NPFCode Implementation

Yasin OZTURK

December 2018

1 Introduction

NPFCode encodes given data V by using different D values to output payload (PL) and disambiguation information (DI) in varying sizes. Increasing the D size reduces the size of DI but also increases the size of PL. When D size gets larger than 8, it becomes harder to compute DI and PL because of the byte-aligned memory architecture. NPFCode splits incoming data by D-bits chunks and outputs each chunk's minimum binary representation (MBR) as payload and corresponding code length in bits as disambiguation information. MBR removes the most significant bit and returns the remaining value with its length. A pseudo-code is given below for calculating MBR value of a unsigned char value.

Algorithm 1: MBR_Char(value, mbrValue, mbrLength)

Input: $value \in [0, 255]$
Output: $mbrValue$: Minimum Binary Representation of $value$.
$mbrValue \in [0, 127]$
$mbrLength$: The number of bits after the most significant bit.
$mbrLength \in [1, 7]$

$mbrValue = 0, mbrLength = 0;$
if $value == 0 \mid\mid value == 1$ **then** $return$;
$orgValue = value;$
while $value \mathrel{!}= 1$ **do**
    $value = value >> 1;$
    $mbrLength = mbrLength + 1;$
end
$bitMask = (1 << mbrLength) - 1;$
$mbrValue = orgValue \ \& \ bitMask;$

NPFCode iterates through V by chunks of D-bits long data and calculates responding MBR values and lengths. Since there can be only 2^D many distinct values to calculate MBR for, these MBR values of every D-bit long data can be pre-processed and saved. It should be noted that resulting MBR lengths can never be equal to D. In order to output PL, all MBR values must be concatenated. Since MBR values can have varying lengths, it is not computationally cheap to concatenate each MBR value. The reason behind that is the resulting concatenated value's length can be different than 8. To resolve the inefficiency caused by shifting concatenated value to fit in 8 bits of data, a Huffman table will be implemented to store the output values of the concatenation and the remaining bits.

Huffman table can be considered as a two-dimensional matrix where the rows of the matrix represent the reverse MBR of the remaining bits of concatenated value and the columns represent the next D-bits length of data. Since remaining bits of data can have at most 7 bits, reverse MBR of the remaining bits can have up to 8 bits of data. The reason behind to get reverse MBR of values is, same MBR values can have different lengths and could be derived from different values. In order to eliminate the ambiguity, it is required to use the original data as an index. It should be noted that 0 and 1 cannot be a valid index since reverse MBR cannot produce these values. When there is no remaining data left, 1 will be used as the index for columns of Huffman table.

$v_1 = 13$ (1101) => MBR(13) = value: 5 (101), length: 3

$v_2 = 37$ (100101) => MBR(37) = value: 5 (00101), length 5

Algorithm 2: ReverseMBR_Char(mbrValue, mbrLength)

Input: $mbrValue \in [0, 128]$
$mbrLength \in [1, 7]$
Output: Reversed MBR value of $mbrValue$, $\in [0, 255]$

```
return (1 << mbrLength) | mbrValue;
```

Each cell of the Huffman table holds MBR length of the corresponding column index (*mbrLength*), output value (*output*), a flag indicating whether output value has more or equal to 8 bits of data (*canOutput*) and reverse MBR value of the remaining bits (*nextTableId*). The pseudo-code for creating the MBR Huffman table for D=8 is as follows:

Algorithm 3: CreateMBRHuffmanTable_D8()

```
Input: permAlph[v], v ∈ [0, 255] : permutated alphabet
Output: mT[t, r], t ∈ [1, 255], r ∈ [0, 255]
for (t = 1; t ≤ 255; t++) do
    inV = 0, inL = 0 ;              /* Initital value, initial length */
    if t != 1 then MBR_Char(t, inV, inL) ;
    for (r = 0; r ≤ 255; r++) do
        tL = 0, rL = 0, rem = 0 ; /* Total length, remaining length,
         remaining */
        v = permAlph[r] ;                                  /* value */
        if v ≤ 253 then
            mV = 0, mL = 0 ;              /* MBR value, MBR length */
            v = v + 2;
            MBR_Char(v, mV, mL);
            mT[t, v].mbrLength = mL;
            tL = inL + mL;
            if tL ≥ 8 then
                mT[t, v].canOutput = 1;
                rL = tL − 8;
                mT[t, v].output = (inV << (8 − inL)) | mV >> rL;
                /* Get the last rL bits of mV, these bits are
                   not outputted and will be used in the next
                   iteration                                    */
                rem = mV & ((1 << rL) − 1);
            else
                mT[t, v].canOutput = 0;
                rL = tL;
                rem = (inV << mL) | mV;
            end
            mT[t, v].nextTableId = ReverseMBR_Char(rem, rL);
        else
            mT[t, v].canOutput = 1;
            rL = inL;
            if value == 254 then
                mT[t, v].output = inV << (8 − inL);
            else if value == 255 then
                mT[t, v].output = (inV << (8 − inL)) | (1 >> rL);
                if rL > 0 then rem = 1;
            end
            mT[t, v].nextTableId = ReverseMBR_Char(rem, rL);
            mT[t, v].mbrLength = 8;
        end
    end
end
```

Created MBR Huffman table has 256 rows representing the all possible remaining bits from the last D-bit length chunk of V, and has 256 columns representing the all possible values that current D-bit length chunk of V can have. Each cell of the table holds 4 bytes of data. The resulting table's total size can be calculated as 256 kiB for D=8.

MBR Huffman table helps us to construct PL, but in order to decode the PL, DI must be constructed as well. DI holds the corresponding MBR lengths of each D-bit length chunk of V. These length representations is shown in the table below:

MBR Length	Codeword
1	0000001
2	000001
3	00001
4	0001
5	001
6	01
7	1
8	0000000

Table 1: MBR Length representations for D=8 in DI

Creating DI also suffers from the memory architecture. Each MBR length codewords must be concatenated to create DI, since most of the codewords contain less than 8 bits of data, it is not computationally cheap to achieve on-the-go concatenation. However, these DI values can also be pre-processed and stored in a Huffman table. Similar to the previously implemented MBR Huffman table, this table also has 256 rows. Each row value corresponds to the reverse MBR value of the remaining values from previous MBR length codeword representations. Columns of the DI Huffman table represents every possible MBR lengths, ranging from 1 to 8. Each cell of the table holds a flag indicating whether the current concatenated codeword's length is bigger than or equal to 8 ($canOutput$), an 8-bit long output value ($output$), and reverse MBR value of the remaining bits to indicate the next row of the table ($nextTableId$). DI Huffman table takes 6 kiB of space for D=8. The pseudo-code for creating DI Huffman table for D=8 is given below:

Algorithm 4: CreateDisInfoHuffmanTable_D8()

```
Output: dT[t, r], t ∈ [1, 255], r ∈ [0, 7]
for (t = 1; t ≤ 255; t++) do
    inV = 0, inL = 0 ;          /* Initital value, initial length */
    if t != 1 then MBR_Char(t, inV, inL) ;
    for (r = 0; r < 8; r++) do
        v = 1, l = 8 − (r + 1) ;                    /* value, length */
        if r == 7 then v = 0, l = 7 ;
        tL = inL + l, rL = 0, rem = 0 ;    /* Total length, remaining
         length, remaining */
        if tL ≥ 8 then
            rem = v;
            rL = tL − 8;
            dT[t, r].canOutput = 1;
            dT[t, r].output = inV << (8 − inL);
            if rL == 0 then
                dT[t, r].output |= v;
                rem = 0;
            end
        else
            rL = tL;
            dT[t, r].canOutput = 0;
            rem = (inV << l) | v;
        end
        dT[t, r].nextTableId = Reverse_MBR(rem, rL);
    end
end
```

Encoding of a file is done by utilizing previously computed Huffman tables. Each D-bit long chunk of V is read and fed to the MBR Huffman table. If selected cell indicates an available output, the indicated value will written to the PL. Then, cell's $mbrLength$ value is fed to DI Huffman table. This cycle continues till file has been completely traversed.

Some information about the encoding process is placed as a header to the DI in the encoding phase. This header stores information such as remaining data, skip amount, D value and whether encoding is seeded or not. Remaining data signifies how many of the file's last bytes have not been encoded. When D size is 8, this information is irrelevant since all of the data can be traversed byte by byte. But when D is set to 20, there can be cases where the last 2 bytes of V might be left out. In such cases, remaining data is set to signify how many byte worth of data is not included in the PL. These remaining bytes of V is added to end of the header. In the decoding phase, these left out bytes of V is added to the end of the decoded file.

In some cases, when all the file has been traversed, DI Huffman table's and MBR Huffman table's current cells might not indicate an available output but there might still some bits left that need to be outputted. After the traversing of V has been completed, these cells are checked. MBR values of these cells row value is written to corresponding files. It is known that these bytes last $8 - MBRlength$ many bits are 0. When decoding the last written DI byte, this many zeroes might result in incorrect number of codeword lengths. Let's say that last DI Huffman table's cell is in row 2. MBR value of 2 is 0 and MBR length is 1. Last 7 bits of last DI byte is not used and is zero. When D=8, 0000000 bit sequence signifies codeword length of 8. This last codeword length must be skipped when decoding the PL, since there is no corresponding codeword in PL. The number of skip that most be done when reading the last DI byte must be written to the header. Header byte structure is given in the table below.

D Value	Skip Amount	Remaining data	Seeded
000 → D4	00 → No Skip	00 → No Remaining	0 → Not Seeded
001 → D8	01 → Skip 1	01 → Remaining 1	1 → Seeded
010 → D12	10 → Skip 2	10 → Remaining 2	
011 → D16	11 → Skip 3	11 → Remaining 3	
100 → D20			

Table 2: Header byte structure

Algorithm 5: Encode_D8()

Input: $file$: Input file
mT : Pre-computed MBR table
dT : Pre-computed disambiguation information table
Output: pL : Payload file
dI : Disambiguation Information file

$fL = length(file)$;
$cP = 0, pLI = 0, dII = 4, dITI = 1, mTI = 1$; `/* current position (`cP`), payload index(`pLI`), disambiguation information index(`dII`), disambiguation information table index(`$dITI$`), mbrtable index (`mTI`) */`
while $cP < fL$ **do**
    $v = file[cP]$;
    $cP = cP + 1$;
    $pL[pLI] = mT[mTI, v].output$;
    $pLI = pLI + mT[mTI, v].canOutput$;
    $mTI = mT[mTI, v].nextTableId$;
    $dI[dII] = dT[dITI, mT[mTI, v].mbrLength - 1].output$;
    $dII = dII + dT[dITI, mT[mTI, v].mbrLength - 1].canOutput$;
    $dITI = dT[dITI, mT[mTI, v].mbrLength - 1].nextTableId$;
end
$skip = 0, dType = 32, remCount = 0, seeded = 0$; `/* skip amount, D size, remaining data count */`
if $dITI \; != 1$ **then**
    $lastDIOut = 0, lastDILen = 0$; `/* last disambiguation information output, last disambiguation information length */`
    $MBR\_Char(dITI, lastDIOut, lastDILen)$;
    $dI[dII] = lastDIOut << (8 - lastDILen)$;
    if $mTI == 1$ **then** $skip = 8$;
end
if $mTI \; != 1$ **then**
    $lastPLOut = 0, lastPLLen = 0$; `/* last payload output, last payload length */`
    $MBR\_Char(mTI, lastPLOut, lastPLLen)$;
    $pL[pLI] = lastPLOut << (8 - lastPLLen)$
end
$dI[0] = dType \mid skip \mid remCount \mid seeded$;
$dI[1] = 0, dI[2] = 0, dI[3] = 0$;

Decoding phase reconstructs the original data from PL and DI. Decoding schema also utilizes pre-computed Huffman tables for efficiency and speed. In this implementation, 3 tables have been used, 2 of them being more significant and used in converting codeword lengths and codewords, the other one for calculating the current bit position in the read PL data.

The table made for parsing DI has only 6 rows, corresponding to the number of zero bits remained from the previous iteration. The remaining zero bit count cannot be 7, because 7 zero bits corresponds to codeword length of 1. Each row has 256 columns for the current DI byte. This table's cell holds the codeword lengths to split the PL. Each cell can hold at most 8 length.When all the bits of the current DI byte is 1, each bit signifies 7 bit long codeword length, which can be at most 8. PL data must be split according to these codeword lengths to achieve decoding. Reverse DI Huffman table takes 17.5 kiB of space for D=8 The pseudo-code for creating the table for D=8 is given below:

Algorithm 6: CreateReverseDisInfoHuffmanTable_D8()

```
Output: revDIT[t, r], t ∈ [0, 6], r ∈ [0, 255]
for (t = 0; t ≤ 6; t++) do
    inL = t ;                                    /* initial length */
    tL = inL + 8 ;                               /* total length */
    bM = 32768 ;              /* bit mask :  10000000 00000000 */
    for (r = 0; r ≤ 255; r++) do
        revDIT[t, r].outputC = 0 ;               /* output count */
        alignedV = r << (16 − tL) ;        /* left aligned value */
        zeroC = 0 ;                                  /* zero count */
        for (i = 0; i < tL; i++) do
            lmBit = (alignedV & bM) == 0?0 : 1 ; /* left most bit */
            alignedV = alignedV << 1;
            if lmBit == 0 then
                zeroC++;
                if zeroC == 7 then
                    revDIT[t, r].outputs[revDIT[t, r].outputC] = 8;
                    revDIT[t, r].outputC++;
                    zeroC = 0;
                end
            else
                revDIT[t, r].outputs[revDIT[t, r].outputC] = 7 − zeroC;
                revDIT[t, r].outputC++;
                zeroC = 0;
            end
        end
        revDIT[t, r].nextTableId = zeroC;
    end
end
```

The PL data byte must be split with the codeword lengths obtained from the reverse DI Huffman table. For each byte, there are 8 positions to split the data. All possible combinations of this splitting can be calculated prior to decoding. This pre-computed Decode Huffman table has 256 rows for every possible PL data byte and 8 rows for each possible codeword length. The read byte is split according to the codeword length and remaining bits are stored to find the next cell in the table. Decode Huffman table takes 6 kiB of space for D=8. The pseudo-code for creating the table for D=8 is given below:

Algorithm 7: CreateDecodeHuffmanTable_D8()

Input: $invPermAlph[v], v \in [0, 255]$: inverse permutated alphabet
Output: $decT[t, r], t \in [0, 255], r \in [0, 7]$

```
for (t = 0; t <= 255; t++) do
    for (r = 0; r < 8; r++) do
        l = r + 1;
        code = t >> (8 - l);
        if l != 8 then
            codeVal = ReverseMBR_Char(code, l);
            decT[t, r].output = invPermAlph[codeVal - 2];
        else
            if code == 0 then  decT[t, r].output = invPermAlph[254] ;
            else if code == 1 then
              decT[t, r].output = invPermAlph[255] ;
        end
        decT[t, r].remaining = t << l;
        decT[t, r].remainingLength = 8 - l;
    end
end
```

A trivial Huffman table can be used to determine whether the current PL data is completely decoded or not. This Length Huffman table raises a flag when all the bits of the current byte is traversed and helps to get the next byte of the PL. This table has 8 rows corresponding to the number of bits that are not used is the current PL byte, and 8 columns for the codeword lengths. When the sum of the remaining bits and codeword length passes 8, *getNextByte* value of the cell's is set to 1. This pre-computed table helps to remove if conditions from the main decoding loop. Length Huffman table takes 128 bytes of space for D=8. The pseudo-code for creating Length Huffman table for D=8 is given below:

Algorithm 8: CreateLengthHuffmanTable_D8()

```
Output: lenT[t, r], t ∈ [0, 7], r ∈ [0, 7]
for (t = 0; t ≤ 7; t++) do
    for (r = 0; r ≤ 7; r++) do
        tL = t + r + 1 ;                          /* total length */
        if tL ≥ 8 then
            lenT[t, r].getNextByte = 1;
            lenT[t, r].remainingLength = tL − 8;
        else
            lenT[t, r].getNextByte = 0;
            lenT[t, r].remainingLength = tL;
        end
    end
end
```

Decoding of PL and DI utilizes previously created Huffman tables for vectorizing the generated code and speeding up the decoding phase. After header bytes of the DI is read, each byte of DI and PL are traversed in a nested loop. First loop reads the current DI byte and determines the codeword lengths using Reverse DI Huffman table. According to the read cell from the table, all of the codeword lengths obtained from this cell is iterated with a for loop. Within the for loop, a payload short is constructed with the previous and current PL byte using Length Huffman table. This payload short is used to determine the next row value of the Decode Huffman table. This table's initial row value is set to the first byte of the PL. Current codeword length is used as the column value for Decode Huffman table. With obtained row and column values, corresponding cell is selected and its *output* value is written to $file$. This cycle continues until all DI values are iterated. The pseudo-code for the decoding schema is given below:

Algorithm 9: Decode_D8()

```
Input: pL : Payload file
dI : Disambiguation information file
decT : Pre-computed decode table
lenT : Pre-computed length table
revDIT : Pre-computed reverse disambiguation information table
Output: file : Decoded file
pLL = length(pL), dIL = length(dI);
fileI = 0, pLI = 1 ;                          /* file index, payload index */
curPLByte = pL[pLI], prevPLByte = pL[pLI − 1];
pLShort = prevPLByte << 8 | curPLByte;
pLShortBI = 0 ;                               /* payload short bit index */
decTI = pL[0], revDITI = 0, lenTI = 0 ;       /* decode table index,
  reverse dis. info. table index, length table index */
hData = dI[0] ;                               /* header data */
dType = hData >> 5, skip = (hData >> 3) & 3;
remCount = (hData >> 1) & 3, seeded = hData &;
dII = 4 ;                                     /* dis. info. index, skip header */
while dII < dIL do
    v = dI[dII];
    dII = dII + 1;

    row = revDIT[revDITI, v];
    for i = 0; i < row.outputC; i + + do
        if dII == dIL && i == row.outputC − skip) then break;
        l = row.outputs[i];
        lenRow = lenT[lenTI, l − 1];
        lenTI = lenRow.remainingLength;
        pLI = pLI + lenRow.getNextByte;
        prevPLByte = curPLByte;
        curPLByte = pL[pLI];
        pLShort = (prevPLByte << 8) | curPLByte;

        decRow = decT[decTI, l − 1];
        file[fileI] = decRow.output;
        fileI = fileI + 1;

        sV = (pLShort << pLShortBI) >> (8 + decRow.remLength);
        decTI = decRow.remaining | sV;
        pLShortBI = lenRow.remainingLength;
    end
    revDITI = row.nextTableId;
end
```

For smaller D values such as 4 and 8, creating the Huffman tables to store pre-processed values work well. When D values gets larger, the space required to store all the tables increases. Using these relatively large tables causes page faults and slows down the encoding and decoding phases. For D values larger than 8, only disambiguation information Huffman table (dT) and reverse disambiguation information Huffman table ($revDIT$) are used. Other pre-processed values are calculated on-the-fly in encoding and decoding loops.