, iterative decoders have attracted a lot of attention. As an example of the attention, LDPC codes - first discovered by Gallager [1] - were rediscoverd by Spielman et al. [3] and Mackay et al. [4] and there have been many researches to analyze the LDPC codes. The LDPC codes have a very important property called threshold phenomenon: for any noise level that is smaller than a certain value, an arbitrary small bit-error probability can be achieved. This phenomenon is first observed by Gallager for binary symmetric channels (BSCs) [1], [2] and generalized by Luby et. al. [6], Richardson and Urbanke [7].

In [7], Richardson and Urbanke proved the decoder performance on random graph converges to its expected value as the length of the code increases. In [10], they proposed an analysis tool for LDPC codes, called Density evolution and analyzed the expected behavior of ensemble of LDPC codes that is cycle-free and has infinitely long length with density evolution. The key idea of density evolution is to track the densities of the messages flowed in the Tanner graph of LDPC code. To obtain the densities of the messages requires high computational complexity that is a serious problem in practice. Thus, Sae-Young chung et al. proposed Gaussian approximation to reduce the computational complexity [8]. In

Gaussian approximation, the message densities are assumed gaussian for the convenience of calculation.

Since LDPC codes can be specified with check and variable node degree distributions and their connection, LDPC code design consist of two steps, code design and code construction. Code design is a process to design ensemble of LDPC codes by determining check and variable node degree distributions. After code design, code construction is followed. Code construction can be done with Random, Progressive Edge Growth (PEG) [13], and Approximated Cycle Extrinsic message degree (ACE) [14]. By using density evolution, code design problem can be changed to a non-linear problem and degree distribution of an ensemble of LDPC codes can be optimized using differential evolution, well-known non-linear program solver [10]. But, in [10], there is no details about how to apply differential to code design. There are only some degree distributions. In practice, sometimes, we will be faced the code design with arbitrary code rate and maximum check/variable node degree. Thus details about code design is important.

In this thesis, we describe details about code design by using differential evolution, especially how to apply differential evolution to degree distribution optimization. And we propose a new iterative simplex algorithm for code rate enhancement and new code design scheme.

1.2 An overview

The remainder of this thesis is organized as follows. In Chapter 2, we give brief description of the LDPC code design problem and two optimization methods, simplex algorithm and differential evolution. Then, we will describe Gaussian approximation for changing code design problem to optimization problem. In Chapter 3, we describe details about threshold optimization with differential evolution. In Chapter 4, we propose a new LDPC code design scheme and compare it with some well-known degree distribution and differential evolution only. And we find some new degree distributions. Conclusions are presented in Chapter 5.

Chapter 2

Degree distribution optimization problem

In this chapter, we will describe LDPC code design problem. Then, we will briefly describe some well-known linear and non-linear program solver, simplex algorithm and differential evolution. Finally, we will convert code design problem to two optimization problems, threshold optimization problem and code rate optimization problem.

2.1 An overview of LDPC code design

As their name suggests, LDPC codes are block codes with parity-check matrices that contain only a very small number of non-zero entries. Aside from the requirement that parity-check matrix H be sparse, an LDPC code itself is no different from any other block code. Indeed, classical block codes will work well with iterative decoding algorithms. However, generally finding a sparse matrix for an existing code is not practical. Instead LDPC codes are designed by constructing a suitable sparse parity-check matrix first and then determining an encoder for the code afterwards.

An ensemble of the LDPC code is specified by variable and check nodes degree distributions $\lambda(x)$, $\rho(x)$. Let d_v, d_c be maximum degree of the variable and check nodes, respectively. Then, degree distribution of the LDPC code is defined as

$$\lambda(x) = \sum_{i=2}^{d_v} \lambda_i x^{i-1},\ \ \rho(x) = \sum_{i=2}^{d_c} \rho_i x^{i-1}$$

where λ_i and ρ_i are the fractions of edges belonging to degree-i variable and check nodes, respectively [9]. Using this expression, the norminal rate R of the code is given by [11]:

$$R = 1 - \frac{\int_0^1 \rho(x)dx}{\int_0^1 \lambda(x)dx}.$$

With these variable and check nodes degree distributions and connection between variable and check nodes, an LDPC code is represented as bipartite graph, called Tanner graph. An example of the Tanner graph is shown in Figure 2.1.

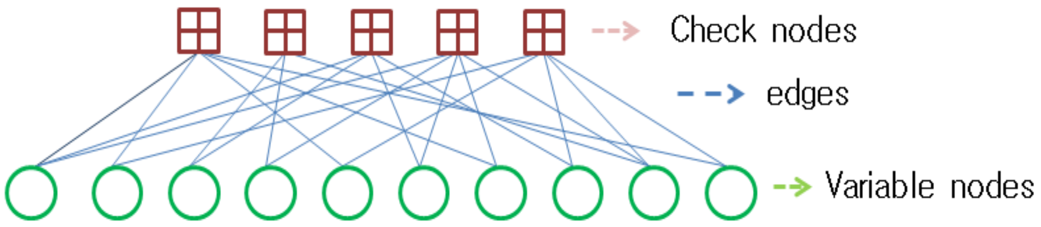


Figure 2.1 Example : Tanner graph of an LDPC code.

If variable and check nodes degree distributions have only one non-zero term

$$\lambda(x) = \lambda_{d_v} x^{d_v - 1}$$

$$\rho(x) = \rho_{d_c} x^{d_c - 1},$$

then the LDPC code is called regular. Otherwise, the LDPC code called irregular.

Since LDPC code can be specified with variable/check nodes degree distributions and their connection, LDPC code design is generally has two steps, 'Code design' and 'Code construction'. In the code design step, we select a good ensemble of LDPC codes by determining variable and check nodes degree distributions. After the code design, we construct bipartite graph using some construction methods.

This is called code construction. There exist some code construction methods such as Random, PEG [13], and ACE [14]. The code design can be achieved with Density evolution and differential evolution [10]. But there exist no details about how to apply differential evolution to code design. Only some degree distributions with code rate 0.5 were presented in [10]. Following table describes some degree distribution and its threshold E_b/N_0 obtained by Gaussian approximation. In the table, $(E_b/N_0)^*_{dB}$ means threshold $(E_b/N_0)_{dB}$. In this thesis, some threshold values will be represented as $(\ )*$.

Since there is no details about how to apply differential evolution to code design, we will be faced with many problems when we want to design ensemble of LDPC codes with arbitrary R, d_v and d_c. And it is also a problem that if we use differential evolution for code design, what is good differential evolution scheme. We will try to solve these problems in this thesis.

There are two objects of LDPC code design, code rate R [8] and threshold $s*$ [10] where the threshold is defined as the maximum noise level such that the probability of error tends to zero as the number of iterations tends to infinity. This two optimization problems are described in Figure 2.2. Since code rate is given in practice, optimizing threshold for given code rate is practically important. To solve the threshold optimization problem, the constraint that bit error probability p_e goes to 0 as iteration l goes to infinity should be some function of $\rho(x)$ and $\lambda(x)$. This process will be done by Gaussian approximation.

Table 2.1 Some well-known degree distributions

	Richardson [10]		
d_v	4	5	6
λ_2	0.38354	0.32660	0.33241
λ_3	0.04237	0.11960	0.24632
λ_4	0.57409	0.18393	0.11014
λ_5		0.36988	
λ_6			0.31112
ρ_5	0.24123		
ρ_6	0.75877	0.78555	0.76611
ρ_7		0.21445	0.23389
R	0.50000	0.50000	0.50839
$(\frac{E_b}{N_0})^*_{dB}$	0.8736	0.8318	0.7997
$(\frac{E_s}{N_0})^*$	0.6114	0.6056	0.6112

Optimize threshold

Minimize s^*

subject to

$$\sum_{i=2}^{d_c} \rho_i = 1$$

$$\sum_{i=2}^{d_v} \lambda_i = 1$$

$$p_e \to 0 \; as \; l \to \infty$$

by tuning

$$\rho(x),\ \lambda(x)$$

for given

$$R,\ d_v,\ d_c$$

Optimize code rate

Maximize R

subject to

$$\sum_{i=2}^{d_c} \rho_i = 1$$

$$\sum_{i=2}^{d_v} \lambda_i = 1$$

$$p_e \to 0 \; as \; l \to \infty$$

by tuning

$$\rho(x),\ \lambda(x)$$

for given

$$s^*,\ d_v,\ d_c$$

Figure 2.2 Two objects of code design and their optimization problems

2.2 Optimization methods for degree distribution optimization problems

Before we describe detail of code design, we will illustrate two well-known optimization algorithms: simplex algorithm and differential evolution. Brief description of two methods is shown in Table 2.2.

Table 2.2 Brief description of simplex algorithm and differential evolution

	Simplex algorithm	**Differential evolution**
Type	Optimization algorithm for linear program (LP) problems	Heuristic
Requirement	Standard form of LP problems	System parameter
Solution	Global optimum	Close to global optimum
Stopping condition	Optimality of current basic solution is guaranteed	Fixed iteration number

2.2.1 Simplex Algorithm

Simplex algorithm is a commonly used linear programming technique. Every linear program can be converted into a 'standard form',

$$\begin{aligned} \textit{maximize} \quad & c_1x_1 + \dots + c_nx_n \\ \textit{subject to} \quad & a_{11}x_1 + \dots + a_{1n}x_n = b_1 \\ & \vdots \\ & a_{m1}x_{m1} + \dots + a_{mn}x_n = b_m \\ & x_1 \geq 0, \dots, x_n \geq 0 \end{aligned}$$

where the objective $c_1x_1 + \dots + c_nx_n$ is maximized, the constraints are equalities and the variables are all non-negative. To solve a linear program by using simplex algorithm, this is done as follows:

- If the problem is $\min z$, convert it to $\max -z$.
- If a constraint is $a_{i1}x_1 + \ldots + a_{in}x_n \geq b_i$, convert it into an equality constraint by adding a nonnegative slack variable. The resulting constraint is $a_{i1}x_1 + \ldots + a_{in}x_n + s = b_i$ where $s \geq 0$.
- If a constraint is $a_{i1}x_1 + \ldots + a_{in}x_n \leq b_i$, convert it into an equality constraint by subtracting a nonnegative slack variable. The resulting constraint is $a_{i1}x_1 + \ldots + a_{in}x_n - s = b_i$ where $s \geq 0$.
- If some variable x_j is unrestricted in sign, replace it everywhere in the formulation by $x'_j - x''_j$, where $x'_j,\ x''_j \geq 0$.

After above process, the equations can be transformed to matrix form. simplex algorithm solve the linear program by using this matrix.

In simplex algorithm, there are two variables, basic and nonbasic. The variables (other than the special variable z) which appear in only one equation are the basic variables. Other variables are nonbasic variables. A basic solution is obtained from the system of equations by setting the nonbasic variables to zero. If there is negative term in the leftmost column of the matrix, it can't have basic feasible solution. There are two steps to find optimal solution by using simplex algorithm, Phase 1 and Phase 2. If there exist negative term in the leftmost column of the matrix, we can transform the matrix to another matrix by pivoting with a row that has negative term in the leftmost. This process is called Phase 1. If there exist feasible solution, then Phase 1 will be skipped. In the Phase 2, we will solve the linear program. Phase 2 is commonly known as simplex algorithm.

There are two rules in Phase 2.

Rule 1. If all variables have a nonnegative coefficient in Row 0, the current basic solution is optimal. Otherwise, pick a variable x_j with a negative coefficient in Row 0. The chosen variable is called the entering variable.

Rule 2. For each Row $i, i \geq 1$, where there is a strictly positive entering variable coefficient, compute the ratio of the Right Hand Side (RHS) to the entering variable coefficient. Choose the pivot row as being the one with MINIMUM ratio.

The simplex algorithm iterates between Rules 1,2 and pivoting until Rule 1 guarantees that the current basic solution is optimal.

Using simplex algorithm to solve linear program, there are two problems, 'degeneracy' and 'unbounded optimum'. The degeneracy problem denotes that, for given iteration, there exist a basic variable which is equal to 0. If degeneracy is occurred for many iterations, the solver repeats same process indefinitely, called cycling. To avoid cycling, we must choose the entering variable with smallest index in Rule 1. But, in commercial, no effort is made to avoid cycling Since cycling is extremely rare and the precision of computer arithmetic takes care of cycling by itself: The cumulative error will be a way that solve degeneracy. The unbounded optimum problem will be occur if there is no positive entry in the column of the entering variable. If it is occurred, the optimal solution is unbounded for some basic variables.

Example

If the linear program problem is given as follow.

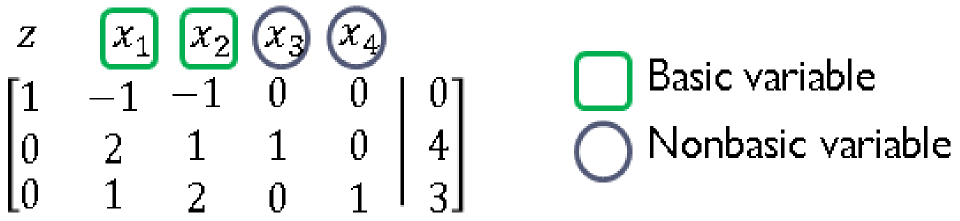
$$\begin{aligned} &\textit{maximize} && z = x_1 + x_2 \\ &\textit{subject to} && 2x_1 + x_2 \le 4 \\ & && x_1 + 2x_2 \le 3 \\ & && x_1, x_2 \ge 0 \end{aligned}$$

At fist, we should transform the problem to its standard form,

$$\begin{aligned} &\textit{maximize} && z = x_1 + x_2 \\ &\textit{subject to} && 2x_1 + x_2 + x_3 = 4 \\ & && x_1 + 2x_2 + x_4 = 3 \\ & && x_1, x_2, x_3, x_4 \ge 0 \end{aligned}$$

where x_3 and x_4 are slack variable.

Matrix representation of the standard form is shown in Figure 2.3.



$$\begin{array}{cccccc} z & \boxed{x_1} & \boxed{x_2} & x_3 & x_4 & \\ 1 & -1 & -1 & 0 & 0 & 0 \\ 0 & 2 & 1 & 1 & 0 & 4 \\ 0 & 1 & 2 & 0 & 1 & 3 \end{array}$$

Figure 2.3 Matrix representation of the standard form

Since there are negative terms in the row 0, the basic solution is not optimal. By rule 1, we pick column 1 for pivoting. Since $\frac{4}{2} = 2$ is smaller than $\frac{3}{1} = 3$, we pick row 1 for pivoting. By pivoting, we get a new matrix as Figure 2.4.

$$\begin{array}{cccccc|c} z & \enclose{circle}{x_1} & \boxed{x_2} & \boxed{x_3} & \enclose{circle}{x_4} & & \\ 1 & 0 & -1/2 & 1/3 & 0 & & 2 \\ 0 & 1 & 1/2 & 1/2 & 0 & & 2 \\ 0 & 0 & 3/2 & -1/2 & 1 & & 1 \end{array}$$

🟩 Basic variable
○ Nonbasic variable

Figure 2.4 Result of first pivoting

Since there is a negative term in the row 0, the basic solution is not optimal. So, we should pick column2 and row 2, and pivot the row. Then, we get a new matrix as Figure 2.5

$$\begin{array}{cccccc|c} z & \enclose{circle}{x_1} & \enclose{circle}{x_2} & \boxed{x_3} & \boxed{x_4} & & \\ 1 & 0 & 0 & 1/3 & 1/3 & & 7/3 \\ 0 & 1 & 0 & 2/3 & -1/3 & & 5/3 \\ 0 & 0 & 1 & -1/3 & 2/3 & & 2/3 \end{array}$$

🟩 Basic variable
○ Nonbasic variable

Figure 2.5 Result of second pivoting

Since there is no negative term in row 0, ***Rule 1*** guarantee that the basic solution is optimal.

※ **Optimal solution**

$$z = \frac{7}{3},\ x_1 = \frac{5}{3},\ x_2 = \frac{2}{3},\ x_3 = 0,\ x_4 = 0$$

2.2.2 Differential evolution

Differential evolution (DE) is a parallel direct search method which utilizes N parameter vectors

$$x_i^G,\ i = 1, 2, ..., N$$

as a population for each generation G. N does not change during the optimization process. The initial vector population is chosen randomly and should cover the entire parameter space. DE generates new parameter vectors by adding the weighted difference between two population vectors to a third vector. This operation is called mutation. The elements of a mutated vector are then mixed with the elements of another predetermined vector, the target vector, to yield the so-called trial vector. Mixing elements of mutated vector is often referred to as 'crossover'. If the trial vector yields a lower cost function value than the target vector, the trial vector replaces the target vector in the following generation. This last operation is called 'selection'. Differential evolution is often represented as

$$DE/X/D/Z$$

where X specifies the vector to be mutated which currently can be 'rand' (a randomly chosen population vector) or 'best' (the vector of lowest cost from the current population), D is the number of difference vectors used to mutate, and Z denotes the crossover scheme. Differential evolution is known as more efficient scheme than 'Annealing methods' [11].

Mutation

For each target vector $x_i^G,\ i = 1, ..., N$, a mutant vector is generated as

$$v_i^{G+1} = x_j^G + F \times \sum_{k=1}^{D} (x_{r_{1k}}^G - x_{r_{2k}}^G)$$

with random indices $r_{1k},\ r_{2k}$ where D is the number of difference vectors used to generate mutated vectors. The randomly chosen indexes $r_{1k},\ r_{2k}$ are chosen to be different. F is a real and constant factor $\in [0,2]$ which controls the amplification of the differential variation.

Crossover

In oder to increase the diversity of the pertubed parameter vectors, crossover is introduced. In this process, the trial vector is mixing of the mutated vector and target vector to generate trial vector u_i^{G+1}.

Selection

To decide whether or not it should become a member of population at generation $G+1$, the trial vector u_i^{G+1} is compared to the target vector x_i^G using the greedy condition. If vector u_i^{G+1} yields a smaller cost function value than x_i^G, then x_i^{G+1} is set to u_i^{G+1}. Otherwise, the old value x_i^G is retained.

Example

Assume that we want to optimize following problem.

$$\begin{aligned} \textit{maximize} \quad & z = a_1 + a_2 \\ \textit{subject to} \quad & 2a_1 + a_2 \le 4 \\ & a_1 + 2a_2 \le 3 \\ & a_1, a_2 \ge 0 \end{aligned}$$

We will solve this problem with differential evolution. The system parameters are given as $N=4$, $DE/best/1/bin$ where the crossover scheme bin means element-wise mixing.

Then, initial populations are generated randomly. We assume that generated initial populations are x_1, x_2, x_3, x_4 in Figure 2.6.

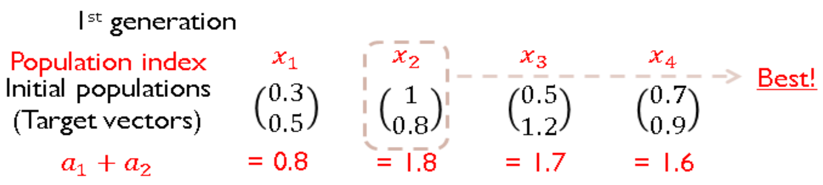


Figure 2.6 Generated initial vectors

Since $a_1 + a_2$ is best when x_2 is selected, x_2 is best population in the initial population.

Then, Mutated vectors are generated as Figure 2.7.

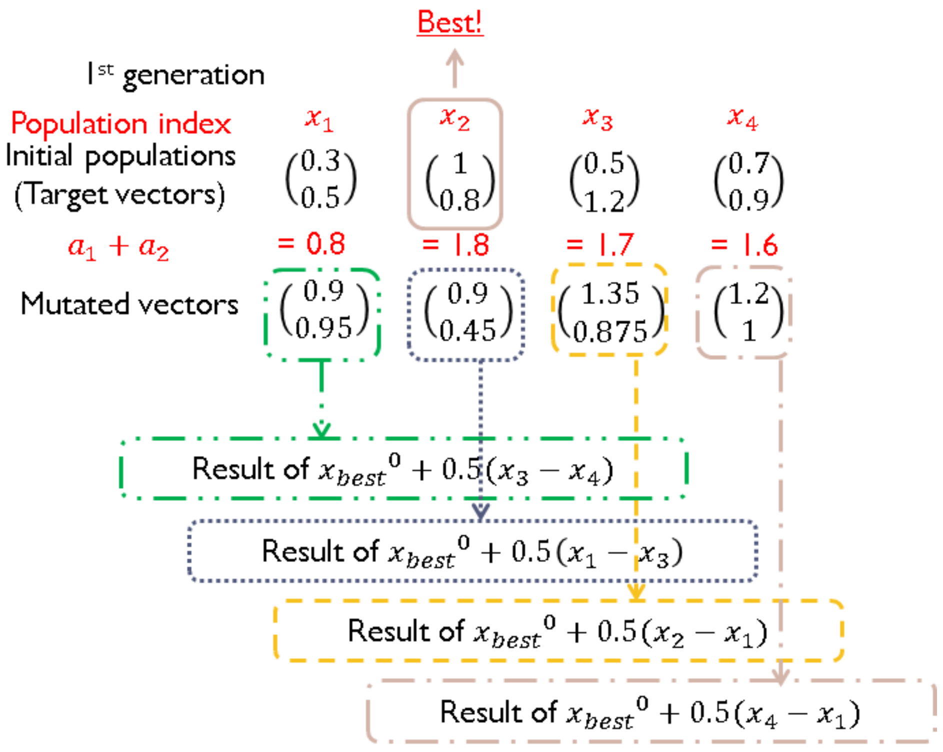
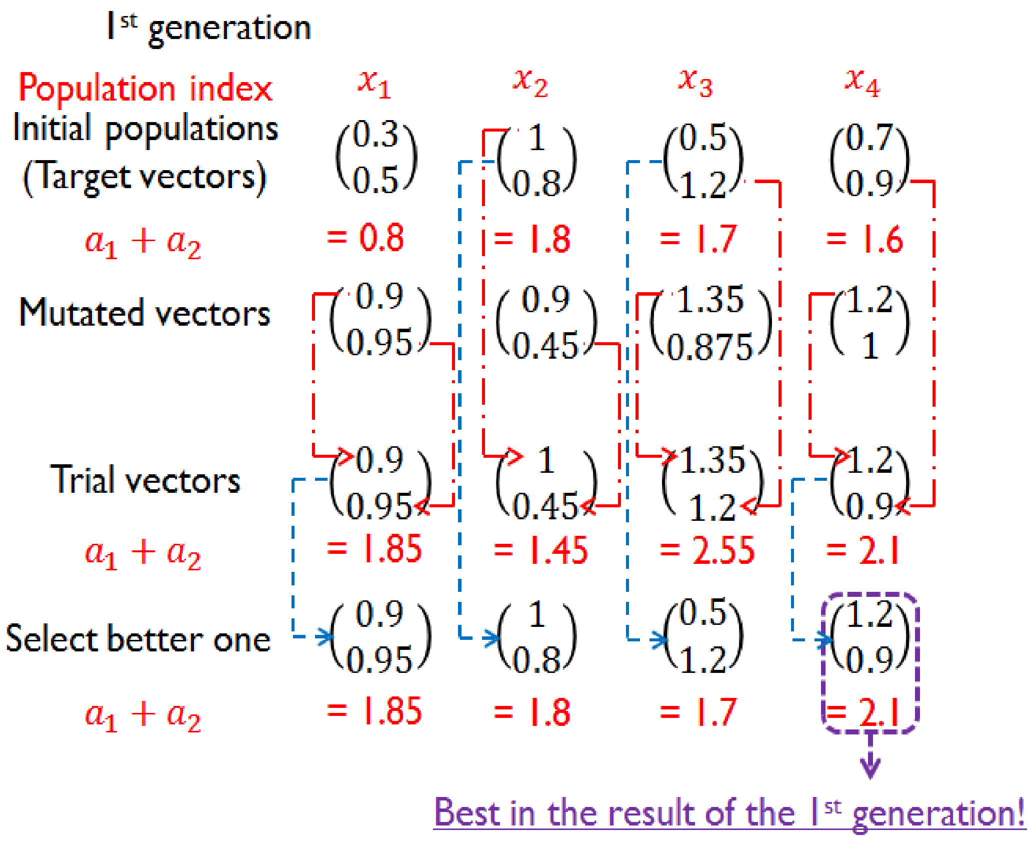


Figure 2.7 Mutation process

After mutation process, we can obtain trial vectors via crossover and select better one between target and trial. Figure 2.8 shows crossover and selection. We will repeat this process for a fixed number of iteration to find solution that is close to optimal.



1st generation
Population index
x_1
x_2
x_3
x_4
Initial populations (Target vectors)
$\binom{0.3}{0.5}$ $\binom{1}{0.8}$ $\binom{0.5}{1.2}$ $\binom{0.7}{0.9}$
$a_1 + a_2$
= 0.8 = 1.8 = 1.7 = 1.6
Mutated vectors
$\binom{0.9}{0.95}$ $\binom{0.9}{0.45}$ $\binom{1.35}{0.875}$ $\binom{1.2}{1}$
Trial vectors
$\binom{0.9}{0.95}$ $\binom{1}{0.45}$ $\binom{1.35}{1.2}$ $\binom{1.2}{0.9}$
$a_1 + a_2$
= 1.85 = 1.45 = 2.55 = 2.1
Select better one
$\binom{0.9}{0.95}$ $\binom{1}{0.8}$ $\binom{0.5}{1.2}$ $\binom{1.2}{0.9}$
$a_1 + a_2$
= 1.85 = 1.8 = 1.7 = 2.1
Best in the result of the 1st generation!

Figure 2.8 Crossover and Selection.

2.3 Density evolution and Gaussian approximation

2.3.1 Density evolution

Under the message passing algorithm (sometimes called sum-product algorithm), variable and check nodes exchange messages at each iteration. At l-th iteration, a check node gets messages from its neighbors, processes the messages and sends the result back to its neighbors. The message that will be sent back from the check node to a variable node will be made with all incoming messages except the incoming message on the edge where the output message will be sent out. A variable node operate similarly as check node. Only difference between variable and check nodes are the process to generate output message with incoming messages.

For convenience of the analysis, we use log-likelihood ratios (LLRs) as messages. So, we use

$$v = \log \frac{p(y|x=0)}{p(y|x=1)}$$

as the output message of a variable node, where x is the value of the variable node and y denotes all the information that is available to the variable node up to the present iteration except the incoming message on the edge where the output message will be sent out. Similarly, the output message of a check node is defined as

$$u = \log \frac{p(y'|x'=0)}{p(y'|x'=1)}$$

where x' is the value of the variable node that gets the message from the check node, and y' denotes all the information that is available to the check node up to the present iteration except the incoming message on the edge where the output message will be sent out.

If the LDPC code is cycle-free, then we can analyze the decoding algorithm straightforwardly because incoming messages to every node are independent. Thus we assume that the LDPC code is cycle-free and infinitely long length.

The main idea of density evolution is tracking densities of messages that flow on the graph. Under the sum-product algorithm, variable and check node equations are given as

$$v = \sum_{i=0}^{d-1} u_i$$

$$\tanh\frac{u}{2} = \prod_{j=1}^{d-1} \tanh\frac{v_j}{2},$$

where d denote degree of the node. Since we know density of channel output, we can track densities of messages. But obtaining densities of messages requires high computational complexity.

2.3.2 Gaussian approximation

Since Density evolution require high complexity, Gaussian approximation is proposed [8]. For the convenience of analysis, we assume that power of transmitted symbol E_s is 1. If the channel is BI-AWGN, the LLR message u_0 from the channel is Gaussian with mean $2/\sigma_n^2$ and variance $4/\sigma_n^2$, where σ_n^2 is the variance of the channel noise. Thus, if all u_i except u_0 are independent and identical Gaussian, then the resulting sum is also Gaussian. Even if the inputs are not Gaussian, by the central limit theorem, the sum would look like a Gaussian if many independent random variables are added.

From the idea described above, the Gaussian approximation assume that all of the messages are gaussian. Since Gaussian is completely specified by its mean and variance, we need to keep only the means and variances during iteration. By enforcing symmetry condition, which can be expressed as $f(x)=f(-x)e^x$, for the approximated Gaussian densities at every iteration, we can greatly improve the accuracy of the approximation [8]. For the Gaussian with mean m and variance σ^2, this condition reduces to $\sigma^2=2m$, which means that we need to keep only the mean.

We denote the mean of u and v by m_u and m_v, respectively. Then (2.1) simply becomes

$$m_v^{(l)}=m_{u_0}+(d_v-1)m_u^{(l-1)} \tag{2.3}$$

where m_{u_0} is the mean of u_0 and l denotes the lth iteration. The index i for u_i is omitted because u_i's are i.i.d. for $1\le i\le d_v$ and have same mean. Note that $m_u^{(0)}$ is zero since the initial message from any check node is 0.

The updated mean $m_u^{(l)}$ at the lth iteration can be calculated by taking expectations on each side of (2.2),

$$E[\tanh\frac{u}{2}]=E[\tanh\frac{v_j}{2}]^{d_c-1} \tag{2.4}$$

where we have omitted the index j and simplified the product because the v_j's are i.i.d. Note that $E[\tanh\frac{x}{2}]$ depends only on the mean m_u of u, we define a function to calculate $E[\tanh\frac{x}{2}]$,

$$\phi(x) = \begin{cases} 1 - \frac{1}{\sqrt{4\pi x}} \int_R \tanh \frac{u}{2} e^{-\frac{(u-x)^2}{4x}} du, & \text{if } x > 0 \\ 1, & \text{if } x = 0 \end{cases}.$$

The function $\phi(x)$ is continuous and monotonically decreasing on $[0, \infty)$, with $\phi(0) = 1$ and $\phi(\infty) = 0$. Using $\phi(x)$, the update rule for m_u can be represented as

$$m_u^{(l)} = \phi^{-1}(1 - [1 - \phi(m_{u_0} + (d_v - 1)m_u^{(l-1)})]^{d_c - 1}) \qquad (2.5)$$

where $m_u^{(0)} = 0$ is the initial value for m_u.

If the LDPC code is irregular, then (2.5) will be (2.6).

$$m_u^{(l)} = \sum_{j=2}^{d_c} \rho_j \phi^{-1}(1 - [1 - \sum_{i=2}^{d_v} \lambda_i \phi(m_{u_0} + (i-1)m_u^{(l-1)})]^{j-1}) \qquad (2.6)$$

For convenience, we denote m_u's update function as

$$g(s,t) = \sum_{j=2}^{d_c} \rho_j \phi^{-1}(1 - [1 - \sum_{i=2}^{d_v} \lambda_i \phi(s + (i-1)t)]^{j-1}).$$

In this case, the threshold $s*$ is the infimum of all $s = 4 \times E_s/N_0$ in R^+ such that $m_u^{(l)}$ goes to ∞ as l goes to ∞. With above equation, the threshold in terms of noise power is equal to $2/s^*$. Since $\phi(x)$ is monotonically decreasing on $0 \leq x < \infty$, $g(s,t)$ is monotonically increasing on both $0 < s < \infty$ and $0 \leq t < \infty$. Thus, $g(s,t)$ will converge to ∞ if and only if $t < g(s,t)$ for all $t \in R^+$.

In [10], by expecting behavior of m_u's update rule for a sufficiently large t, stability condition is derived as

$$\lambda_2 < \lambda_2^* = e^{1/2\sigma_n^2} / \prod_{j=2}^{d_c} (j-1)^{\rho_j}. \qquad (2.7)$$

An alternative expression of (2.6) is

$$r^l = h(s, r^{(l-1)}) = \sum_{i=2}^{d_v} \lambda_i \phi[s + (i-1) \sum_{j=2}^{d_c} \rho_j \phi^{-1}(1-(1-r^{(l-1)})^{j-1})], \quad (2.8)$$

where $h(s,r)$ denote r's update function and r^l goes to zero if and only if $r > h(s,r)$ for $0 < r < \phi(s)$. With (2.6) and (2.7) (or (2.7) and (2.8)), the degree distribution optimization can be done.

It is known that $\rho(x) = \rho x^{d_c - 1} + (1-\rho)x^{d_c}$ is good enough to obtain good performance [8]. With above, the code design problem can represented as Figure 2.9. For code rate optimization problem, it can be linear program if $\rho(x)$ (or $\lambda(x)$) is given.

There is a problem that we can't calculate $s*$ directly since the representation of $s*$ in terms of $\rho(x)$ and $\lambda(x)$ is not known. Thus, we can obtain threshold $s*$ through trial and error.

Optimize threshold

Minimize $s*$

subject to

$$\rho_{d_c-1}+\rho_{d_c}=1$$

$$\sum_{i=2}^{d_v}\lambda_i=1$$

$$r>h(s,r)\ \text{for}\ ^{\forall}r$$

$$0<r<\phi(s)$$

$$(\text{or}\ t<g(s,t)\ \text{for}\ t\in R^{+})$$

$$\lambda_2<e^{1/2\sigma_n^2}/\prod_{j=2}^{d_c}(j-1)^{\rho_j}$$

by tuning

$$\rho(x),\ \lambda(x)$$

for given

$$R,\ d_v,\ d_c$$

Optimize code rate

Maximize R

subject to

$$\rho_{d_c-1}+\rho_{d_c}=1$$

$$\sum_{i=2}^{d_v}\lambda_i=1$$

$$r>h(s,r)\ \text{for}\ ^{\forall}r$$

$$0<r<\phi(s)$$

$$(\text{or}\ t<g(s,t)\ \text{for}\ t\in R^{+})$$

$$\lambda_2<e^{1/2\sigma_n^2}/\prod_{j=2}^{d_c}(j-1)^{\rho_j}$$

by tuning

$$\rho(x),\ \lambda(x)$$

for given

$$s*,\ d_v,\ d_c$$

Figure 2.9 Two optimization problems
with Gaussian approximation

Chapter 3
Practical application of differential evolution to degree distribution optimization

In this Chapter, we will describe details about how to apply differential evolution to optimize threshold $s*$. Then, we will explain about effect of parameters in differential evolution to find good differential evolution scheme.

3.1 Differential evolution scheme for practical degree distribution optimization

As mentioned in Section 2.1, differential evolution can be used to design ensemble of LDPC codes. Since differential evolution consists of mutation, crossover, and selection, we should define each step to solve the threshold optimization problem.

At first, initial populations are chosen randomly. Since computers manipulate finite length floating point, we choose initial populations uniformly that satisfy the constraints

$$r > h(s,r) \text{ for } 0 < r < \phi(s) \tag{3.1}$$

$$\sum_{i=2}^{d_v} \lambda_i = 1 \tag{3.2}$$

$$\rho_{d_c-1}+\rho_{d_c}=1. \tag{3.3}$$

$$\lambda_2<\lambda_2^*=e^{1/2\sigma_n^2}/\prod_{j=2}^{d_c}(j-1)^{\rho_j} \tag{3.4}$$

The populations at G-generation has following form,

$$x_i^G=\begin{bmatrix}\rho_{d_c-1,i}^G\\ \rho_{d_c-1,i}^G\\ \lambda_{2,i}^G\\ \vdots\\ \lambda_{d_v,i}^G\end{bmatrix}.$$

In mutation process, the control parameter F should be 1 to satisfy (3.2) and (3.3). Thus, we use following mutation process

$$v_i^G=x_{best}^{G-1}+\sum_{k=1}^{D}(x_{r_1,k}^{G-1}-x_{r_2,k}^{G-1}).$$

There are some issues of implementing threshold optimizer with differential evolution. First, the resolution of $\phi(x)$ and $\phi^{-1}(x)$ should be carefully determined. Since the function $\phi(x)$ and $\phi^{-1}(x)$ affect calculating threshold and checking (3.1), an error introduced from the resolution of $\phi(x)$ and $\phi^{-1}(x)$ cause wrong checking result or calculated threshold. It is very critical to this scheme. Second, r should be appropriately quantized for checking (3.1) and threshold calculation. Third, the cumulative error of computer arithmetic cause a population not to satisfy (3.2) and (3.3) after large iteration. It means that the output best population is not degree distribution. Thus, we normalize the sum of $\rho(x)$ and $\lambda(x)$ at each mutation. Finally, the number of iterations should be large enough. And the number of iterations should be increased if size of the parameter space is increased. Increasing size of the parameter space means that the searching space is increased. Thus, required the number of iterations to find good solution should be increased. By simulation, It turned out that the size N only affects the expected best threshold at initial population, and the size D affect the slope of changing best threshold. Since increasing N and D cause high computational complexity, the values of N and D must be carefully selected

3.2 Threshold optimization result

Table 3.1 show that some well-known degree distribution and our optimization result. For optimization, tolerable error of code rate is less than 0.001 and differential evolution scheme is mentioned above. From the table, we can see that optimization is successfully done and the degree distribution is as good as well-known degree distributions.

Table 3.1 Some well-known degree distributions and optimization results

	Richardson [10]				**Differential evolution**			
d_v	4	5	6	8	4	5	6	8
λ_2	0.38354	0.32660	0.33241	0.30013	0.37648	0.32063	0.31265	0.30819
λ_3	0.04237	0.11960	0.24632	0.28395	0.05434	0.04681	0.17269	0.24250
λ_4	0.57409	0.18393	0.11014		0.56918	0.25301	0.10492	0.00601
λ_5		0.36988				0.37955	0.24427	0.01471
λ_6			0.31112				0.16547	0.00148
λ_7								0.00856
λ_8				0.41592				0.41855
ρ_5	0.24123				0.23938			
ρ_6	0.75877	0.78555	0.76611	0.22919	0.76062	0.62913	0.65951	0.16636
ρ_7		0.21445	0.23389	0.77081		0.37087	0.34049	0.83364
R	0.50000	0.50000	0.50839	0.50013	0.49908	0.49906	0.49953	0.49918
$(\frac{E_b}{N_0})^*_{dB}$	0.8736	0.8318	0.7997	0.5778	0.8579	0.8297	0.7836	0.5628

Chapter 4

A new LDPC code design scheme

In this chapter, we propose an iterative simplex algorithm for code rate enhancement that tune $\rho(x)$ and $\lambda(x)$ iteratively for improving code rate. Since we need initial $\rho(x)$ to enhance code rate, we will use some well-known degree distributions to enhance code rate and observe an effect of code rate enhancer for given degree distribution. We propose a new LDPC code design scheme and compare it to optimization result of differential evolution only. Finally, we find some new degree distributions.

4.1 An iterative simplex algorithm for code rate enhancement

We can't solve threshold optimization problem since there is no known representation of $s*$ in terms of $\rho(x)$ and $\lambda(x)$. But the code rate optimization problem is a linear program if one of $\rho(x)$ and $\lambda(x)$ is given. Thus we can think of a scheme that optimizes $s*$ by tuning $\rho(x)$ and $\lambda(x)$ iteratively by using simplex algorithm. This scheme is shown in Figure 4.1. In this scheme, we must select an initial $\rho(x)$ to enhance code rate. We can easily see that the code rate at each iteration will converge as iteration number goes to ∞.

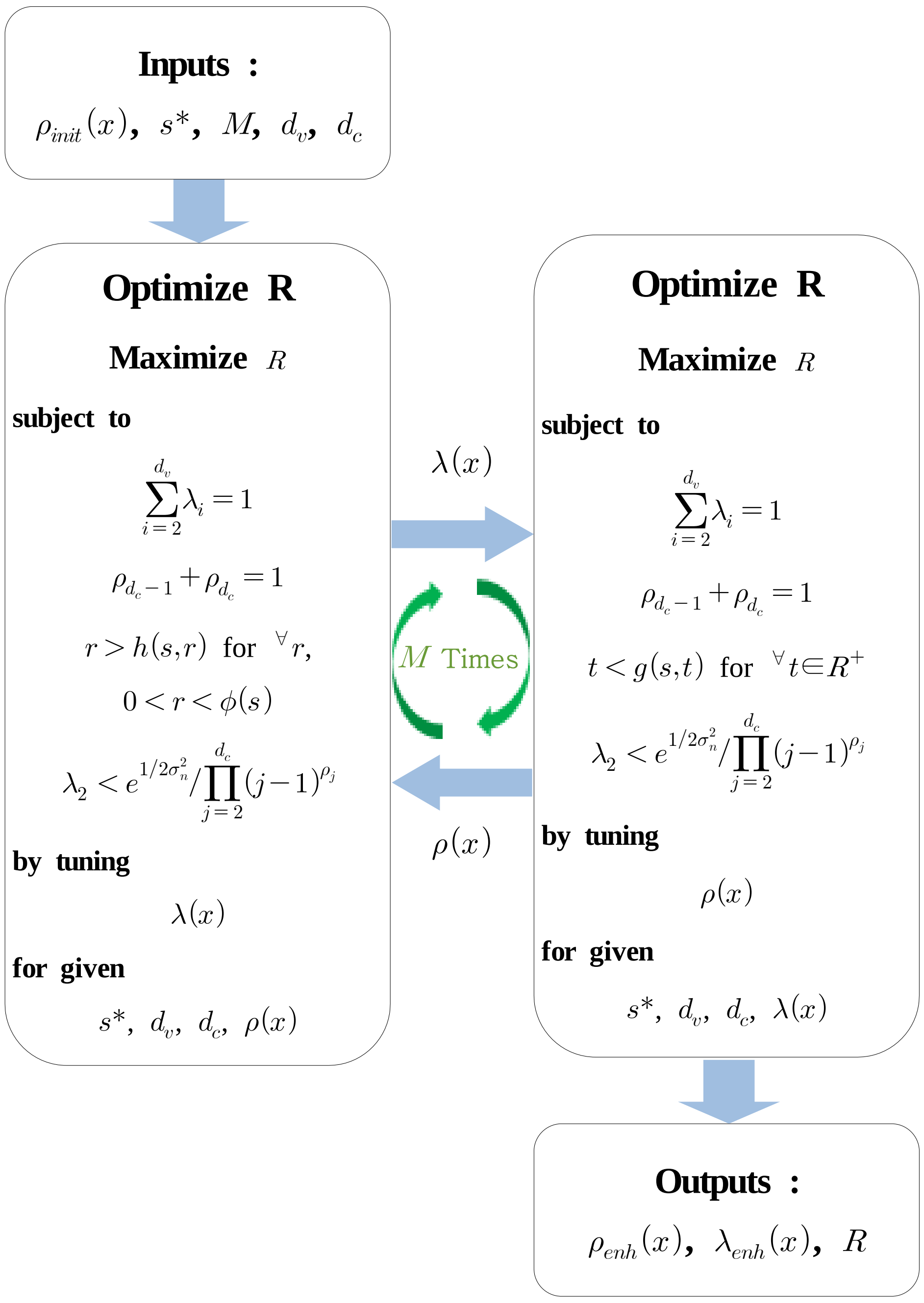

Inputs :
$\rho_{init}(x)$, $s*$, M, d_v, d_c
Optimize R
Maximize R
subject to
$\sum_{i=2}^{d_v}\lambda_i = 1$
$\rho_{d_c-1}+\rho_{d_c}=1$
$r > h(s,r)$ for $\forall r$,
$0<r<\phi(s)$
$\lambda_2 < e^{1/2\sigma_n^2}/\prod_{j=2}^{d_c}(j-1)^{\rho_j}$
by tuning
$\lambda(x)$
for given
$s*$, d_v, d_c, $\rho(x)$
$\lambda(x)$
M Times
$\rho(x)$
Optimize R
Maximize R
subject to
$\sum_{i=2}^{d_v}\lambda_i = 1$
$\rho_{d_c-1}+\rho_{d_c}=1$
$t < g(s,t)$ for $\forall t \in R^+$
$\lambda_2 < e^{1/2\sigma_n^2}/\prod_{j=2}^{d_c}(j-1)^{\rho_j}$
by tuning
$\rho(x)$
for given
$s*$, d_v, d_c, $\lambda(x)$
Outputs :
$\rho_{enh}(x)$, $\lambda_{enh}(x)$, R

Figure 4.1 An iterative simplex algorithm for code rate enhancement

Theorem 4.1. Let the linear program solver of the two optimization process in Figure 4.1 be simplex algorithm. Then, the code rate will converge as iteration number goes to ∞ because the solution of simplex algorithm is global optimum.

proof)

Let $\lambda^{(k)}(x)$, $\rho^{(l)}(x)$ be an optimization result for given $\rho^{(l-1)}(x)$ (or $\lambda^{(k-1)}(x)$). Then, by iterative optimization, the $\lambda^{(k)}(x)$, $\rho^{(l)}(x)$ pairs are given as Figure 4.2.

$$\begin{tikzcd}
\rho^{(i)}(x),\ \lambda^{(i)}(x) \arrow[r] & \rho^{(i)}(x),\ \lambda^{(i+1)}(x) \arrow[dl] \\
\rho^{(i+1)}(x),\ \lambda^{(i+1)}(x) \arrow[r] & \rho^{(i+1)}(x),\ \lambda^{(i+2)}(x) \arrow[dl] \\
\vdots \arrow[r] & \vdots
\end{tikzcd}$$

Figure 4.2 Optimization process for given $\rho^{(l)}(x)$ or $\lambda^{(k)}(x)$

Let

$$R_{i,j} = 1 - \frac{\sum_{n=2}^{d_c} \frac{\rho_n^{(i)}}{n}}{\sum_{m=2}^{d_v} \frac{\lambda_m^{(j)}}{m}}$$

be the intermediate code rate that represented in terms of $\rho^{(i)}(x)$ and $\lambda^{(j)}(x)$. For given $\rho^{(i)}(x)$, tuned $\lambda^{(i+1)}(x)$ is the best choice since the solution of simplex algorithm is global optimum. But, if we want to tune $\rho^{(i+1)}(x)$ for $\lambda^{(i+1)}(x)$, there may exist another choice better than $\rho^{(i)}(x)$. If there is no choice better than $\rho^{(i)}(x)$, then $\rho^{(i)}(x)$ is the best choice. It means that

$$R_{i,i+1} \geq R_{i,i} \geq R_{i-1,i}.$$

Above statement means that the sequence of $R_{i,j}$ is non-decreasing. Since the code rate is upper bounded by 1 and the sequence of $R_{i,j}$ is non-decreasing, $R_{i,j}$ will converge. **(End of proof)**

By computer simulation, if E_s/N_0 is small enough (0 to 1.2), then $R_{i,j}$ will converge with few iterations (less than 10). We denote the converged code rate in the above iterative simplex algorithm for code rate enhancement scheme by R_{enh}.

An implementation of the iterative simplex algorithm for code rate enhancement has some problems. First, since we can't guarantee that R_{enh} is local or global optimum. Thus, the choice of $\rho_{init}(x)$ is very important to obtain code rate that close to local or global optimum. Second, r and t should be quantized to make linear program. Thus, we should carefully choose the quantization size of r and t.

4.2 Application and verification of the iterative simplex algorithm

As mentioned in Section 4.1, we should choose $\rho_{init}(x)$ carefully to use proposed code rate enhancement scheme. For the performance of code rate enhancement, we choose $\rho_{init}(x)$ in some well-known degree distribution and enhance its code rate with the iterative simplex algorithm for code rate enhancement. Table 4.1 shows some well-known degree distributions and its results of enhancing code rate. From Table 4.1, we can see that threshold $(E_b/N_0)^*_{dB}$ is become smaller than well-known degree distribution by the effect of increased code rate.

Table 4.1 Some well-known degree distributions
and its code rate enhancement results

	Richardson [10]				**Results of code rate enhancement**			
d_v	4	5	6	8	4	5	6	8
λ_2	0.38354	0.32660	0.33241	0.30013	0.38720	0.35140	0.35206	0.30681
λ_3	0.04237	0.11960	0.24632	0.28395	0.03290	0.15408	0.27317	0.26644
λ_4	0.57409	0.18393	0.11014		0.57990			
λ_5		0.36988				0.49450		0.00819
λ_6			0.31112				0.37477	
λ_7								0.03787
λ_8				0.41592				0.38069
ρ_5	0.24123				0.24123			
ρ_6	0.75877	0.78555	0.76611	0.22919	0.75877	0.78555	0.76611	0.22919
ρ_7		0.21445	0.23389	0.77081		0.21445	0.23389	0.77081
R	0.50000	0.50000	0.50839	0.50013	0.50021	0.50436	0.51116	0.50038
$(\frac{E_b}{N_0})^*_{dB}$	0.8736	0.8318	0.7997	0.5778	0.8727	0.7943	0.7760	0.5756
$(\frac{E_s}{N_0})^*$	0.6114	0.6056	0.6112	0.5713	0.6114	0.6056	0.6112	0.5713

4.3 A new LDPC code design scheme combining differential evolution and iterative simplex algorithm

From the result of Section 4.2, we can observe that threshold E_b/N_0 will be decreased and code rate R will be increased via the iterative simplex algorithm for code rate enhancement. Thus we can think of a new LDPC code design scheme in Figure 4.4.

At the first stage, we find a good $\rho(x)$, $\lambda(x)$ pair, that has low $s*$, with differential evolution. Then, we can optimize code rate by using the iterative simplex algorithm for code rate enhancement presented in Section 4.1. The result of code rate enhancement is $\rho_{enh}(x)$ and $\lambda_{enh}(x)$ that has enhanced code rate with same E_s/N_0. By the effect of enhanced code rate with same $s*$, the $(E_b/N_0)*$ will be decreased.

In the proposed new code design scheme, differential evolution is used to find a good $\rho(x)$ and $s*$ and the iterative simplex algorithm for code rate enhancement is used to enhance code rate for given $\rho_{init}(x)$ and $s*$. The optimized result is compared with some well-known degree distributions and differential evolution only in Tables 4.2, 4.3, 4.4, 4.5. In these tables, the results of differential evolution only are the results of pre-processing in the iterative simplex algorithm for code rate enhancement. By the effect of increased code rate, threshold $(E_b/N_0)^*_{dB}$ becomes smaller than the result of differential evolution only.

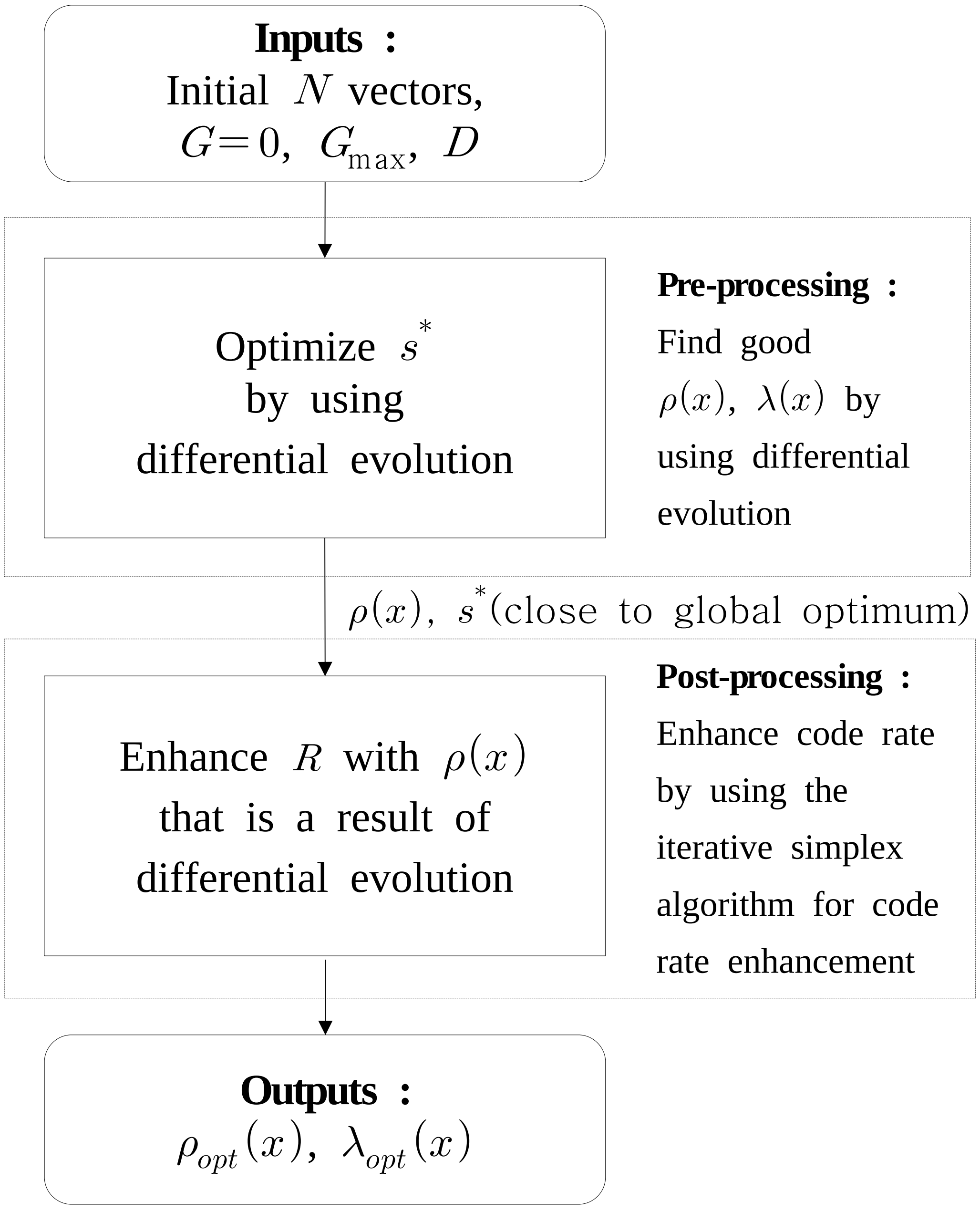


Figure 4.4 A new LDPC code design scheme

Table 4.2 Degree distributions for d_v= 4, 5.

	Richardson [10]		**Differential evolution only**		**Proposed**	
d_v	4	5	4	5	4	5
λ_2	0.38354	0.32660	0.37648	0.32063	0.38586	0.34065
λ_3	0.04237	0.11960	0.05434	0.04681	0.02148	0.11491
λ_4	0.57409	0.18393	0.56918	0.25301	0.59266	
λ_5		0.36988		0.37955		0.54443
ρ_5	0.24123		0.23938		0.23938	
ρ_6	0.75877	0.78555	0.76062	0.62913	0.76062	0.62913
ρ_7		0.21445		0.37087		0.37087
R	0.50000	0.50000	0.49908	0.49906	0.49851	0.50291
$(\frac{E_b}{N_0})^*_{dB}$	0.8736	0.8318	0.8579	0.8297	0.8629	0.7960

Table 4.3 Degree distributions for d_v= 6, 8.

	Richardson [10]		**Differential evolution only**		**Proposed**	
d_v	6	8	6	8	6	8
λ_2	0.33241	0.30013	0.31265	0.30819	0.34061	0.30259
λ_3	0.24632	0.28395	0.17269	0.24250	0.24400	0.26274
λ_4	0.11014		0.10492	0.00601		
λ_5			0.24427	0.01471		
λ_6	0.31112		0.16547	0.00148	0.41539	
λ_7				0.00856		
λ_8		0.41592		0.41855		0.43468
ρ_6	0.76611	0.22919	0.65951	0.54864	0.65951	0.16636
ρ_7	0.23389	0.77081	0.34049	0.45136	0.34049	0.83364
R	0.50839	0.50013	0.49953	0.49918	0.50584	0.49927
$(\frac{E_b}{N_0})^*_{dB}$	0.7997	0.5778	0.7836	0.5628	0.7288	0.5619

Table 4.4 Degree distributions for d_v= 9, 10.

	Richardson [10]		**Differential evolution only**		**Proposed**	
d_v	9	10	9	10	9	10
λ_2	0.27684	0.25105	0.28683	0.28408	0.28432	0.28413
λ_3	0.28342	0.30938	0.24431	0.23683	0.24548	0.22737
λ_4		0.00104	0.01290	0.01003	0.03260	0.01930
λ_5			0.01349	0.01460		0.09475
λ_6			0.00009	0.05226		
λ_7			0.01384	0.00484		
λ_8			0.01169	0.09040		
λ_9	0.43974		0.416867	0.05946		
λ_{10}		0.43853		0.24750	0.43759	0.37446
ρ_6	0.01568					
ρ_7	0.85244	0.63676	0.85899	0.80889	0.85900	0.80939
ρ_8	0.13188	0.36324	0.14101	0.19111	0.14100	0.22737
R	0.50013	0.50000	0.49975	0.49925	0.50015	0.500299
$(\frac{E_b}{N_0})^*_{dB}$	0.5498	0.5456	0.5392	0.5351	0.5358	0.5260

Table 4.5 Degree distributions for d_v= 12, 15.

	Richardson [10]		**Differential evolution only**		**Proposed**	
d_v	12	15	12	15	12	15
λ_2	0.24426	0.23802	0.25903	0.24454	0.25120	0.25039
λ_3	0.25907	0.20997	0.17584	0.23239	0.18200	0.19942
λ_4	0.01054	0.03492	0.04345	0.00996	0.10740	
λ_5	0.05510	0.12015	0.03463	0.00243		0.06000
λ_6			0.00092	0.00342		0.15268
λ_7		0.01587	0.01625	0.04248		
λ_8	0.01455		0.00164	0.02307		
λ_9			0.03947	0.04856		
λ_{10}	0.01275		0.02230	0.02322		
λ_{11}			0.06518	0.10635		
λ_{12}	0.40373		0.34130	0.03850	0.45944	
λ_{13}				0.13364		
λ_{14}		0.00480		0.00144		
λ_{15}		0.37627		0.09000		0.33752
ρ_7	0.25475					
ρ_8	0.73438	0.98013	0.99303	0.97786	0.99294	0.97786
ρ_9	0.01087	0.01987	0.00697	0.02214	0.00707	0.02214
R	0.50016	0.50001	0.49955	0.49965	0.50316	0.50443
$(\frac{E_b}{N_0})^*_{dB}$	0.5355	0.5287	0.5514	0.5288	0.5202	0.4874

Table 4.6 New degree distributions from the proposed algorithm for d_v= 7, 13, 14.

d_v	7	13	14
λ_2	0.32721	0.25085	0.24250
λ_3	0.27415	0.15928	0.14597
λ_4		0.13275	
λ_5		0.03823	0.14764
λ_6			
λ_7	0.39864		
λ_{13}		0.41889	
λ_{14}			0.42869
ρ_6	0.52918		
ρ_7	0.47082		
ρ_8		0.97248	0.76507
ρ_9		0.02752	0.23493
R	0.50164	0.50465	0.50205
$(\frac{E_b}{N_0})^*_{dB}$	0.6473	0.5055	0.4910

Also, we find optimized degree distributions with $d_v = 7, 13, 14$. Table 4.6 shows these results. We find good degree distributions for $d_v = 7$, $d_v = 13$, and $d_v = 14$. We can see that the optimized results are good by comparing well-known degree distributions that have similar maximum variable node degrees.

Chapter 5

Conclusion

5.1 Summary

In this thesis, we describe a way to find good degree distribution by using differential evolution and an iterative simplex algorithm.

We discuss some implementation issues of threshold optimizer with differential evolution.

We propose an iterative simplex algorithm for code rate enhancement that tunes $\rho(x)$ and $\lambda(x)$ iteratively, and a new LDPC code design scheme, that uses differential evolution for finding good $\rho(x)$, $s*$ and use an iterative simplex algorithm for code rate enhancement scheme to enhance code rate.

We compare the optimization results of proposed LDPC code design scheme with the optimization results of differential evolution only and some well-known degree distributions. The optimized result of proposed LDPC code design has better $(E_b/N_0)*$ than those obtained by differential evolution only.

5.2 Future work

In the future research, the following problems are desired to be studied.

- For finding good $\rho(x)$ and $s*$, is there any other way that is better than Differential evolution?
- How many generations in Differential evolution only scheme are required to obtain optimized degree distribution as good as proposed scheme? And what scheme requires low computational complexity?

Bibliography

[1] R. G. Galleger, "Low-density parity-check codes," *IRE Trans. Information Theory*, vol IT-8, pp. 21-28, Jan. 1962.

[2] R. G. Galleger, Low-Density Parity-Check Codes. Cambridge, MA:MIT Press, 1963.

[3] M. Sipser and D. A. Spielman, "Expander codes," IEEE Trans. Information Theory, vol. 42, pp. 1710-1722, Nov. 1996.

[4] D. J. C. MacKay and R. M. Neal, "Near Shannon limit performance of low-density parity-check codes," *Electron. Letter*, vol. 32, pp. 1645-1646, Aug. 1996.

[5] D. J. C. MacKay, "Good error-correcting codes based on very sparse matrices," *IEEE Trans. Information Theory*, vol. 45, pp. 399-431, Mar. 1999.

[6] M. Luby, M. Mitzenmacher, A. Shokrollahi, and D. Spielman, "Analysis of low density codes and improved designs using irregular graphs," *in Proc. 30th Annu. ACM Symp. Theory of Computing*, pp. 249-258, 1998.

[7] T. J. Richardson and R. Urbanke, "The capacity of low-density parity-check codes under message-passing decoding," *IEEE Trans. Information Theory*, vol. 47, pp. 599-618, Feb. 2001.

[8] Sae-Young Chung, Thomas J. Richardson, and Rüdiger L. Urbanke, "Analysis of Sum-Product Decoding of Low-Density Parity-Check Codes Using a Gaussian Approximation," *IEEE Trans. Information Theory*, vol. 47, pp. 657-670. Feb. 2001.

[9] M. Luby, M. Mitzenmacher, A. Shokrollahi, D. Speilman, and V. stemann, "Practical loss-resilient codes," *in Proc. 29th annu. ACM Symp. Theory of Computing*, pp. 150-159, 1997.

[10] T. J. Richardson, A. Shokrollahi, and R. Urbanke, "Design of capacity-approaching Irregular low-density parity-check codes," *IEEE Trans. Information Theory*, vol. 47, pp. 619-637, Feb. 2001.

[11] R. Storn and K. Price, "Differential evolution - A simple and efficient heuristic for global optimization over continuous spaces," *Journal of Global Optimization*, vol. 11, 341-359, 1997.

[12] K. Price, "Differential Evolution : A Fast and Simple Numerical Optimizer," NAFIPS'96, pp. 524-527, 1996.

[13] X. Y. Hu, E. Eleftheriou, and D. M. Arnold, "Regular and irregular progressive edge-growth tanner graphs," *IEEE Trans. Information Theory*, vol. 51, pp. 386-398, 2005

[14] D Vukobratovic, A Djurendic, V. Senk, "ACE Spectrum of LDPC Codes and Generalized ACE Design," *IEEE International Conference on Communications*, pp. 665-670, 2007.

국문요약

Differential evolution과 Simplex 알고리듬을 이용한 새로운 LDPC 부호 설계 방법

본 논문에서는 differential evolution과 반복적인 simplex 알고리듬을 혼합한 새로운 LDPC 부호 설계 방법을 제시한다. 이 새로운 방법은, differential evolution을 이용하여 좋은 임계값을 갖는 체크 노드와 변수 노드의 차수 분포를 구하고, 이를 바탕으로 반복적인 simplex 알고리듬을 이용하여 부호율을 향상시킨다. 이를 위해, differential evolution을 이용한 임계값 최적화의 구현상의 문제에 대해서 다룬다. 그리고 변수 노드와 체크 노드의 차수 분포를 최적화하는 두 simplex 알고리듬으로 구성된 반복적인 simplex 알고리듬을 다룬다.

모의 실험 결과를 통하여, 제안하는 새로운 LDPC 부호 설계 방법이 잘 알려진 차수 분포보다 좋은 결과를 얻을 수 있음을 확인하였다. 또한 제안하는 새로운 LDPC 부호 설계 방법을 이용하여 몇 가지 새로운 차수 분포를 찾았다.

핵심되는 말 : LDPC, 부호 설계, simplex 알고리듬, differential evolution, 차수 분포 최적화