© The British Computer Society 2020. All rights reserved.
For permissions, please e-mail: journals.permissions@oup.com
doi: 10.1093/comjnl/bxaa074

UISA: Decoupling the Frequency Model From the Context Model in Prediction-Based Compression

VINICIUS FULBER-GARCIA[1,*] AND SÉRGIO LUIS SARDI MERGEN[2]

[1]*Laboratory of Networks and Distributed Systems, Department of Informatics, Federal University of Paraná, Curitiba, Paraná, Brazil*
[2]*Department of Languages and Computer Systems, Federal University of Santa Maria, Santa Maria, Rio Grande do Sul, Brazil*
**Corresponding author: vfgarcia@inf.ufpr.br*

Prediction-based compression methods, like prediction by partial matching, achieve a remarkable compression ratio, especially for texts written in natural language. However, they are not efficient in terms of speed. Part of the problem concerns the usage of dynamic entropy encoding, which is considerably slower than the static alternatives. In this paper, we propose a prediction-based compression method that decouples the context model from the frequency model. The separation allows static entropy encoding to be used without a significant overhead in the meta-data embedded in the compressed data. The result is a reasonably efficient algorithm that is particularly suited for small textual files, as the experiments show. We also show it is relatively easy to built strategies designed to handle specific cases, like the compression of files whose symbols are only locally frequent.

Keywords: data compression; lossless compression; prediction by partial matching; prediction tree; LUISA; compression methodology

Received 17 May 2019; Revised 21 March 2020; Editorial Decision 19 May 2020; Accepted 19 May 2020
Handling editor: Natasha Alechina

1. INTRODUCTION

Prediction by partial matching (PPM) is a compression method that encodes the probability of a symbol appearing after a context (a sequence of preceding symbols) [5]. This method achieves overwhelming compression ratios for texts written in natural language, where exact contexts appear frequently, and acceptable ratios for other file formats. The pitfall is execution time. The cost of compression/decompression is a prohibitive factor. Also, the overall process is highly sensitive to the frequency of the symbols. If a symbol is very frequent in a context, it is greatly compressed. Compression is harmed if a symbol starts to appear less often.

Those problems are (at least partially) related to the fact that the frequency model is highly coupled with the context model. To encode a symbol, the method finds the probability of the symbol occurring in the current context. Then, an entropy encoding method (usually arithmetic encoding) is applied in order to achieve compression. This scheme demands the usage of dynamic entropy encoding, which is considerably slower than the static alternative. The reason is that it would be too expensive in terms of space to use a static method where frequencies need to be known for every known context.

In this paper, we propose LUISA, a compression method based on PPM that decouples the context model and the frequency model. Instead of finding the probability of a symbol in its context, the method finds a key that uniquely identifies a symbol in its context. On a later (and separate) stage, the key is compressed using any form of entropy encoding. This scheme allows several techniques to be used, which would be cumbersome to implement if the frequency model was built as part of the context model. For instance, LUISA enables straightforward implementations that do not necessarily rely on symbols frequencies and the application of static arithmetic encoding. Experimental results show how the techniques can be combined in order to achieve fast compression/decompression associated with compelling compression ratios. We also show

Downloaded from https://academic.oup.com/comjnl/article-abstract/doi/10.1093/comjnl/bxaa074/5868170 by University of New England user on 11 July 2020

how the method can be tuned to enhance compression when symbols are only locally frequent.

This paper is organized as follows: Section 2 briefly explains how compression is achieved with PPM. Variations of PPM are also discussed. Section 3 presents the proposed method and how it differs from PPM. We also discuss alternative strategies that arise for the problem of generating unique keys. In Section 4, we present the experimental results, where alternative strategies are combined into different versions that trade compression ratio and execution time. Comparisons against PPM and other state-of-the-art methods are provided as well. Section 5 brings our final remarks.

2. PREDICTION BY PARTIAL MATCHING IN A NUTSHELL

PPM is a prediction-based method that transforms symbols from a finite alphabet A into a coded output. The prediction process is streamed, so that incoming symbols are encoded one at a time. Firstly, the method locates the symbol to encode within a context model. The model states which symbols have already appeared for contexts of varying lengths. Contexts are represented as Markov models of orders ranging from zero to a previously defined maximum (n).

Once the symbol is found, a prediction is obtained. The prediction is a probability interval that uniquely identifies the symbol to be encoded. Then, dynamic arithmetic coding is used to combine the probability intervals of the incoming symbols into an increasingly lower probability interval, which eventually is transformed into a bit sequence. Larger intervals (reserved to the most probable symbols) lead to smaller sequences and greater compression.

The probability of a symbol in a context is built based on its frequency, i.e. how many times the symbol has occurred in that context. The 'Null Probability' problem occurs when an incoming symbol has not yet been seen in a context. In those cases, the interval probability of the escape symbol is issued. The escape symbol is not part of the alphabet. Its sole purpose is to notify the compression method that the symbol to encode does not exist in the current context of length i, and the search must continue in the order $i-1$. A special order (-1) contains all symbols of the alphabet. In the worst-case scenario, a sequence of escape symbols is issued until order -1 is reached, where the symbol to predict definitely exists.

To understand, consider the example illustrated in Table 1. Assume a small alphabet A composed by the symbols {'A', 'B', 'C', 'D', 'E'}. Also, assume the maximum order is two ($n = 2$), and that the current context is 'DE'. The example shows lists of symbols that have already appeared in contexts of lengths 2 ('DE'), 1 ('E'), 0 ('') and -1, where all symbols exist. Frequencies are in parenthesis. For instance, symbol 'A' has already appeared three times after context 'E'. The escape symbol intrinsically appears in all lists but the last one.

TABLE 1. Context information used by PPM.

Order	Context	Symbols				
2	'DE'	E(2)				
1	'E'	A(3)	E(2)	C(1)		
0	''	A(5)	E(4)	B(4)	C(4)	
-1	-	A(1)	B(1)	C(1)	D(1)	E(1)

The frequency of the escape can be determined in different ways. One of the earlier solutions, called PPM-C, computes the frequency as the number of times the escape was issued [21].

Suppose the next symbol to encode is 'E', as in the third character of 'DEED'. In this case, a single probability interval is issued, identifying the symbol 'E' in order 2. If, however, the next symbol to encode was 'C' (as in the third character of 'DECADE'), two probability intervals are issued, identifying the escape symbol at order 2 and the symbol 'C' in order 1. After the coding, the model is updated by increasing the frequency of the symbol. If a symbol is new in an order, its initial frequency is set (usually the initial value is one). The most frequent symbols are assigned with the larger probability intervals. Hence, the occurrence of frequent symbols results in smaller codes.

Over the years, many PPM-based methods were proposed. In general, the main concern was on improving the prediction to enhance the compression of conventional textual files. This was achieved by several means, such as adjusting the escape probability estimation [25], cleaning irrelevant contexts [9] and allowing unbounded context lengths [27]. One of the most compelling alternatives (PPMII) achieves good compression with several tweaks, which includes setting the initial frequency of a symbol using information inherent from a lower-order context [26]. Despite the achievements, very little was done about PPM's most severe weakness: execution time.

One of the costlier operations is arithmetic coding [22]. Existing dynamic arithmetic encoding implementations are slow [17]. On the other hand, it is practically unfeasible to use static coding for orders higher than one, giving the space overhead required to store the frequencies for all contexts (the frequencies need to be packed along with the compressed data to enable loss-less decompression).

Another side effect of using frequencies in PPM style is that recency is not properly handled. Symbols that are only locally frequent have low counts, resulting in low compression. The problem can be alleviated by using a scale factor to increment the frequencies. However, it is at best a workaround that does not address the root problem: using the global frequencies to issue predictions.

The most recent developments are quite old. The efforts are focused on the usage of PPM for the compression of specific file formats, such as log files [28], JPEG [32] and XML [29]. In those particular scenarios, and given some assumptions about

Downloaded from https://academic.oup.com/comjnl/article-abstract/doi/10.1093/comjnl/bxaa074/5868170 by University of New England user on 11 July 2020

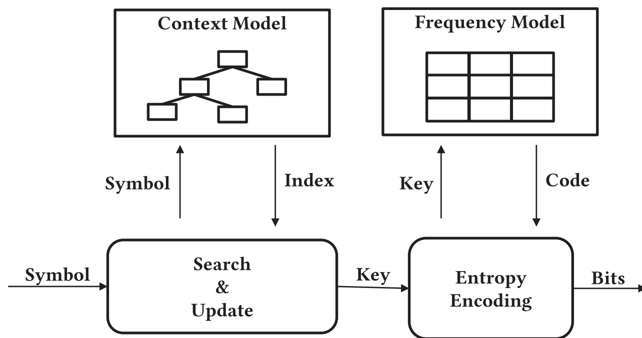

FIGURE 1. LUISA's architecture: the context model and the frequency model are detached.

the data, efficiency and recency are handled. However, these are still open challenges when it comes to the compression of text files in general.

Although the PPM core implementation has not modified in recent years, their application in different areas has been extensively explored. For example, in [24], [16], [18], [8] and [23], the PPM method is used as part of solutions for genome and amino acid compression, while authors in [2] used it for data transmission over networks.

Finally, the PPM prediction model can be adopted for highly abstracted tasks, such as anticipating Web page accesses according to the users' browsing history [14] and automated language identification and discrimination between similar languages [20]. There is a myriad of other uses, such as the ones presented in [31], [1], [30] and [11]. These recent applications show that the PPM is a state-of-the-art compression method that is still open for improvements and adjustments to particular scenarios.

3. LUISA

LUISA[1] shares the same core processing as PPM, which involves (i) using the context model to locate the symbol to encode, (ii) using the frequency model to locate the probability interval that uniquely identifies the symbol to the encoded, (iii) encoding that interval and (iv) updating the context model. The main difference is that LUISA detaches the frequency model from the context model.

Figure 1 shows how LUISA transforms input symbols into a compressed output. Two independent stages are defined. We call them 'Search & Update' and 'Entropy Encoding'. The first searches the symbol to encode and updates the model. Before updating the model, it emits a key that uniquely identifies the input symbol. In the second stage, the key is encoded using any entropy encoding method. The role of the 'Search and Update' stage is to reduce the entropy of the original message, so the later stage can actually achieve compression.

Note that the concept of producing an intermediary message with a highly skewed frequency distribution is also the base of block-sorting compression, where the 'Move to Front' stage generates long runs of small-valued codes [13].

Additionally, symbol rank refers to a family of methods that keep the symbols within a context ranked by their likelihood to appear and encode the rank. One of the latest developments (whose implementation is available) is sr2/sr3 [19]. They are improvements over srank, a method that keeps only two orders (order-3 and order-0), where the highest order stores the three most recent symbols [12].

In general, symbol rank methods propose specific solutions to either the problem of keeping contexts information manageable (such as srank/sr2/sr3) or the problem of efficiently encoding the rank (e.g. by using quasi-arithmetic coding [15]).

We, on the other hand, focus our work on completely decoupling the probability estimator from the context model. By keeping these components in separate layers, we can easily switch between alternative strategies to generate keys, update the model (rank the symbols) and compress the keys. Also, no strong constraints are imposed on the context model or the entropy encoder. It enables one to try different settings over a full context model (up to a maximum defined order) and to possibly conceive new strategies to work with files whose symbol distribution is somehow biased.

In what follows, we present some of the techniques that can be used. These techniques are reversible so that the compression is lossless. We also discuss alternative context model implementations that support the required search and update operations.

3.1. Key generation

The 'Search & Update' stage is ultimately responsible for producing a key that identifies a symbol. The key is an index that uniquely locates the symbol in the current context. As with PPM, the search starts in the highest possible order. If the symbol is not found, the search proceeds to the next order. The process continues until the symbol is found.

Part of the problem involves coding symbols that have not been seen yet in higher orders. As noted in Section 2, this is called the zero/null probability problem, which PPM handles by using an escape code. In LUISA, the never-seen symbols are handled by an abstraction that we call 'Impossible Key'. The Impossible Key is an index that does not refer to any of the symbols that exist in the current order. Its presence means the search must proceed to the inferior orders.

To understand, consider the same example used in the past section. Figure 2 brings an updated view of the symbols in each context. The indexes above the lists mark both the absolute position of the symbols in the context and the relative position of the symbols, ignoring all symbols that already appeared

1 Available at https://github.com/ViniGarcia/LUISA.

Downloaded from https://academic.oup.com/comjnl/article-abstract/doi/10.1093/comjnl/bxaa074/5868170 by University of New England user on 11 July 2020

Order	Context	Symbols				
		0 <= absolute index ***0*** <= relative index				
2	'DE'	E(2)				
		0 <= 'without exclusion' key ***0*** <= 'with exclusion' key				
		0 ***1***	**1**	**2** ***2***		
1	'E'	A(3)	E(2)	C(1)		
		1 ***1***		**3** ***2***		
		0	**1**	**2** ***3***	**3**	
0	''	A(5)	E(4)	B(4)	C(4)	
				6 ***3***		
		0	**1**	**2**	**3** ***4***	**4**
-1		A(1)	B(1)	C(1)	D(1)	E(1)
					11 ***4***	

FIGURE 2. Context information used by LUISA.

in superior orders. The values below the lists mark the key attached to each symbol by using the techniques of 'With Exclusion' and 'Without Exclusion' (discussed next).

Given that the current order is the highest order in which the symbol to encode appears, we devise two strategies for the unique key generation:

Without Exclusion. The unique key is the count of all symbols that appear in the superior orders and the absolute index of the desired symbol in the current order. This single key uniquely locates both the order and the position of the symbol within the order.

With Exclusion. The unique key is the relative index of the desired symbol in the current order. It resembles the 'Exclusion' technique used in PPM, which computes the probabilities in a context considering only the symbols that have not appeared in the higher orders [6].

Table 2 shows the keys produced for every symbol of the alphabet by the 'Without Exclusion' and the 'With Exclusion' techniques. When using exclusions, every symbol is accounted only once, so the maximum possible key is $|A|$-1. When no symbol is excluded, the same symbol may be accounted multiple times. In the worst (and very unlikely) case scenario, where most symbols occur in all orders, the maximum key is proportional to $(|A| - 1) \times n$.

Keys produced with the 'Exclusion' technique tend to concentrate more in few indexes, leading to messages with lower entropy. The counterpart is the computational cost. Algorithms 1 and 2 show pseudo-codes for decoding a symbol based on the key. As we can see, 'Without Exclusion' implementation is rather straightforward. On the other hand, 'With Exclusion' needs more work to count only unseen symbols.

TABLE 2. Exclusion table.

Symbol	'With Exclusion' Key	'Without Exclusion' Key
E	0	0
A	1	1
C	2	3
B	3	6
D	4	11

Algorithm 1 Decoding without exclusion

```
decode(key)
1: for i = n to 0 do
2:    if length(order_i) <key then
3:       return get(order_i, key);
4:    end if
5:    key ← key - length(order_i);
6: end for
```

Algorithm 2 Decoding with exclusion

```
decode(key)
1: for i ← 0 to |A|-1 do
2:    seen[symbol(i)] = false;
3: end for
4: key2 ← −1;
5: for i ← n to 0 do
6:    for j ← 0 to length(order_i) do
7:       symbol ← get(order_i, j);
8:       if seen[symbol] = false then
9:          seen[symbol] ←true;
10:          key2++;
11:          if key2 = key then
12:             return symbol;
13:          end if
14:       end if
15:    end for
16: end for
```


Downloaded from https://academic.oup.com/comjnl/article-abstract/doi/10.1093/comjnl/bxaa074/5868170 by University of New England user on 11 July 2020

These two algorithms serve merely to (conceptually) state the basic differences between the approaches. There is a lot of room for optimization. For instance, the 'With Exclusion' strategy can be adjusted to reduce the computation cost by starting to count from the order immediately above the order in which the symbol was actually found. The 'Without Exclusion' strategy can be adjusted to reduce the range of possible keys by comparing the lengths of two orders, and ignoring all symbols of the lower order if the lengths are the same, as no new information is presented.

3.2. Context update

Whenever a symbol is encoded, its occurrence is used to update the context model. The idea is to continuously enhance the model used to predict the forthcoming symbols. In PPM, updating the context means increasing the frequency of the symbol in the orders where it exists. This technique assumes that the most frequent symbols in a context are most likely to appear again. The classical example is the compression of files written in natural language. However, it is not suited if symbols are only locally frequent. Examples include archives that combine several files into one, where patterns change drastically from one file to the next. In those cases, a symbol that is very frequent within a file may cease appearing in the next one. Another example is files composed of ordered words, such as dictionaries. A symbol stops appearing as soon as the next symbol in the lexicographic order starts appearing.

There are other problems associated with the management of frequencies. For instance, the memory overhead for storing the values and the need to periodically re-scale the frequencies to prevent overflows [21].

In contrast, the update method adopted by LUISA is not necessarily bounded to frequencies. What really matters is to order the symbols in a context by their estimated probability of occurrence. The symbol deemed most likely to occur again appears at index zero, and it is followed by the second symbol most likely to appear, and so on.

If the symbols to encode are found in the front of their corresponding lists, the entropy encoding stage will receive few keys with high frequencies, which leads to better compression. Therefore, it is important to devise techniques that promote the most probable symbols to the front of their corresponding lists.

The promotion can be based merely on frequency, as with PPM and other adaptive compressors [4]. However, techniques that do not rely on frequencies can be easily conceived. To explain, consider the task of updating the symbol 'C' that has just appeared in the context 'DE'. In what follows, we discuss three alternatives to carry the update (illustrated in Fig. 3).

- **Frequency Swap**: The symbol is moved by comparing its frequency with the ones from their front neighbors. In the given example, the symbol 'C' would jump two positions ahead, and symbols 'E' and 'B' would be pushed back by one position.

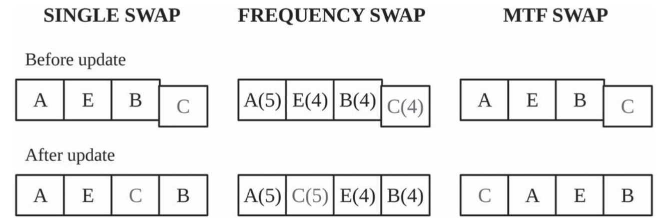

FIGURE 3. Strategies to update the absolute position of the symbols.

- **Transpose Swap**: No frequency is used. Instead, the symbol is moved one position ahead. In the given example, the symbol 'C' would be swapped with the symbol 'B'.
- **Move to Front Swap**: No frequency is used. Instead, the symbol is moved to the head of the list (as in BWT). In the given example, the symbol 'C' would jump to the first position, and symbols 'A', 'E' and 'B' would be pushed back one position.

Regarding the compression ratio, 'Frequency' is preferable if the symbols are globally frequent, which is the general case. If symbols are only locally frequent (such as in dictionaries), the 'Move to Front' swap is potentially better. 'Transpose' is a strategy that can be placed between these two extremes. Frequent symbols will continuously exchange positions at the front part of the list. When compared to the strategy that relies on frequencies, 'old' symbols that are currently much more frequent than the rest will leave the front sooner.

'Transpose' requires a single swap and no comparisons are ever made. This branch-less design favors the instructions pipelining used by super-scalar processors. 'Move to front' also requires no comparisons. If carefully implemented, updating the position is cheap (just a matter of updating pointers of a linked list). The number of comparisons and swaps of 'Frequency' is variable. For long enough files, if symbols are globally frequent, most symbols will hardly need to swap more than once.

One optimization proposed for PPM is called 'Update Exclusion' [21]. When used, only the order where the symbol was found is updated. Inferior orders remain untouched. This mode reduces the update cost. Besides, experiments show that compression is generally improved. The same optimization is used inside LUISA.

The presented alternatives are not an exhaustive list of possibilities, neither are they new. All three were originally conceived as heuristics to self-organized sequential searches [3]. Our intention here is to demonstrate that it is straightforward to come up with strategies tailored at specific ends. We expand this discussion in the experimental section, where different update implementations are compared both in terms of compression ratio and execution time.

3.3. Entropy encoding

One of the most important properties of LUISA is that the 'Entropy Encoding' stage is completely decoupled from the

Downloaded from https://academic.oup.com/comjnl/article-abstract/doi/10.1093/comjnl/bxaa074/5868170 by University of New England user on 11 July 2020

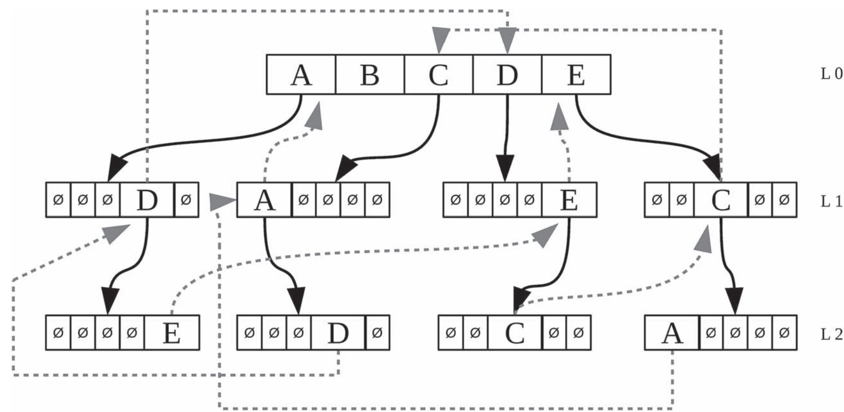

FIGURE 4. Context model storing the all contexts up to three characters of the sequence 'DECADE'.

'Search & update' stage, as depicted in Fig. 1. It makes the first stage an independent module whose probability model has no relation with the context model. The separation allows virtually any entropy compressor to be used as a black box. Also, keys generated by the 'Search & Update' stage can be buffered before entering the second stage. This enables static entropy encoding to be used, in a block-based pipeline.

An obvious choice is using static Huffman as a fast method to achieve compression. While arithmetic encoding struggles with the processing of multiplications and divisions (at least during decompression), Huffman offers a better solution based on table lookups. A recent implementation (Huff0) makes the process even faster by removing branches and decoding multiple symbols at once [7].

However, it is not as effective as arithmetic encoding at approximating the entropy of a message. Huffman needs at least one bit to encode a symbol, whereas arithmetic encoding can use a fraction of a bit to encode a symbol. Recently, a breakthrough in entropy encoding was proposed to overcome the speed barrier. The method, called finite state entropy (FSE), is based on asymmetric numeral systems, a theoretical way to approximate Shannon entropy by using low state entropy coding automata [10]. Experimental results show that the method is as fast as Huffman and provides a reasonable compression when compared to arithmetic encoding. Both FSE and Huff0 were evaluated as part of LUISA, as we detailed in Section 4.

3.4. Implementation designs

The core processing of LUISA uses the context model to (i) perform the symbol search, (ii) update the model and (iii) propagate the symbol location to the entropy encoder. It is important to build the model in such a way that these operations can achieve highly efficient results in terms of execution time.

Essentially, the n-order Markov models that form the contexts are represented as n-ary tries (or prefix trees). Nodes represent symbols, and the path from the root to a node forms a context. The children of a node are the symbols that have already occurred in the corresponding context. Suffix links connect a symbol in order i with the same symbol in order $i-1$. If a match is not found at a given order, the suffix link leads directly to the next order.

In what follows, we discuss two distinct context model implementations: one memory-intensive solution based on symbol tables and one memory-friendly solution based on dynamic sets.

3.4.1. *Using symbol tables*

The symbol table is an array of fixed size $|A|$ that informs which symbols from the alphabet A have occurred for a given context. Figure 4 illustrates an example where a small alphabet composed by the symbols $\{A, B, C, D, E\}$ is identified by codes ranging from zero to four.

There is one table per context, and it provides direct access to every symbol in an order. For instance, index zero always leads to symbol A, and index four always leads to symbol E. Indexes that lead to null elements represent symbols that have never occurred in that particular context. Suffix links appear as dotted lines.

The tables can be seen as collision-free hash functions that come at the expense of high memory consumption. Each of the i orders of the model would demand $|A|^{i-1}$ arrays with $|A|$ elements each. Moreover, the tables can be quite sparse, especially at the highest orders.

A table t contains two parts: the position pos of the symbol identified by i, and the index idx of the symbol whose position is i. For instance, the index of the best ranked symbol ($pos = 0$) can be found in $t[0].idx$. Similarly, the position of the symbol identified by the index 3 can be found in $t[3].pos$. During compression, after the symbol s to encode is found, the purpose is to issue its position within the current context. This information is found at $t[s].pos$. During decompression, given a key k, the correspondent symbol within the current context is found at $t[k].idx$.

Downloaded from https://academic.oup.com/comjnl/article-abstract/doi/10.1093/comjnl/bxaa074/5868170 by University of New England user on 11 July 2020

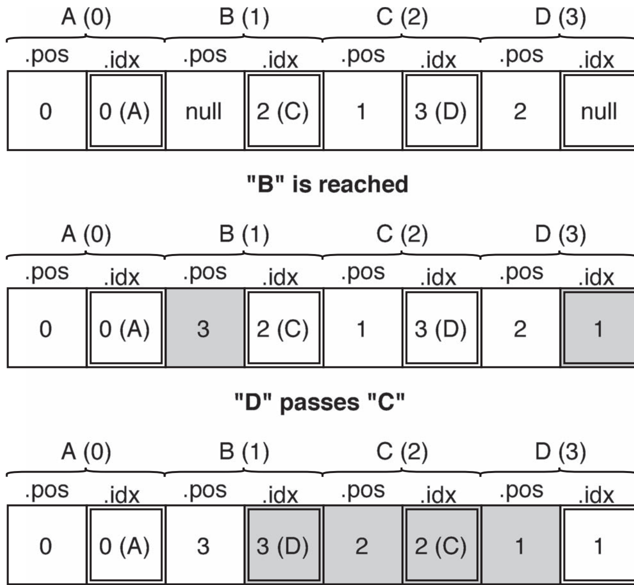

FIGURE 5. Symbol table update process using a limited dictionary (ABCD).

Figure 5 shows examples of symbol table update. In the first event, the new symbol 'B' is reached by the algorithm. To update this, symbol two operations are, in order, executed: $t[B].pos = first_null_idx(t)$ (where $first_null_idx(t)$ returns the lowest position of a symbol table t where $.idx$ is null) and $t[t[B].pos].idx = D$. In the second event, 'D' symbol is promoted, assuming the position of 'C'. Thus, *.pos* attributes of both 'D' and 'C' are exchanged, enabling the execution of $t[t[D].pos].idx = D$ and $t[t[C].pos].idx = C$.

It is relatively easy to conceive algorithms to update the positions of the symbols. Algorithm 3 shows in pseudo-code how to update the table when the transpose strategy is used. Lines 1–4 are devoted to locating the symbols to swap. Function $index(s)$ simply returns the decimal representation of the symbol s, a value between 0 and 255. Given this index, the position is immediately found (line 2), as well as the symbol that occupies the previous position (line 4). Lines 5–8 are devoted to swapping the positions of the two symbols.

Algorithm 3 Updating the model when using the transpose strategy

updateModel(Symbol s)

- 1: index_cur ← index(s)
- 2: pos_cur ← t[index_cur].pos
- 3: index_prev ← t[pos_cur-1].idx
- 4: pos_prev ← pos_cur-1;
- 5: t[pos_prev].idx ← index_cur
- 6: t[pos_cur].idx ← index_prev
- 7: t[index_prev].pos ← pos_cur
- 8: t[index_cur].pos ← pos_prev

Different strategies work similarly. For instance, moving a symbol located at position *pos* to the front of the list requires changing the position and index values of all elements whose positions are smaller than *pos*, as described by Algorithm 4.

Algorithm 4 Updating the model when using the MTF strategy

updateModel(Symbol s)

- 1: index_cur ← index(s)
- 2: **for** pos_cur ← t[index_cur].pos, 1 **do**
- 3: pos_prev ← pos_cur-1;
- 4: index_prev ← t[pos_Prev].idx
- 5: t[index_prev].pos ← pos_Cur
- 6: t[pos_Cur].idx ← index_prev
- 7: **end for**
- 8: t[index_cur].pos ← 0;
- 9: t[0].idx ← index_cur;

3.4.2. *Dynamic sets*

Unlike the symbol tables solution, dynamic sets use just enough memory to store the occurred symbols. This approach is desirable for general cases, where the compression ratio requires setting a maximum order that is just not manageable with symbol tables.

We support dynamic sets by modifying the source code of PPMII, a state of the art PPM compressor designed with a very efficient memory management [26]. We stripped the arithmetic encoding out of the context model and added the part related to the key generation. A separate layer uses FSE to achieve compression. By leveraging from existing PPM code, we are able to properly compare the effect of changing from adaptive to static entropy encoding. We get back to this in the experimental section.

PPMII ignores symbols seen at the highest orders using a mask array. Also, it has a unique way to accumulate frequencies, based on empirical evaluations. Our implementation maintained these features, as they are deeply blended into the code.

However, there was some room for optimizations, especially during decompression. Since our primal concern is finding the position of the symbol to decode based on a key, only two orders need to be accessed: the one where the symbol exists (i) and the one above ($i+1$). In order $i+1$, the mask is updated with the found symbols. In order i, the mask is used only to check which symbols have already appeared. Order i is located as the one where the number of symbols is at least equal to the key.

Algorithm 5 shows how the symbol is located at order i. The key to decode is *key*. Observe that the mask is never updated (only checked). Unseen symbols cause k to be incremented.

Downloaded from https://academic.oup.com/comjnl/article-abstract/doi/10.1093/comjnl/bxaa074/5868170 by University of New England user on 11 July 2020

The search stops when k equals *key*, meaning the symbol to decode was found ($p->symbol$).

```
inline void LUISA_CONTEXT::decodeSymbol2
(int key, int *position, unsigned
char *DecodedArray){
  UINT Sym;
  STATE* p = Stats-1;
  k--;
  do {
    do {
      Sym=p[1].Symbol; p++;
    } while (Mask[Sym] == EscCount);
  } while (++k!=key);
  DecodedArray[*position] = p->Symbol;
  *position = *position + 1;
  update2(p);
}
```

4. EXPERIMENTATION AND RESULTS

This section reports the results achieved in terms of compression ratio and speed. The compression ratio is measured as bpc (bits per code), the average number of bits required to code each byte. Speed is measured as the number of KBytes processed per millisecond. We report the average speed from 30 executions (compression + decompression), ignoring the lowest and highest 10%. Algorithms were executed on an Intel Core I5-3330 (3.0 GHz, 4 cores, 32KB L1, 256KB L2, 6144KB L3) server with 8 GB of RAM (DDR3, synchronous, 1333 MHz) running Windows 7 (64 bits) OS.

The input files come from Calgary, Silesia, Canterbury and Pizza&Chilli corpora. The first three are collections commonly used as benchmarks of compression tools. The Pizza&Chilli corpus contains text collections for experimenting and validating compressed indexes. There is also an extra file containing an English dictionary. It comes from the expansion of.aff and.dic files with hunspell software, thus creating a lexicographically ordered dictionary with words in singular and plural.[2] For the sake of space, only some results are discussed, which we believe are representative enough.[3]

Two distinct implementations of LUISA were evaluated: the one that uses symbol tables (used in preliminary tests in Subsection 4.1) and the one based on PPMII (used in comparing tests in Subsections 4.2 and 4.3). Both were compiled with GCC/G++ compiler from TDM-GCC (version 5.1.0) using the -O3 optimization flag to obtain maximum efficiency. Also, keys are buffered in blocks of size 125 000 bytes, defined empirically. This information is valuable when using static entropy encoding. Larger blocks have almost no effect on compression.

4.1. Parameter tuning

LUISA can be tuned in different ways to achieve better compression or execution time. The tuning involves changing the model update strategy (frequency, transpose, MTF), the key generation strategy (with/without exclusion) and the entropy encoder, using static methods (FSE/Huff0) or a dynamic method (AE). In what follows, we test different combinations. Our baseline is a setting that balances speed and compression ratio, by combining the FSE encoding method, the frequency-based update model and the key generation with exclusion.

This experiment uses the implementation based on symbol tables. Since this version was built from scratch, we can easily alternate between the different strategies (which could not be done with the version derived from PPMII).

The maximum order was set to five. Longer contexts lead to compression deterioration due to memory issues: the OS imposes a limit on the amount of memory available to the program. When the limit is reached, we stop adding symbol tables to prevent overflows.

We remark that the symbol table version is intended to be used as a testbed. It serves as a reference by which the different strategies can be compared. Contexts longer than five are not needed to fairly compare the alternatives. Besides, as we discuss later, the maximum order of five usually leads to better results.

Table 3 shows information about the 10 files used in this experiment. The chosen files vary greatly in size, structure and format. Emphasis is given to textual files (Book1, Bible, Dickens, English, Dict) since LUISA works best when compressing textual information.

Table 4 shows the results when varying the key generation strategy. UE and WUE refer to update exclusion and without update exclusion, respectively. The other parameters were fixed by using baseline values (FSE encoding/frequency update model).

Keeping already visited symbols does not translate into an expressive speed-up. On the other hand, the compression ratio when excluding visited nodes shows a compensating trade-off, especially when files are small (Book1, Obj1). In those cases, the contexts are not yet stable, and there is a higher chance of finding the symbol to encode only in lower orders. The accumulation of counts from the higher orders increases the entropy and leads to a degraded bpc. The compression ratio is similar on larger files (such as English), where the symbol to encode is likely to be found at the highest order.

Table 5 shows the results when varying the encoding method. The other parameters were fixed by using baseline values (frequency update model/key generation with exclusion). Note that ‘speed’ refers to the number of KBytes processed per millisecond (KB/ms).

[2] The file can be found at https://github.com/ViniGarcia/LUISA.

[3] The complete results, files corpora, evaluated compressors, testing framework and configuration files can be found at https://github.com/ViniGarcia/LUISA.

Downloaded from https://academic.oup.com/comjnl/article-abstract/doi/10.1093/comjnl/bxaa074/5868170 by University of New England user on 11 July 2020

TABLE 3. A brief description of the files used to evaluate the compression methods.

File (corpus)	Size (bytes)	Description
Book1 (Calgary)	768 771	Far from the Madding Crowd, by Thomas Hardy
Obj1 (Calgary)	49 379	Compilation of Pascal code
Bible (Canterbury)	4 047 392	The King James version of the Bible
Ecoli (Canterbury)	4 638 690	Complete genome of the E. Coli bacterium
Kennedy (Canterbury)	1 029 744	Excel spreadsheet
Dickens (Silesia)	10 192 446	A Child's History of England, by Charles Dickens
XML (Silesia)	5 345 280	Collected XML files
ooffice (Silesia)	6 152 192	A dll from Open Office.org 1.01
English (Pizza&Chilli)	52 428 800	English texts from project Gutenberg
Dict	5 074 110	English dictionary expanded from.aff and.dic files of LibreOffice[6] with hunspell software

[6] https://github.co\ignorespacesm/LibreOffice/dictionaries

TABLE 4. Changing the key generation strategy.

File	UE		WUE	
	bpc	Speed	bpc	Speed
Dict	2.91	3653	3.11	3722
Book1	2.65	2125	2.86	2183
Obj1	4.23	470	4.47	487
Ooffice	5.24	4434	5.32	4880
English	3.56	14 099	3.58	14 539
XML	1.29	6843	1.34	7035
Ecoli	1.95	26 004	1.96	25 850
Kennedy	2.07	1905	2.61	1966
Bible	2.00	6374	2.09	6547
Dickens	2.32	4649	2.42	4806

TABLE 5. Changing the encoding strategy.

File	FSE		HUFF0		AE	
	bpc	Speed	bpc	Speed	bpc	Speed
Dict	2.91	3653	2.95	3673	2.79	1926
Book1	2.65	2125	2.68	2131	2.62	1213
Obj1	4.23	470	4.19	472	4.18	553
Ooffice	5.24	4434	5.25	4569	4.05	341
English	3.56	14 099	3.59	14 454	2.38	1122
XML	1.29	6843	1.61	6909	1.33	2466
Ecoli	1.95	26 004	2.01	26 970	1.96	3139
Kennedy	2.07	1905	2.34	1907	1.92	829
Bible	2.00	6374	2.11	6420	1.98	1886
Dickens	2.32	4649	2.38	4675	2.31	1354

TABLE 6. Changing the model update strategy

file	FREQ		TRANSP		MTF	
	bpc	Speed	bpc	Speed	bpc	Speed
Dict	2.91	3653	2.62	4152	2.08	4199
Book1	2.65	2125	2.75	2382	2.82	2353
Obj1	4.23	470	4.41	484	4.19	478
Ooffice	5.24	4434	5.11	5402	4.82	3412
English	3.56	14 099	3.59	16 341	3.90	13 626
XML	1.29	6843	1.34	7616	1.38	7531
Ecoli	1.95	26 004	1.99	29 480	1.99	27 266
Kennedy	2.07	1905	1.92	1987	0.84	1878
Bible	2.00	6374	2.01	7705	2.07	7344
Dickens	2.32	4649	2.41	5688	2.55	5272

Note that the usage of dynamic entropy encoding (AE) is usually associated with greater compression. For the larger file (English), the compression gain is expressive (∼20%). On the other hand, it is notable that this method leads to a significant slowdown. The speed reduction varies from file to file, ranging from 42% to 92%. This result demonstrates the importance of using non-adaptive entropy encoding to gain response time. HUFF0 and FSE behave similarly, where the former is faster and the latter compresses more. It called our attention the fact that FSE is much better at compressing the XML file. Another curious fact is that AE beats the competitors in terms of speed when compressing obj (where the three methods are slow).

Table 6 shows the results when varying the model update strategy among frequency swap (FREQ), transpose swap (TRANSP) and MTF swap (MTF). The other parameters were fixed by using baseline values (FSE/key generation with exclusion). It is important to notice that 'speed' refers to the number of KBytes processed per millisecond (KB/ms).

The results show an interesting trade-off between FREQ and TRANSP. When files are small (Book1, Obj1), the frequency information is more important to estimate the next symbol to encode. As files get larger, TRANSP gradually moves the most frequent symbol to the front, reducing the compression gap

Downloaded from https://academic.oup.com/comjnl/article-abstract/doi/10.1093/comjnl/bxaa074/5868170 by University of New England user on 11 July 2020

TABLE 7. Compression ratio on the Bible file.

Method	Orders			
	2	3	4	5
FSE-TRANSP-UE	2.78	2.30	2.10	2.01
FSE-FREQ-UE	2.82	2.35	2.11	2.00
HUFF0-FREQ-UE	2.84	2.39	2.19	2.11
FSE-FREQ-WUE	2.83	2.38	2.16	2.09
FSE-MTF-UE	2.95	2.42	2.19	2.07
AE-FREQ-UE	2.81	2.34	2.10	1.98

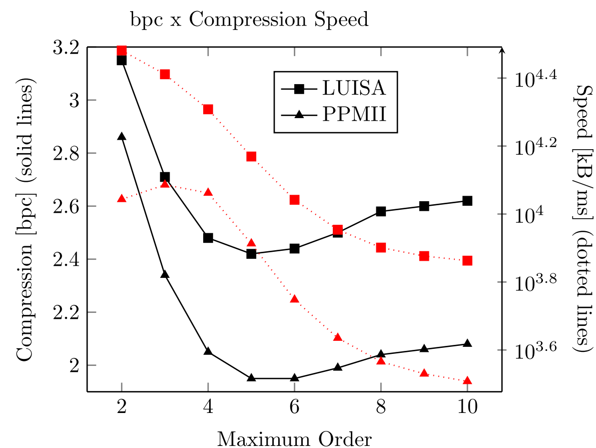

FIGURE 6. Compression of textual files from the Pizza&Chilli corpus.

to FREQ. In some particular cases, their compression ratio is practically identical, while TRANSP shows superior speed. For instance, the compression on the Bible and English files got a speedup of 20% and 15%, respectively.

Another interesting case is the Ecoli file. The frequency-based strategy reaches a compression ratio of 1.95, which is similar to the one obtained by PPMII (1.96 bpc, using the same five orders). Since the alphabet size is small (basically four letters representing nucleotides), and the distribution of the symbols is relatively uniform, both methods end up using similar frequency tables.

The results also show that the MTF strategy outperforms the competitors in some particular cases (Dict, Obj1, Ooffice and Kennedy). These are examples of files where recency is a prevailing factor, being more important than global frequencies. MTF is particularly suited for this kind of file. The reason is that a symbol that just occurred in a context will very likely occur again the next time the context is found, and MTF is able to quickly promote this symbol to the head of the list.

Surprisingly, Dict and Kennedy compete with PPMII (set with the same five orders). LUISA requires 2.08 bpc and 0.84 bpc to compress the dictionary and the spreadsheet (kennedy), respectively, while PPMII requires 2.06 bpc and 0.95 bpc to compress the same files. Moreover, when using a maximum of three orders, the dictionary is encoded with only 1.54 bpc. This improvement is related to the fact that the dictionary is mostly composed of small grams, like affixes and stems.

We conclude this section with Table 7. It presents the compression ratio on the Bible file, considering all alternatives mentioned above with a maximum order ranging from 2 to 5. Observe that all methods behave similarly as the number of orders changes. In this particular case, compression keeps improving until order 5. More importantly, the strategy based on symbols transpositions is practically identical to the one based on frequencies. As with the Tables 5 and 6, 'speed' refers to the number of KBytes processed per millisecond (KB/ms).

One can argue that better results can be achieved using other methods (for instance, Front-Coding for dictionary files). However, our primal concern here was not to overcome the strong competitors, but to show that our context model approach can achieve interesting results without relying on frequency information. By keeping the probability model outside the context model, we can easily conceive alternative strategies to handle files with a distinguished set of features. As demonstrated, a simple change was required in the model update module to handle recency. Other prediction-based methods may struggle to properly model recency, especially those that rely on the symbols' frequency.

4.2. Comparing LUISA With PPM

In this section, we compare the version of LUISA built from PPMII against the original PPMII. The two compressors use the same context model (taken from PPMII), but a completely different encoding. While PPMII encodes symbols based on local frequencies, LUISA uses FSE and a decoupled global frequency table to encode the keys that identify the symbols. The decoupling comes as an alternative to classical PPM implementations that require adaptive entropy encoding. We expect to achieve greater speed at the expense of a deteriorated compression ratio. In what follow, we investigate this trade-off.

Figure 6 shows the results of compression ratio (black lines) and compression speed (red lines) when using the textual files taken from the Pizza&Chilli corpus (four english files with sizes of 50MB, 100MB, 200MB and 1024MB). We compressed the corpus files individually, and the average between their results is presented as the final result. The maximum order ranges from 2 to 10. Both methods behave similarly according to the number of orders increasing. They achieve the greatest compression when using five orders, where PPMII is around 20% more effective. On the other hand, it is 45% slower.

The compressors speed degrades with higher orders, with one exception: PPMII shows superior performance when moving from two to three orders. In this particular case, the amount of work required for managing longer contexts is compensated

Downloaded from https://academic.oup.com/comjnl/article-abstract/doi/10.1093/comjnl/bxaa074/5868170 by University of New England user on 11 July 2020

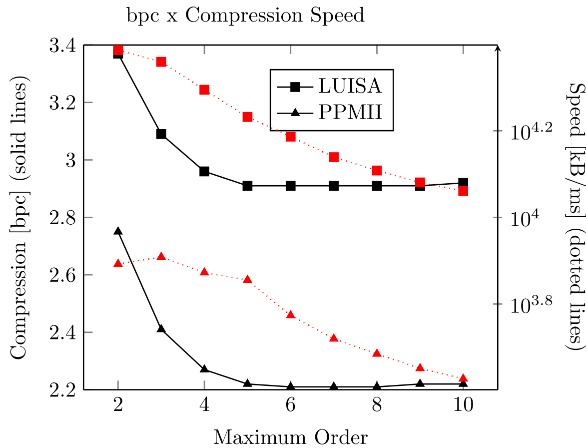

FIGURE 7. Compression of files from the Silesia corpus.

TABLE 8. Compressors settings for best compression ratio (-c) and highest speed (-s).

Compressor	Default setup (-d)	Compression setup (-c)	Speed setup (-s)
ZSTD	-3	-19	-1
7ZIP	-mx5	-mx9	-mx1
PPMII	-o4	-o7	-o2
LUISA	-o 4	-o 7	-o 2
BZIP2	-9	n/a	-1
BSC	-e1	-e2	-p
SR3	n/a	n/a	n/a

by a much more efficient encoding (the compression ratio increases ∼19% when working with three orders instead of two).

Figure 7 shows the results of compression ratio (black lines) and compression speed (red lines) when using all files from the Silesia corpus. We compressed the corpus files individually, and the average between their results is presented as the final result. This corpus is much more diverse than the previous one, including non-textual files. Again, both methods behave similarly. The greatest compression is achieved using five orders, where PPMII is ∼24% more efficient. On the other hand, it is 58% slower.

Observe that, in both charts, PPMII cannot compete with LUISA in terms of speed, even when using settings that reduce compression. In other words, the best performance of PPMII does not overcome LUISA in its worst performance.

4.3. Comparing against state-of-the-art methods

In this section, we compared LUISA (the one built from PPMII context model) against representatives from state-of-the-art loss-less compression methods. The chosen methods are modern and highly optimized C/C++ codes based on different compression concepts, such as block-sorting (BZIP2, BSC), dictionary look-up (7ZIP, ZSTD) and context modeling (PPMII, SR3).

Most methods allow defining compression levels to trade between compression ratio and execution time (BZIP2, BSC, 7ZIP, ZSTD). We tried three settings: the default one (identified with the suffix ‘d’ and configured with standard compression level setup), the one targeted at compression (identified with the suffix ‘c’ and configured with maximum compression level) and the one targeted at speed (identified with the suffix ‘s’ and configured with minimum compression level). For instance, ZSTD-c is the ZSTD compressor tuned to achieve the greatest compression (compression level -19). The standard setting of BZIP2 is already tuned to achieve the best compression, so BZIP2-c was discarded. PPMII uses 5 as the default maximum order. We also tried setting the maximum order to 2 and 7, to obtain minimum execution time and greater compression ratio, respectively. LUISA uses the same orders. SR3 does not offer compression levels. Table 8 summarizes the settings of default configuration (-d), best compression ratio (-c) and highest speed (-s).

Execution time results consider the average time (in seconds) to perform a round trip (compression + decompression). We observe that LUISA’s compression is symmetric. The execution time to search the symbol, update the model and apply the entropy encoding is roughly the same when these modules are executed in the reverse order. Optimizations in the source code (discussed in Section 3.4) made it around 15% faster in decompression. The other context model methods are also symmetrical. Block-sorting methods are faster in decompression, while the methods based on dictionary lookups are much faster in decompression. This occurs mainly because there is no need to perform lookups during decompression.

Figure 8 shows a scatter plot comparing round-trip execution time and compression ratio when compressing/decompressing a subset of the Calgary corpus composed by small textual files (Book1, Book2, Paper1,..., Paper6). Furthermore, Table 9 numerically specifies the compression ratio, the compression execution time and the decompression execution time of processing the Calgary corpus. The best results are in the extremes: PPMII-c offers the greatest compression, but it is the slowest alternative. Conversely, ZSTD-s is the fastest and has the poorest compression. It is important to state that LUISA-d is in the Pareto Frontier. Thus, none of the competitors is at the same time faster and more effective in compression/decompression.

Figure 9 shows the results achieved when compressing/decompressing a subset of the Pizza&Chilli corpus composed by large textual files (English documents with 50MB, 100MB, 200MB and 1024MB). Also, Table 10 numerically defines the compression ratio, the compression execution time and the decompression execution time of processing the Pizza&Chilli

Downloaded from https://academic.oup.com/comjnl/article-abstract/doi/10.1093/comjnl/bxaa074/5868170 by University of New England user on 11 July 2020

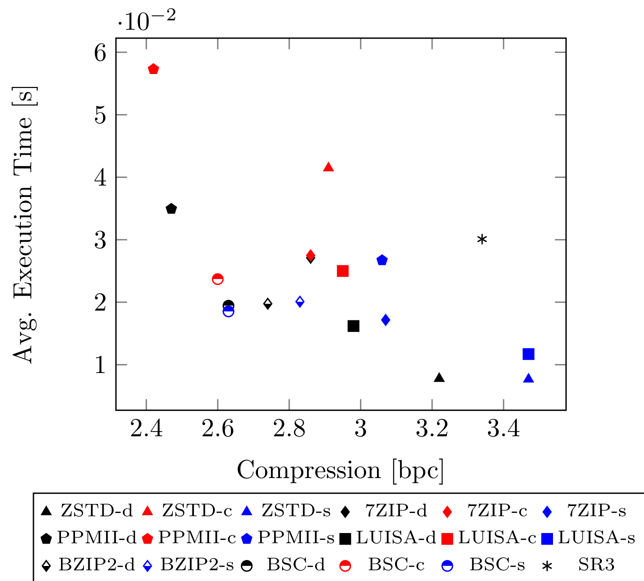

FIGURE 8. Avg. execution time/compression ratio results achieved when compressing/decompressing textual files from the Calgary corpus.

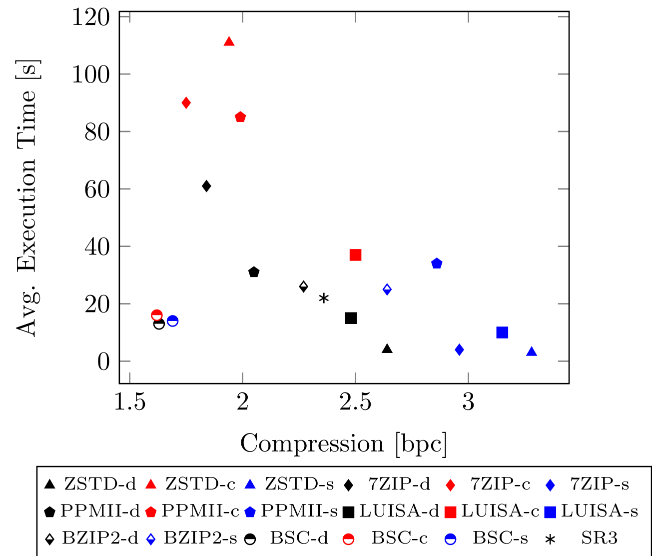

FIGURE 9. Results achieved when compressing/decompressing textual files from the Pizza&Chilli corpus.

TABLE 9. Execution time results achieved when compressing/decompressing textual files from the Calgary corpus.

Compressor	bpc	Compression exec. time (s)	Decompression exec. time (s)
ZSTD-d	3.22	0.0084	0.0071
ZSTD-c	2.91	0.0752	0.0077
ZSTD-s	3.47	0.0087	0.0065
7ZIP-d	2.86	0.0434	0.0108
7ZIP-c	2.86	0.0441	0.0108
7ZIP-s	3.07	0.0231	0.0112
PPMII-d	2.47	0.0330	0.0368
PPMII-c	2.42	0.0552	0.0593
PPMII-s	3.06	0.0253	0.0281
LUISA-d	2.98	0.0173	0.0151
LUISA-c	2.95	0.0268	0.0232
LUISA-s	3.47	0.0127	0.0107
BZIP2-d	2.74	0.0261	0.0134
BZIP2-s	2.83	0.0271	0.0131
BSC-d	2.63	0.0237	0.0152
BSC-c	2.60	0.0277	0.0197
BSC-s	2.63	0.0229	0.0141
SR3	3.34	0.0278	0.0323

TABLE 10. Execution time results achieved when compressing/decompressing textual files from the Pizza&Chilli corpus.

Compressor	bpc	Compression exec. time (s)	Decompression exec. time (s)
ZSTD-d	2.64	5.2196	1.8633
ZSTD-c	1.94	219.5249	1.6923
ZSTD-s	3.28	3.9007	1.5912
7ZIP-d	1.84	120.0036	1.5333
7ZIP-c	1.75	176.9292	2.0758
7ZIP-s	2.96	6.9308	1.6571
PPMII-d	2.05	29.7625	32.9671
PPMII-c	1.99	81.8390	88.1192
PPMII-s	2.86	31.5487	36.5537
LUISA-d	2.48	17.1037	14.5414
LUISA-c	2.50	38.3281	35.4933
LUISA-s	3.15	11.6247	8.8882
BZIP2-d	2.27	37.8516	14.6214
BZIP2-s	2.64	35.7727	13.5898
BSC-d	1.63	18.0478	8.7282
BSC-c	1.62	20.3927	11.6904
BSC-s	1.69	18.6960	8.6330
SR3	2.36	21.2303	23.6613

corpus. When compared to the previous case, some methods benefit from large textual files, such as BSC and SR3. LUISA competes with most of the best compressors, except for BSC. The block sorting compression method shows outstanding compression and it is among the fastest approaches. The symbol rank method (SR3) compresses more than LUISA, which did not happen when texts were smaller.

Figure 10 shows the results achieved when compressing/decompressing the Silesia corpus, composed by files of varying sizes and formats (not only text). Additionally, Table 11 numerically indicates the compression ratio, the compression execution time and the decompression execution time of processing

Downloaded from https://academic.oup.com/comjnl/article-abstract/doi/10.1093/comjnl/bxaa074/5868170 by University of New England user on 11 July 2020

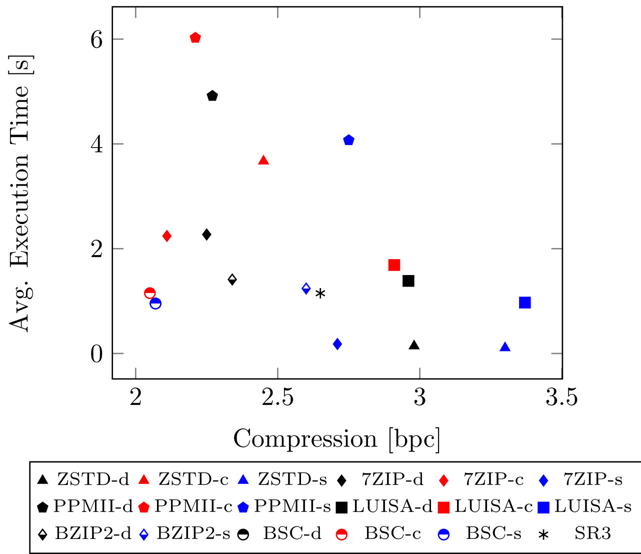

FIGURE 10. Results achieved when compressing/decompressing the Silesia corpus.

TABLE 11. Execution time results achieved when compressing/decompressing textual files from the Silesia corpus.

Compressor	bpc	Compression exec. time (s)	Decompression exec. time (s)
ZSTD-d	2.98	0.2069	0.0768
ZSTD-c	2.45	7.2652	0.0766
ZSTD-s	3.30	0.1421	0.0703
7ZIP-d	2.25	4.3267	0.2152
7ZIP-c	2.11	4.2711	0.2139
7ZIP-s	2.71	0.2758	0.0848
PPMII-d	2.27	4.6080	5.2237
PPMII-c	2.21	5.6863	6.3638
PPMII-s	2.75	3.7977	4.3435
LUISA-d	2.96	1.4089	1.1790
LUISA-c	2.91	1.8050	1.5718
LUISA-s	3.37	1.0793	0.8635
BZIP2-d	2.34	2.1986	0.6245
BZIP2-s	2.60	1.9059	0.5734
BSC-d	2.07	1.2396	0.6704
BSC-c	2.05	1.4076	0.8954
BSC-s	2.07	1.2382	0.6666
SR3	2.65	1.0908	1.2077

the Silesia corpus. Again, BSC shows unbeaten compression and competitive execution time. On the other hand, LUISA does not perform well. As a reference, it is defeated both in terms of compression and execution time by SR3 (whose results are reasonable). It demonstrates that the current setting (FSE+UE+FREQ) is not suited when files are non-textual.

Perhaps, it is possible to build more effective solutions by using different strategies, especially regarding the model update. For instance, one can incorporate the SR3 main ideas into the more general LUISA's architecture, like giving some sort of priority to the most recent symbols that appear within a context. The investigation of complementary techniques is left for future work.

5. CONCLUSIONS

We have presented LUISA, a PPM-based compression method that decouples the context model and the frequency model. This decoupling allows several strategies to be used, trading speed for compression ratio. The architecture is clear and concise, and the modules can easily be adjusted to particular compression situations.

Experiments show that the compression ratio is compelling and the compression/decompression speed is acceptable. Compared to a well-known prediction-based approach (PPMII), the speed gain is considerable, which makes LUISA a viable solution. The proposed method is not a replacement for some strong compressors that are undefeated at combining acceptable compression ratio and outstanding speed when compressing general files. However, when files are small or have a low entropy, the overall performance of LUISA is acceptable.

There is still much to be done to achieve greater compression and speed. Some paths we consider include building new model update strategies and executing compression on segmented parts of the file in parallel. Furthermore, in future works, we will study alternatives to create a version of LUISA optimized for text compression. Thus, we will evaluate multiple LUISA versions compared with semi-static and dynamic word-based compressors.

REFERENCES

[1] I. P. Andrezza. E C. Borges., and L. V. Batista. Heart arrhythmia classification using the prediction by partial matching algorithm. *Int. J. Comput. Appl. Technol.*, 52: 285–291, 2015.

[2] S. Beg, M. F. Khan, and F. Baig. Transference and retrieval of compress voice message over low signal strength in satellite communication. *Int. J. Syst. Syst. Eng.*, 4: 174–186, 2013.

[3] Jon L. Bentley and Catherine C. McGeoch. Amortized analyses of self-organizing sequential search heuristics. *Comm. ACM*, 28: 404–411, 1985.

[4] N. R. Brisaboa, A. Fariña, G. Navarro, and J. R. Parama. New adaptive compressors for natural language text. *Softw. Pract. Exp.*, 38: 1429–1450, 2008.

[5] J. Cleary and I. Witten. Data compression using adaptive coding and partial string matching. *IEEE Trans. Commun.*, 32: 396–402, 1984.

[6] Cleary, J.G., Teahan, W.J. and Witten, I.H. Unbounded Length Contexts for PPM. In *Conference on Data Compression*, p. 52. IEEE Computer Society, USA, Washington, DC, USA, 28–30 March 1995.

Downloaded from https://academic.oup.com/comjnl/article-abstract/doi/10.1093/comjnl/bxaa074/5868170 by University of New England user on 11 July 2020

[7] Cyan. *Quick look back at huff0: an entropy coder analysis*. http://fastcompression.blogspot.com/2011/01/quick-loock-back-at-huff0-entropy-coder.html, 2011 (accessed January 28, 2019).
[8] S. Deorowicz, J. Walczyszyn, and A. Debudaj-Grabysz. Comsa: compression of protein multiple sequence alignment files. *Bioinformatics*, 35: 227–234, 2018.
[9] Drinic, M., Kirovski, D. and Potkonjak, M. (2003) PPM Model Cleaning. In *Data Compression Conf.*, Snowbird, UT, USA, March 25–27 pp. 163–172. Institute of Electrical and Electronics Engineers, USA.
[10] Duda, J., Tahboub, K., Gadgil, N.J. and Delp, E.J. (2015) The Use of Asymmetric Numeral Systems as an Accurate Replacement for Huffman Coding. In *Picture Coding Symposium*, Cairns, QLD, Australia, May 31–June 3, pp. 65–69. Institute of Electrical and Electronics Engineers, USA.
[11] A. Farayez, M. B. I. Reaz, and N. Arsad. Spade: activity prediction in smart homes using prefix tree based context generation. *IEEE Access*, 7: 5492–5501, 2019.
[12] P. Fenwick. Symbol ranking text compressors: review and implementation. *Softw. Pract. Exp.*, 28: 547–559, 1998.
[13] P. Fenwick. Burrows–Wheeler compression: principles and reflections. *Theor. Comput. Sci.*, 387: 200–219, 2007.
[14] A. Gellert and A. Florea. Web prefetching through efficient prediction by partial matching. *World Wide Web*, 19: 921–932, 2016.
[15] P. G. Howard and J. S. Vitter. Design and analysis of fast text compression based on quasi-arithmetic coding. *Inform. Process. Manag.*, 30: 777–790, 1994.
[16] Z. Huang, Z. Wen, Q. Deng, Y. Chu, Y. Sun, and Z. Zhu. Lw-fqzip 2: a parallelized reference-based compression of fastq files. *BMC Bioinform.*, 18: 179, 2017.
[17] Yunwei Jia, En-hui Yang, Da-ke He, and Steven Chan. A greedy renormalization method for arithmetic coding. *IEEE Trans. Commun.*, 55: 1494–1503, 2007.
[18] Y. Liu, H. Peng, L. Wong, and J. Li. High-speed and high-ratio referential genome compression. *Bioinformatics*, 33: 3364–3372, 2017.
[19] M. Mahoney. *Data compression programs*. http://mattmahoney.net/dc/, 2019. (accessed November 22, 2019).
[20] McNamee, P. (2016) Language and Dialect Discrimination Using Compression-Inspired Language Models. In *Workshop on NLP for Similar Languages, Varieties and Dialects*, Osaka, Japan, December 11–12, pp. 195–203. The COLING 2016 Organizing Committee, Japan.
[21] A. Moffat. Implementing the PPM data compression scheme. *IEEE Trans. Commun.*, 38: 1917–1921, 1990.
[22] A. Moffat, R.M. Neal, and I. H. Witten. Arithmetic coding revisited. *ACM Trans. Inf. Syst.*, 16: 256–294, 1998.
[23] Pratas, D., Hosseini, M. and Pinho, A.J. (2019) Compression of Amino Acid Sequences. In *Int. Conf. Practical Applications of Computational Biology and Bioinformatics*, Toledo, Spain, June, pp. 105–113. Springer International Publishing, USA.
[24] S. Saha and S. Rajasekaran. NRGC: a novel referential genome compression algorithm. *Bioinformatics*, 32: 3405–3412, 2016.
[25] A. Sayyed and S. Agarwal. (2017) PPM Revisited With New Idea on Escape Probability Estimation. In IEEE *Int. Conf. Computational Intelligence and Multimedia Applications*, Sivakasi, Tamil Nadu, India, December 13–15 2007, pp. 152–156. Institute of Electrical and Electronics Engineers, USA.
[26] Shkarin, D. (2002) PPM: One Step to Practicality. In *Data Compression Conference*, Snowbird, UT, USA, April 2–4, pp. 202–211. Institute of Electrical and Electronics Engineers, USA.
[27] Skibinski, P. and Grabowski, S. (2004) Variable-Length Contexts for PPM. In *Data Compression Conference*, Snowbird, UT, USA, March 23–25, pp. 409–418. Institute of Electrical and Electronics Engineers, USA.
[28] Skibiński, P. and Swacha, J. (2007) Fast and Efficient Log File Compression. In *CEUR Workshop Proc. 11th East-European Conf. Advances in Databases and Information Systems (ADBIS)*, Varna, Bulgaria, September 29–October 3, pp. 330–342. Association for Computing Machinery, USA.
[29] Skibiński, P., Swacha, J. and Grabowski, S. (2008) A Highly Efficient XML Compression Scheme for the Web. In *Int. Conf. Current Trends in Theory and Practice of Computer Science*, Nový Smokovec, Slovakia, January 19–25, pp. 766–777. Springer International Publishing, USA.
[30] J. G. Wolff. Information compression as a unifying principle in human learning, perception, and cognition. *Complexity*, 2019: 1–38, 2019.
[31] P. Wu and W. J. Teahan. A new PPM variant for Chinese text compression. *Nat. Lang. Eng.*, 14: 417–430, 2008.
[32] Y. Zhang and D. A Adjeroh. Prediction by partial approximate matching for lossless image compression. *IEEE Trans. Image Process.*, 17: 924–935, 2008.

Downloaded from https://academic.oup.com/comjnl/article-abstract/doi/10.1093/comjnl/bxaa074/5868170 by University of New England user on 11 July 2020