Selective Dynamic Compression

Shmuel T. Klein[1], Elina Opalinsky[2], and Dana Shapira[2]

[1] Dept. of Computer Science, Bar Ilan University, Ramat Gan 52900, Israel
tomi@cs.biu.ac.il

[2] Dept. of Computer Science, Ariel University, Ariel 40700, Israel
shapird@g.ariel.ac.il, elinao@ariel.ac.il

Abstract. Dynamic compression methods continuously update the model of the underlying text file to be compressed according to the already processed part of the file, assuming that such a model accurately predicts the distribution in the remaining part. Since this premise is not necessarily true, we suggest to update the model only selectively. We give empirical evidence that this hardly affects the compression efficiency, while it obviously may save processing time and allow the use of the compression scheme for cryptographic applications.

1 Introduction

Compression systems often comprise three major components: the model, the encoding process, and the corresponding inverse decoding process. The model is basically the definition of the set of symbols, called the alphabet, and their distribution. The alphabet defined in the model is not limited to characters only, and can also include words and phrases, and we shall use the term *alphabet* in this border sense. Generally, as the model becomes more and more accurate and adapted to the underlying text, the compression efficiency achieved by the encoding phase becomes better. However, the space overhead imposed by extending the model to be more precise, might diminish the savings achieved by the enhanced encoding, implying a crucial trade-off. Thus, determining the model has a crucial impact on the compression efficiency.

Text compression systems can be partitioned into static and dynamic compression techniques. Static variants determine the model in a preprocessing stage, and the model remains the same throughout the compression stage. The model can be constructed based on known distributions as well as on statistics gathered through a double pass over the file, the latter being suitable to off-line compressions. Dynamic variants, called also *adaptive compression*, save the additional pass, and usually consist of three main steps for each processed symbol:

1. reading the following symbol;
2. encoding the symbol according to the current model;
3. updating the model by incrementing the frequency of the currently read symbol.

The intuition of the dynamic variants is that as more information is collected about the characters in the text seen so far, the more accurate the probabilities become, and, thus, better approximation is achieved to the probabilities of the characters that are still to be seen. Indeed, all adaptive compression methods are based on the assumption that the distribution of the characters in the text starting from the current position onwards will be similar to the distribution in the part of the text preceding this current position. However, this is a statistic observation, which is not necessarily

Shmuel T. Klein, Elina Opalinsky, Dana Shapira: Selective Dynamic Compression, pp. 102–110.
Proceedings of PSC 2019, Jan Holub and Jan Žďárek (Eds.), ISBN 978-80-01-06618-8 © Czech Technical University in Prague, Czech Republic

true, especially not for non-homogeneous texts. In these kind of texts it may happen that basing the prediction of the character probabilities on a random subset of the recently read characters may yield a compression performance that is not inferior, and sometimes even better, than using the entire history. Such an example is given in Section 3.2.

Evidence for basing the distribution of a certain alphabet in a given file on a subset of its characters, rather than on the entire file, can be found in the reported numbers for letter frequency analysis, which are often used in cryptography and linguistics. For example, Norvig [3] generated tables of counts for letters, words and letter sequences, on *Google books*, a collection of 23GB of the books that have been scanned by Google. The probabilities are given with an accuracy of at most 5 digits after the decimal point, corresponding to frequencies within a text of size 100,000, less than a fraction of 10^{-5} of the actual text. Even if the entire text has been used to generate the probabilities, the lower precision suffices to discriminate between them and therefore using a higher precision would have been a overkill.

Another example for using a restricted history, rather than the entire available one, can be found in the LZSS [4] compressor, which represents a given file T as a sequence of substring copies and single characters. The copies are described in the form of ordered pairs (off, len), meaning that the substring starting at the current position can be copied from *off* characters before the current position in the decompressed file, and the length of this substring is len. The variable *off* is often bounded by the size of a predefined *sliding window*, which limits the history in practice. GZIP uses a default window size of $32K$, and thus the *off* values can be written in 15 bits. The use of a limited history implemented as a sliding window is motivated by the saving incurred by the required bits to represent offsets within the restricted window. In this research we show that a limited history may have advantages beyond explicit storage savings.

The core motivation for basing the statistics of subsequent encoding only on a part of the previously seen symbols, rather than on the entire history, is time savings due to the processing of a smaller set. There is, however, a concern whether the restricted history may hurt the compression efficiency. A *Compression-Crypto System* based on arithmetic coding is introduced in [2], where the model gets updated according to a secret key shared only by encoder and decoder. This is yet another motivation for the encoding that is based on selectively updating the model, called *selective encoding* for short, however, here we extend the method to general adaptive compressors as well as for other purposes, other than encryption needs.

We focus on three different adaptive algorithms and examine the effect on time and compression performance in case the model is updated selectively. Our empirical results suggest that the loss in compression efficiency incurred by turning to a selective updating procedure as suggested, is hardly noticeable. Moreover, the encoding and decoding processing times can only be improved using the selective method, as expected, by saving the operations in case the model does not get updated, giving noticeable practical time savings.

Traditional dynamic Huffman codes turn the encoded file into an extremely vulnerable one in case of even a single bit error [1]. As a solution to this problem, blockwise dynamic Huffman variants are suggested, where the Huffman tree is periodically, rather than constantly, updated. Experiments show that the new scheme is more robust against single errors introduced in the encoded file. Here we suggest a "semi-blockwise" variant in which not all occurrences get updated.

The paper is constructed as follows. Section 2 presents the general method for selective encoding. Section 3 adapts the general selective algorithms to the three basic adaptive codings: dynamic Huffman, LZW and arithmetic coding. Section 4 presents empirical results and concludes.

2 General Method

Traditional dynamic algorithms update the frequencies adaptively after every character, according to the assumption that better compression can be achieved when all previous characters are taken into account. This seems to justify the slow processing time of some of the adaptive methods, such as *dynamic Huffman* coding. The selective method calls for omitting a subset of these updates in a predefined setting that is synchronized between the encoder and decoder. The selection may be done periodically, randomly, or by supplying the specific cases in which the model should get updated. We present here only the random version, with probability $\frac{1}{2}$ for the occurrences of `0` and `1` bits, which can be easily updated to the other selection variants.

Algorithm 1: SELECTIVE-ENCODE

```
SELECTIVE-ENCODE(T = x_1 ··· x_n )
initialize the model
initialize a random bit generator
for i ← 1 to n do
    encode x_i according to the current model
    bit ← random()
    if bit = 1 then
        Update the model
```

The general random selective encoding and corresponding decoding algorithms are presented in Algorithm 1 and Algorithm 2, respectively. The selective encoding algorithm is applied on a given text $T = x_1 \cdots x_n$ where each x_i is a symbol of the alphabet Σ.

Algorithm 2: SELECTIVE-DECODE

```
SELECTIVE-DECODE(E(T))
initialize the model
initialize a random bit generator identical to the one in SELECTIVE-ENCODE
for i ← 1 to n do
    decode x_i according to the current model
    bit ← random()
    if bit = 1 then
        Update the model
```

The coding method is chosen in advance and known to both the encoder and decoder. The model gets updated in case the bit returned by the random number generator is set to 1. We refer to the chosen encoding and corresponding decoding applications as a black box, and deal with specific encodings in the following section. The selective decoding algorithm is applied on a given compressed text denoted by

$\mathcal{E}(\mathcal{T})$, where $\mathcal{E}$ is the encoding function. The model and random bit generator are initialized identically in both procedures.

As mentioned above, instead of using a randomized algorithm, basing the selection on a random bit generator, one may decide in advance of setting a constant k, so that the model gets updated exactly every k symbols. The randomized version can be approximated by the constant updates with $k = 2$.

3 Specific Variants

The general selective approach may require adaption when applied to a specific method. Here we examine our proposed selective algorithms on three main adaptive compression techniques: arithmetic coding [7], dynamic Huffman coding [5] and LZW [6]. The following describes the adaptation suggested for each variant.

3.1 Arithmetic Coding

One of the most effective compression schemes, in theory as well as in practice, is arithmetic coding, for which the compression efficiency approaches the underlying text's entropy. The arithmetic compressor is initialized with the interval $[low, high) = [0, 1)$, which is narrowed for each processed character of the input file, according to the character's probability.

More formally, given a text $T = x_1 \cdots x_n$ over an alphabet Σ of size σ, the current interval $[low, high)$ is partitioned into σ subintervals, each corresponding to one of the symbols $\sigma_i \in \Sigma$, where the size of the subinterval assigned to σ_i is proportional to its probability p_i. After iterating on all symbols of T, the compressed text is represented by a single number within the final interval that corresponds to T.

For example, if $\Sigma = \{\mathtt{A}, \mathtt{B}, \mathtt{C}\}$, initialized with uniform probabilities, one may start with (fictitious) frequencies 1 for each of the three characters. The partition is $\{[0, \frac{1}{3}), [\frac{1}{3}, \frac{2}{3}), [\frac{2}{3}, 1)\}$, and the intervals may be assigned lexicographically. If the text to be compressed is `BCB`, then the initial interval $[0, 1)$ is narrowed to $I_1 = [\frac{1}{3}, \frac{2}{3})$ after having read the first `B`, and the probabilities are updated to $P = \{\frac{1}{4}, \frac{1}{2}, \frac{1}{4}\}$; the interval then narrows to $I_2 = [\frac{7}{12}, \frac{2}{3})$ after the processing of `C` and the probabilities are updated to $P = \{\frac{1}{5}, \frac{2}{5}, \frac{2}{5}\}$; and finally, after reading the second `B`, the interval narrows to $I_3 = [\frac{3}{5}, \frac{19}{30})$. Any real number within I_3 can be chosen to represent the compressed file, e.g., 0.625 whose binary representation 0.101 is the shortest.

However, when the model gets updated selectively, for example for every second character ($k = 2$ in our notation), the initial interval $[0, 1)$ gets narrowed in the first step, as in the traditional arithmetic coding, to $S_1 = [\frac{1}{3}, \frac{2}{3})$ after reading the first character `B`, and P becomes $P = \{\frac{1}{4}, \frac{1}{2}, \frac{1}{4}\}$; processing the character `C` does not cause an update of the model, yet the interval narrows to $S_2 = [\frac{7}{12}, \frac{2}{3})$; but then, processing the following `B` narrows the interval to $S_3 = [\frac{29}{48}, \frac{31}{48})$ and updates the probabilities to $P = \{\frac{1}{5}, \frac{2}{5}, \frac{2}{5}\}$. The same binary number 0.101 can represent this interval. The compressed file will thus consist of a three bits in both cases.

Let $P = (p_1, \ldots, p_\sigma)$ be the probability distribution of all the characters of Σ in the entire text, and let P' be the probability distribution of the characters within the subtext that has been selected by the model, consisting by those corresponding to the 1-bits chosen by the random number generator. Because of the random selection and if the subset is large enough, the distribution remains almost the same, and therefore

$H(P) \simeq H(P')$, where $H(P) = -\Sigma_{i=1}^{\sigma} p_i \log p_i$ is the entropy of the distribution P. Since we know that the size of the encoded text is $n \cdot H(P)$ in the first case and $n \cdot H(P')$ in the second, we conclude that there is practically no loss in compression efficiency.

In the cryptographic application of [2], the bit sequence generated by the random bit generator is kept as a secret key K shared only by encoder and decoder. Using different keys yields completely different output files, and there seems to be no easy way to decipher the message without guessing K, yet the sizes of the compressed files were practically unchanged for different keys, as long as their 1-bit density was kept at $\frac{1}{2}$.

3.2 Dynamic Huffman Coding

We assume, for simplicity, that all dynamic Huffman variants initialize the tree with all the characters of the alphabet. We suggest two approaches for applying the proposed selective method in case of dynamic Huffman coding. Both algorithms still update the model periodically or via a random bit generator. The difference between them relates to the way the model gets updated.

The first approach, analogous to the one used for arithmetic coding, updates the dynamic Huffman tree by advancing the frequency of the current character only, ignoring the previous characters that have been skipped, and using Vitter's Algorithm in order to fix the Huffman tree in case the sibling property is violated. The first variant is denoted by Huf-subset.

The second approach, denoted by Huf-full, performs the updates according to the changes in the frequencies of *all* the characters seen since the last update. Practically, the updates in this case are spaced out and only done at the end of a *block* of several characters, therefore, immediately after the processing of a set bit in Algorithm 1, the updated Huffman tree reflects all symbols processed so far. As Vitter's Algorithm is based on updating the Huffman tree when only a single character frequency has been incremented by 1, we generated a static Huffman tree from scratch to implement the second suggestion.

Obviously, a bad selection may in the worst case jeopardize any compression savings, for example, when processing fixed length lines of 80 characters and choosing $k = 80$. The subset of Σ would then consist of the newline character only! On the other hand, the selective Huffman algorithm can produce a compressed file that is not only marginally smaller than the file constructed by traditional dynamic Huffman, but may even reach only about $\frac{3}{4}$ of its size, as shown in the following example.

Let $T = \mathtt{B}\{\mathtt{CCBB}\}^t$ for some positive integer t. For both variants, the Huffman tree is initialized with $\Sigma = \{\mathtt{A}, \mathtt{B}, \mathtt{C}\}$ as shown in Figure 1(a). Consider first the traditional dynamic algorithm as proposed in [5]. The first `B` is encoded as two bits, 10, and `B` is exchanged with `A`, as shown in Figure 1(b). When processing `C` of the first quadruple `CCBB`, only the second `C` causes a change in the structure of the Huffman tree, but this happens *after* the two `C`s have already been encoded by 11, using 2 bits each. The Huffman tree after reading the prefix `BCC` of T, is depicted in Figure 1(c). When the following two `B`s of the first quadruple `CCBB` are processed, again the positions of the `B` and `C` nodes are swapped only *after* the frequency of `B` exceeds that of `C`, so each of the `B`s is also encoded by two bits (11). This alternating structure between two different Huffman trees proceeds until the end of the file, thus every character of T uses 2 bits, for a total of $2|T| = 8t + 2$ bits.

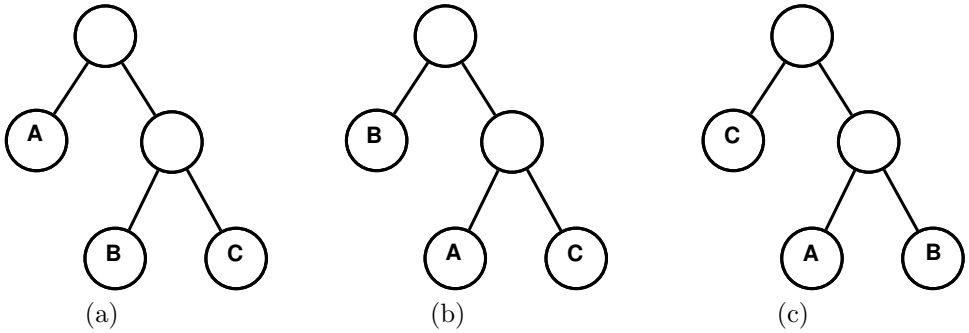

Figure 1. Example for which selective Huffman coding produces a file $\frac{3}{4}$ of the size of that constructed by standard Huffman.

However, when selective Huffman coding is used with $k = 2$, the first `B` is again encoded as two bits, 10, and `B` is exchanged with `A`, as shown in Figure 1(b). When processing the following `C`, it does not change the model, and is encoded with 2 bits as 11. The following `C` does cause an update of the frequencies, but does not change the structure of the Huffman tree. When the following two `B`s of the first quadruple `CCBB` are processed, they are both encoded as 0, using a single bit, and only the second `B` causes an update of the frequencies in the tree, but no change in its structure. This behavior proceeds until the end of the file, where the structure of the tree remains the same, and only the frequencies of the `B`s and `C`s get updated, encoding all `B`s as 0, and all `C`s by 11. Thus every quadruple `CCBB` uses 6 bits, for a total of $6t + 2$ bits, which is about $\frac{3}{4}$ of the regular encoding.

A random selection of the characters that are chosen for updating the model may be used to turn a dynamic Huffman encoder into a cryptosystem. Although in the long run, the distribution of the characters in the selected subset will be so close to their distribution in the entire text that there will be no noticeable loss in the compression, the small details of when exactly to increment which frequency will have a cumulative impact, producing ultimately completely different output sequences. Nevertheless, the sizes of these output files will be very close. As example, we compressed a French text file of 7.3MB with five independently generated random keys, and got compression ratios 0.57801, 0.57812, 0.57803, 0.578058 and 0.578061.

3.3 LZW

We turn to LZW [6], which is a member of the family of dictionary methods introduced by Ziv and Lempel [8,9], unlike the statistical compressions dealt with in the previous sections. The dictionary is initialized by the single characters of the alphabet, and then is updated dynamically by adding newly encountered substrings that have not been seen previously in the parsing of the underlying text. The text is thereby partitioned into a sequence of longest possible phrases that already occur in the dictionary, and the encoded file is a list of indices, each pointing to the entry of the corresponding phrase. LZW starts with a dictionary of size 512 which is filled to half its capacity by the alphabet of ASCII symbols, and each entry index is encoded by 9 bits. Once the dictionary has filled up after adjoining 256 new phrases, its size is doubled to 1024 entries, and all dictionary entries are encoded from this point on using 10-bit pointers.

In general, after processing $2^9, 2^{10}, \ldots, 2^i, \ldots$ more elements of the input file, the size of the dictionary is doubled to $2^{11}, 2^{12}, \ldots, 2^{i+2}, \ldots$ entries, up to a predetermined maximal size, which is 2^{16} in our implementation. We consider two variants:

1. restarting the dictionary from scratch each time the dictionary reaches its maximum size, denoted here by LZW-restart; or
2. considering the dictionary as static once it gets full, and not adjoining any more strings, denoted by LZW-static.

The dictionary built by LZW maintains a *prefix property*, that is, for each substring in the dictionary, all its prefixes also are included in the dictionary. In both cases of the selective variant, the dictionary gets filled up at a slower pace than for the traditional approach. As not all elements of the dictionary are necessarily used later, there are situations where the selective variant may improve the space savings of the traditional one, as can be seen in our experimental results presented in the following section.

Similarly to what could be done for Huffman coding, if the subset of newly encountered substrings is chosen randomly according to a secret key K, it will be hard to decode the file for whoever has to guess the 1-bits in K. Even though the compressed file consists of a sequence of integers and those are easy to decipher, it is their interpretation as representing certain substrings that is only known to the encoder and decoder. As before, different keys produce completely different dictionaries and thus different output strings, but there is hardly any change in the size of these files. For the same example input as in the previous section, the obtained compression ratios for 5 different keys were 0.3988, 0.3978, 0.3994, 0.3987 and 0.3981.

4 Experimental Results and conclusion

We examined the traditional and selective methods comparing the compression efficiency as well as the encoding and decoding processing times using the three basic adaptive compression algorithms mentioned above. We considered the 50MB file *English*, downloaded from the Pizza&Chili Corpus, which is the concatenation of English text files selected from etext02 to etext05 collections of the Gutenberg Project, from which the headers related to the project were deleted so as to leave just the real text. All experiments were conducted on a machine running 64 bit Windows 10 with an Intel Core i5-8250 @ 1.60GHz processor, 6144K L3 cache, and 8GB of main memory.

Our first experiment compares the compression efficiency of the traditional vs. selective variants of the various adaptive compression methods: Huf-subset, Huf-full, LZW-restart, LZW-static and arithmetic codings. The results are given in Figure 2 depicting the compression ratio (defined as the size of the compressed over the size of the original file) for $k = 1, \ldots, 8$ and $k = 16, 32$. The traditional dynamic algorithms correspond to $k = 1$, where the model gets updated after every character.

As can be seen, LZW-restart is less effective as the blocks become larger. However, when the dictionary stays constant once it gets filled up, the compression improves for larger blocks, until a point (not depicted in the graph), where the compression gain declines. According to our results, the compression efficiency becomes worse than for the traditional LZW for $k = 256$. This phenomenon can be explained by the fact that the static variant needs only a single learning period, while the other method undergoes, after each restart, a new learning phase during which the compression is less effective. In any case, the difference between selective and traditional techniques is less than 3%.

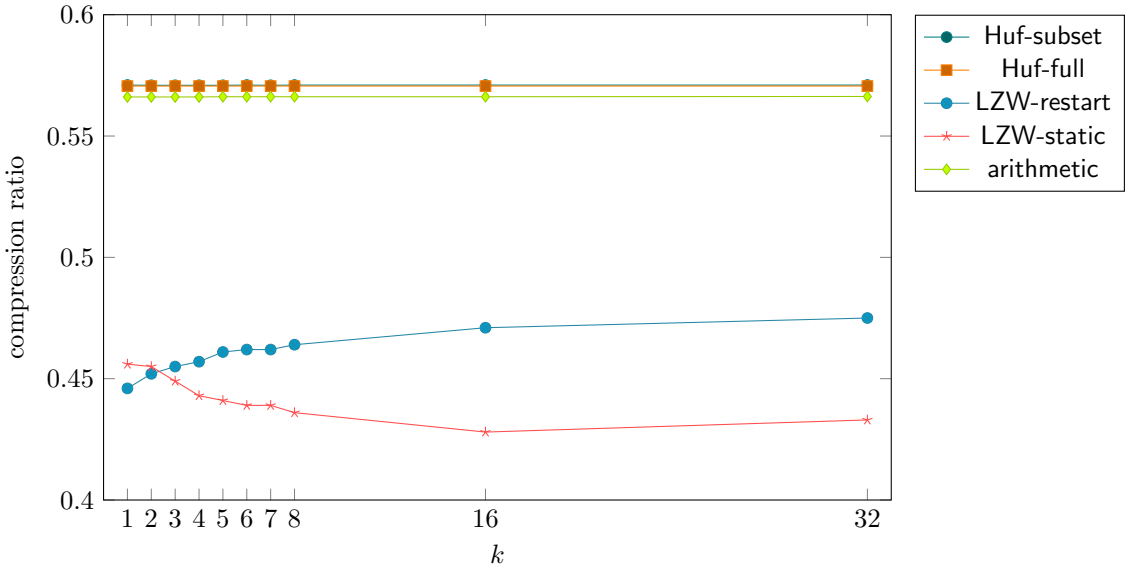

Figure 2. Compression Efficiency as a function of k.

The performance of the blockwise dynamic Huffman encoding, Huf-full, is practically identical to that of the partial update variant, Huf-subset, and their plots are overlapping in Figure 2. As for arithmetic coding, our results coincide with the theory mentioned above, and selective arithmetic compression remains very similar to the traditional one, as the probabilities are quite the same.

Figure 3 shows the processing times for both compression and decompression, for the standard and selective methods. For each of the test files, the encoding and decoding times were averaged over 10 runs. The displayed times are averages, given in seconds. Obviously, the compression and decompression times improve as the spacing becomes larger, because a smaller number of model updates is required.

	Compression ratio		Encoding time		Decoding time	
	Trad	Rand	Trad	Rand	Trad	Rand
Huf-subset	0.571	0.571	4447	2789	2388	1190
LZW-restart	0.452	0.446	10.667	9.781	5.321	4.348
LZW-static	0.456	0.454	8.796	8.771	3.096	3.097
arithmetic	0.5661	0.5661	41.61	27.11	46.65	32.39

Table 1. Results for Random as a function of k.

Table 1 presents the results for a random selection in which the choice to update the model is determined by a random bit generator (columns Rand). The columns headed Trad bring the results of the traditional, non-selective, approach. A random selection is similar to a fixed length selection model with $k = 2$, but has the advantage of almost surely avoiding worst case scenarios, like, e.g., a file for which even indexed characters belong to a different and disjoint alphabet than the odd indexed characters. As can be seen, the compression efficiencies are about the same, and that of the selective choice may sometimes be better. The processing times are consistently better, suggesting the usefulness of a selective encoder.

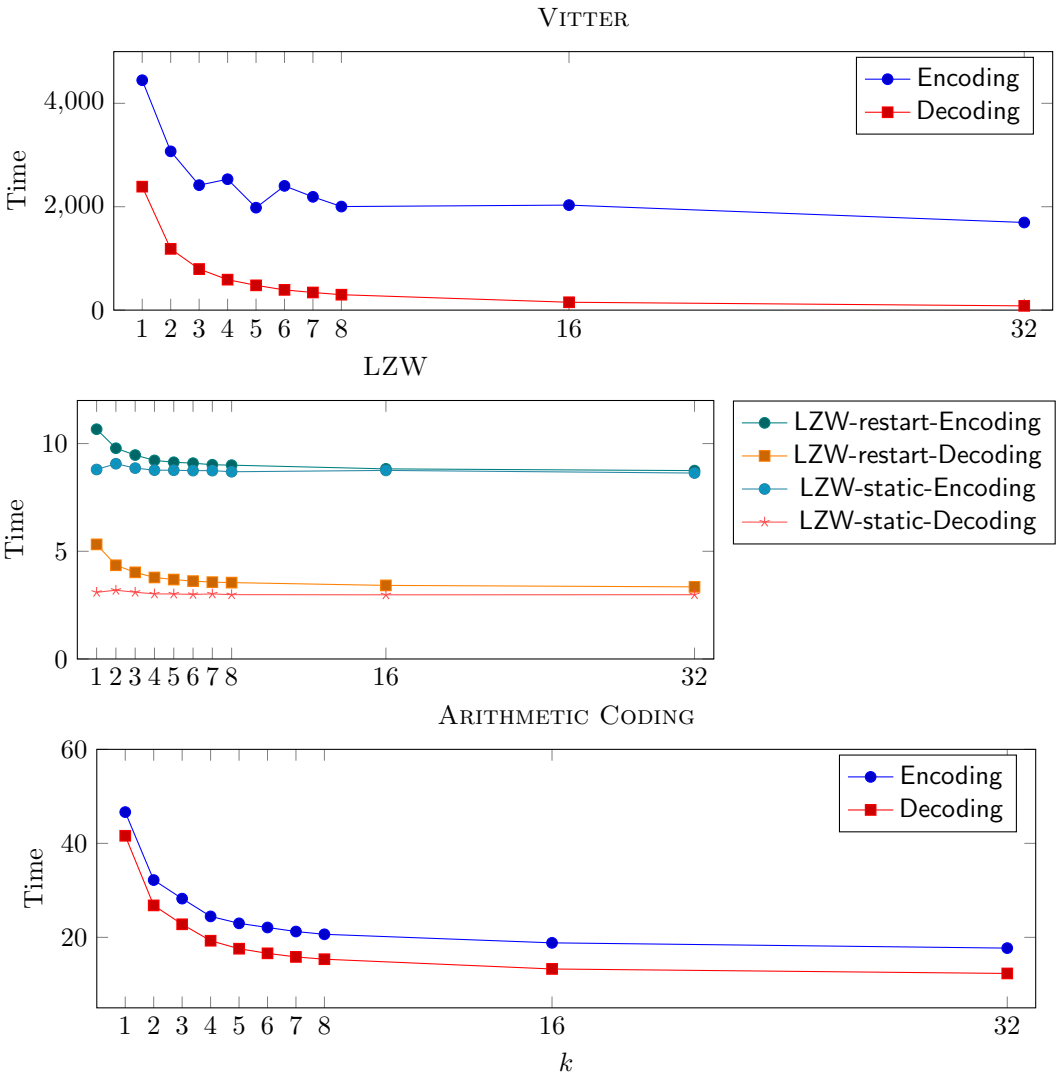

Figure 3. Processing Times as a function of k.

References

1. S. T. Klein, E. Opalinsky, and D. Shapira: *Synchronizing dynamic Huffman codes*, in Proceedings of the Prague Stringology Conference 2018, Czech Technical University in Prague, Czech Republic, 2018, pp. 27–37.
2. S. T. Klein and D. Shapira: *Integrated encryption in dynamic arithmetic compression*, in Language and Automata Theory and Applications - 11th International Conference, LATA 2017, Umeå, Sweden, March 6-9, 2017, Proceedings, 2017, pp. 143–154.
3. P. Norvig: *English letter frequency counts: Mayzner revisited or etaoin srhldcu*, 2018, `http://norvig.com/mayzner.html`.
4. J. A. Storer and T. G. Szymanski: *Data compression via textual substitution.* J. ACM, 29(4) 1982, pp. 928–951.
5. J. Vitter: *Design and analysis of dynamic Huffman codes.* J. ACM, 34(4) 1987, pp. 825–845.
6. T. Welch: *A technique for high-performance data compression.* IEEE Computer, 17(6) 1984, pp. 8–19.
7. I. Witten, R. Neal, and J. Cleary: *Arithmetic coding for data compression.* Commun. ACM, 30(6) 1987, pp. 520–540.
8. J. Ziv and A. Lempel: *A universal algorithm for sequential data compression.* IEEE Trans. Information Theory, 23(3) 1977, pp. 337–343.
9. J. Ziv and A. Lempel: *Compression of individual sequences via variable-rate coding.* IEEE Trans. Information Theory, 24(5) 1978, pp. 530–536.