Detailed explanation of 5G NR Polar design and implementation of CA-SCL decoder

Jie Liu hahaliu2001@gmail.com

1 Introduction

Polar code was invented by Turkish professor Erdal Arikan in 2009. It quickly became popular due to its superior performance, theoretically reaching the Shannon limit, and simple encoding and decoding algorithms. 3GPP started the discussion of introducing Polar code into 5G specification in 2016(as far as what I found in 3GPP TDoc) and finally chosen Polar code to replace traditional convolutional code and is used for DCI, UCI and BCH control channel encoding. LDPC code was chosen for PDSCH/PUSCH channel coding.

Some new features are added into 5G specification Polar code comparing with Polar code invented by Professor Arikan.

The first part of the paper is to descript 5G polar new features in detail, such as Distributed CRC which is used for Polar decoding early termination, Channel interleaver in Polar rate matching.

The second part of the paper is about 5G Polar decoder which includes:

1. One N=8 polar SC decoder example to step-by-step show decoding procedure
2. Two SC decoder implementation
    a. One is to follow SC algorithm in "List decoding of polar codes"
    b. Another is to follow decoding procedure used for N=8 polar SC decoder example

    Both implementations give the same result.
3. Two SC decoder memory optimization to save LLR and B memory from N*log2(N) to N
4. 5G NR LLR based SCL decoder
    a. Support Distributed CRC based early termination
    b. Support parity bit check based early termination
    c. Support CRC-aided SCL decoder

I have finished 5G Polar encoder and SC, SCL decoder coding and testing. The code is in:

https://github.com/hahaliu2001/python_5gtoolbox.git : py5gphy/polar

To be clear, this paper is focused on Polar implementation and is not involved to Polar theory and performance analysis.

1.1 reference

【1】 polar 码基本原理 v1 :

https://github.com/luxinjin/polar-code/blob/master/polar%E7%A0%81%E5%9F%BA%E6%9C%AC%E5%8E%9F%E7%90%86v1.docx

【2】 Valerio Bioglio; Carlo Condo; Ingmar Land “Design of Polar Codes in 5G New Radio” IEEE Communications Surveys & Tutorials (Volume: 23, Issue: 1, Firstquarter 2021)

【3】 I. Tal and A. Vardy, “List decoding of polar codes,” in Proceedings of 2011 IEEE International Symposium on Information Theory (ISIT), Jul. 2011, pp. 1–5.

【4】 A. Balatsoukas-Stimming, M. Bastani Parizi and A. Burg, “LLR-based

successive cancellation list decoding of polar codes,”arXiv:1401.3753v3

【5】 Kai Niu1,3,*, Ping Zhang2, Jincheng Dai1, Zhongwei Si1, Chao Dong “A Golden Decade of Polar Codes: From Basic Principle to 5G Applications” https://arxiv.org/abs/2303.14614

【6】 3GPP R1-1708833, “Design details of Distributed CRC”, Nokia, Alcatel-Lucent

【7】 R1-164039, Huawei, HiSilicon, “Polar codes – encoding and decoding,” 3GPP TSG RAN WG1 #85, Nanjing, China, 23-27 May 2016.

【8】 R1-1611254, Huawei, HiSilicon, “Details of the polar code design,” 3GPP TSG RAN WG1 #87, Reno, NV, 14-18 Nov. 2016.

【9】 R1-1705861, “Design details of distributed CRC,” Nokia, Alcatel-Lucent Shanghai Bell.

【10】 R1-1700979 “Discussion on CA-Polar and PC-Polar Codes” , Samsung

【11】 R1-1708047 “Early Termination of Polar Decoding” Samsung

【12】 E. Arıkan, “Channel Polarization: A Method for Constructing Capacity-Achieving Codes for Symmetric Binary-Input Memoryless Channels”, IEEE Transactions on Information Theory, vol. 55, no. 7, July 2009

2 5G NR polar code design

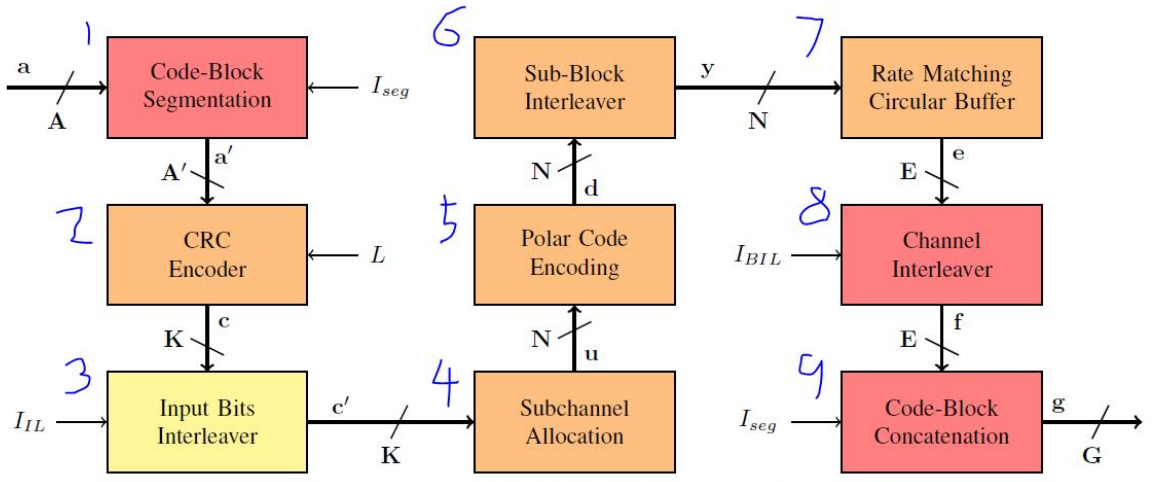


Fig. 6: 5G polar codes encoding chain; yellow, red and orange blocks are implemented in downlink, uplink and both respectively.

Above figure comes from "Design of Polar Codes in 5G New Radio" and shows 5G polar processing from CRC-> polar encoding -> Polar rate match.

This section is to explain in detail a few features used in 5G polar.

2.1 5G Polar interleaver and Distributed CRC design

Step 3 Input Bits interleaver defined at 38.212 5.3.1 is used for DL DCI and BCH only. The purpose is for UE to check CRC bit and early terminate polar decoder processing in the middle of Polar decoding. It can be used by UE during PDCCH searching to save UE power.

This polar interleaver is based on Distributed CRC design and will explain in below.

2.1.1 Distributed CRC example: 4 bit CRC polynomial Distributed CRC

Assume 4 bit CRC polynomial is [0, 1, 0, 1]. For 8 bits information bits, CRC output is the same 8 bits information bits + 4 bit CRC. CRC matrix generator is:

$$P_{1X12} = U_{1X8}(I_{8x8}|G_{8x4})$$

Where :

U_{1X8} is 8 bits information bits

P_{1X12} is 12 bits output

I_{8x8} is 8X8 diagonal matrix

$$G_{8X4} = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{bmatrix}$$ is CRC generation matrix

Each line in G_{8x4} shows how one of information bits is involved in CRC calculation. 1 mean involved, 0 mean not.

For example first line [1 0 1 0] indicate first information bit.

It means CRC calculation for [a,0,0,0,0,0,0,0] vector,

If a = 1, CRC bit 0 and bit2 = 1, CRC bit 1 and bit 3 =0

If a = 0, CRC bit 0 and bit2 = 0, CRC bit 1 and bit 3 =0

CRC bit 1 and bit 3 are always zero no matter what 'a' value is.

Similarly second line [0,1,0,1] indicate second information bit, and CRC bit0 and bit2 are always 0 .

Let's change location of '1' in G_{8x4} to information bit index.

$$Gindex_{8X4} = \begin{bmatrix} 0 & -1 & 0 & -1 \\ -1 & 1 & -1 & 1 \\ 2 & -1 & -1 & -1 \\ -1 & 3 & -1 & -1 \\ -1 & -1 & 4 & -1 \\ -1 & -1 & -1 & 5 \\ 6 & -1 & 6 & -1 \\ -1 & 7 & -1 & 7 \end{bmatrix}$$

In each line, -1 means CRC bits is always zero for correspondent information bits. 0-7 means information bit index.

Each column shows which information bits are used for current CRC bit calculation.

For example first column [0, -1,2,-1,-1,-1,6,-1] means information bits [0,2,6] are used for CRC bit0 calculation.

Similarly

Information bit [1,3,7] are used for CRC bit 1 calculation.

Information bit [0,4,6] are used for CRC bit 2 calculation.

Information bit [1,5,7] are used for CRC bit 3 calculation.

In this way, if bit sequence that inputs to polar encoder is [0,2,6,CRC0,1,3,7,CRC1,4,CRC2,5,CRC3], When first 4 bits are decoded in Polar decoder, it can calculate CRC0 with first three decoded bit. If CRC0 doesn't match, it can early terminate Polar decoder. This is very helpful for UE blind searching DL DCI.

Polar interleaver is to indicate how to generate [0,2,6,CRC0,1,3,7,CRC1,4,CRC2,5,CRC3] kind of sequence.

Above example is for 8 bits information bits. Same CRC generation matrix can be used for less than 8 information bits case.

For example for 7 information bits, last 7 lines of $Gindex_{8X4}$ are used. To generate 7 bits sequence:

1. Remove 0 from 8 bits sequence [0,2,6,CRC0,1,3,7,CRC1,4,CRC2,5,CRC3] to get [2,6,CRC0,1,3,7,CRC1,4,CRC2,5,CRC3]
2. All information index - 1 to get [1,5,CRC0,0,2,6,CRC1,3,CRC2,4,CRC3]

This method can be used to calculate all less than 8 bits sequence.

It means one 8 bit interleaving table is enough for all <=8 bit cases.

Below python code are used to generate CRC table and CRC index table

```
def gen_crc_interleaver_table(K,crcpoly):
    """ gen interleave table
        crcidx_matrix: CRC index matrix, -1 means not involved
            each column show that CRC input data indexs that are involed for this bit CRC calculation
    """
    L = crcpoly.size  # CRC size
    crcmatrix = np.zeros((K,L),'i2')
    crcidx_matrix = np.zeros((K,L),'i2')
    for n in range(K):
        inbits = np.zeros(K,'i1')
        inbits[n] = 1
        blkandcrc = crc_encode(inbits, crcpoly)
        crc_bits = blkandcrc[-L:]
        crcmatrix[n,:] = crc_bits
        crcidx_matrix[n,:] = crc_bits*(n+1)

    crcidx_matrix = crcidx_matrix -1
    return crcmatrix, crcidx_matrix
```

2.1.2 5G polar interleave design

38.212 Table 5.3.1.1-1: Interleaving pattern is generated based on CRC24C and maximum 140 information bits.

$$g_{\mathrm{CRC24C}}(D)=[D^{24}+D^{23}+D^{21}+D^{20}+D^{17}+D^{15}+D^{13}+D^{12}+D^{8}+D^{4}+D^{2}+D+1]$$

Below python function is to genrate Interleaving pattern

```
def gen_polar_pitable(K, crcidx_matrix):
    """ K is data size, K=140 for 38.212 Table 5.3.1.1-1: Interleaving pattern
    """
    CRC_size =  crcidx_matrix.shape[1]
    pitable = -1*np.ones(CRC_size+K,'i2')
    pos = 0
    for n in range(CRC_size):
        #find crc input bit index that is used to calculate CRC bit n
        d1 = crcidx_matrix[:,n]
        d2 = d1[d1 >= 0] #

        #exclude all values that exist in pitabe
        d3=[v for v in d2 if v not in pitable]
        pitable[pos:pos+len(d3)] = d3 #
        pitable[pos+len(d3)] = n + K #crc index
        pos = pos + len(d3) + 1

    return pitable
```

Run below python code to generate 38.212 Table 5.3.1.1-1: Interleaving pattern, I have verified the code.

```
crcmatrix, crcidx_matrix = gen_crc_interleaver_table(140,np.array([1, 0, 1, 1, 0, 0, 1, 0, 1, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 1, 0, 1, 1,
1],'i1'))
pitable = gen_polar_pitable(140, crcidx_matrix)
```

2.2 Subchannel Allocation

Refer to 38.212 5.3.1.2 Polar encoding.

For N bits size polar encoder input bits, each bit has different reliability. K bits information+CRC bits need allocate into highest K reliabilities location and set 0 for other N-K lowest reliabilities location(Frozen bits).

5G DL support N values in the range of $2^{\wedge[5,6,7,8,9]}$, 5G UL support N values in the range of $2^{\wedge[5,6,7,8,9,10]}$

38.212 Table 5.3.1.2-1: Polar sequence $\mathbf{Q}_0^{N_{\max}-1}$ and its corresponding reliability $W\left(Q_i^{N_{\max}}\right)$

Gives N=1024 reliability sequence.

38.212 5.4.1.1 shows how to generate $\overline{\mathbf{Q}}_I^N$ (information bit location) for any N value

2.3 5G polar encoder is natural order encoder

There are two Polar encode order("A Golden Decade of Polar Codes: From Basic Principle to 5G Applications"2.2),

One is bit-reversal order encoder, where

$$x_1^N = u_1^N \mathbf{G}_N = u_1^N \mathbf{B}_N \mathbf{F}_2^{\otimes n},$$

Another is natural order encoder, where

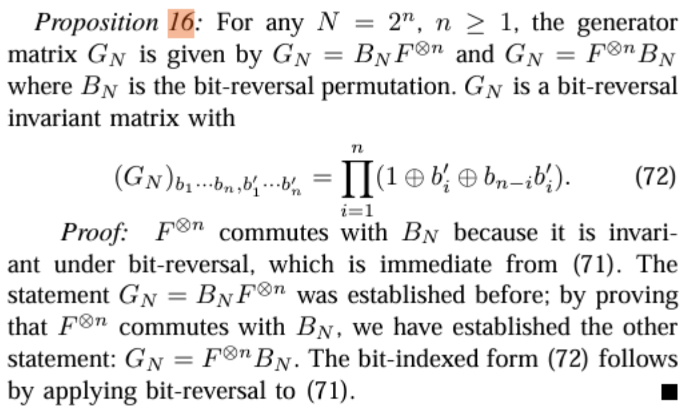
$$x_1^N = u_1^N \mathbf{G}_N = u_1^N \mathbf{F}_2^{\otimes n}.$$

$F_2^{\otimes n}$is n -th Kronecker power of matrix of F_2, where $F_2 = \begin{bmatrix} 1 & 0 \\ 1 & 1 \end{bmatrix}$

B_N is the bit-reversal permutation matrix

And it has: $B_N F_2^{\otimes n} = F_2^{\otimes n} B_N$

Refer to "Channel Polarization: A Method for Constructing Capacity-Achieving Codes for Symmetric Binary-Input Memoryless Channels" Proposition 16



Proposition 16: For any $N = 2^n$, $n \geq 1$, the generator matrix G_N is given by $G_N = B_N F^{\otimes n}$ and $G_N = F^{\otimes n} B_N$ where B_N is the bit-reversal permutation. G_N is a bit-reversal invariant matrix with

$$(G_N)_{b_1 \cdots b_n, b'_1 \cdots b'_n} = \prod_{i=1}^{n} (1 \oplus b'_i \oplus b_{n-i} b'_i). \qquad (72)$$

Proof: $F^{\otimes n}$ commutes with B_N because it is invariant under bit-reversal, which is immediate from (71). The statement $G_N = B_N F^{\otimes n}$ was established before; by proving that $F^{\otimes n}$ commutes with B_N, we have established the other statement: $G_N = F^{\otimes n} B_N$. The bit-indexed form (72) follows by applying bit-reversal to (71). ■

5G standard choose natural order encoder, while SC polar decoder is for bit-reversal order encoder.

Based on above equation we get: $x_1^N B_N = u_1^N F_2^{\otimes n} B_N = u_1^N B_N F_2^{\otimes n}$

It means 5G polar decoder LLR input data need bit-reverse before running SC decoder.

2.4 Puncturing and shorting difference in Polar rate matching

```
if $K/E \le 7/16$    -- puncturing
    for $k = 0$ to $E-1$
        $e_k = y_{k+N-E}$;
    end for
else    -- shortening
    for $k = 0$ to $E-1$
        $e_k = y_k$;
    end for
end if
```

Above is defined in 38.211 5.4.1.2 polar rate matching.

The difference:

- Puncturing is to remove first N-E bits which are valid bits
- Receiver shall set puncturing bits LLR = 0 because these bits are unknown.
- Shorting is to remove last N-E bit which are all zero
- Receiver shall set shorting bits LLR to positive highest value because these bits are known to be zero

Below is Polar processing output after polar encoding and rate matching Sub-block interleaving.

K=51， E= 216, N=256，K/E < 7/16, need puncture first 40 bits which is mixed with '0' and '1'

```
[1, 1, 0, 1, 0, 1, 1, 1, 0, 1, 0, 0, 0, 1, 1, 1, 0, 1, 0, 1, 0, 0,
 1, 1, 0, 1, 0, 1, 1, 1, 0, 0, 0, 1, 0, 1, 0, 1, 0, 1, 0, 0, 1, 1,
 0, 0, 1, 1, 1, 1, 0, 1, 1, 0, 0, 0, 0, 0, 1, 0,.....]
```

Below is Polar processing output after polar encoding and rate matching Sub-block interleaving.

K=59,E=108, N=128， K/E>7/16, need shorting last 20 bits which are all '0'

```
[0, 0, 0, 0, 1, 0, 0, 1, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0, 1,
 0, 0, 0, 0, 1, 1, 0, 1, 0, 0, 1, 1, 1, 1, 1, 1, 0, 1, 1, 0, 1, 0,
 0, 0, 0, 0, 1, 0, 0, 1, 1, 0, 0, 0, 0, 0, 1, 1, 1, 0, 0, 1, 0, 0,
 1, 0, 0, 1, 0, 0, 0, 1, 1, 0, 1, 0, 1, 1, 0, 1, 0, 0, 0, 0, 0, 1,
 0, 1, 0, 1, 1, 0, 0, 0, 0, 1, 0, 0, 1, 0, 1, 0, 0, 1, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0],
```

2.5 5G polar code channel interleaver after rate matching

38.212 5.4.1.3 Interleaving of coded bits defined Triangle interleaver which is used for channel interleaing after Polar rate matching repetition/puncturing/shortening.

The reason (from my understanding) is that Polar decoding is based on bit reliability. It decodes bit from higher reliability to lower reliability bit. It assumes input LLR has the same SNR.

While for high-rank modulation such as 16QAM and 64QAM, SNR among these bits is not the same.

For example for receiving 16QAM 4 bit [a0,a1,a2,a3]. SNR_a0 = SNR_a1 > SNR_a2=SNR_a3. This SNR difference destroy Polar bit's reliability sequence.

Based on 3GPP study, channel interleaver can be used to improve Polar decoding performance for high-rank modulation, and channel interleaver is not necessary for QPSK.

This is why in 5G, Channel interleaver is only enabled on UL UCI. the reason is that UL UCI could transmit in 16QAM/64QAM PUSCH.

2.5.1 3GPP studying of polar channel interleaver

Below are based on what I searched on 3GPP drafts.

"3GPP R1-1612656, Interleaver for Polar codes, Interdigital, RAN1#87, Reno, USA, Nov. 2016"

Proved that channel interleaver performance is 0.5dB better than without interleaver for 16QAM/64QAM

Different interleaver methods are proposed. Among them, Triangle interleaver from Qualcomm and row-column interleaver have the best performance.

Triangle interleaver from Qualcomm was finally chosen into 3GPP 5G.

The patent "Efficient interleaver design for polar codes" from Qualcomm has protected this Triangle interleaver.

All related reference:

"R1-1708649, Interleaver design for Polar codes, Qualcomm, RAN1#89, Hangzhou, China, May 2017"

"R1-1713474 Design and evaluation of interleaver for Polar codes, Qualcomm, 3GPP TSG-RAN RAN1#90 August 21th – 25th, 2017"

“R1- 1712649 Channel Interleaver for Polar Codes,Ericsson, 3GPP TSG RAN WG1 Meeting #90 21th – 25th August 2017”

Patent link:
（https://patents.google.com/patent/US12081333B2/en?q=(Channel+interleaver+design+polar+coding)&assignee=qualcomm&oq=Channel+interleaver+design+for+polar+coding+qualcomm
）

2.6 Summary of 5G polar coding and rate matching configuration

	UL DCI CRC payload size <=11	UL DCI CRC payload size >=12	DL DCI	BCH
CRC	CRC6	CRC11	CRC24C 24 ‘1’ CRCpadding, RNTI mask	CRC24C
Polar code block segmentation	no	If ($A \geq 360$ and $E \geq 1088$) or if $A \geq 1013$, $I_{seg} = 1$;	no	no
Polar config	$n_{\max} = 10$, $I_{IL} = 0$, $n_{PC} = 3$, $n_{PC}^{wm} = 1$ if $E_r - K_r + 3 > 192$ and $n_{PC}^{wm} = 0$ if $E_r - K_r + 3 \leq 192$	$n_{\max} = 10$, $I_{IL} = 0$, $n_{PC} = 0$, and $n_{PC}^{wm} = 0$	$n_{\max} = 9$, $I_{IL} = 1$, $n_{PC} = 0$, and $n_{PC}^{wm} = 0$.	$n_{\max} = 9$, $I_{IL} = 1$, $n_{PC} = 0$, and $n_{PC}^{wm} = 0$.
Rate match channel interleaver	$I_{BIL} = 1$	$I_{BIL} = 1$	$I_{BIL} = 0$.	$I_{BIL} = 0$.

3 5G NR polar decoder

Almost all Polar decoder paper I found sourced from below two papers:

- List decoding of polar codes
  Introduce two decoder algorithms: Successive cancellation (SC) decoding and SC list (SCL) decoding.
  SC can be regarded as list size =1 SCL
- LLR-based successive cancellation list decoding of polar codes.
  Provide simplified LLR calculation equation and PM (path metric) calculation equation

5G polar decoder algorithm can be regarded as LLR based CRC-aided+ distributed CRC-aided + Parity-check-aided SCL.

- CRC-aided
  After finishing L path Polar decoder, CRC check for each active path.
  Used for UL UCI

- Distributed CRC-aided
  Check CRC bits in the middle of SCL decoder and terminate the paths if CRC check failed.
  Used for DL DCI and BCH
- Parity check-aided
  Parity bit check in the middle of SCL decoder and terminate the paths if PC check failed.
  Used for UL UCI with small payload size

It took me quite some time to learn polar decoder. Based on my experience, it is better to learn SC first starting from an N=8 example. After understanding SC algorithm, it is very straightforward to understand SCL.

This section is organized as follows:

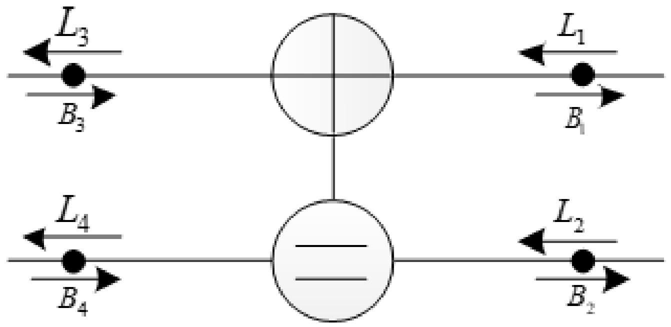
1. One N=8 polar SC decoder example to step-by-step show decoding procedure
2. Two SC decoder implementation
   a. One is to follow SC algorithm in "List decoding of polar codes"
   b. Another is to follow decoding procedure used for N=8 polar SC decoder example

   Both implementations give the same result.

   The difference is that in (a), LLR and B recursively calculate all branches value in current layer. While in (b) LLR and B only recursively its own related branch.

   Annex A1 and Annex A2 contain the logs of both SC algorithm for N=8 case.
3. Two SC decoder memory optimization to save LLR and B memory from N*log2(N) to N
4. 5G NR LLR based SCL decoder which exactly support 5G specification

3.1 basic computation unit of the polar encoder



This is one basic computation unit for polar decoder.

L1 – L4 are LLR values which are transferring from right to left.

B1-B4 are hard-decision bit which are transferring from left to right.

During the calculation, L_1, L_2 are known, it needs estimate $L_3 \rightarrow B_3 \rightarrow L_4 \rightarrow B_4 \rightarrow B_1$ *and* B_2.

The process and equation are:

Step 1: get L3 by $L_3 = sign(L_1)sign(L_2)\min(|L_1|, |L_2|)$

Step 2: $B_3 = \begin{cases} 0 & L_3 > 0 \\ 1 & L_3 < 0 \end{cases}$

Step 3: $L_4 = (-1)^{B_3} L_1 + L_2$

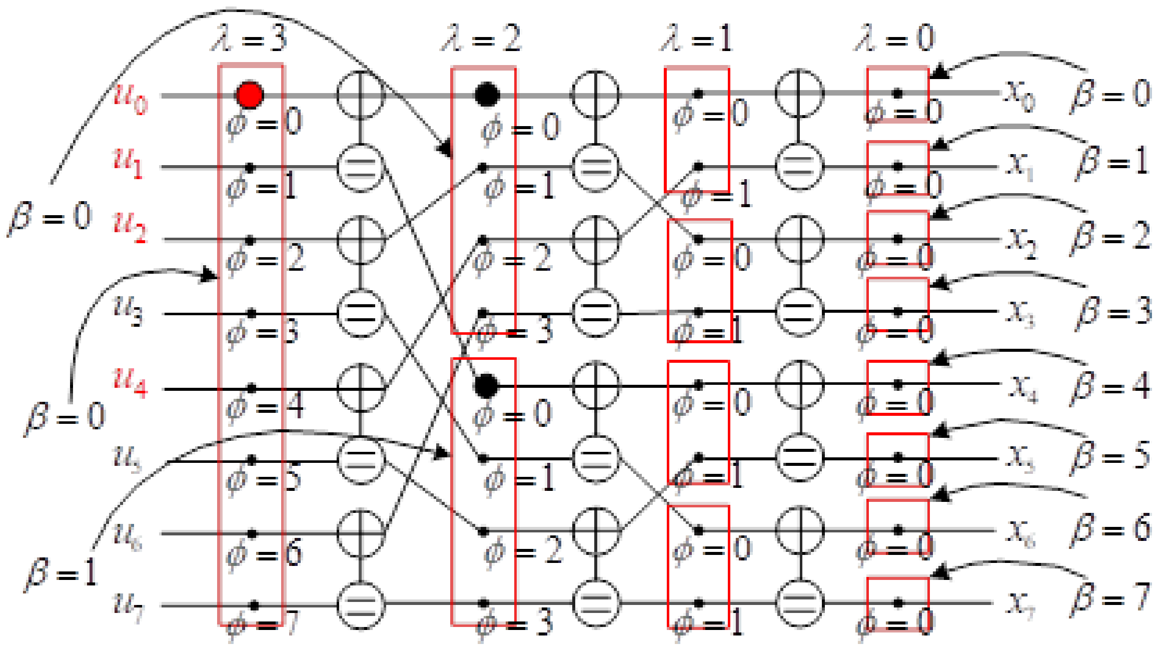
Step 4: $\mathrm{B}_4 = \begin{cases} 0 & \mathrm{L}_4 > 0 \\ 1 & \mathrm{L}_4 < 0 \end{cases}$

Step 5: $\mathrm{B}_1 = \mathrm{B}_3 \oplus \mathrm{B}_4$

Step 6: $\mathrm{B}_2 = \mathrm{B}_4$

All equations in above description comes from “LLR-based successive cancellation list decoding of polar codes”

3.2 N=8 Polar SC decoder example



In above figure, λ means layer, β means branch, ϕ is phase in the branch

Total layer number = log2(N)+1 = 4, layer index in above figure is 3-2-1-0 from left to right.

There are 2^{3-s} branch for layer index s, and each branch has 2^s phases.

For layer [3,2,1,0], branch number is [1,2,4,8], phase number per branch is [8,4,2,1]

Each point in above figure can be expressed with (layer, branch, phase)

Four red point u_0, u_1, u_2, u_4 mapping to B[3,0,0], B[3,0,1], B[3,0,2], B[3,0,4] are frozen bits.

LLR values in layer 0 is known for polar decoder.

SC decoder procedure is:

```
S1 cal LLR[3,0,0] which need LLR[2,0,0], LLR[2,1,0]
   S1.1 cal LLR[2,0,0] which need LLR[1,0,0], LLR[1,1,0]
        S1.1.1 set LLR[1,0,0] using LLR[0,0,0],LLR[0,1,0]
        S1.1.2 set LLR[1,1,0] using LLR[0,2,0],LLR[0,3,0]
        S1.1.3 set LLR[2,0,0] using LLR[1,0,0],LLR[1,1,0]
   S1.2 cal LLR[2,1,0] which need LLR[1,2,0], LLR[1,3,0]
        S1.1.1 set LLR[1,2,0] using LLR[0,4,0],LLR[0,5,0]
        S1.1.2 set LLR[1,3,0] using LLR[0,6,0],LLR[0,7,0]
        S1.1.3 set LLR[2,1,0] using LLR[1,2,0],LLR[1,3,0]
   S1.3 set LLR[3,0,0] using LLR[2,0,0], LLR[2,1,0]
S2 set B[3,0,0] to zero for this is frozen bit
S3 set LLR[3,0,1] using LLR[2,0,0], LLR[2,1,0],B[3,0,0]
S4 set B[3,0,1] to zero for this is frozen bit,
   S4.1 set B[2,0,0] using B[3,0,0], B[3,0,1]
   S4.2 set B[2,1,0] using B[3,0,1]
S5 cal LLR[3,0,2] which need LLR[2,0,1], LLR[2,1,1]
   S5.1 set LLR[2,0,1] using LLR[1,0,0],LLR[1,1,0],B[2,0,0]
   S5.2 set LLR[2,1,1] using LLR[1,2,0],LLR[1,3,0],B[2,1,0]
   S5.3 set LLR[3,0,2] using LLR[2,0,1],LLR[2,1,1]
S6 set B[3,0,2] to zero for this is frozen bit
S7 set LLR[3,0,3] using LLR[2,0,1], LLR[2,1,1],B[3,0,2]
S8 set B[3,0,3] using LLR[3,0,3]
   S8.1 set B[2,0,1] using B[3,0,2], B[3,0,3]
        S8.1.1 set B[1,0,0] using B[2,0,0],B[2,0,1]
        S8.1.2 set B[1,1,0] using B[2,0,1]
```

```
  S8.2 set B[2,1,1] using B[3,0,3]
      S8.1.1 set B[1,2,0] using B[2,1,0],B[2,1,1]
      S8.1.2 set B[1,3,0] using B[2,1,1]
S9 cal LLR[3,0,4] which need LLR[2,0,2], LLR[2,1,2]
   S9.1 cal LLR[2,0,2] which need LLR[1,0,1], LLR[1,1,1]
       S9.1.1 set LLR[1,0,1] using LLR[0,0,0],LLR[0,1,0],B[1,0,0]
       S9.1.2 set LLR[1,1,1] using LLR[0,2,0],LLR[0,3,0],B[1,1,0]
       S9.1.3 set LLR[2,0,2] using LLR[1,0,1], LLR[1,1,1]
   S9.2 cal LLR[2,1,2] which need LLR[1,2,1], LLR[1,3,1]
       S9.2.1 set LLR[1,2,1] using LLR[0,4,0],LLR[0,5,0],B[1,2,0]
       S9.2.2 set LLR[1,3,1] using LLR[0,6,0],LLR[0,7,0],B[1,3,0]
       S9.2.3 set LLR[2,1,2] using LLR[1,2,1], LLR[1,3,1]
  S9.3 set LLR[3,0,4] using LLR[2,0,2], LLR[2,1,2]
S10 set B[3,0,4] to zero for this is frozen bit
S11 set LLR[3,0,5] using LLR[2,0,2], LLR[2,1,2], B[3,0,4]
S12 set B[3,0,5] using LLR[3,0,5]
   S12.1 set B[2,0,2] using B[3,0,4],B[3,0,5]
   S12.2 set B[2,1,2] using B[3,0,5]
S13 cal LLR[3,0,6] which need LLR[2,0,3], LLR[2,1,3]
    S13.1 set LLR[2,0,3] using LLR[1,0,1],LLR[1,1,1],B[2,0,2]
    S13.2 set LLR[2,1,3] using LLR[1,2,1],LLR[1,3,1],B[2,1,2]
    S13.3 set LLR[3.0.6] using LLR[2,0,3], LLR[2,1,3]
S14 set B[3,0,6] using LLR[3,0,6]
S15 set LLR[3,0,7] using LLR[2,0,3], LLR[2,1,3],B[3,0,6]
S16 set B[3,0,7] using LLR[3,0,7]
    S16.1 set B[2,0,3] using B[3,0,6],B[3,0,7]
          S16.1.1 set B[1,0,1] using B[2,0,2],B[2,0,3]
                S16.1.1.1 set B[0,0,0] using B[1,0,0],B[1,0,1]
                S16.1.1.2 set B[0,1,0] using B[1,0,1]
          S16.1.2 set B[1,1,1] using B[2,0,3]
                S16.1.2.1 set B[0,2,0] using B[1,1,0],B[1,1,1]
```

            S16.1.2.2 set B[0,3,0] using B[1,1,1]

    S16.2 set B[2,1,3] using B[3,0,7]

        S16.2.1 set B[1,2,1] using B[2,1,2],B[2,1,3]

            S16.2.1.1 set B[0,4,0] using B[1,2,0],B[1,2,1]

            S16.2.1.2 set B[0,5,0] using B[1,2,1]

        S16.2.2 set B[1,3,1] using B[2,1,3]

            S16.2.2.1 set B[0,6,0] using B[1,3,0],B[1,3,1]

            S16.2.2.2 set B[0,7,0] using B[1,3,1]

Summary of LLR processing to cal LLR[m,n,s]:

- If s == even:
    - Recursively LLR[m-1,2*n,s//2], LLR[m-1,2*n+1,s//2]
    - When Recursively return, Cal LLR[m,n,s] using LLR[m-1,2*n,s//2], LLR[m-1,2*n+1,s//2]
- Else s == odd:
    - No need Recursively
    - Cal LLR[m,n,s] using LLR[m-1,2*n,s//2], LLR[m-1,2*n+1,s//2], B[m,n,s-1]

Summary of B processing for B[m,n,s]:

- Cal B[m,n,s] using LLR[m,n,s]
- If s == odd:
    - Cal B[m-1,2*n,s//2] and B[m-1,2*n+1,s//2] using B[m,n,s-1], B[m,n,s]
    - If (s//2) == odd:
        - Recursively Calc m-1 layer B values

3.3 LLR-based Polar SC(SUCCESSIVE CANCELLATION) decoder

Here is LLR-based SC algorithm based on "List decoding of polar codes".

Below is Python implementation of SC decoder.

First it need create a Path class which includes both LLR matrix and B matrix.

Path class shall initialize layer0 LLR values with input LLR.

```
class SC_Path_no_opt ():
    """ define Polar SC decoder path without path buffer optimization, buffer size = N*log2(N)
      based on "List Decoding of Polar Codes" from Algorithm1 to Algorithm 4
      one path include both LLR matrix and bit matrix
    """
    def __init__(self, LLR0, N):
        """  N is polar decode input size
          LLR0 is layer 0 LLR values
        """
        m = int(np.ceil(np.log2(N))) #m = log2(N)
        #define m+1 layer X N value per layer, for layer n, there are total 2^(m-n) branch with 2^n value per each branch
        #LLR is LLR values for each layers, B is estimated hard bits estimatio for each layers
        self.LLR = np.zeros((m+1,N))
        self.LLR[0,:] = LLR0
        self.B = -1*np.ones((m+1,N),'i2')
        self.m = m
        self.N = N

    def setB(self,value, layer, branch, phase):
        offset = branch * 2**layer + phase
        self.B[layer][offset] = value

    def getB(self, layer, branch, phase):
        offset = branch * 2**layer + phase
        return self.B[layer][offset]

    def setLLR(self,value, layer, branch, phase):
        offset = branch * 2**layer + phase
        self.LLR[layer][offset] = value

    def getLLR(self, layer, branch, phase):
        offset = branch * 2**layer + phase
        return self.LLR[layer][offset]

    def get_u_seq(self):
        """ return polar decoded bits, which is all values in layer=m"""
        u_seq = self.B[self.m,:]
        return u_seq
```

SC decoder main function

```
def PolarSCDecoder(LLRin, F, N):
    """ polar SC decoder"""
    #main function
    m = int(np.ceil(np.log2(N))) #m = log2(N)
   #Bit-wise reverse LLRin to get layer0 LLR
    LLR0 = np.zeros(N)
    for branch in range(N):
        #bit reverse LLRin
        br=int( '{:0{width}b}'.format(branch, width=m)[::-1],2)
        LLR0[branch] = LLRin[br]

    #init path
    path = polar_path.SC_Path_no_opt(LLR0, N)

    #SC main loop
    for n in range(N):
        recursivelyCalcLLR(path,m,n)
        if F[n] == 1:  #frozen bit
            path.setB(0, m, 0, n)
        else:
            LLR = path.getLLR(m,0,n)
            if LLR > 0:
                path.setB(0, m, 0, n)
            else:
                path.setB(1, m, 0, n)
        if (n%2)==1:
            recursivelyUpdateB(path,m,n)

    #get decoded bits
    decodedbits = path.get_u_seq()

    return decodedbits, True
```

```
def recursivelyUpdateB(path,layer,phase):
    m = path.m
    newphase = phase // 2
    for branch in range(2**(m-layer)):
        b1 = path.getB(layer,branch,phase-1)
        b2 = path.getB(layer,branch,phase)

        path.setB((b1+b2)%2, layer-1, 2*branch, newphase)
        path.setB(b2, layer-1, 2*branch+1, newphase)
    if (newphase % 2):
        recursivelyUpdateB(path,layer-1,newphase)
```

```
def recursivelyCalcLLR(path,layer, phase):
    if layer == 0:
        return
    newphase = phase // 2
    m = path.m
    if (phase % 2) == 0:
        recursivelyCalcLLR(path,layer-1, newphase)
    for branch in range(2**(m-layer)):
        LLR1 = path.getLLR(layer-1, 2*branch, newphase)
        LLR2 = path.getLLR(layer-1, 2*branch+1, newphase)

        if (phase % 2) == 0:
            value = np.sign(LLR1)*np.sign(LLR2)*min(abs(LLR1), abs(LLR2))
            path.setLLR(value, layer, branch, phase)
        else:
            B = path.getB(layer,branch,phase-1)
            value = LLR2 + (-1)**B * LLR1
            path.setLLR(value, layer, branch, phase)
```

3.4 LLR-based Polar SC(SUCCESSIVE CANCELLATION) decoder optionB

The main difference with A is recursivelyCalcLLR and recursivelyCalcB implementation

```
def PolarSCDecoder(LLRin, F, N):
    """ polar SC decoder"""
    #main function
    m = int(np.ceil(np.log2(N))) #m = log2(N)
    #Bit-wise reverse LLRin to get layer0 LLR
    LLR0 = np.zeros(N)
    for branch in range(N):
        #path.setLLR(LLRin[branch], 0, branch, 0)
        #bit reverse LLRin
        br=int( '{:0{width}b}'.format(branch, width=m)[::-1],2)
        LLR0[branch] = LLRin[br]

    #init path
    path = polar_path.SC_Path_no_opt(LLR0, N)

    #SC main loop
    for n in range(N):
        recursivelyCalcLLR(path,m,0, n)
        if F[n] == 1:  #frozen bit
            path.setB(0, m, 0, n)
        else:
            LLR = path.getLLR(m,0,n)
            if LLR > 0:
                path.setB(0, m, 0, n)
            else:
                path.setB(1, m, 0, n)

        if (n%2)==1:
            recursivelyUpdateB(path,m,0,n)

    #get decoded bits
    decodedbits = path.get_u_seq()

    return decodedbits, True
```

```
def recursivelyUpdateB(path,layer,branch,phase):
    #update path.B:
    newphase = phase // 2

    b1 = path.getB(layer,branch,phase-1)
    b2 = path.getB(layer,branch,phase)

    path.setB((b1+b2)%2, layer-1, 2*branch, newphase)
    if (newphase % 2):
        recursivelyUpdateB(path,layer-1,2*branch,newphase)

    path.setB(b2, layer-1, 2*branch+1, newphase)
    if (newphase % 2):
        recursivelyUpdateB(path,layer-1,2*branch+1,newphase)
```

```
def recursivelyCalcLLR(path,layer, branch,phase):
    if layer == 0:
        return
    newphase = phase // 2

    if (phase % 2) == 0:
        recursivelyCalcLLR(path,layer-1, 2*branch, newphase)
        recursivelyCalcLLR(path,layer-1, 2*branch+1, newphase)

    LLR1 = path.getLLR(layer-1, 2*branch, newphase)
    LLR2 = path.getLLR(layer-1, 2*branch+1, newphase)

    if (phase % 2) == 0:
        value = np.sign(LLR1)*np.sign(LLR2)*min(abs(LLR1), abs(LLR2))
        path.setLLR(value, layer, branch, phase)
    else:
        B = path.getB(layer,branch,phase-1)
        value = LLR2 + (-1)**B * LLR1
        path.setLLR(value, layer, branch, phase)
```

3.5 SC decoder memory optimization

In SC_Path_no_opt class,

LLR memory size = (log2(N)+1)*N

B memory size = 2*(log2(N)+1)*N

Total layer number $m = log2(N) + 1$

There are $2^{\mathrm{m}-s}$ branch for layer index s, and each branch has 2^{s} phases.

Below memory optimization is to save LLR and B memory size.

3.5.1 Memory optimization 1

This optimization is shown in “List decoding of polar codes” III. SPACE-EFFICIENT SUCCESSIVE CANCELLATION DECODING

From N=8 Polar SC decoder example we can see:

When start using LLR[layer,branch ,s]

1. all LLR[layer,branch ,phase] values with phase<n will not be used in the later decoding
2. all LLR[layer,branch ,phase] values with phase>n are not involved for decoding

for example, for layer=2 branch=0 LLR value.

All of LLR[2,0,0] get/set operations happen before decoder uses LLR[2,0,1], LLR[2,0,2], LLR[2,0,3]

When decoder start using LLR[2,0,1], LLR[2,0,0] value becomes useless.

And LLR[2,0,2], LLR[2,0,3] are not involved before the last LLR[2,0,1] get/set operation.

Similar rule also works for LLR[2,0,2], LLR[2,0,3]

it means that one LLR value is enough for any [layer,branch] group, while 2^{s} values are used in SC_Path_no_opt class

in this way, each layer only need $2^{\mathrm{m}-s}$ values,

and total LLR memory size = 1+2+4+…+N = 2*N – 1

similar happen for B value.

For example for layer=2, branch=0 B values.

All of B[2,0,0] and B[2,0,1] get/set operation happen before using B[2,0,2],B[2,0,3].

When decoder start using B[2,0,2] and B[2,0,3], B[2,0,0] and B[2,0,1] become useless.

It means two B values are enough for any [layer,branch] group, while 2^s values are used in SC_Path_no_opt class.

The total B memory size is 2*(1+2+4+..+N) = 2*(2*N-1)

It also needs additional N size memory to store last layer B values

Python function is below:

```
class SC_Path_opt1 ():
   """ define Polar SC decoder path with path buffer optimization,
      LLR buffer size = 2*N-1, B buffer size = N+2*(2N-1)
      based on "List Decoding of Polar Codes" from Algorithm5 to Algorithm 7
      one path include both LLR matrix and bit matrix
   """
   def __init__(self, LLR0, N):
      """  N is polar decode input size
         LLR0 is layer 0 LLR values
      """
      m = int(np.ceil(np.log2(N))) #m = log2(N)
      #there are total m+1 layers, for layer n, branch number = 2^(m-n)
      #LLR is LLR values for each layers, B is estimated hard bits estimatio for each layers
      #LLR of each branch need one value size, B ofeach branch need two value size
      #
      self.LLR = np.zeros(2*N-1)
      self.LLR[0:N] = LLR0
      self.B = -1*np.ones(2*(2*N-1),'i2')
      self.U = -1*np.ones(N,'i2') #decoded bit seq,
      self.m = m
      self.N = N

   def setB(self,value, layer, branch, phase):
      #each branch has two B value, put even value on position 0, put odd value on position 1
      offset = 2*(2**(self.m+1) - 2**(self.m+1-layer))  + branch*2 + (phase % 2)
      self.B[offset] = value
      if layer == self.m:
         self.U[phase] = value

   def getB(self, layer, branch, phase):
      #each branch has two B value, put even value on position 0, put odd value on position 1
      offset = 2*(2**(self.m+1) - 2**(self.m+1-layer))  + branch*2 + (phase % 2)
      return self.B[offset]

   def setLLR(self,value, layer, branch, phase):
      #each branch has one LLR value
      offset = (2**(self.m+1) - 2**(self.m+1-layer))  + branch
      self.LLR[offset] = value

   def getLLR(self, layer, branch, phase):
      #each branch has one LLR value
      offset = (2**(self.m+1) - 2**(self.m+1-layer))  + branch
      return self.LLR[offset]

   def get_u_seq(self):
      """ return polar decoded bits, which is all values in layer=m"""
      return self.U
```

3.5.2 Memory optimization 2

This is what I found when analyzing Memory optimization 1.

About B memory.

B values in layer 0 is not needed, it can save N values.

The total B memory size = 2*(1+2+4+…+N/2) = 2*(N-1)

About LLR memory.

There are N LLR values for layer=0. These layer0 LLR value are the input from external module.

It doesn’t need additional memory to copy these layer0 LLR to. In this way N LLR values for layer 0 can be saves.

The total LLR memory size = (1+2+4+…+ N/2) = N - 1

Python code are:

```
class SC_Path_opt2 ():
  """ define Polar SC decoder path with path buffer optimization method 2,
    there are N B value in layer 0 which is not necessary. removing layer0 B value can reduce B buffer size to N+2*(N-1)
    for SCL decoder, all path shared the same layer0 LLR value.LLR buffer doesn;t need N layer0 LLR value which can
reduce LLR buffer size to N-1
    one path include both LLR matrix and bit matrix
  """
  def __init__(self, LLR0, N):
    """  N is polar decode input size
      LLR0 is layer 0 LLR values
    """
    m = int(np.ceil(np.log2(N))) #m = log2(N)
    #there are total m+1 layers, for layer n, branch number = 2^(m-n)
    #LLR is LLR values for each layers, B is estimated hard bits estimatio for each layers
    #LLR of each branch need one value size, B ofeach branch need two value size
    #
    self.LLR = np.zeros(N-1)
    self.LLR0 = LLR0
    self.B = -1*np.ones(2*(N-1),'i2')
    self.U = -1*np.ones(N,'i2') #decoded bit seq,
    self.m = m
    self.N = N

  def setB(self,value, layer, branch, phase):
    #each branch has two B value, put even value on position 0, put odd value on position 1
    if layer == 0:
      return
    offset = 2*(2**self.m - 2**(self.m-(layer-1)))  + branch*2 + (phase % 2)
    self.B[offset] = value
    if layer == self.m:
      self.U[phase] = value

  def getB(self, layer, branch, phase):
    #each branch has two B value, put even value on position 0, put odd value on position 1
```

```
        if layer == 0:
            return -1
        offset = 2*(2**self.m - 2**(self.m-(layer-1)))  + branch*2 + (phase % 2)
        return self.B[offset]

    def setLLR(self,value, layer, branch, phase):
        #each branch has one LLR value
        if layer == 0:
            assert 0
        offset = (2**self.m - 2**(self.m-(layer-1)))  + branch
        self.LLR[offset] = value

    def getLLR(self, layer, branch, phase):
        #each branch has one LLR value
        if layer == 0:
            return self.LLR0[branch]
        offset = (2**self.m - 2**(self.m-(layer-1)))  + branch
        return self.LLR[offset]

    def get_u_seq(self):
        """ return polar decoded bits, which is all values in layer=m"""
        return self.U
```

3.6 5G NR LLR based CRC-aided+ distributed CRC-aided + Parity-check-aided SCL decoder

SCL(successive cancellation list) algorithm is introduced in “List decoding of polar codes”, but I read the paper a couple of time and still can not get idea about the description and implementation of SCL in this paper.

From software point of view the idea of SCL is a “Tree splitting” question.

SC decoder support only one path and hard-decide B value to be 0 or 1 on each step.

SCL decoding support L paths.

For L paths SCL, it needs:

1. A method to split the path.
   SCL will split the path to two, by setting B value in one path to be 0 and setting B value is another path to be 1.
2. A path metric to identify path quality

   When the tree is growing up to be more than L paths, select best L paths and then reject other paths.

A Path Metric(PM) value need be added to each path which is used to decide path quality.

Refer to “LLR-based successive cancellation list decoding of polar codes” equation (12),

$$PM_{new} = \begin{cases} PM\_old & if\ B = \frac{1}{2}(1 - sign(LLR)) \\ PM_{old} + |LLR| & otherwise \end{cases}$$

Smaller PM value means better path quality.

The basic design of 5G SCL:

1. Create SCL_path class which
    a. inherited from SC_path,
    b. add current PM value and two nextPM values for B=0 and B=1
2. create Pathlist class to:
    a. active/de-active path
    b. split and then clone path to new path
3. the function to choose the best L paths when the tree is split to more than L paths
4. parity-bit check when the decoded bit is parity bit and remove those paths with parity-check failure
5. CRC bit check for distributed CRC when the decoded bit is CRC bit, and remove those paths with CRC-bit check failure
6. CRC verification after SCL decoding is complete.

Below is the copy of python code.

```
class SCL_Path (SC_Path_opt2):
  """ define SCL Polar decoder path based on SC_Path_opt2
    and add Path Metric(PM) related processing
    PM calculation is based on "LLR-Based Successive Cancellation List Decoding of Polar Codes" equation 12

  """
  def __init__(self, LLR0, N):
    super().__init__(LLR0,N)
    self.PM = 0 #currect PM
    self.nextPM = [0,0] #next stage PM for u=0 and u= 1

  def clone(self,clone2path):
    super().clone(clone2path)
    clone2path.PM = self.PM
    clone2path.nextPM = self.nextPM.copy()

  def update_BandPM(self,phase,u):
    """ set B on the phase of layer m, and then update PM
    """
    self.setB(u, self.m, 0, phase)
    LLR = self.getLLR(self.m,0,phase)
    if u != (1-np.sign(LLR))/2:
      self.PM = self.PM + abs(LLR)

  def gen_nextPM(self,phase):
    """ generate next stage PM values for u=0 and u=1
    nextPM[0] is u=0 PM, nextPM[1] is u=1 PM
    """
    LLR = self.getLLR(self.m,0,phase)

    if 0 == (1-np.sign(LLR))/2: #u=0 a
      self.nextPM[0] = self.PM
```

```
        self.nextPM[1] = self.PM + abs(LLR)
    else:
        self.nextPM[0] = self.PM + abs(LLR)
        self.nextPM[1] = self.PM
```

```
class Pathlist ():
    """ polar SCL decoder path list
        N: polar length
        L: path list size
    """
    def __init__(self, LLR0, N, L):
        self.L = L
        self.paths = [polar_path.SCL_Path(LLR0, N) for m in range(L)]
        self.paths_status = [0]*L #0 is inactive, 1: active
        self.paths_status[0] = 1 #active first path

    def get_path(self,index):
        return self.paths[index]

    def active_path(self, index):
        self.paths_status[index] = 1

    def inactive_path(self, index):
        self.paths_status[index] = 0

    def get_path_status(self, index):
        return self.paths_status[index]

    def get_total_active_paths(self):
        return sum(self.paths_status)

    def get_inactive_path_idx(self):
        """ find first inactive path
            -1: no inacive path
        """
        idx = self.paths_status.index(0) if 0 in self.paths_status else -1
        return idx

    def get_active_paths(self):
        active_paths = []
        for idx in range(self.L):
            if self.get_path_status(idx) == 1: #active
                active_paths.append([idx,self.get_path(idx)])
        return active_paths
```

```
def nr_decode_polar_SCL(LLRin, E, K, L, nMax, iIL, CRCLEN=24, padCRC=0, rnti=0 ):

    #get F array, PC array and other values
    F, qPC, N, nPC, nPCwm = polar_construct.construct(K, E, nMax)

    m = int(np.ceil(np.log2(N))) #m = log2(N)

    #initial a few tables
    #bit seqeucne ck is used to polar interleaving to get ckbar, ckbar map to u seq at non-frozen location
    #here need get u_seq to ckbar mapping table,
    ckbar_indices = [idx for idx in range(N) if (F[idx]==0) and (idx not in qPC)]
    #deinterleave table, used for iIL == 1
    if iIL == 1:
        depitable = polar_interleaver.gen_deinterleave_table(K)
        crcidx_matrix = crc.gen_CRC24C_encoding_matrix(K-24)
        crc_mask = crc.gen_crc_mask(K-24, padCRC, rnti)

    #Bit-wise reverse LLRin to get layer0 LLR
    LLR0 = np.zeros(N)
    for branch in range(N):
        #path.setLLR(LLRin[branch], 0, branch, 0)
        #bit reverse LLRin
        br=int( '{:0{width}b}'.format(branch, width=m)[::-1],2)
        LLR0[branch] = LLRin[br]

    #init pathlist, active first path,
    pathlist = polar_path_list.Pathlist(LLR0, N, L)

    #SCL main loop
    for phase in range(N):
        active_paths = pathlist.get_active_paths()
        #recursivelyCalcLLR for each active path
        for _, path in active_paths:
            recursivelyCalcLLR(path,m,phase)

        if F[phase] == 1: #frozen bits
            for _, path in active_paths:
                path.update_BandPM(phase,0) #set B=0 and update PM
        else: #unfrozen bits
            continuePaths_UnfrozenBit(pathlist, phase)

            active_paths = pathlist.get_active_paths() #get new active paths

            if nPC > 0 and (phase in qPC): #this this is PC bit
                # PC check
                for idx, path in active_paths:
                    u_seq = path.get_u_seq()
                    pc = u_seq[phase]
                    u_seq = u_seq[0:phase] #select u seq from 0 to phase -1
                    if pc != cal_polar_pc(u_seq, F, qPC, phase):
                        #bad path
                        pathlist.inactive_path(idx)
            elif iIL  :
                #if polar interleaver = 1
                # distributed CRC check, CRC poly must be 24C and CRC size = 24
                #get ckbar index for phase in u_seq
                ckbar_loc = ckbar_indices.index(phase)
                if ckbar_loc in depitable[-24:]: #this is CRC bits
                    for idx, path in active_paths:
                        u_seq = path.get_u_seq()
                        ckbar = u_seq[ckbar_indices] #get ckbar bit sequence
                        ck = ckbar[depitable] #deinterleave to get ck seq
                        crc_bit_loc = depitable[-24:].index(ckbar_loc)
                        if  crc.check_distributed_CRC24C(ck, crc_bit_loc, crcidx_matrix, crc_mask) == False:
                            #CRC bit can not match
                            pathlist.inactive_path(idx)

            if pathlist.get_total_active_paths() == 0:
                #early terminate
                return [-1], False

        active_paths = pathlist.get_active_paths() #get new active paths
        if (phase % 2) == 1:
            for _, path in active_paths:
                recursivelyUpdateB(path,m,phase)
    if pathlist.get_total_active_paths() == 0:
        #early terminate
        return [-1], False

    #sort active paths based on PM and then do CRC check for each active path until pass CRC check
    active_paths = pathlist.get_active_paths() #get new active paths
    PM_list = [[idx,path.PM] for idx, path in active_paths]
    PM_list.sort(key=lambda x: x[1]) #sort by the second element of sublist

    if iIL:
        #distributed CRC has passed during CA-SCL decoding, here return best active path
        path = pathlist.get_path(PM_list[0][0])
        decodedbits = path.get_u_seq()
        ckbar = decodedbits[ckbar_indices] #get information bit and CRC bits only, not including pc bits
        ck = ckbar[depitable] #deinterleave to get ck seq
        return np.array(ck,'i1'), True
    else:
        for idx, _ in PM_list:
            path = pathlist.get_path(idx)
            #get decoded bits
            decodedbits = path.get_u_seq()
            ckbar = decodedbits[ckbar_indices] #get information bit and CRC bits only, not including pc bits
            ck = ckbar
            poly = {6:'6', 11: '11', 24: '24C'}[CRCLEN]
            _, err = crc.nr_crc_decode(np.array(ck,'i1'), poly, rnti)
            if err == 0:
                return np.array(ck,'i1'), True

    #all path CRC failed
    return [-1], False
```

```
def continuePaths_UnfrozenBit(pathlist, phase):
    active_paths = pathlist.get_active_paths()
    #generate nextPM list for B=0 and B=1 for all active path
    #PM_list sublist [idx, B, PM]
    PM_list = []
    for idx, path in active_paths:
        path.gen_nextPM(phase)
        PM_list.append([path.nextPM[0],idx,0])
        PM_list.append([path.nextPM[1],idx,1])

    L = pathlist.L
    if pathlist.get_total_active_paths()*2 <= L:
        #split each active path to two,update current path with B=0 and new path with B=1
        for _, path in active_paths:
            #clone to one inactive path
            inactive_idx = pathlist.get_inactive_path_idx()
            clone2path = pathlist.get_path(inactive_idx)
            path.clone(clone2path)

            path.update_BandPM(phase,0) #update current path with B=0
            clone2path.update_BandPM(phase,1) #update new path with B=1
            pathlist.active_path(inactive_idx) #set new path to active
    else:
        PM_list.sort() #sort by PM
        tmp = [v[0] for v in PM_list]
        threshold = statistics.median(tmp)

        # inactive paths with both nextPM value that is >= threshold
        # to free bad path
        for idx, path in active_paths:
```

```
        if (path.nextPM[0] >= threshold) and (path.nextPM[1] >= threshold):
            pathlist.inactive_path(idx)

    active_paths = pathlist.get_active_paths() #get new active paths

    #if only one nextPM of the active path < threshod, update this path
    for _, path in active_paths:
        if (path.nextPM[0] < threshold) and (path.nextPM[1] >= threshold):
            path.update_BandPM(phase,0)

        if (path.nextPM[0] >= threshold) and (path.nextPM[1] < threshold):
            path.update_BandPM(phase,1)

    #duplicate active paths that both nextPM < threshold
    # it may be possible that multiple nextPM value is equal to threhold,
    for _, path in active_paths:
        if (path.nextPM[0] < threshold) and (path.nextPM[1] < threshold):
            #clone to one inactive path
            inactive_idx = pathlist.get_inactive_path_idx()
            if inactive_idx == -1:
                assert 0 #should not reach here
            else:
                clone2path = pathlist.get_path(inactive_idx)
                path.clone(clone2path)

                path.update_BandPM(phase,0) #update current path with B=0
                clone2path.update_BandPM(phase,1) #update new path with B=1
                pathlist.active_path(inactive_idx) #set new path to active
```

3.7 5G NR LLR based CRC-aided+ distributed CRC-aided + Parity-check-aided SCL decoder optionB

Refer to nr_polar_decoder_CA_PC_SCL_optionB.py

Similar with above SCL decoder, the only difference are recursivelyCalcLLR and recursivelyCalcB

3.8 Polar decoder simulation

Below are polar decoder simulation results for K=64, N=128, ½ rate.

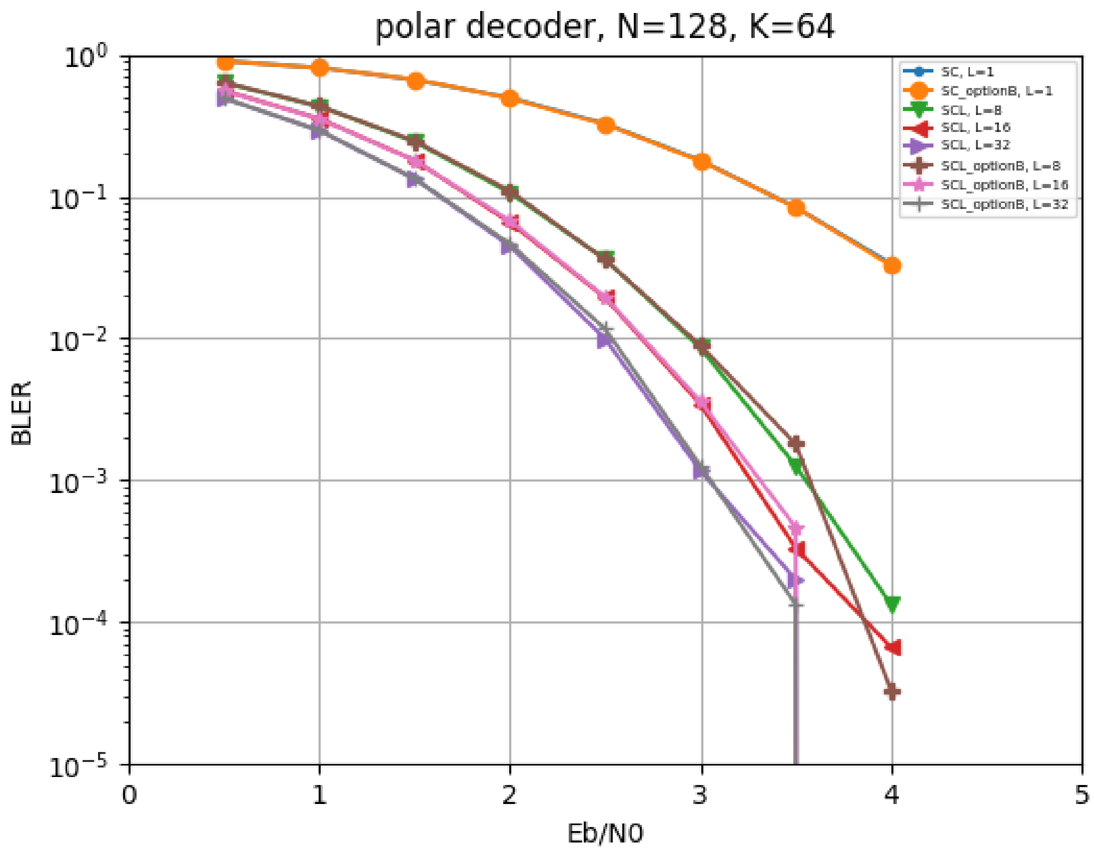
In the simulation

- It simulate SC, SC optionB, SCL L=8, SCL L=16, SCL L=32 with and without optionB
- Simulate SNR from 0.5 to 4dB with 0.5dB step
- Each SNR run 30K times of test
- The simulation takes a few days to finish

From the figure we can see:

1. SC performance is around 2dB worse than SCL under 10% BLER
2. L=32 SCL is around 0.25dB better than L=16 SCL
3. L=16 SCL is around 0.25dB better than L=8 SCL
4. optionB decoder performance is similar with non optionB decoder.

The code is [https://github.com/hahaliu2001/python_5gtoolbox.git : script](https://github.com/hahaliu2001/python_5gtoolbox.git)



4 Annex A1 traditional Polar SC N=8 logs

```
traditional SC in main func, phase= 0
cal LLR_value [3, x, 0]
cal LLR_value [2, x, 0]
cal LLR_value [1, x, 0]
recursivelyCalcLLR get LLR_value [0, 0, 0],[0, 1, 0]
recursivelyCalcLLR set LLR_value [1, 0, 0]
recursivelyCalcLLR get LLR_value [0, 2, 0],[0, 3, 0]
recursivelyCalcLLR set LLR_value [1, 1, 0]
recursivelyCalcLLR get LLR_value [0, 4, 0],[0, 5, 0]
recursivelyCalcLLR set LLR_value [1, 2, 0]
recursivelyCalcLLR get LLR_value [0, 6, 0],[0, 7, 0]
recursivelyCalcLLR set LLR_value [1, 3, 0]
recursivelyCalcLLR get LLR_value [1, 0, 0],[1, 1, 0]
recursivelyCalcLLR set LLR_value [2, 0, 0]
recursivelyCalcLLR get LLR_value [1, 2, 0],[1, 3, 0]
recursivelyCalcLLR set LLR_value [2, 1, 0]
recursivelyCalcLLR get LLR_value [2, 0, 0],[2, 1, 0]
recursivelyCalcLLR set LLR_value [3, 0, 0]
main func B_value set frozen bit,[3, 0, 0]
traditional SC in main func, phase= 1
cal LLR_value [3, x, 1]
recursivelyCalcLLR get LLR_value [2, 0, 0],[2, 1, 0]
recursivelyCalcLLR B_value get [3, 0, 0]
recursivelyCalcLLR set LLR_value [3, 0, 1]
main func B_value set frozen bit,[3, 0, 1]
recursivelyUpdateB B_value get [3, 0, 0],[3, 0, 1]
recursivelyUpdateB B_value set [2, 0, 0]
recursivelyUpdateB B_value set [2, 1, 0]
traditional SC in main func, phase= 2
cal LLR_value [3, x, 2]
cal LLR_value [2, x, 1]
recursivelyCalcLLR get LLR_value [1, 0, 0],[1, 1, 0]
recursivelyCalcLLR B_value get [2, 0, 0]
recursivelyCalcLLR set LLR_value [2, 0, 1]
recursivelyCalcLLR get LLR_value [1, 2, 0],[1, 3, 0]
recursivelyCalcLLR B_value get [2, 1, 0]
recursivelyCalcLLR set LLR_value [2, 1, 1]
recursivelyCalcLLR get LLR_value [2, 0, 1],[2, 1, 1]
recursivelyCalcLLR set LLR_value [3, 0, 2]
main func B_value set frozen bit,[3, 0, 2]
traditional SC in main func, phase= 3
cal LLR_value [3, x, 3]
recursivelyCalcLLR get LLR_value [2, 0, 1],[2, 1, 1]
recursivelyCalcLLR B_value get [3, 0, 2]
recursivelyCalcLLR set LLR_value [3, 0, 3]
main func B_value set unfrozen bit,[3, 0, 3], LLR= 4.0
recursivelyUpdateB B_value get [3, 0, 2],[3, 0, 3]
recursivelyUpdateB B_value set [2, 0, 1]
recursivelyUpdateB B_value set [2, 1, 1]
recursivelyUpdateB B_value get [2, 0, 0],[2, 0, 1]
recursivelyUpdateB B_value set [1, 0, 0]
recursivelyUpdateB B_value set [1, 1, 0]
recursivelyUpdateB B_value get [2, 1, 0],[2, 1, 1]
recursivelyUpdateB B_value set [1, 2, 0]
recursivelyUpdateB B_value set [1, 3, 0]
traditional SC in main func, phase= 4
cal LLR_value [3, x, 4]
cal LLR_value [2, x, 2]
cal LLR_value [1, x, 1]
recursivelyCalcLLR get LLR_value [0, 0, 0],[0, 1, 0]
recursivelyCalcLLR B_value get [1, 0, 0]
recursivelyCalcLLR set LLR_value [1, 0, 1]
recursivelyCalcLLR get LLR_value [0, 2, 0],[0, 3, 0]
recursivelyCalcLLR B_value get [1, 1, 0]
recursivelyCalcLLR set LLR_value [1, 1, 1]
recursivelyCalcLLR get LLR_value [0, 4, 0],[0, 5, 0]
recursivelyCalcLLR B_value get [1, 2, 0]
recursivelyCalcLLR set LLR_value [1, 2, 1]
recursivelyCalcLLR get LLR_value [0, 6, 0],[0, 7, 0]
recursivelyCalcLLR B_value get [1, 3, 0]
recursivelyCalcLLR set LLR_value [1, 3, 1]
recursivelyCalcLLR get LLR_value [1, 0, 1],[1, 1, 1]
recursivelyCalcLLR set LLR_value [2, 0, 2]
recursivelyCalcLLR get LLR_value [1, 2, 1],[1, 3, 1]
recursivelyCalcLLR set LLR_value [2, 1, 2]
recursivelyCalcLLR get LLR_value [2, 0, 2],[2, 1, 2]
recursivelyCalcLLR set LLR_value [3, 0, 4]
main func B_value set frozen bit,[3, 0, 4]
traditional SC in main func, phase= 5
cal LLR_value [3, x, 5]
recursivelyCalcLLR get LLR_value [2, 0, 2],[2, 1, 2]
recursivelyCalcLLR B_value get [3, 0, 4]
recursivelyCalcLLR set LLR_value [3, 0, 5]
main func B_value set unfrozen bit,[3, 0, 5], LLR= 4.0
recursivelyUpdateB B_value get [3, 0, 4],[3, 0, 5]
recursivelyUpdateB B_value set [2, 0, 2]
recursivelyUpdateB B_value set [2, 1, 2]
traditional SC in main func, phase= 6
cal LLR_value [3, x, 6]
cal LLR_value [2, x, 3]
recursivelyCalcLLR get LLR_value [1, 0, 1],[1, 1, 1]
recursivelyCalcLLR B_value get [2, 0, 2]
recursivelyCalcLLR set LLR_value [2, 0, 3]
recursivelyCalcLLR get LLR_value [1, 2, 1],[1, 3, 1]
recursivelyCalcLLR B_value get [2, 1, 2]
recursivelyCalcLLR set LLR_value [2, 1, 3]
recursivelyCalcLLR get LLR_value [2, 0, 3],[2, 1, 3]
recursivelyCalcLLR set LLR_value [3, 0, 6]
main func B_value set unfrozen bit,[3, 0, 6], LLR= -4.0
traditional SC in main func, phase= 7
cal LLR_value [3, x, 7]
recursivelyCalcLLR get LLR_value [2, 0, 3],[2, 1, 3]
recursivelyCalcLLR B_value get [3, 0, 6]
recursivelyCalcLLR set LLR_value [3, 0, 7]
main func B_value set unfrozen bit,[3, 0, 7], LLR= 8.0
recursivelyUpdateB B_value get [3, 0, 6],[3, 0, 7]
recursivelyUpdateB B_value set [2, 0, 3]
recursivelyUpdateB B_value set [2, 1, 3]
recursivelyUpdateB B_value get [2, 0, 2],[2, 0, 3]
recursivelyUpdateB B_value set [1, 0, 1]
recursivelyUpdateB B_value set [1, 1, 1]
recursivelyUpdateB B_value get [2, 1, 2],[2, 1, 3]
recursivelyUpdateB B_value set [1, 2, 1]
recursivelyUpdateB B_value set [1, 3, 1]
recursivelyUpdateB B_value get [1, 0, 0],[1, 0, 1]
recursivelyUpdateB B_value set [0, 0, 0]
recursivelyUpdateB B_value set [0, 1, 0]
recursivelyUpdateB B_value get [1, 1, 0],[1, 1, 1]
recursivelyUpdateB B_value set [0, 2, 0]
recursivelyUpdateB B_value set [0, 3, 0]
recursivelyUpdateB B_value get [1, 2, 0],[1, 2, 1]
recursivelyUpdateB B_value set [0, 4, 0]
recursivelyUpdateB B_value set [0, 5, 0]
recursivelyUpdateB B_value get [1, 3, 0],[1, 3, 1]
recursivelyUpdateB B_value set [0, 6, 0]
recursivelyUpdateB B_value set [0, 7, 0]
```

5 Annex A2 new Polar SC N=8 logs

```
new SC in main func, phase= 0
cal LLR_value [3, 0, 0],need [2, 0, 0],[2, 1, 0]
cal LLR_value [2, 0, 0],need [1, 0, 0],[1, 1, 0]
cal LLR_value [1, 0, 0],need [0, 0, 0],[0, 1, 0]
recursivelyCalcLLR get LLR_value [0, 0, 0],[0, 1, 0]
recursivelyCalcLLR set LLR_value [1, 0, 0]
cal LLR_value [1, 1, 0],need [0, 2, 0],[0, 3, 0]
recursivelyCalcLLR get LLR_value [0, 2, 0],[0, 3, 0]
recursivelyCalcLLR set LLR_value [1, 1, 0]
recursivelyCalcLLR get LLR_value [1, 0, 0],[1, 1, 0]
recursivelyCalcLLR set LLR_value [2, 0, 0]
cal LLR_value [2, 1, 0],need [1, 2, 0],[1, 3, 0]
cal LLR_value [1, 2, 0],need [0, 4, 0],[0, 5, 0]
recursivelyCalcLLR get LLR_value [0, 4, 0],[0, 5, 0]
recursivelyCalcLLR set LLR_value [1, 2, 0]
cal LLR_value [1, 3, 0],need [0, 6, 0],[0, 7, 0]
recursivelyCalcLLR get LLR_value [0, 6, 0],[0, 7, 0]
recursivelyCalcLLR set LLR_value [1, 3, 0]
recursivelyCalcLLR get LLR_value [1, 2, 0],[1, 3, 0]
recursivelyCalcLLR set LLR_value [2, 1, 0]
recursivelyCalcLLR get LLR_value [2, 0, 0],[2, 1, 0]
recursivelyCalcLLR set LLR_value [3, 0, 0]
main func B_value set frozen bit,[3, 0, 0]
new SC in main func, phase= 1
cal LLR_value [3, 0, 1],need [2, 0, 0],[2, 1, 0]
recursivelyCalcLLR get LLR_value [2, 0, 0],[2, 1, 0]
recursivelyCalcLLR B_value get [3, 0, 0]
recursivelyCalcLLR set LLR_value [3, 0, 1]
main func B_value set frozen bit,[3, 0, 1]
recursivelyUpdateB B_value get [3, 0, 0],[3, 0, 1]
recursivelyUpdateB B_value set [2, 0, 0]
recursivelyUpdateB B_value set [2, 1, 0]
new SC in main func, phase= 2
cal LLR_value [3, 0, 2],need [2, 0, 1],[2, 1, 1]
cal LLR_value [2, 0, 1],need [1, 0, 0],[1, 1, 0]
recursivelyCalcLLR get LLR_value [1, 0, 0],[1, 1, 0]
recursivelyCalcLLR B_value get [2, 0, 0]
recursivelyCalcLLR set LLR_value [2, 0, 1]
cal LLR_value [2, 1, 1],need [1, 2, 0],[1, 3, 0]
recursivelyCalcLLR get LLR_value [1, 2, 0],[1, 3, 0]
recursivelyCalcLLR B_value get [2, 1, 0]
recursivelyCalcLLR set LLR_value [2, 1, 1]
recursivelyCalcLLR get LLR_value [2, 0, 1],[2, 1, 1]
recursivelyCalcLLR set LLR_value [3, 0, 2]
main func B_value set frozen bit,[3, 0, 2]
new SC in main func, phase= 3
cal LLR_value [3, 0, 3],need [2, 0, 1],[2, 1, 1]
recursivelyCalcLLR get LLR_value [2, 0, 1],[2, 1, 1]
recursivelyCalcLLR B_value get [3, 0, 2]
recursivelyCalcLLR set LLR_value [3, 0, 3]
main func B_value set unfrozen bit,[3, 0, 3], LLR= 4.0
recursivelyUpdateB B_value get [3, 0, 2],[3, 0, 3]
recursivelyUpdateB B_value set [2, 0, 1]
recursivelyUpdateB B_value get [2, 0, 0],[2, 0, 1]
recursivelyUpdateB B_value set [1, 0, 0]
recursivelyUpdateB B_value set [1, 1, 0]
recursivelyUpdateB B_value set [2, 1, 1]
recursivelyUpdateB B_value get [2, 1, 0],[2, 1, 1]
recursivelyUpdateB B_value set [1, 2, 0]
recursivelyUpdateB B_value set [1, 3, 0]
new SC in main func, phase= 4
cal LLR_value [3, 0, 4],need [2, 0, 2],[2, 1, 2]
cal LLR_value [2, 0, 2],need [1, 0, 1],[1, 1, 1]
cal LLR_value [1, 0, 1],need [0, 0, 0],[0, 1, 0]
recursivelyCalcLLR get LLR_value [0, 0, 0],[0, 1, 0]
recursivelyCalcLLR B_value get [1, 0, 0]
recursivelyCalcLLR set LLR_value [1, 0, 1]
cal LLR_value [1, 1, 1],need [0, 2, 0],[0, 3, 0]
recursivelyCalcLLR get LLR_value [0, 2, 0],[0, 3, 0]
recursivelyCalcLLR B_value get [1, 1, 0]
recursivelyCalcLLR set LLR_value [1, 1, 1]
recursivelyCalcLLR get LLR_value [1, 0, 1],[1, 1, 1]
recursivelyCalcLLR set LLR_value [2, 0, 2]
cal LLR_value [2, 1, 2],need [1, 2, 1],[1, 3, 1]
cal LLR_value [1, 2, 1],need [0, 4, 0],[0, 5, 0]
recursivelyCalcLLR get LLR_value [0, 4, 0],[0, 5, 0]
recursivelyCalcLLR B_value get [1, 2, 0]
recursivelyCalcLLR set LLR_value [1, 2, 1]
cal LLR_value [1, 3, 1],need [0, 6, 0],[0, 7, 0]
recursivelyCalcLLR get LLR_value [0, 6, 0],[0, 7, 0]
recursivelyCalcLLR B_value get [1, 3, 0]
recursivelyCalcLLR set LLR_value [1, 3, 1]
recursivelyCalcLLR get LLR_value [1, 2, 1],[1, 3, 1]
recursivelyCalcLLR set LLR_value [2, 1, 2]
recursivelyCalcLLR get LLR_value [2, 0, 2],[2, 1, 2]
recursivelyCalcLLR set LLR_value [3, 0, 4]
main func B_value set frozen bit,[3, 0, 4]
new SC in main func, phase= 5
cal LLR_value [3, 0, 5],need [2, 0, 2],[2, 1, 2]
recursivelyCalcLLR get LLR_value [2, 0, 2],[2, 1, 2]
recursivelyCalcLLR B_value get [3, 0, 4]
recursivelyCalcLLR set LLR_value [3, 0, 5]
main func B_value set unfrozen bit,[3, 0, 5], LLR= 4.0
recursivelyUpdateB B_value get [3, 0, 4],[3, 0, 5]
recursivelyUpdateB B_value set [2, 0, 2]
recursivelyUpdateB B_value set [2, 1, 2]
new SC in main func, phase= 6
cal LLR_value [3, 0, 6],need [2, 0, 3],[2, 1, 3]
cal LLR_value [2, 0, 3],need [1, 0, 1],[1, 1, 1]
recursivelyCalcLLR get LLR_value [1, 0, 1],[1, 1, 1]
recursivelyCalcLLR B_value get [2, 0, 2]
recursivelyCalcLLR set LLR_value [2, 0, 3]
cal LLR_value [2, 1, 3],need [1, 2, 1],[1, 3, 1]
recursivelyCalcLLR get LLR_value [1, 2, 1],[1, 3, 1]
recursivelyCalcLLR B_value get [2, 1, 2]
recursivelyCalcLLR set LLR_value [2, 1, 3]
recursivelyCalcLLR get LLR_value [2, 0, 3],[2, 1, 3]
recursivelyCalcLLR set LLR_value [3, 0, 6]
main func B_value set unfrozen bit,[3, 0, 6], LLR= -4.0
new SC in main func, phase= 7
cal LLR_value [3, 0, 7],need [2, 0, 3],[2, 1, 3]
recursivelyCalcLLR get LLR_value [2, 0, 3],[2, 1, 3]
recursivelyCalcLLR B_value get [3, 0, 6]
recursivelyCalcLLR set LLR_value [3, 0, 7]
main func B_value set unfrozen bit,[3, 0, 7], LLR= 8.0
recursivelyUpdateB B_value get [3, 0, 6],[3, 0, 7]
recursivelyUpdateB B_value set [2, 0, 3]
recursivelyUpdateB B_value get [2, 0, 2],[2, 0, 3]
recursivelyUpdateB B_value set [1, 0, 1]
recursivelyUpdateB B_value get [1, 0, 0],[1, 0, 1]
recursivelyUpdateB B_value set [0, 0, 0]
recursivelyUpdateB B_value set [0, 1, 0]
recursivelyUpdateB B_value set [1, 1, 1]
recursivelyUpdateB B_value get [1, 1, 0],[1, 1, 1]
recursivelyUpdateB B_value set [0, 2, 0]
recursivelyUpdateB B_value set [0, 3, 0]
recursivelyUpdateB B_value set [2, 1, 3]
recursivelyUpdateB B_value get [2, 1, 2],[2, 1, 3]
recursivelyUpdateB B_value set [1, 2, 1]
recursivelyUpdateB B_value get [1, 2, 0],[1, 2, 1]
recursivelyUpdateB B_value set [0, 4, 0]
recursivelyUpdateB B_value set [0, 5, 0]
recursivelyUpdateB B_value set [1, 3, 1]
recursivelyUpdateB B_value get [1, 3, 0],[1, 3, 1]
recursivelyUpdateB B_value set [0, 6, 0]
recursivelyUpdateB B_value set [0, 7, 0]
```