변형된 윈도우 복호를 이용한 SC-LDPC 코드와 QC-LDPC 코드의 성능 비교

하유진, 김강산, 채상원, 송홍엽

연세대학교

{eugeneha1, gs.kim, sw.chae, hysong}@yonsei.ac.kr

Performance Comparison of SC-LDPC Codes and QC-LDPC Codes with Modified Window Decoding Scheduling

Eugene Ha, Gangsan Kim, Sangwon Chae, and Hong-Yeop Song

Yonsei Univ.

요 약

본 논문은 SC-LDPC 코드의 윈도우 복호 방식의 변형에 따른 성능 및 복잡도의 변화를 비교하고 이에 대한 알고리즘을 제시한다. 그 중 가장 좋은 성능을 보인 윈도우 복호 방식으로 복호된 SC-LDPC 코드와 BP flooding 방식으로 복호된 QC-LDPC 코드의 성능을 비교해 보았다.

Ⅰ. 서 론

5G 이동통신의 표준 오류 정정 부호로 사용되는 Low-density parity-check(LDPC) 코드는 1963년도에 Gallager에 의해 소개되었으며, 이후 약 30년 동안 잊혀 있었다. 하지만 Mackay가 확률적 반복 복호 방식을 적용한 LDPC 코드가 뛰어난 성능을 나타낸다는 사실을 발견하여 그 중요성이 다시 떠오르게 되었다. SC-LDPC 코드는 구조적인 특성을 활용한 윈도우 복호를 통해 낮은 복잡도로 복호를 수행할 수 있다. 하지만 윈도우 복호 방식에 따라 성능 및 복잡도의 차이가 발생하는 특징이 있다. 본 논문은 윈도우 복호 방식의 차이가 성능 및 복잡도에 미치는 영향을 비교하고, 가장 좋은 성능을 보인 윈도우 복호 방식을 이용하여 QC-LDPC와의 비교를 진행한다.

Ⅱ. 본 론

가. SC-LDPC

Spatially coupled LDPC(SC-LDPC)는 프로토그래프(protograph)를 연달아 합성하여 설계되며, 이웃한 프로토그래프의 엣지 간의 연결을 통해 복호 과정에서 공간 결합적인 의존성을 부여한다. 행렬의 관점으로는 프로토그래프를 $n_c \times n_v$ 크기의 구성 행렬(component matrix) C_i로 이루어진 기반 행렬(base matrix) $B = [C_0; \cdots; C_w]$로 나타낼 수 있다. 이후 기반 행렬을 복제하여 패리티 검사 행렬 H의 대각 방향을 따라 위치시키고, 각 원소를 $z \times z$ 크기의 순환치환행렬(circulant permutation matrix)로 리프팅하여 H를 생성할 수 있다. 이때 결합 길이 L은 기반 행렬이 복제된 개수를 의미하며, 결합 폭 w는 이웃한 기반 행렬 사이에서 동일한 체크 노드를 공유하는 구성 행렬 C_i의 개수를 의미한다.

오류 정정 부호는 확률적 반복 복호 방식을 통해 Shannon limit에 근접한 성능을 얻을 수 있으며, message-passing 알고리즘을 통해 복호기를 구현할 수 있다. Belief-propagation(BP) 복호는 log-likelihood ratio(LLR)을 이용하여 soft decision을 수행하는 확률적 반복 복호 방식이다. BP 복호는 다양한 스케줄링 방법을 통해 구현할 수 있으며, 특히 SC-LDPC 코드는 동일한 기반 행렬이 대각 방향으로 복제되어 형성되는 구조적인 특성을 활용하여 윈도우 복호(Window decoding)를 사용한다. 윈도우 복호는 윈도우 내부에 Wn_cz개의 체크 노드와 연결된 변수 노드들이 모두 포함되도록 $Wn_cz \times (w+W)n_vz$ 크기의 윈도우 영역을 설정하며, 영역 내부에 있는 노드들 사이에서만 반복적인 BP 복호가 수행된다. 그림 1에서와 같이 각 윈도우에서 목표 변수 노드가 복호되며 윈도우가 대각 방향으로 이동하면서 전체 노드에 대한 복호가 이루어진다.

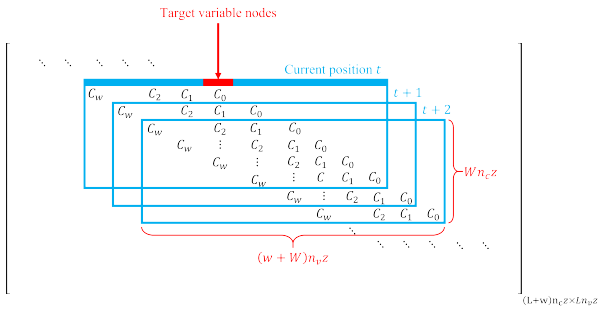


그림 1. 윈도우 복호 과정

나. 윈도우 복호 변형

본 논문에서 사용한 SC-LDPC 코드는 기반 행렬 $B = [1\,1; 1\,1; 1\,1]$, $n_c \times n_v = 1 \times 2$, $L = 50$, $z = 100$으로 생성한 코드 길이 $N = 10000$, 부호율 $R = 0.48$의 패리티 검사 행렬을 사용했으며, 최대 반복 횟수 $I_{\max} = 50$으로 복호를 진행했다.

실험에서 사용된 윈도우 복호는 크게 두 부분이 변형되었다. 먼저 윈도우 복호를 위해 구현한 알고리즘 1의 3번째 줄에서는 마지막 윈도우 위치를 일반적으로 사용되는 $L - W + 1$이 아닌 $L - W + w$로 설정했다. 마지막 위치에서의 윈도우 크기가 줄어들어 복잡도가 개선되는 효과를 기대했다. 다음으로 알고리즘 1의 12번째 줄에서는 현재 윈도우에서 패리티 검사를 만족한 경우 다음 윈도우에서 사용될 메시지에 비선형 활성화 함수(activation function)를 적용했다. 실험에서 사용한 함수는 다음과 같다.

$$\text{Rectifier:}\quad g(x)=\begin{cases}x & (H\underline{v}^T=\underline{0})\\ 0 & (H\underline{v}^T\neq\underline{0})\end{cases} \tag{1}$$

$$\text{Scaling:}\quad g(x)=\begin{cases}2x & (H\underline{v}^T=\underline{0})\\ x & (H\underline{v}^T\neq\underline{0})\end{cases} \tag{2}$$

$$\text{Cube:}\quad g(x)=\begin{cases}x^3 & (H\underline{v}^T=\underline{0})\\ x & (H\underline{v}^T\neq\underline{0})\end{cases} \tag{3}$$

수식 (1)은 패리티 검사를 만족하지 못한 메시지들이 다음 윈도우에서 부정적인 영향을 주는 것을 차단한다. 수식 (2), (3)은 패리티 검사를 만족했을 때 메시지 값을 증폭시키는 피드백을 준다. 수식 (2)는 [1]에 제시된 방법이며 참조 논문에서는 오류 마루가 개선되는 효과를 보였다.

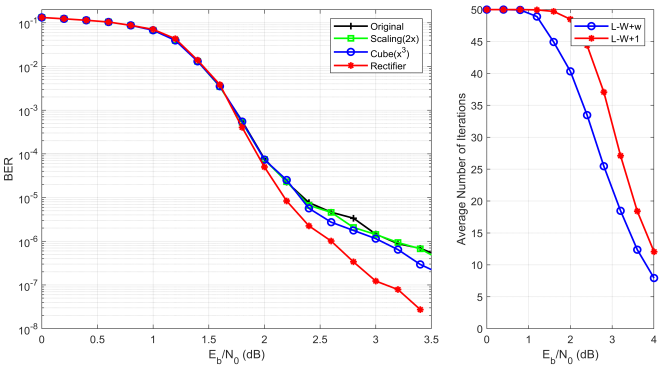
Algorithm 1 Window Decoding

Inputs: L, W, R_i, $I_{\max}$

// LLR of received signals R_i
Every VNs ← R_i
// Window Decoding start
current position ← 0
while current position ≤ $L-W+w$ **do**
  // Initialization phase
  Calculate window arguments
  Load window VNs, CNs, edge messages
  // Iterative decoding
  while $I < I_{\max}$ or $H\underline{v}^T=\underline{0}$ **do**
    Variable nodes to Check nodes :
    $M_{i,j}=\sum_{j'\in A,j'\neq j}E_{j',i}+R_i$
    Check nodes to Variable nodes :
    $E_{j,i}=\log\frac{1+\prod_{i'\notin B_j,i'\neq i}\tanh(M_{j,i'}/2)}{1-\prod_{i'\notin B_j,i'\neq i}\tanh(M_{j,i'}/2)}$
    Hard decision :
    $L_i=\sum_{j\in A_i}E_{j,i}+R_i$
    $v_i=\begin{cases}0 & (L_i\leq 0)\\ 1 & (L_i>0)\end{cases}$
    Test $H\underline{v}^T=\underline{0}$
  end while
  // Keep decoding results for the next position
  Apply $g(x)$ to edge messages according to $H\underline{v}^T$
  Update target VN's L_i and v_i
  current position ← current position + 1
end while

알고리즘 1. 변형된 윈도우 복호의 의사코드

$g(x)$에 따른 복호 성능은 그림 2-(a)와 같이 나타났다. 세 함수 모두 오류 마루 영역에서 기존의 윈도우 복호와 차이를 보이며, 수식 (1)을 사용한 결과에서 오류 마루가 가장 개선되는 것을 볼 수 있다. 또한 그림 2-(b)와 같이 $L-W+w$까지 진행한 복호에서 복잡도가 감소하였다.



(a) (b)

그림 2. $g(x)$에 따른 BER 및 마지막 윈도우 위치에 따른 평균 반복 횟수

다. SC-LDPC와 QC-LDPC의 성능 비교

다음으로는 SC-LDPC와 QC-LDPC의 성능을 비교하는 실험을 진행했다. 동일한 지연시간을 기준으로 코드의 BER 성능을 비교하였으며, 지연시간은 [2]에 제시된 계산식을 이용했다.

QC-LDPC는 Golomb Ruler $(0,1,4,10,12,17)$ [3]을 사용하여 3×6 exponent 행렬을 $z=200$으로 리프팅한 패리티 검사 행렬을 사용했으며, 지연시간 $\Lambda_{QC}=nz=1200$의 BP 복호를 $I_{\max}=20$으로 진행했다.

SC-LDPC는 $B=[1\,1;1\,1;1\,1]$ 기반 행렬과 $L=50$을 사용했으며, QC-LDPC와 동일한 지연시간을 갖도록 $\Lambda_{SC}=Wn_v z=1200$을 만족하는 윈도우 크기 W와 리프팅 계수 z를 선택했다. SC-LDPC의 복호는 그림 2-(a)에서 가장 좋은 개선을 보인 Rectifier 함수를 이용하여 윈도우 복호를 마지막 윈도우 위치 $L-W+w$까지 $I_{\max}=50$으로 진행했다.

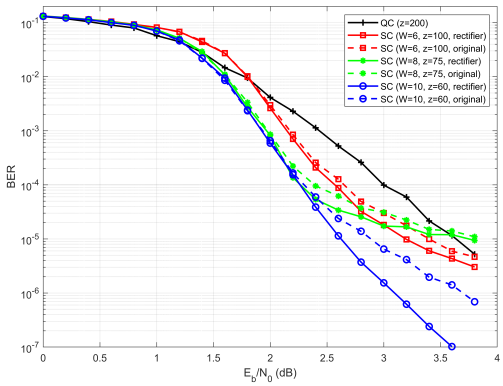


그림 3. SC-LDPC와 QC-LDPC의 BER 성능

그림 3에서 동일한 지연시간에 대해 QC-LDPC보다 SC-LDPC의 성능이 대체로 좋게 나타났으며, Rectifier 함수를 사용했을 때 오류 마루가 개선 되었다. 특히 윈도우 크기와 리프팅 계수에 따라 Rectifier 함수의 오류 마루의 개선 효과가 달라지는 것으로 나타났다.

Ⅲ. 결 론

본 논문에서는 SC-LDPC의 윈도우 복호의 마지막 윈도우 위치에 따른 복잡도와, 다음 윈도우에서 사용될 메시지에 적용하는 비선형 함수의 종류에 따른 오류 마루의 변화를 비교했다. Rectifier 함수를 사용한 윈도우 복호에서 오류 마루 영역의 큰 성능 개선을 보였으며, QC-LDPC와의 성능 비교에서도 동일한 지연시간에서 더 높은 성능을 확인할 수 있었다.

ACKNOWLEDGMENT

이 (성과)는 정부(과학기술정보통신부)의 재원으로 한국연구재단의 지원을 받아 수행된 연구임(No.RS-2023-00209000).

참 고 문 헌

[1] I. Ali, J. -H. Kim, S. -H. Kim, H. Kwak, and J. -S. No, "Improving Windowed Decoding of SC LDPC Codes by Effective Decoding Termination, Message Reuse, and Amplification," *IEEE Access*, vol. 6, pp. 9336-9346, 2018.

[2] M. Lentmaier, M. M. Prenda, and G. P. Fettweis, "Efficient message passing scheduling for terminated LDPC convolutional codes," in *Proc. 2011 IEEE International Symposium on Information Theory*, St. Petersburg, Russia, pp. 1826-1830, 2011.

[3] W. -J. Kim, H. -W. Cho, H. -Y. Song, and M. -K. Song, "Some Variations of Tanner's Construction for Short Length QC-LDPC Codes," *IET Electronics Letters*, vol. 60, no. 3, pp. e13088, January 2024.