On Implementing the Binary Interpolative Coding Algorithm

GIULIO ERMANNO PIBIRI, ISTI-CNR

1 THE ALGORITHM

The *Binary Interpolative Coding* algorithm was invented by Moffat and Stuiver [1, 2] and can be used to compress a sorted integer sequence $\mathcal{S}[0..n)$. Throughout this technical report, we assume that $\mathcal{S}$ is *strictly increasing*, i.e., $\mathcal{S}[i] > \mathcal{S}[i-1]$ for every $0 < i < n$, although the algorithm can also be applied to non-decreasing sequences with a straightforward modification of the code we are going to illustrate. The key idea of the algorithm is to exploit the order of the already-encoded elements to compute the number of bits needed to represent the elements that will be encoded next.

Encoding. At the beginning of the encoding phase, suppose we are specified two quantities $lo \le \mathcal{S}[0]$ and $hi \ge \mathcal{S}[n-1]$. Given such quantities, we can encode the element in the middle of the sequence, say $\mathcal{S}[m]$ with $m = \lfloor n/2 \rfloor$, in some appropriate manner, knowing that $lo \le \mathcal{S}[m] \le hi$. For example, we can write $\mathcal{S}[m] - lo - m$ using just $\lceil \log_2(hi - lo - n + 1) \rceil$ bits. After that, we can apply the same step to both halves $\mathcal{S}[0, m)$ and $\mathcal{S}[m+1, n)$ with *updated knowledge* of lower and upper values (lo, hi) that are set to $(lo, \mathcal{S}[m] - 1)$ and $(\mathcal{S}[m] + 1, hi)$ for the left and right half respectively. The following code listing shows the idea, where we assume the input sequence $\mathcal{S}$ to be a sequence of 32-bit unsigned integers.

```
void encode(uint32_t const* S, uint32_t n, uint32_t lo, uint32_t hi) {
    if (!n) return;
    assert(lo <= hi);
    uint32_t m = n / 2;
    uint32_t x = S[m];
    write(x - lo - m, hi - lo - n + 1);
    encode(S, m, lo, x - 1);
    encode(S + m + 1, n - m - 1, x + 1, hi);
}
```

```
void write(uint32_t x, uint32_t r) {
    assert(x <= r);
    if (!r) return;
    uint32_t b = msb(r) + 1;
    append(x, b);
}
```

The function $\text{write}(x, r)$ just writes the value x using $b = \lceil \log_2(r+1) \rceil$ bits; $\text{append}(x, b)$ appends to the output bitstream the representation of x on b bits. The function $\text{msb}(r)$ returns the position in $[0, 31]$ of the most significant bit of r, hence $b = \lceil \log_2(r+1) \rceil$ can be computed as $\text{msb}(r) + 1$. The msb function can be efficiently implemented via the built-in function[1] $\text{clz}(r)$ that returns the

[1] See http://gcc.gnu.org/onlinedocs/gcc/Other-Builtins.html

Author's address: Giulio Ermanno Pibiri, ISTI-CNR, Pisa, Italy, giulio.ermanno.pibiri@isti.cnr.it.

number of leading zeros in r, starting at the most significant bit position. (Beware that if r is 0, the result is undefined, thus is good practice to assert $r > 0$ to remember this.)

```
uint32_t msb(uint32_t r) {
    assert(r > 0);
    return 31 - __builtin_clz(r);
}
```

Decoding. Decoding simply works by reversing the encoding process. The following listings illustrate the decoding algorithm. We assume that the original sequence $\mathcal{S}$ is decoded as an array of 32-bit unsigned integers. The function take(b) consumes b bits from the encoded bitstream and interprets them as an unsigned integer value.

```
void decode(uint32_t* S, uint32_t n, uint32_t lo, uint32_t hi) {
    if (!n) return;
    assert(lo <= hi);
    uint32_t m = n / 2;
    uint32_t x = read(hi - lo - n + 1) + lo + m;
    S[m] = x;
    if (n == 1) return;
    decode(S, m, lo, x - 1);
    decode(S + m + 1, n - m - 1, x + 1, hi);
}
```

```
uint32_t read(uint32_t r) {
    if (!r) return 0;
    uint32_t b = msb(r) + 1;
    uint32_t x = take(b);
    assert(x <= r);
    return x;
}
```

An encoding example. We now show and encoding example applied to the sequence $\mathcal{S}[0..12)$ = [3, 4, 7, 13, 14, 15, 21, 25, 36, 38, 54, 62]. As it is always safe to choose $lo = 0$ and $hi = \mathcal{S}[n-1]$, we do so, thus at the beginning of the encoding phase we have $lo = 0$ and $hi = 62$. Since we set $hi = \mathcal{S}[n-1]$, the last value of the sequence is first encoded and we process $\mathcal{S}[0..n-1)$ only.

Fig. 1 shows the sequence of recursive calls performed by the encoding algorithm oriented as a binary tree. In each node of the tree we report the values assumed by the quantities m, n, lo and hi, plus the processed subsequence and the number of bits needed to encode the middle element. By *pre-order* visiting the tree, we obtain the sequence of written values, that is [10, 5, 3, 3, 5, 5, 18, 8, 5, 16, 1] with associated codeword lengths [6, 4, 3, 2, 3, 3, 6, 5, 4, 5, 5]. The Interpolative code produced by the example consumes 46 bits for 11 values, that is $46/11 = 4.18$ bits per value.

At the beginning of the decoding phase, we must know the initial value for hi (lo is known to be 0). Since we set $hi = \mathcal{S}[11] = 62$, we also save this value using the following write_binary function, that first writes the needed bit-width b (5 bits suffice) and then write the value in binary using $b+1$ bits.

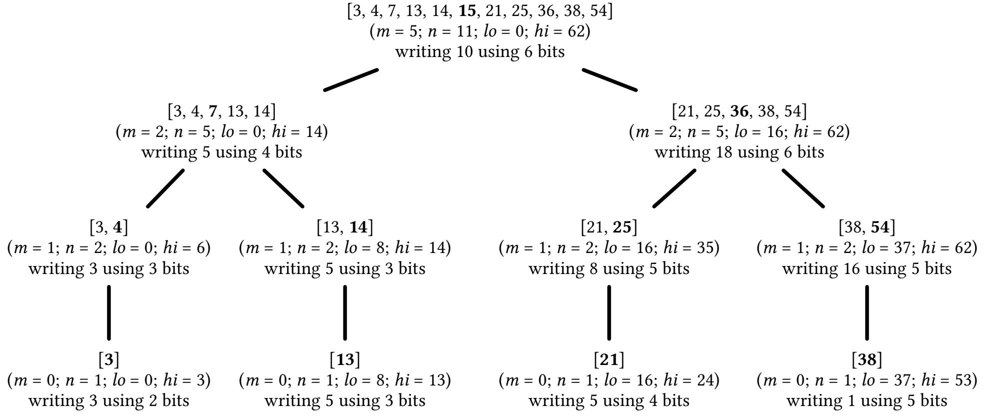

Fig. 1. The recursive calls performed by the Binary Interpolative Coding algorithm when applied to the sequence [3, 4, 7, 13, 14, 15, 21, 25, 36, 38, 54] with initial knowledge of lower and upper bound values $lo = 0$ and $hi = 62$. In bold font we highlight the middle element being encoded.

```
void write_binary(uint32_t x) {
    uint32_t b = 0;
    if (x) b = msb(x);
    assert(b <= 31);
    append(b, 5);
    append(x, b + 1);
}
```

Therefore, in total we consume $46 + 5 + 6 = 57$ bits, that is $57/12 = 4.75$ bits per value (if we also save the value of n using the write_binary function, we consume a total of 66 bits, i.e., 5.5 bits per value). Reading the output of the write_binary function is very easy and can be done as follows.

```
uint32_t read_binary() {
    uint32_t b = take(5);
    return take(b + 1);
}
```

Exploiting minimal binary codes. However, the encoding process obtained by the use of the simple write(x, r) function is wasteful. In fact, as discussed in the original work [1, 2], more succinct encodings can be achieved with a *minimal* binary encoding. More precisely, when the range $r > 0$ is specified, all values $0 \leq x \leq r$ are assigned fixed-length codewords of size $\lceil \log_2(r+1) \rceil$ bits. But the more r is distant from $2^{\lceil \log_2(r+1) \rceil}$ the more this allocation of codewords is wasteful because $c = 2^{\lceil \log_2(r+1) \rceil} - r - 1$ codewords can be made 1 bit shorter without loss of unique decodability. Therefore we proceed as follows. We identify the range of smaller codewords, delimited by the values r_{lo} and r_{hi}, such that every value $x \leq r$ such that $r_{lo} < x < r_{hi}$ is assigned a shorter $(\lceil \log_2(r+1) \rceil - 1)$-bit codeword and every value outside this range is assigned a longer one on $\lceil \log_2(r+1) \rceil$ bits. To maintain unique decodability, we first always read $\lceil \log_2(r+1) \rceil - 1$ bits and

(a)

0	1	2	3	4	5	6	7	8	9	10	11	12
–	–	–	**0**	**1**	**0**	**1**	**0**	**1**	**0**	**1**	**0**	**1**
0	0	0	0	0	1	1	1	1	1	1	1	1
0	0	1	1	1	0	0	0	0	1	1	1	1
0	1	0	1	1	0	0	1	1	0	0	1	1

(b)

0	1	2	3	4	5	6	7	8	9	10	11	12
0	**0**	**0**	**0**	**0**	–	–	–	**1**	**1**	**1**	**1**	**1**
0	0	0	0	1	1	1	1	0	0	0	0	1
0	0	1	1	0	0	1	1	0	0	1	1	0
0	1	0	1	0	1	0	1	0	1	0	1	0

Table 1. Minimal binary codeword assignment to the values in 0..12 for a left-most assignment rule (a) and a centered assignment rule (b). In bold we highlight the extra bit written/read during encoding/decoding.

interpret these as the value x. Then we check if condition $r_{lo} < x < r_{hi}$ is satisfied: if so, we are done; otherwise, the codeword must be extended by 1 bit.

In a *left-most* minimal binary code assignment, the first c values are assigned the shorter codewords, thus $r_{hi} = 2^{\lceil \log_2(r+1) \rceil} - r - 1$ (and we only check whether $x < r_{hi}$ since r_{lo} is assumed to be -1). In a *centered* minimal binary code assignment, the values in the centre of the range are assigned the shorter codewords, thus $(r_{lo}, r_{hi}) = (\lfloor r/2 \rfloor - \lfloor c/2 \rfloor - 1, \lfloor r/2 \rfloor + \lfloor c/2 \rfloor + 1)$ if r is even, or $(r_{lo}, r_{hi}) = (\lfloor r/2 \rfloor - \lfloor c/2 \rfloor, \lfloor r/2 \rfloor + \lfloor c/2 \rfloor + 1)$ if r is odd. The rationale behind using centered minimal codes is that a reasonable guess is to assume the middle element to be about half of the upper bound. Note that, in a centered minimal assignment, the lowest $\lceil \log_2(r+1) \rceil - 1$ bits of the codewords assigned to integers falling outside the range (r_{lo}, r_{hi}) are always symmetric. More precisely, if $\widehat{x_i}$ denotes such lowest bits of the integer x_i, that we have $\widehat{x_i} = \widehat{x}_{i+r_{hi}}$ for $i = 0, \ldots, r_{lo}$. This implies that, during decoding, we can just check whether the read $(\lceil \log_2(r+1) \rceil - 1)$-bit integer x is less than or equal to r_{lo}: if so, we must extend the codeword by 1 more bit.

For example, when $r = 12$, we have that $\lceil \log_2(12+1) \rceil = 4$ and $2^4 - 12 - 1 = 3$ codewords can be made 3-bit long and the remaining 10 codewords have a standard length of 4 bits. In the left-most assignment, these 3 shorter codewords are assigned to the values 0, 1 and 2; in the centered assignment, these are assigned to the values 5, 6 and 7. In the latter assignment, we thus have $r_{lo} = 4$ and $r_{hi} = 8$. Notice that the lowest 3 bits of the values 0 and 8 are the same, as well as the ones for 1 and 9, 2 and 10 and so on. Therefore, we can just read a 3-bit integer and check if it is less than or equal to $r_{lo} = 4$ to know whether we need a longer codeword or not. Table 1 shows the comparison between left-most and centered minimal assignments[2].

Continuing our example in Fig. 1, using a left-most minimal assignment we have the following sequence of codeword length [5, 4, 3, 2, 3, 3, 5, 4, 3, 5, 4], resulting in a code of 41 bits (ignoring the encoding of n and $\mathcal{S}[n-1]$), thus 5 bits shorter than the one using a simple binary assignment. Lastly, the centered minimal assignment produces a further saving of 1 bit on that example sequence.

The following listings[3] illustrate how we can fast encode and decode minimal binary codes for a left-most assignment and centered assignment respectively. These are the assignments used also for the example in Table 1. To experiment with such encodings, it is sufficient to replace the write (resp. read) function in the encode (resp. decode) algorithm with one of the two options shown below (and without touching anything else).

[2]See also Table 2 from [2] for another example of centered minimal codes assigned using a different rule than the one used here. In fact, it should be noted that the exact assignment of codewords is irrelevant and many assignments could be possible: what matters is to assign correct lengths and maintain the property of unique decodability.

[3]The code is valid for little-endian architectures.

```
void leftmost_write(uint32_t x, uint32_t r) {
    if (!r) return;
    assert(x <= r);
    uint32_t b = msb(r);
    uint32_t hi = (uint64_t(1) << (b + 1)) - r - 1;
    if (x < hi) append(x, b);
    else {
        x += hi;
        append(x >> 1, b);
        append(x & 1, 1);
    }
}

uint32_t leftmost_read(uint32_t r) {
    if (!r) return 0;
    uint32_t b = msb(r);
    uint32_t hi = (uint64_t(1) << (b + 1)) - r - 1;
    uint32_t x = take(b);
    if (x >= hi) x = (x << 1) + take(1) - hi;
    assert(x <= r);
    return x;
}
```

```
void centered_write(uint32_t x, uint32_t r) {
    if (!r) return;
    assert(x <= r);
    uint32_t b = msb(r);
    uint32_t c = (uint64_t(1) << (b + 1)) - r - 1;
    int64_t half_c = c / 2;
    int64_t half_r = r / 2;
    int64_t lo = half_r - half_c;
    int64_t hi = half_r + half_c + 1;
    if (r % 2 == 0) lo -= 1;
    if (x > lo and x < hi) append(x, b);
    else append(x, b + 1);
}

uint32_t centered_read(uint32_t r) {
    if (!r) return 0;
    uint32_t b = msb(r);
    uint32_t c = (uint64_t(1) << (b + 1)) - r - 1;
    int64_t half_c = c / 2;
    int64_t half_r = r / 2;
    int64_t lo = half_r - half_c;
    if (r % 2 == 0) lo -= 1;
    uint32_t x = take(b);
    if (x <= lo) x += take(1) << b;
    assert(x <= r);
    return x;
}
```

Run-awareness. The recursive implementation we have shown can be made faster by stopping the recursion whenever we detect a *run*, i.e., a sequence of consecutive values: if $hi - lo + 1 = n$, then we stop recursion. If the condition is satisfied during encoding, we omit no bits at all. During decoding, we simply output the values $lo, lo + 1, \ldots, lo + n - 1$ with a for loop. This means that the Interpolative code can actually use codewords of 0 bits to represent more than one integer, hence attaining to a rate of less than one bit per integer – a remarkable property that makes the code very succinct for highly clustered sequences. (Since runs are handled in this way, we can also avoid checking whether r is equal to 0 in the write/read functions shown before.) The following two listings show the updated code for the encode/decode algorithm.

```
void encode(uint32_t const* S, uint32_t n, uint32_t lo, uint32_t hi) {
    if (!n) return;
    if (hi - lo + 1 == n) return; // <-- run
    assert(lo <= hi);
    uint32_t m = n / 2;
    uint32_t x = S[m];
    write(x - lo - m, hi - lo - n + 1);
    encode(S, m, lo, x - 1);
    encode(S + m + 1, n - m - 1, x + 1, hi);
}
```

```
void decode(uint32_t* S, uint32_t n, uint32_t lo, uint32_t hi) {
    if (!n) return;
    assert(lo <= hi);
    if (hi - lo + 1 == n) { // <-- run
        for (uint32_t i = 0; i != n; ++i) out[i] = lo++;
        return;
    }
    uint32_t m = n / 2;
    uint32_t x = read(hi - lo - n + 1) + lo + m;
    S[m] = x;
    if (n == 1) return;
    decode(S, m, lo, x - 1);
    decode(S + m + 1, n - m - 1, x + 1, hi);
}
```

2 EXPERIMENTS

In this section we quantify the compression effectiveness and the sequential decoding speed achieved by the Interpolative Coding algorithm on standard and large text collections.

Source code. All our C++ code is available at https://github.com/jermp/interpolative_coding.

Datasets. We performed our experiments on the following datasets, whose statistics are summarized in Table 2.

- Gov2 is the TREC 2004 Terabyte Track test collection, consisting in roughly 25 million .gov sites crawled in early 2004. Documents are truncated to 256 KB.
- ClueWeb09 is the ClueWeb 2009 TREC Category B test collection, consisting in roughly 50 million English web pages crawled between January and February 2009.

	Gov2	ClueWeb09	CCNews
Documents	24,622,347	50,131,015	43,530,315
Sequences	35,636,425	92,094,694	43,844,574
Integers	5,742,630,292	15,857,983,641	20,150,335,440

Table 2. Basic statistics for the tested collections.

Method	Gov2		ClueWeb09		CCNews	
simple binary	3.532		4.990		5.502	
left-most minimal	3.362	(−4.81%)	4.754	(−4.73%)	5.210	(−5.31%)
centered minimal	3.361	(−4.85%)	4.766	(−4.50%)	5.147	(−6.47%)

Table 3. Compression measured in average number of bits spent per integer.

Method	Gov2		ClueWeb09		CCNews	
	r.a.	not r.a.	r.a.	not r.a.	r.a.	not r.a.
simple binary	2.76	3.43	3.64	4.11	3.90	4.01
left-most minimal	4.37	5.70	5.89	7.15	6.88	7.98
centered minimal	4.52	5.89	6.09	7.35	7.24	8.37

Table 4. Decoding time measured in average nanoseconds spent per decoded integer, for the implementation that is and is not run-aware (r.a.).

- CCNews is an English subset of the freely available news from CommonCrawl[4], consisting of articles crawled from 09/01/16 to 30/03/18.

Identifiers were assigned to documents by following the lexicographic order of their URLs [3].

Experimental setting and methodology. Experiments are performed on a server machine equipped with Intel i9-9900K cores (@3.60 GHz), 64 GB of RAM DDR3 (@2.66 GHz) and running Linux 5 (64 bits). Each core has two private levels of cache memory: 32K L1 cache (one for instructions and one for data); 256K for L2 cache. A shared L3 cache spans 16384K. We compiled the code with gcc 9.2.1 using the highest optimization setting, i.e., with compilation flags `-O3` and `-march=native`.

We encoded all sequences one after the other in a single file on disk that is then memory-mapped for decoding. We measured the time needed to decode every single integer in the test collection and report the average nanoseconds spent per decoded integer. The decoding algorithm runs on a single CPU core.

Compression effectiveness. Table 3 shows the average number of bits spent per represented integer. We can see that the usage of minimal codes always improve over the simple binary codeword allocation by roughly 5%. Also, centered minimal codes may slightly improve compression with respect to the left-most assignment, as we can observe on Gov2 and CCNews.

[4]http://commoncrawl.org/2016/10/news-dataset-available

Decoding time. Table 4 shows, instead, the sequential decoding speed expressed in average nanoseconds spent per decoded integer. We compare the optimized run-aware version against the not optimized one.

For the simplicity of the approach and branchless code, the usage of simple binary codewords strongly favours decoding efficiency, being significantly faster than the version using minimal codes. The decoding time of the two minimal arrangements tested is practically the same because similar instructions are executed, with centered minimal being consistently and slightly slower than the left-most one. Lastly, note that the run-aware implementation pays off as it is always faster: more effective on the more clustered Gov2, but less effective on the less clustered ClueWeb09 and CCNews.

3 CONCLUSIONS

In this technical report we described a concrete C++ implementation on the Binary Interpolative Coding algorithm and illustrated some examples on real-world datasets. Our implementation is publicly available.

From the experimental analysis, we can conclude that we should prefer the choice of a simple binary codeword assignment – especially paired with *run-awareness* – if our primary concern is decoding efficiency. The usage of minimal binary codewords slightly improves compression effectiveness, at the cost of significantly slowing down decoding.

REFERENCES

[1] Alistair Moffat and Lang Stuiver. 1996. Exploiting Clustering in Inverted File Compression. In *Data Compression Conference*. 82–91.

[2] Alistair Moffat and Lang Stuiver. 2000. Binary Interpolative Coding for Effective Index Compression. *Information Retrieval Journal* 3, 1 (2000), 25–47.

[3] Fabrizio Silvestri. 2007. Sorting Out the Document Identifier Assignment Problem. In *Proceedings of the 29th European Conference on IR Research*. 101–112.