Improving PPM Algorithm Using Dictionaries

Yichuan Hu
Department of Electrical and Systems Engineering
University of Pennsylvania
Email: yichuan@seas.upenn.edu

Jianzhong (Charlie) Zhang, Farooq Khan and Ying Li
Standards Research Laboratory
Samsung Telecommunications America
Email: {jzhang,fkhan,yli2@sta.samsung.com}

Abstract

We propose a method to improve traditional character-based PPM text compression algorithms. Consider a text file as a sequence of alternating words and non-words, the basic idea of our algorithm is to encode non-words and prefixes of words using character-based context models and encode suffixes of words using dictionary models. By using dictionary models, the algorithm can encode multiple characters as a whole, and thus enhance the compression efficiency. The advantages of the proposed algorithm are: 1) it does not require any text preprocessing; 2) it does not need any explicit codeword to identify switch between context and dictionary models; 3) it can be applied to any character-based PPM algorithms without incurring much additional computational cost. Test results show that significant improvements can be obtained over character-based PPM, especially in low order cases. [1]

Keywords

Text compression; Markov model; PPM; Dictionary model.

I. INTRODUCTION

Prediction by partial matching (PPM) [2] has set a benchmark for text compression algorithms due to its high compression efficiency. In PPM, texts are modeled as Markov processes in which the occurrence of a character only depends on its context, i.e., n preceding characters, where n is called the context order and is a parameter of PPM. In each context, probabilities of next character is maintained. When a new character comes, its probability is estimated using context models and is encoded by an arithmetic coder. During the encoding/decoding process, the context models (probability tables) are updated after a character is encoded/decoded. By using such adaptive context models, PPM is able to predict text as well as human do [3], and achieves higher compression ratio than other compression algorithms [4].

However, one limitation of PPM is that the prediction is character-based, i.e., characters are encoded one by one sequentially, which is not quite efficient. In [5], word-based PPM is proposed in which every word is predicted by its preceding words, but its performance is even worse than character-based PPM [4], mainly because the alphabet size of English words is so large that very long texts are required to collect sufficient statistical information for word-based models. Similarly, Horspool [6] introduced an algorithm that alternatively switches between word-based and character-based PPM, but it needs to explicitly encode the length of characters when character-based PPM is used, resulting in unnecessary overhead. Recently, Skibiński [7] extended the alphabet of PPM to long repeated strings by pre-processing the whole texts before encoding, which showed performance improvement over traditional PPM.

In this paper, we propose an enhanced algorithm that combines traditional character-based context models and dictionary models. The basic idea is that, for most English words, given the first a few characters (prefix) the rest of the word (suffix) can be well predicted. Specifically, in addition to context models for character prediction used in traditional PPM [2] we introduce dictionary models that contain words with a common prefix. By doing so, a word can be predicted and encoded given its prefix, i.e., the first a few characters. Therefore, different from traditional PPM [2] and its variations [8], [9], [10], [11], proposed algorithm can achieve variable-length prediction by partial matching (VLPPM). The remainder of the paper is organized as follows. Section II describes the dictionary model used for variable-length prediction and the framework of proposed scheme. Section III details the encoding and decoding algorithms. Test results and conclusions are presented in Section IV and V, respectively.

II. VARIABLE-LENGTH PREDICTION

A. Dictionary Model For Variable-Length Prediction

First of all, a simple example is given to show how variable-length prediction can be achieved by a dictionary model. Suppose we are encoding a sequence of texts like this: "...information...". The first 3 characters "inf" have been encoded

---

Char to be encoded	Context
o	inf
r	nfo
m	for
a	orm
t	rma
i	mat
o	ati
n	tio

encode one by one

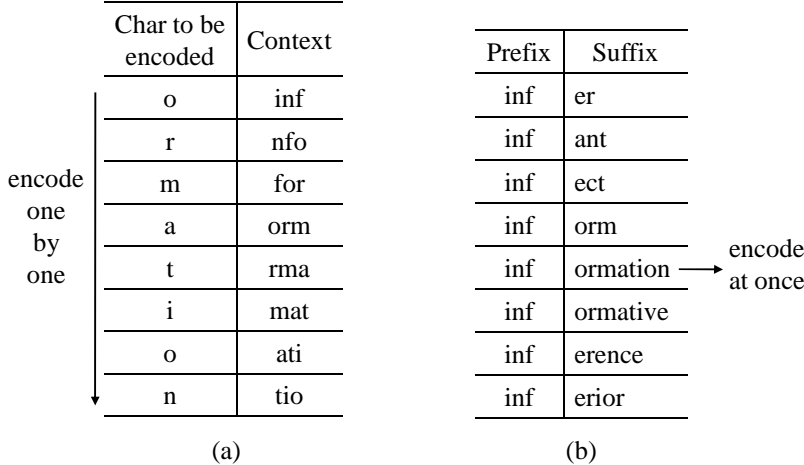| Prefix	Suffix
inf	er
inf	ant
inf	ect
inf	orm
inf	ormation	→ encode at once
inf	ormative
inf	erence
inf	erior

(a)   (b)

Figure 1: Comparison of encoding the word "information" given that the first 3 characters "inf" have been encoded. (a) order-3 PPM encoder which processes characters one by one. (b) proposed variable-length prediction using a dictionary model that estimates rest characters of the word at once.

and characters to be encoded next are "ormation...". If we use order-3 PPM, characters are encoded one after another given its context, i.e. 3 preceding characters, as shown in Fig. 1 (a). On the other hand, if we divide every word into two parts: a fixed-length prefix and a variable-length suffix, and suppose we have a dictionary that contains words with prefix "inf" as shown in Fig. 1 (b). Instead of predicting characters one by one, we can predict suffixes in this dictionary at one time, provided that we know the prefix is "inf". Since the dictionary contains suffixes with different length, the prediction is variable-length.

Next, let's take a look at the advantage of using dictionary model. We still focus on the above example, and estimate how many bits are required to encode "ormation". For traditional order-3 PPM, assume 1.5 bits are needed on average for encoding one character, then we need $1.5 \times 8 = 12$ bits to encode "ormation". Notice that 1.5 bpc is not an unreasonable assumption because order-3 PPM usually achieves more than 2 bpc for text compression [4], [8]. On the other hand, as we can see from Fig. 1 (b), there are 8 possible words that start with "inf" in the dictionary and "information" is one of them. Therefore, if we assign different indexes to these 8 words, we can easily encode "ormation" with as few as $\log_2 8 = 3$ bits, achieving 9 bits saving over order-3 PPM.

In practice, the dictionary model contains not only words but also their counts so that an arithmetic coder can be used to encode any word in the dictionary. Specifically, every dictionary D includes three parts: a common prefix P, a list of strings W_i that are suffixes of words with prefix P, and corresponding counts C_i. Given a prefix, we can find the associated dictionary model and then perform encoding/decoding. Moreover, different from context models in character-based PPM in which contexts from order-n to order-1 are used, we only maintain dictionary models with fixed-length prefix (fixed context order). If the prefix is too short, each dictionary will contain a lot of words which is not good for efficient compression. On the other hand, if the prefix is too long, number of characters that can be predicted by dictionary model will be very small. In order to take full advantage of dictionary model, we choose the length of prefix as 3.

B. Combining Dictionary Model and Context Model

As we can see, the dictionary model works on word basis. By word we mean a sequence of consecutive English letters. We can always parse a text file into a sequence of alternating words and non-words. For non-words, they cannot be encoded by dictionary model, and even for the prefix of a word we need to encode character by character. Therefore, dictionary model is combined with context model used in traditional character-based PPM. At the beginning of encoding/decoding, both context model and dictionary model are empty, and texts are encoded on character basis. After a word is encoded, corresponding dictionary model is updated. Detailed encoding and decoding algorithms will be introduced in Section III.

Compared with the methods in [6] and [7], the above framework has two advantages. First, no extra bits are required to indicate the switch from context model to dictionary model, because every time after the prefix of a word (3 consecutive English letters) is encoded or decoded by context model, encoder or decoder will automatically switch to dictionary model. Moreover, since dictionary models are constructed and updated during encoding/decoding, no pre-processing is required to build initial dictionaries.

III. Algorithm Details

A. Model Switching Using Finite State Machine (FSM)

Any compression algorithms using more than one model face the problem of model switching [12]. For example, in traditional PPM in which up to $n+1$ context models (order-n to order-0) might be used when encoding a character, $escape$ code is sent as a signal to let decoder know switch from current order to lower order. In our case, we need a mechanism to control the switch between dictionary model and context model. We use finite state machine (FSM), and for both encoder and decoder there are 3 states: S_0, S_1 and S_2. The transition rules between states for encoding and decoding are different, which will be described next.

B. Encoding Algorithm

The encoding process starts at S_0, and the state transition rules are as follows:

- At S_0: Encode the next character using context model. If it is an English letter, assign it to an empty string P and move to S_1; otherwise stay at S_0.
- At S_1: Encode the next character using context model. If it is an English letter, append it to string P; otherwise, go back to S_0. If the length of P reaches 3, move to S_2.
- At S_2: Read consecutive English letters, i.e. suffix string of current word, denoted as W. If a dictionary D associated with prefix P if found and W exists in D, encode W using dictionary model D. Otherwise, encode characters in W one by one using context models (an $escape$ code should be encoded using dictionary model D if W cannot be found in D). Move to S_0.

The encoding state transition diagram is depicted in Fig. 2. Pseudo code of the algorithm is provided below. On line 6 and 10, $English(c)$ is a function that checks character c is an English letter or not. Due to limited space, "**break**" at the end of each "**case**" clause and the "**default**" clause are omitted.

Algorithm 1: VLPPM-Encoder

---
1 $state \leftarrow S_0$;
2 **while** $c \neq EOF$ **do**
3   encode next character c using context model;
4   **switch** $state$ **do**
5   **case** S_0
6   **if** $English(c)$ **then**
7                  $P \leftarrow c$;
8                  $state \leftarrow S_1$;
9   **case** S_1
10   **if** $English(c)$ **then** $P \leftarrow P + c$;
11   **else** $state \leftarrow S_0$;
12   **if** $length(P) = 3$ **then** $state \leftarrow S_2$;
13   **case** S_2
14   read suffix W;
15   **if** $find$ $dictionary$ D $with$ $prefix$ P **then**
16   **if** $find$ W in D **then** encode W using D;
17   **else**
18   encode $escape$ code using D;
19   encode W using context model;
20   **else** encode W using context model;
21   update dictionary model;
22              $state \leftarrow S_0$;

At state S_2, the probability of $escape$ code is calculated by

$$P_{escape} = \frac{1}{1 + \sum_{W_i \in D} C_i}, \tag{1}$$

and the probability for suffix W_i is

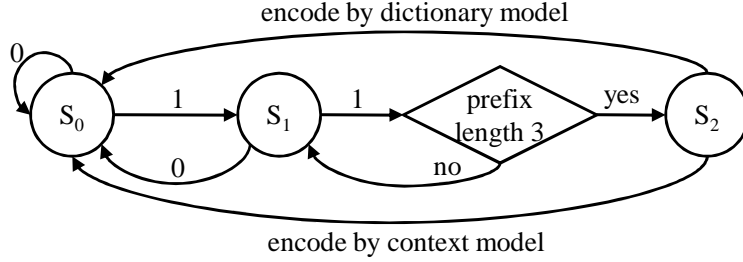$$P_{W_i} = \frac{C_i}{1 + \sum_{W_j \in D} C_j}. \tag{2}$$

Figure 2: Encoding algorithm using FSM. "1" and "0" denote the condition that next character to be encoded is an English or a non-English letter, respectively.

C. Decoding Algorithm

Similar to the encoding algorithm, the decoding algorithm also uses FSM with 3 states and starts at S_0, but with different state transition rules:

- At S_0: Decode the next character using context model. If it is an English letter, assign it to an empty string P and move to S_1; otherwise stay at S_0.
- At S_1: Decode the next character using context model. If it is an English letter, append it to string P; otherwise, go back to S_0. If the length of P reaches 3, decode using dictionary model. If a string of characters are decoded, move to S_0; if an *escape* code is decoded, move to S_2.
- At S_2: Decode characters one by one using context model until a non-English letter is decoded. Move to S_0.

Algorithm 2: VLPPM-Decoder

---

1 $state \leftarrow S_0$;
2 **while** $c \neq EOF$ **do**
3   decode c using context model;
4   **switch** $state$ **do**
5   **case** S_0
6   **if** *English(c)* **then**
7                 $P \leftarrow c$;
8                 $state \leftarrow S_1$;
9   **case** S_1
10   **if** *English(c)* **then** $P \leftarrow P + c$;
11   **else** $state \leftarrow S_0$;
12   **if** *length(P) = 3* **then**
13   **if** *find dictionary D with prefix P* **then**
14   decode W using D;
15   **if** *W is escape code* **then** $state \leftarrow S_2$;
16   **else** update dictionary model;
17   **else** $state \leftarrow S_2$;
18   **case** S_2
19   decode c using context model;
20   **if** *!English(c)* **then** $state \leftarrow S_0$;

D. Exclusion

As we can see from the decoding algorithm, decoder automatically switches from context model to dictionary model once 3 consecutive English letters are decoded. By doing so, we don't need to waste any bits to indicate switches between dictionary model and context model. However, this leads to another problem: if a prefix in the dictionary is a word of length 3 (e.g., "let" can be either a word or a prefix of "lettuce"), then every time this word occurs extra bits will be used by the dictionary model to encode an *escape* code, resulting in performance degradation. To resolve this problem, we introduce an exclusion mechanism: after a word is encoded/decoded, if it is a prefix of a dictionary, this dictionary is discarded and this prefix is put in a "blacklist" for future reference. During encoding/decoding, if a prefix is in "blacklist", encoder/decoder skip dictionary model and use context model directly.
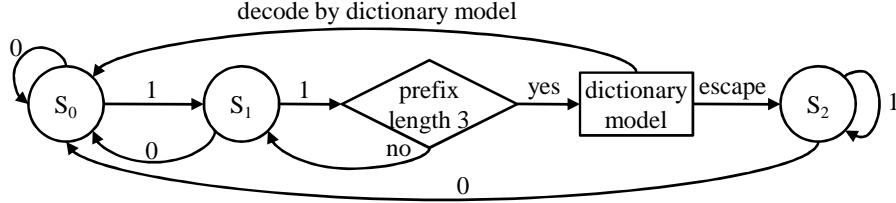
Figure 3: Decoding algorithm using FSM. "1" and "0" denote the condition that the next decoded character is an English or a non-English letter, respectively.

Table I: Compression Ratio Comparison of PPM and VLPPM

File	order-2			order-3		
	PPM	VLPPM	Gain	PPM	VLPPM	Gain
bib	2.66	2.25	18.2%	2.12	1.99	6.5%
book1	2.92	2.57	13.6%	2.48	2.36	5.1%
book2	2.89	2.34	23.5%	2.27	2.07	9.7%
news	3.26	2.87	13.6%	2.65	2.52	5.2%
paper1	2.94	2.60	13.1%	2.50	2.40	4.2%
paper2	2.89	2.52	14.7%	2.47	2.36	4.7%
progc	2.91	2.71	7.4%	2.52	2.47	2.0%
progl	2.40	2.13	12.7%	1.92	1.86	3.2%
progp	2.29	2.05	11.7%	1.86	1.81	2.8%
trans	2.38	2.08	14.4%	1.78	1.69	5.3%
alice29.txt	2.72	2.41	12.9%	2.31	2.23	3.6%
asyoulik.txt	2.81	2.58	8.9%	2.53	2.47	2.4%
lcet10.txt	2.76	2.15	28.4%	2.19	1.97	11.2%
plrabn12.txt	2.83	2.51	12.7%	2.44	2.33	4.7%
cp.html	2.73	2.55	7.1%	2.38	2.34	1.7%
fields.c	2.44	2.33	4.7%	2.18	2.15	1.4%
grammar.lsp	2.72	2.60	4.6%	2.51	2.47	1.6%
xargs.1	3.31	3.17	4.4%	3.08	3.05	1.0%
Average	2.86	2.46	16.3%	2.36	2.22	6.3%

IV. PERFORMANCE EVALUATION

A. Compression Efficiency

In order to show the compression efficiency of the proposed algorithm, VLPPM encoder/decoder are implemented and tested on a large set of texts. Specifically, we implemented PPMC encoder/decoder as described in [8], and further developed VLPPM based on PPMC. Text files from two popular data compression corpora, Calgary corpus [4] and Canterbury corpus [13], are chosen as the data to be compressed. Compression ratio of VLPPM is presented in Table I in terms of bits per character (bpc), and is compared with traditional PPM (PPMC). As we can see, using proposed VLPPM algorithm leads to considerable performance improvements: 16.3% and 6.3% gains are achieved over traditional PPM for order-2 and order-3, respectively. Moreover, although VLPPM implemented here is based on PPMC, it is applicable to any other character-based predictive compression schemes, such as all the variations of PPM [8], [9], [10], [11].

In traditional PPM, characters are encoded one by one sequentially. As a result, the number of bits required to encode a word increases almost linearly with the word length. However, from the information theoretic point of view this is not the case because longer word doesn't necessarily contain more information. VLPPM, on the other hand, predicts several characters at once, achieving higher compression efficiency. This is illustrated in Fig. 4, in which the average number of bits required to encode words with different length is depicted for both PPM and VLPPM. As we can see, the number of bits increases almost linearly with word length when PPM is used, in accordance with our analysis above. Compared with PPM, VLPPM requires similar number of bits when word length is short, but much fewer bits when word length is long, and the longer the word is, the more bits can be saved.

B. Computational Complexity

Since VLPPM is based on PPM, it is natural to compare its complexity with that of PPM. Table II compares the time and memory consumption of VLPPM and PPM encoders for order-2 and order-3. In both cases, the results of VLPPM are presented in terms of fraction of time and memory compared with PPM. As we can see, the speed of VLPPM is comparable to PPM, and it is even faster than PPM at order-2, due to the variable-length prediction ability. Furthermore,
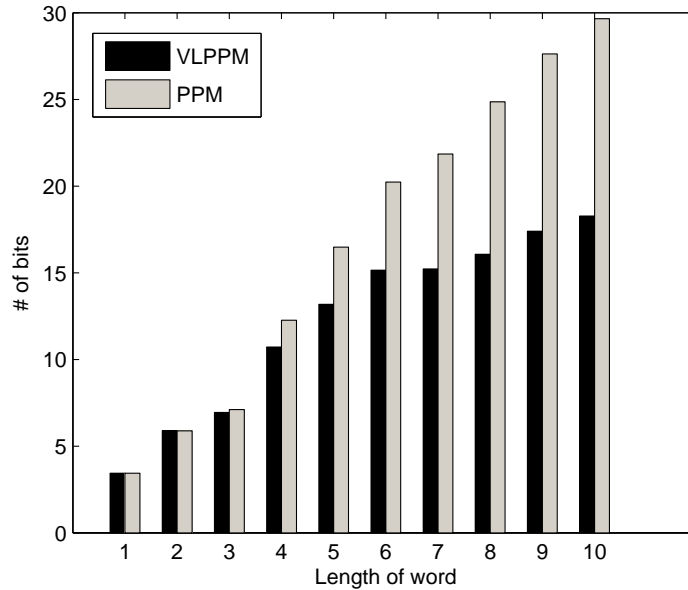
Figure 4: Average number of bits per word when `lcet10.txt` is compressed by order-2 PPM and VLPPM.

Table II: Computational Complexity of PPM and VLPPM

	order-2		order-3	
	PPM	VLPPM	PPM	VLPPM
Time	100%	98%	100%	108%
Memory	100%	113%	100%	108%

although VLPPM uses dictionary models in addition to context models, the memory consumption only increases by a small percentage: 13% and 8% for order-2 and order-3, respectively. This is because VLPPM only maintains dictionaries with fixed-length prefix, i.e. prefix with length 3, which means only those words longer than 3 will be stored. Moreover, using the exclusion mechanism introduced in the Section III-D excludes certain words from being stored into dictionary which further reduces memory use.

V. CONCLUSION

We have presented a text compression algorithm using variable-length prediction by partial matching (VLPPM). By introducing dictionary model which contains words with common prefix and combing it with context model used in traditional character-based PPM, the proposed method can predict one or more characters at once, further improving the compression efficiency without increasing computational complexity a lot. Moreover, the proposed method does not require any text preprocessing and can be applied to any other character-based predictive compression algorithms without increasing much computational complexity.

REFERENCES

[1] Y. Hu, J. Zhang, K. Farooq and Y. Li, "Improving PPM algorithm using dictionaries," *Proc. DCC'11*, March 2011.

[2] J. G. Cleary and I. H. Witten, "Data compression using adaptive coding and partial string matching," *IEEE Trans. Communications*, vol. COM-32, no. 4, April 1984.

[3] W. J. Teahan and J. G. Cleary, "The entropy of English using PPM based models," *Proc. DCC'96*, March 1996.

[4] T. Bell, I. H. Witten, and J. G. Cleary, "Modeling for text compression," *ACM Computing Surveys*, vol. 21, no. 4, pp. 557-591, December 1989.

[5] A. Moffat, "Word based text compression," *Research Report*, Dept. of Computer Science, Univ. of Melbourne, 1987.

[6] R. N. Horspool and G. V. Cormack, "Constructing word-based text compression algorithms," *Proc. DCC'92*, March 1992.

[7] P. Skibiński, "PPM with the extended alphabet", *Information Sciences,* vol. 176, no. 7, pp. 861-874, April 2006.

[8] A. Moffat, "Implementing the PPM data compression scheme," *IEEE Trans. Communications*, vol. 38, no. 11, pp. 1917-1921, November 1990.

[9] J. G. Cleary and W. J. Teahan, "Unbounded length contexts for PPM," *The Computer Journal*, vol. 40, no. 2/3, pp. 67-75, 1997.

[10] C. Bloom, "PPMZ: high compression markov predictive coder," *http://www.cbloom.com/src/ppmz.html*, 1999.

[11] D. Shkarin, "PPM: one step to practicality," *Proc. DCC'02*, March 2002.

[12] P. A. J. Volf and F. M. J. Willems, "Switching between two universal source coding algorithms," *Proc. DCC'98*, March 1998.

[13] R. Arnold and T. Bell, "A corpus for the evaluation of lossless compression algorithms", *Proc. DCC'97,* March 1997.