5G NR 系统 Polar 码设计详解以及 CA_SCL 译码实现

刘杰 hahaliu2001@gmail.com

1 前言

Polar 码在 2009 由土耳其教授 Erdal Arıkan 发明，由于具有性能优越，理论可达香农限，并且编译码算法简单等优点迅速成为编码界的研究热点。我查到的 3GPP 提案里 2016 年 3GPP 就开始讨论把 Polar 码引入 5G 标准，最终选择 Polar 码替代传统的卷积码，用于 DCI，UCI，BCH 等控制信道的短码信道编码，业务信道 PDSCH 和 PUSCH 选择了 LDPC 码。

5G 标准中的 Polar 码相比 Erdal Arıkan 发明的 Polar 码不完全相同，增加了一些新的特性。

本文第一部分详细描述 5G polar 编码的新特性，比如分布 CRC（Distributed CRC）用于提前判决并且终止 Polar 译码， Polar 速率匹配中的 channel interleaver 等。

第二部分详细描述 5G polar decoder，包括：

1. N =8 的 Polar SC decoder 例子详解。
2. 两个 SC decoder 实现
    a. 第一个 SC decoder follow “List decoding of polar codes”中的描述
    b. 第二个 SC decoder 完全 follow N =8 的 Polar SC decoder 例子里的过程

    两个 SC 实现的结果相同，只不过逻辑上有些区别。
3. 两个 SC decoder memory 优化，节约 LLR 和 B 存储大小从 N*log2(N)到 N
4. 5G NR LLR-based SCL decoder
    a. 支持 Distributed CRC based 提前终止
    b. 支持 parity bit check based 提前终止
    c. 支持 CRC-aided

本文不涉及 Polar 码的原理和性能分析.

我已经完成了 Polar 编码以及 SC， SCL 译码 Python 实现和测试，代码在：

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

1.1 参考资料

【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 码设计

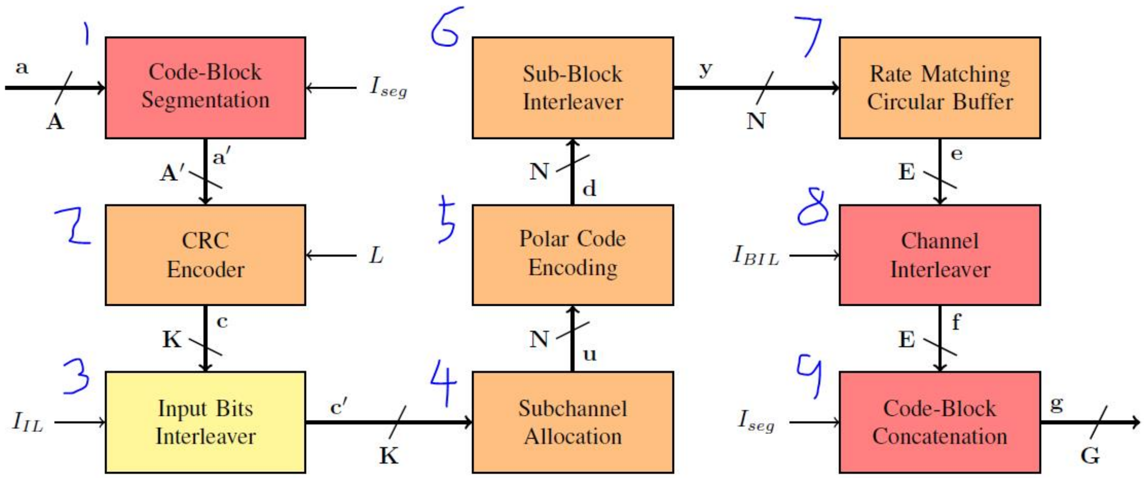


Fig. 6: 5G polar codes encoding chain; yellow, red and orange blocks are implemented in downlink, uplink and both respectively.

上图来自【2】“Design of Polar Codes in 5G New Radio”，描述了 5G polar 从 CRC 到速率匹配的处理流程。

本节主要描述 5G polar 中一些新的特性。

2.1 5G Polar interleaver 和 Distributed CRC 设计

第 3 步 Polar interleaver(38.211 5.3.1)是针对 Distributed CRC 的设计，只用于下行 DCI 和 BCH，目的是 UE 搜索 PDCCH 和可以在完成 Polar 译码就判决 CRC，如果 CRC 错就提前终止 Polar 译码

2.1.1 了解 CRC

CRC 可以简单认为是多项式除法，满足加法律，

也就是说如果 C=C+C2+...CN，CRC=CRC+CRC2+...+CRCN，

还有就是如果 C 为全零，CRC 结果也是全 0

$\frac{C}{poly} = \frac{C1+C2+\cdots+CN}{poly} = CRC1 + CRC2 + \cdots + CRCN = CRC$。（1.1.1）

$\frac{all\ zero\ bits}{poly} = [0,0,\ldots 0]$（1.1.2）

2.1.2 Distributed CRC 例子：4 bit CRC polynomial Distributed CRC

以 4bit CRC [0，1, 0, 1]多项式为例，如果输入 8 bits 信息位，CRC 编码后输出为 8bits 信息位+4bit CRC, CRC 生成矩阵为：

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

其中

U_{1X8}为输入 8 bit 行向量，

P_{1X12}为输出 12bit 行向量

I_{8x8}为 8X8 对角矩阵，

G_{8x4}为 CRC 生成矩阵：$G_{8X4} = \begin{bmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{bmatrix}$

其中每一行代表一个 bit 是否参与到 CRC 计算，1 代表参与，0 代表不参与。

比如第一行[1 0 1 0]，表示[a,0,0,0,0,0,0,0]向量做 CRC，

如果 a=1, CRC bit0 and bit2 =1

如果 a=0, CRC bit0 and bit2 =0

CRC bit 1 and 3 不管 a 值如何，结果都是 0

把G_{8x4}中’1’改为信息位索引，把’0’改为-1，表示无效，得到：

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

其中每一列中非‘-1’值表示哪些信息位用于这个 CRC bit 的计算。

比如，第一列[0, -1,2,-1,-1,-1,6,-1]表示信息位[0,2,6]用于计算 CRC bit0

同理可以得到，

对 CRCbit1，输入 bit [1,3,7]参与到 CRC 计算

对 CRCbit2，输入 bit [0,4,6]参与到 CRC 计算

对 CRCbit3，输入 bit [1,5,7]参与到 CRC 计算

也就是说，如果输入到 polar encoder 的序列为[0,2,6,CRC0,1,3,7,CRC1,4,CRC2,5,CRC3], polar 译码时译码出前四个 bit，就可以计算 CRC0 来判决 CRC 校验是否成功，如果失败就提前终止 polar 译码。

这就是 polar interleaver 的目的，通过分散传送信息 bit 和 CRC bit 实现 Distributed CRC.

polar interleaver table 用来描述如何生成类似[0,2,6,CRC0,1,3,7,CRC1,4,CRC2,5,CRC3]序列。

上面的例子是针对 8 bit 信息位，如果输入 7bit 信息位，矩阵的最后 7 行可以用于计算 CRC，8 bit 输入到 polar encoder 的序列[0,2,6,CRC0,1,3,7,CRC1,4,CRC2,5,CRC3]去掉 0，其余 index-1 就得到 7bit 序列[1,5,CRC0,0,2,6,CRC1,3,CRC2,4,CRC3]。同理可以得到所有小于 8bit 输入的 polar encoder 的序列。

也就是说只要提供一个 8 比特的交织序列表，就可以计算出来所有小于 8 bit 的交织序列。这就是为什么 5G 协议只提供最到 164 的交织序列表。5G 协议最大支持的 DCI 和 BCH bit 长度为 KIL_max=164 – 24 CRC bit = 140

下面 Python code 用于生成上面两个矩阵

```python
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
```

2.1.3 5G polar interleaver 设计

5G 38.212 Table 5.3.1.1-1: Interleaving pattern 表是基于 CRC24C 和最大长度 140 的信息位产生的。

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

下面的 python 函数用于生成 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
```

运行下面 code 可以生成 38.212 Table 5.3.1.1-1: Interleaving pattern

```
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

参考 38.212 5.3.1.2 Polar encoding。

对于长度 N 的 Polar 码比特序列，每个 bit 的可靠性不同，Subchannel Allocation 的作用就是把信息比特放到可靠性高的位置，可靠性低的比特为 Frozen bit，放 0。

对下行信道 N 取值范围为 2^[5,6,7,8,9], 对上行信道 N 取值范围为 2^[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 $W(Q_i^{N_{max}})$

给出了 1024 长度的码序列每个 bit 的可靠性。

$\overline{\mathbf{Q}}_I^N$ 是信息比特在 polar 码中放置位置列表，38.212 5.4.1.1 速率匹配部分介绍了如何产生这个列表。针对速率匹配的打孔（Puncturing）、缩短（Shortening）和重复（Repetition），产生 $\overline{\mathbf{Q}}_I^N$ 的方式不同。

2.3 5G Polar code Encoder 采用 natural order encoder

参见“A Golden Decade of Polar Codes: From Basic Principle to 5G Applications”2.2，有两种 Polar 编码 order，

一种是位反序(bit-reversal) order encoder,

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

一种是自然顺序（natural）order encoder，

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

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

B_N是位反序矩阵（bit-reversal matrix），$B_N = B_N^T = B_N^{-1}$

并且 $B_N F_2^{\otimes n} = F_2^{\otimes n} B_N$

参见

“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 采用的是自然顺序（natural）order encoder，而 SC Polar decoder 算法针对的是位反序(bit-reversal) order encoder。

由公式 1.3.2，1，3，3 得到：$x_1^N B_N = u_1^N F_2^{\otimes n} B_N = u_1^N B_N F_2^{\otimes n}$

也就是说 SC 译码前输入的 LLR 数据需要做位反序（bit-reversal），然后可以使用标准的 polar SC decoder.

2.4 Polar 速率匹配打孔（Puncturing）和缩短（Shortening）的区别

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

上面是 38.212 5.4.1.2 速率匹配打孔和缩短的实现。

打孔和缩短区别：

- 打孔是去掉前 N- E 个 bit，里面包含有效信息。
- 缩短是去掉后面 N-E 个 bit，里面的 bit 信息是全零。
- 解速率匹配时要补全 N 个 LL R 信息，因为打孔 bit 值未知，LLR=0.
- 缩短 bit 值为 0，LLR 需要取极大值

下面一组 polar 数据经过 polar encoding 和速率匹配 Sub-block interleaving 的输出结果。其中 K=51， E= 216, N=256，K/E < 7/16，需要打孔前 40 个 bit，里面‘0’，‘1’混杂

```
[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,.....]
```

下面是 polar 数据经过 polar encoding 和速率匹配 Sub-block interleaving 的输出结果.

其中 K=59,E=108, N=128， K/E>7/16，需要缩短去掉后 20 bit，这些 bit 全零。

```
[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 速率匹配后 Channel interleaver

38.212 5.4.1.3 Interleaving of coded bits 定义了 Triangle interleaver，用于 Polar rate matching repetition/puncturing/shortening 后的 channel interleaving.

我查到的最早的关于 Polar channel interleaver 的 3GPP 提案是“3GPP R1-1612656, Interleaver for Polar codes, Interdigital, RAN1#87, Reno, USA, Nov. 2016”，证明了对于高阶调制(16QAM, 64QAM)，交织后的性能比没有交织性能好大概 0.5dB。

我理解的原因是 polar 解码是根据 bit 的可靠性从高到低进行，先 decode 可靠性高的比特，然后 decode 可靠性低的 bit。高阶调制比如 16QAM/64QAM 破坏了 bit 的可靠性。比如接收 16QAM 的 4 个 bit [a0,a1,a2,a3], SNR_a0 = SNR_a1 > SNR_a2=SNR_a3，不同比特 SNR 不同破坏了 Polar bit 的可靠性排序。

Polar channel interleaver 只在高阶调制有效，低阶无效。

下行 DCI 和 BCH 使用 QPSK，所以协议跳过了 Polar channel interleaver。

上行 UCI 支持在 PUSCH 传输，而 PUSCH 支持高阶调制，所以用到 Polar channel interleaver

2.5.1 3GPP 关于 Polar channel interleaver 的研究

多家公司提出了不同的 channel interleaver 方案，其中 Qualcomm 的 Triangle interleaver 和华为的 row-column interleaver 性能最好，最后的结果选择了 Qualcomm 的 Triangle interleaver。

详细的 3GPP 讨论可以参考：

“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”

Quancomm 在“Efficient interleaver design for polar codes”专利中已经保护了这个 Triangle interleaver，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 5G polar 配置汇总

	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

大部分关于 Polar 译码的文章就基于两个 paper：

1. “List decoding of polar codes”
   介绍了两种 Polar 译码算法：Successive cancellation (SC) decoding 和 SC list (SCL) decoding。SC 可以认为是 List size =1 的 SC L 实现.
2. “LLR-based successive cancellation list decoding of polar codes”
   给出简化的适合产品实现的 LLR 的计算公式
   以及路径权重（path metric）的计算公式

5G 采用的 Polar 译码算法是 LLR based CRC-aided + distributed CRC-aided + Parity-check-aided SCL

- CRC-aided：L 个 path 估计结束后，每个 path 做 CRC check，用于 DL DCI, BCH, UL DCI
- distributed CRC-aided：SCL 译码过程中计算 check CRC bit，用于 DL DCI 和 BCH
- Parity-check-aided：SCL 译码过程中做 Parity bit check，用于 UL DCI small payload size

我自己的学习体会，理解 SC L 必须首先理解 SC，理解 SC 最好从一个 N=8 的例子开始学习。

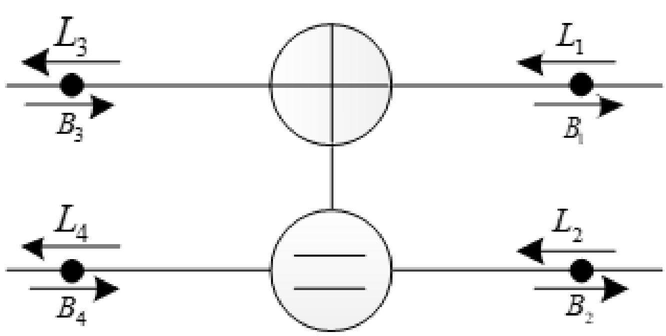
本章包括如下部分：

1. 一个 N=8 的 Polar SC decoder 例子，详细描述 SC decoding 过程
2. 两个 SC decoder 算法实现
    a. 第一个采用“List decoding of polar codes”中的实现，每次 LLR 和 B 递归，会计算当前 layer 所有 branch 的值
    b. 第二个采用 N=8 的 Polar SC decoder 例子中介绍的流程，每次 LLR 和 B 递归，只会计算相关的 branch。

    更详细的区别可以参考附录 A 1 和 A2 中给出的两个 SC 算法针对 N=8 输出的 log

3. 两个 SC decoder memory 优化的方法
   LLR 和 B memory 大小从 N*log2(N)优化到 N
4. 5G NR LLR-based SCL decoder

3.1 单位因子图概率传递公式



上面是 polar 码的单位因子图，在该因子图上有 8 个值，分别是 代表向左传递的 LLR 值， 代表向右传递的硬比特信息。

已知L_1，L_2，按顺序计算$L_3 \rightarrow B_3 \rightarrow L_4 \rightarrow B_4 \rightarrow B_1\ and\ B_2$的流程:

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

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

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

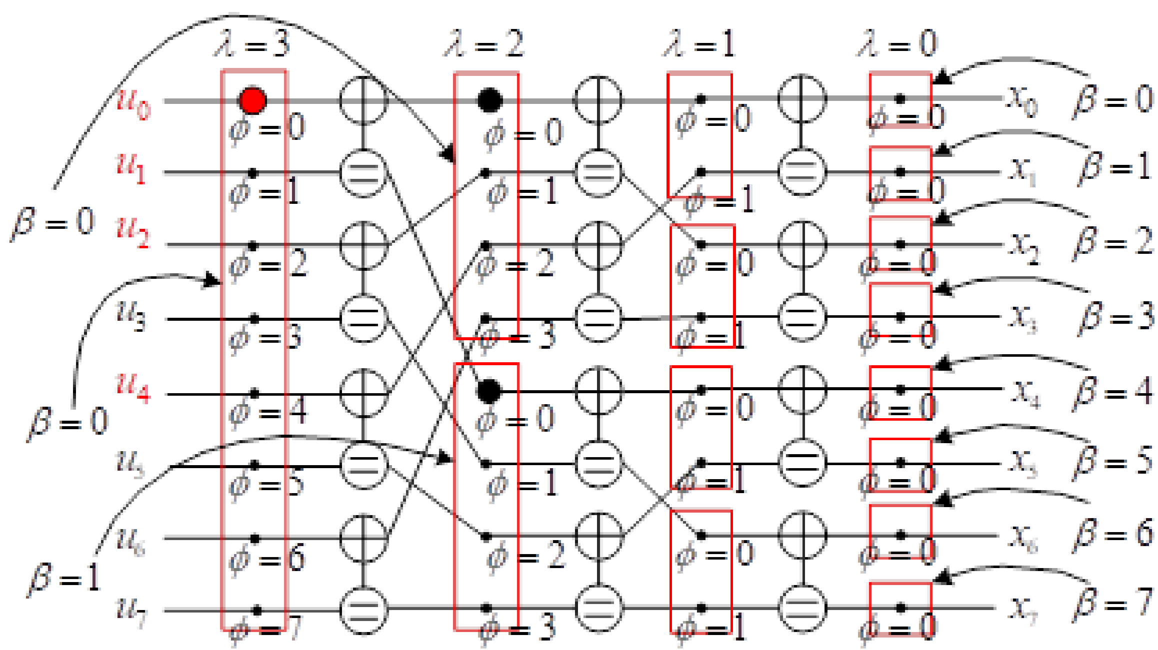
Step 4:$\mathrm{B}_4 = \begin{cases} 0 & \mathrm{L}_4 > 0 \\ 1 & \mathrm{L}_4 < 0 \end{cases}$

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

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

上面公式来自“LLR-based successive cancellation list decoding of polar codes”

3.2 N=8 SC decoder 递归顺序例子



上图λ代表 layer，β代表 branch，ϕ代表 branch 中的 phase

总 layer 数=log2(N) + 1= 4，上图从左到右为 layer 3-2-1-0

对 layer=s, 有2^{3-s} 个 branch，每个 branch 有2^s 个 phase 值

对 layer [3,2,1,0], branch 个数为[1,2,4,8], 每个 branch 内 phase 值个数[8,4,2,1]

上图中每个点的位置可以用(layer, branch,phase)表示。

每个点的 LLR 值用 LLR[layer, branch, phase)]表示

每个点的 bit 值用 B[layer, branch, phase)]表示。

上图红色四个点u_0, u_1, u_2, u_4对应着 B[3,0,0], B[3,0,1], B[3,0,2], B[3,0,4]为 frozen bit

在 polar decoder 中已知的是 layer=0 的 LLR 值(通过信道均衡得到)

SC decoder 译码递归顺序为

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]

总结 LLR 处理过程：

- If phase= even
  - 需要递归计算 LLR[layer-1, 2*branch,phase//2] 和 LLR[layer-1, 2*branch+1, phase//2]值
  - 递归返回后，用 LLR[layer-1, 2*branch,phase//2], 和 LLR[layer-1, 2*branch+1, phase//2]计算 LLR[layer,branch,phase]
- If phase==odd
  - 不需要递归，
  - 用 LLR[layer-1, 2*branch,phase//2], LLR[layer-1, 2*branch+1, phase//2]和 B[layer, branch, phase-1]计算 LLR[layer,branch,phase]

总结 B 处理过程：

- 基于 LLR[layer,branch,phase]计算 B[layer,branch,phase]
- If phase== odd，进入递归计算
  - 用 B[layer,branch,phase-1]和 B[layer,branch,phase]计算得到 B[layer-1, 2*branch,phase // 2]和 B[layer-1, 2*branch, phase // 2]
  - 如果 phase // 2 is odd，继续递归计算下一个 layer

3.3 LLR-based Polar SC(SUCCESSIVE CANCELLATION) decoder

下面是 python 实现的 SC decder，参考“List decoding of polar codes,”

首先是 create a Path 类，里面包含 LLR matrix 和 B matrix。矩阵 memory size (log2(N)+1)*N

初始化 LLR[layer=0]值为输入的 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
```

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

SC_Path_no_opt 类中 LL R 和 B 占用的 memory 大小为：

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

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

其中，

总 layer 数=log2(N) + 1

每个 layer 有$2^{\log 2(N)-layer}$ branch，每个 branch 有2^{layer} phase 值

下面的方法用于优化 LLR 和 B memory

3.5.1 Optimization 1

这是“List decoding of polar codes,” III. SPACE-EFFICIENT SUCCESSIVE CANCELLATION DECODING 介绍的优化。

仔细观察 N=8 的 SC decoder 例子。

当读写操作 LLR[layer, branch, s]时：

1. 所有 phase < s 的 LLR[layer, branch, phase]值在后面的 decoding 中都不会再使用
2. 所有 phase > s 的 LLR[layer, branch, phase]值都还没有参与到 dedoding

比如，对 layer=2 branch=0 的 LLR 值，

- get/set LLR[2,0,0]时，LLR[2,0,1]，LLR[2,0,2]，LLR[2,0,3]还没有参与到 decoding
- get/set LLR[2,0,1]时, LLR[2,0,0]在 decoding 中不再使用，LLR[2,0,2]，LLR[2,0,3]还没有参与到 decoding

也就是说对任何一组[layer, branch]，只需要保持一个 LLR 值的大小。

这样每个 layer 只需要$2^{\log2(N)-layer}$个值，

总的 LLR size = 1+2+4+...+N = 2*N-1

B 值 memory 使用类似。

例如 layer=2, branch=0 B 值.

所有 B[2,0,0] and B[2,0,1] get/set 操作之间都不会使用 B[2,0,2]和 B[2,0,3].

当开始使用 B[2,0,2]和 B[2,0,3]， B[2,0,0]和 B[2,0,1] 都不会再用到。

也就是说对任何一组[layer, branch]，只需要保持 2 个 B 值的大小。

总的 B size = 2*(1+2+4+...+N) =2*(2*N-1)

另外还需要额外 N 个值保存最后一个 layer 的输出结果

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

这是我在分析 Memory optimization 1 方法是发现的。

对 B memory：

Layer =0 的 N 个 B values 是不需要的,这样可以省 N 个值

总 B memory size = 2*(1+2+4+...+N/2) = 2*(N-1)

对 LLR memory.

Layer=0 的 N 个 LLR 值是外部输入，不需要额外的 memory 存储这些值，这样可以省去 layer=0 的 N 个值。.

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) 算法来自 “List decoding of polar codes”, 不过我读了几遍还是没有搞明白作者的思路，所以还是按照自己的思路设计。

从软件角度 SCL 是解决“Tree splitting” 问题。

SC decoder 只支持一个 Path，每一个都硬判决 B 值为 0 或 1。

SCL decoding 支持 L paths，

L paths SCL 处理需要：

1. 一个 split path 的方法
   SCL 按照 B 值取 0 和取 1， split path 为两个 path
2. 一个 path 度量值 PM(path metric)来标识 path 质量
   每个 path 都有一个 PM 值。

   当 tree 分叉超过 L paths 时，需要通过 PM 值选择最好的 L 个 paths.

PM 公式来自于 “LLR-based successive cancellation list decoding of polar codes” equation (12),

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

PM 值越小代表 path 质量越好。

5G SCL 基本思路：

1. 创建 SCL_path 类
   a. 继承于 SC_path,
   b. 增加当前 PM 值和两个 nextPM 值代表 split path to B=0 and B=1
2. 创建 Pathlist 类管理 L SCL_Path:
   a. active/de-active path
   b. split and then clone path to new path
3. 创建函数用来选择最好的 L 个 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
```

```python
        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)
```

```python
        if err == 0:
            return np.array(ck,'i1'), True

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

```python
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.

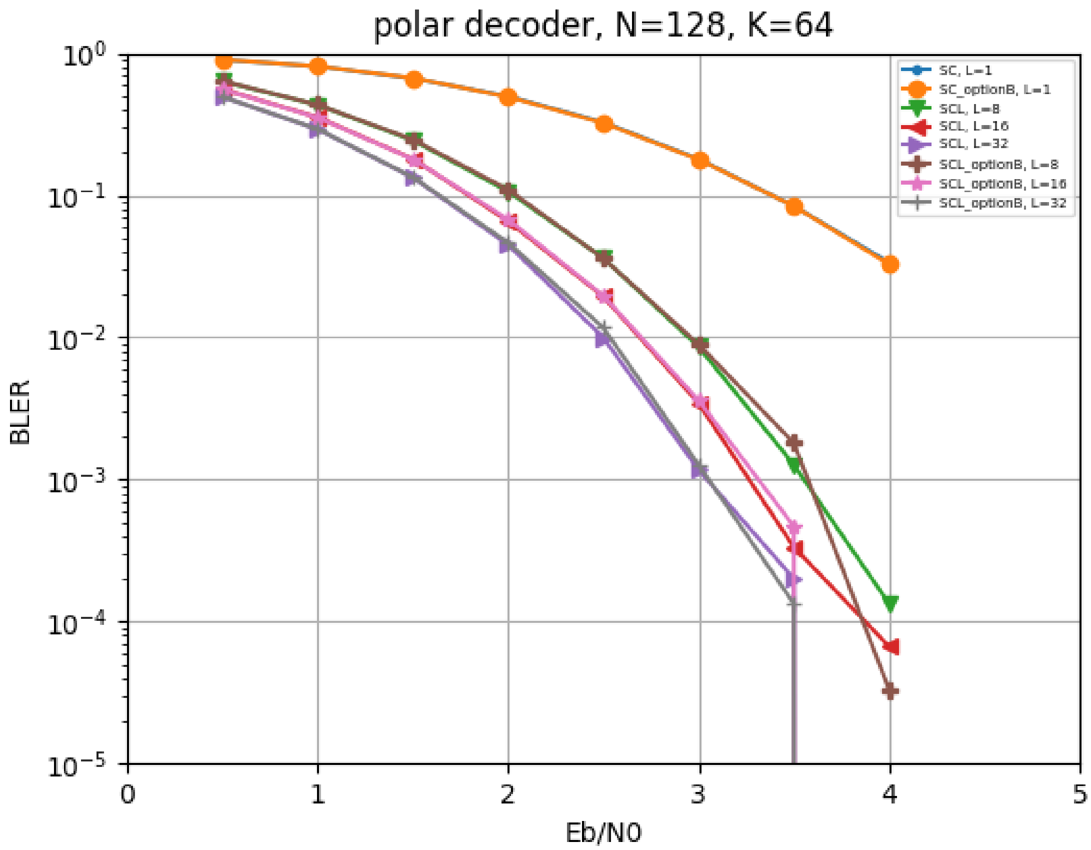
In the simulation

- It simulates 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]
```