Final Report for EE274 Bits-back coding and compression of data with permutation invariance

Yifei Wang

1 Introduction

In this project, we will investigate the compression of multisets where the relative order of the symbols are irrelevant, for instance, collections of files. Severo et al. (2022) proposes a new compression algorithm which saves computational cost by avoiding encoding the order between symbols. We follow this paper as the context and background.

2 Asymmetric Numeral Systems and Bits-Back Coding

We first briefly review asymmetric numeral systems (ANS), which serves as the basis for bits-back coding. For a symbol $x \in \mathcal{A}$ with mass function D, the encode and decode functions form an inverse pair:

$$\begin{aligned}\texttt{encode} :& \mathbb{N} \times \mathcal{A} \to \mathbb{N}, (s, x) \to s',\\ \texttt{decode} :& \mathbb{N} \to \mathbb{N} \times \mathcal{A}, s' \to (s, x).\end{aligned} \tag{1}$$

Here $\log s' \approx \log s + \log 1/D(x)$. To implement `encode` and `decode`, the quantized distribution D must be specified by a precision parameter N together with two lookup functions:

$$\begin{aligned}\texttt{forward\_lookup} &: x \to (c_x, p_x),\\ \texttt{reverse\_lookup} &: i \to (x, c_x, p_x).\end{aligned} \tag{2}$$

Here c_x and p_x are quantized cumulative distribution function and probability mass satisfying

$$\frac{p_x}{N} = D(x), \quad c_x = \sum_{y<x} p_x. \tag{3}$$

The `reverse_lookup` takes the input $i \in \{0, \ldots, N\}$ and it must output (x, c_x, p_x) such that $i \in [c_x, c_x + p_x)$.

The key to apply ANS to bits-back coding is the observation that we can use `decode` for a quantized distribution D as an invertible sampler. To be specific, the ANS state s is thus used as a random seed. By executing `decode` to s, it removes $-\log D(x)$ bits from the state s. In short, the random seed is slowly consumed as symbols are sampled.

ANS can be directly applied to compress a multiset. Consider a multiset $\mathcal{M} = \{x_1, \ldots, x_m\}$, where $m = |\mathcal{M}|$. We can view $x_1, \ldots, x_m$ as a sequence of i.i.d. symbols $x^m = x_1 \ldots x_m$ from a distribution D over alphabet $\mathcal{A}$. We can compress x^m by initializing the state $s_0 = 0$ and recursively update $s_n = \texttt{encode}(s_{n-1}, x_n)$ with D. The finial ANS state s_m will be approximately

$$\log s_m \approx \sum_{n=1}^{m} \log 1/D(x_n).$$

Now, we describe the algorithm of bits-back coding as follows. To encode, we store a multiset of remaining symbols $\mathcal{M}_n$. Firstly, we set $\mathcal{M}_1 = \mathcal{M}$. In the n-th iteration, we sample z^n without replacement from $\mathcal{M}_n$ and encode it. Let z^{n-1} be the sequence of previously sampled symbols. Then, we note that

$$P(z_n|z^{n-1}) = \frac{\mathcal{M}_n(z_n)}{|\mathcal{M}_n|}. \tag{4}$$

To sample z_n decreases the state s_n, while the encoding step increases s_n. To be specific, the number of bits of the state increase at the n-th round is approximately

$$\Delta_n = \log \frac{D(z_n)}{P(z_n|z^{n-1})}. \tag{5}$$

The total increase in the state is

$$\sum_{n=1}^{m} \Delta_n = \sum_{n=1}^{m} \log \frac{D(z_n)}{P(z_n|z^{n-1})} = \log \prod_{z\in\mathcal{M}} D(z)^{\mathcal{M}(z)} - \log \frac{m!}{\prod_{z\in\mathcal{M}} \mathcal{M}(z)!} = \log P(\mathcal{M}). \tag{6}$$

This is exactly the negative information content of the multiset. Suppose that we start with $\log s_0 \approx \sum_{n=1}^{m} \log 1/D(x_n)$. Then, we have

$$\log s_m \approx -\log \frac{m!}{\prod_{z\in\mathcal{M}} \mathcal{M}(z)!}. \tag{7}$$

In summary, by using the bits-back coding to compress multisets, we can save $-\log P(\mathcal{M})$ bits compared to directly apply ANS to compress multisets.

To efficiently implement the invertible sampling without replacement, we utilizes the binary search tree (BST). In summary, we describe the multiset encoder and decoder in Algorithm 1.

Algorithm 1 Multiset Encoder and Decoder for bits-back coding based on ANS.

```
1:  function MultisetEncode($\mathcal{M}$)
2:      Initiate a random state $s_0$. Set $m = |\mathcal{M}|$ and $\mathcal{M}_1 = \mathcal{M}$.
3:      for $n = 1, \dots, m$ do
4:          $(s'_n, z_n) = \texttt{decode}(s_{n-1})$ with $P(\cdot|\mathcal{M}_n)$          ▷ $O(\log m)$
5:          $\mathcal{M}_{n+1} = \mathcal{M}_n / \{z_n\}$                                  ▷ $O(\log m)$
6:          $s_n = \texttt{encode}(s'_n, z_n)$ with $D(\cdot)$                        ▷ $O(D_{\text{Encode}})$
7:      end for
8:      return $s_m$
9:  end function
10: function MultisetDecode($s_m$)
11:     Initialize $\mathcal{M}_{m+1} = \varnothing$.
12:     for n=m,...,1 do
13:         $(s'_n, z_n) = \texttt{decode}(s_n)$ with $D(\cdot)$                      ▷ $O(D_{\text{Decode}})$
14:         $\mathcal{M}_n = \mathcal{M}_{n+1} \cup \{z_n\}$                             ▷ $O(\log m)$
15:         $s_{n-1} = \texttt{encode}(s'_n, z_n)$ with $P(\cdot|\mathcal{M}_n)$.       ▷ $O(\log m)$
16:     end for
17:     return $\mathcal{M}_1$
18: end function
```

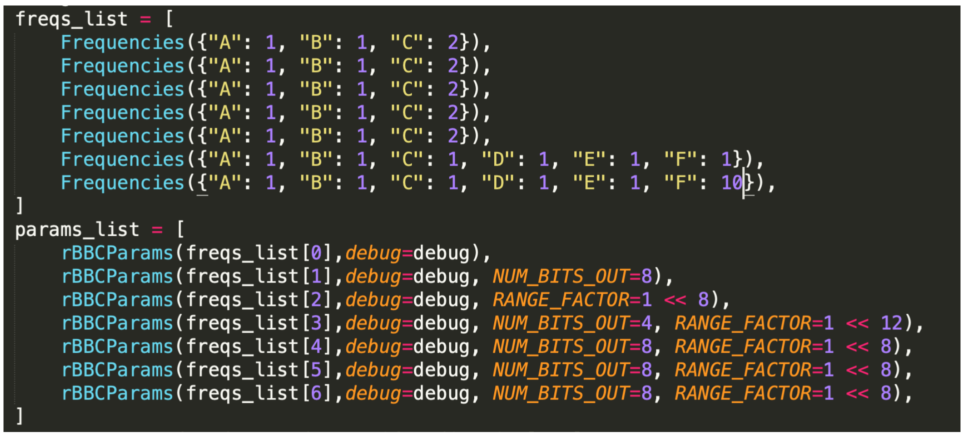
3 Bits back coding upon rANS

As ANS is not directly implemented in SCL, we implement the bits back coding upon rANS, which require that the state before encoding / after decoding lies in the interval $[L, H]$. We briefly summarize the difference of rANS and ANS as follows. In each encoding iteration, rANS has an additional `shrink_state` operation on the state s before encoding the symbol x into the state s. This operation streams out the lower bits of the state, until the state is below some threshold. Then, it output bits to the stream to bring the state in the range for the next encoding. Similarly, in each decoding iteration, we remap the state into the acceptable range after the decoding step. Therefore, we summarize the bits back coding algorithm based on rANS as follows.

The reason why we use `remapLocal` and `shrinkLocal` is that we start with the decoding step in bits back coding. We don't know about the bitarray for `remap` in rANS. Therefore, we simply use local information to implement `remapLocal` and `shrinkLocal` memorilessly.

4 Numerical implementation

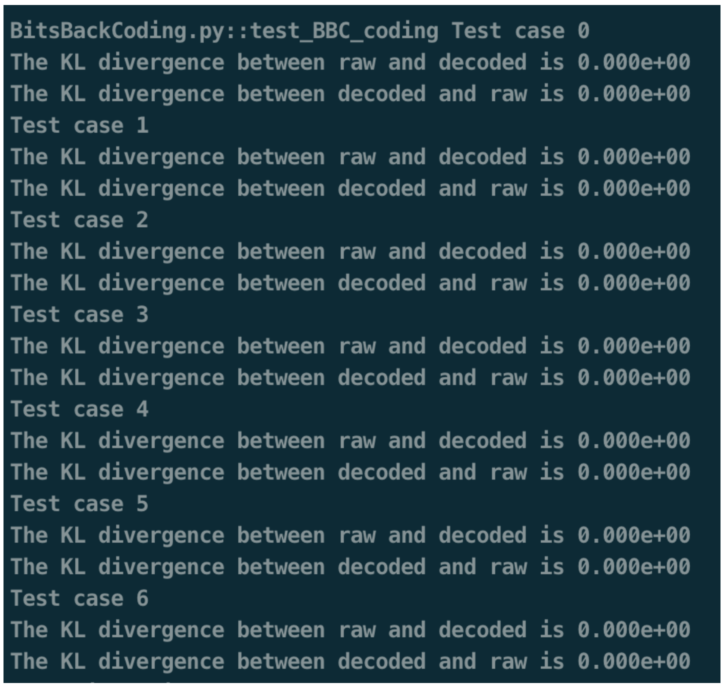
For simplicity, we write our implementation of bits-back coding based on rANS as rBBC. As rBBC does not compress the order information, the decoded data block may not be identical to the raw data block. Therefore, we evaluate the correctness of rBBC by comparing the KL divergence between the empirical distributions of the raw data block and the decoded block. In Figure 1, we present the description of our test cases. The first five test cases use the same data distribution and we aim to study under which the range factor and number of bits, rBBC outperforms rANS. The last two test cases study under which distribution rBBC has better performance than rANS.



```
freqs_list = [
    Frequencies({"A": 1, "B": 1, "C": 2}),
    Frequencies({"A": 1, "B": 1, "C": 2}),
    Frequencies({"A": 1, "B": 1, "C": 2}),
    Frequencies({"A": 1, "B": 1, "C": 2}),
    Frequencies({"A": 1, "B": 1, "C": 2}),
    Frequencies({"A": 1, "B": 1, "C": 1, "D": 1, "E": 1, "F": 1}),
    Frequencies({"A": 1, "B": 1, "C": 1, "D": 1, "E": 1, "F": 10}),
]
params_list = [
    rBBCParams(freqs_list[0],debug=debug),
    rBBCParams(freqs_list[1],debug=debug, NUM_BITS_OUT=8),
    rBBCParams(freqs_list[2],debug=debug, RANGE_FACTOR=1 << 8),
    rBBCParams(freqs_list[3],debug=debug, NUM_BITS_OUT=4, RANGE_FACTOR=1 << 12),
    rBBCParams(freqs_list[4],debug=debug, NUM_BITS_OUT=8, RANGE_FACTOR=1 << 8),
    rBBCParams(freqs_list[5],debug=debug, NUM_BITS_OUT=8, RANGE_FACTOR=1 << 8),
    rBBCParams(freqs_list[6],debug=debug, NUM_BITS_OUT=8, RANGE_FACTOR=1 << 8),
]
```

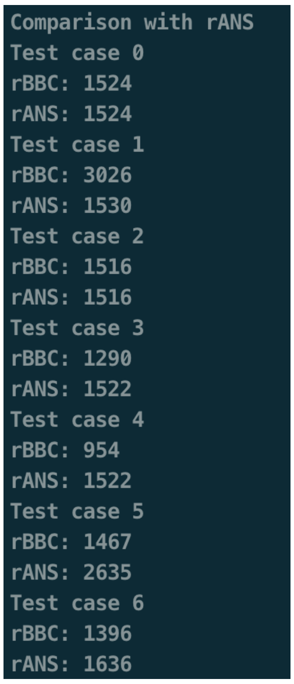
Figure 1: Quote of test cases.

The results in Figure 2 validate the correctness of our implementation. In Figure 3, we numerically compare the improvement of rBBC for compressing a multiset compared to directly apply rANS to compress it. From the first five cases, we note that using 8 number of bits out and a range factor of 2^8 makes rBBC compress more compared to rANS. From the last test cases, we note that rBBC has better compress performance for data distribution which is close to a uniform distribution.



```
BitsBackCoding.py::test_BBC_coding Test case 0
The KL divergence between raw and decoded is 0.000e+00
The KL divergence between decoded and raw is 0.000e+00
Test case 1
The KL divergence between raw and decoded is 0.000e+00
The KL divergence between decoded and raw is 0.000e+00
Test case 2
The KL divergence between raw and decoded is 0.000e+00
The KL divergence between decoded and raw is 0.000e+00
Test case 3
The KL divergence between raw and decoded is 0.000e+00
The KL divergence between decoded and raw is 0.000e+00
Test case 4
The KL divergence between raw and decoded is 0.000e+00
The KL divergence between decoded and raw is 0.000e+00
Test case 5
The KL divergence between raw and decoded is 0.000e+00
The KL divergence between decoded and raw is 0.000e+00
Test case 6
The KL divergence between raw and decoded is 0.000e+00
The KL divergence between decoded and raw is 0.000e+00
```

Figure 2: Terminal output for evaluating the correctness of rBBC.



```
Comparison with rANS
Test case 0
rBBC: 1524
rANS: 1524
Test case 1
rBBC: 3026
rANS: 1530
Test case 2
rBBC: 1516
rANS: 1516
Test case 3
rBBC: 1290
rANS: 1522
Test case 4
rBBC: 954
rANS: 1522
Test case 5
rBBC: 1467
rANS: 2635
Test case 6
rBBC: 1396
rANS: 1636
```

Figure 3: Terminal output for comparing rBBC and rANS.

5 Conclusion

The bits back coding is a interesting randomized data-compression scheme based on ANS which neglect the order information. Therefore, it can compress multi-set to the information limit. In this work, we extend bits back coding to rBBC which is based on rANS. Our numerical experiment shows that rBBC has better performance if the data distribution is close to a uniform distribution. This also coincide with the theory where BBC can compress the multi-set to the information limit.

References

Severo, D., Townsend, J., Khisti, A., Makhzani, A., and Ullrich, K. (2022). Compressing multisets with large alphabets. In *2022 Data Compression Conference (DCC)*, pages 322–331. IEEE.

Algorithm 2 Multiset Encoder and Decoder for bits-back coding based on rANS.

```
1: function MultisetEncode($\mathcal{M}$)
2:     Initiate a random state $s_0$. Set $m = |\mathcal{M}|$, $\mathcal{M}_1 = \mathcal{M}$, $b_0 = 0$ and $B_0 = b_0$.
3:     for $n = 1, \dots, m$ do
4:         $(s_n^{(1)}, z_n) =$ decode($s_{n-1}$) with $P(\cdot|\mathcal{M}_n)$        ▷ $O(\log m)$
5:         $s_n^{(2)} =$ remapLocal($s_n^{(1)}$).
6:         $\mathcal{M}_{n+1} = \mathcal{M}_n / \{z_n\}$        ▷ $O(\log m)$
7:         $s_n^{(3)}, b_n =$ shrink($s_n^{(2)}$).
8:         $s_n =$ encode($s_n^{(3)}, z_n$) with $D(\cdot)$        ▷ $O(D_{\text{Encode}})$
9:         $B_n = B_{n-1} + b_n$.
10:    end for
11:    return $s_m, B_m$
12: end function
13: function MultisetDecode($s_m, B_m$)
14:    Initialize $\mathcal{M}_{m+1} = \varnothing$.
15:    for n=m,...,1 do
16:        $B_n, b_{n+1} = B_{n+1}$.
17:        $(s_n^{(3)}, z_n) =$ decode($s_n$) with $D(\cdot)$        ▷ $O(D_{\text{Decode}})$
18:        $s_n^{(2)} =$ remap($s_n^{(3)}, b_{n+1}$).
19:        $\mathcal{M}_n = \mathcal{M}_{n+1} \cup \{z_n\}$        ▷ $O(\log m)$
20:        $s_n^{(1)} =$ shrinkLocal($s_n^{(2)}$).
21:        $s_{n-1} =$ encode($s_n^{(1)}, z_n$) with $P(\cdot|\mathcal{M}_n)$.        ▷ $O(\log m)$
22:    end for
23:    return $\mathcal{M}_1$
24: end function
25: function remapLocal($s$)
26:    while $s < L$ do
27:        $s = 2s$.
28:    end while
29:    return $s$
30: end function
31: function shrinkLocal($s$)
32:    while $s > H$ do
33:        $s = s//2$.
34:    end while
35:    return $s$
36: end function
```