Theoretical Computer Science 127 (1994) 181–186
Elsevier

Note

P-complete problems in data compression

Sergio De Agostino
Computer Science Department, Brandeis University, Waltham, MA 02254, USA

Communicated by F.P. Preparata
Received July 1990
Revised April 1993

Abstract

De Agostino, S., P-complete problems in data compression, Theoretical Computer Science 127 (1994) 181–186.

In this paper we study the parallel computational complexity of some methods for compressing data via textual substitution. We show that the Ziv–Lempel algorithm and two standard variations are P-complete. Hence an efficient parallelization of these algorithms is not possible unless P = NC.

1. Introduction

The purpose of data compression is to develop methods for representing information in the minimum amount of space. The two most common applications of data compression are, therefore, data storage and data communication, since compressing data allows more data to be placed in some device and speeds up transmission. In this paper we deal with *lossless text* data compression, where the decompressed data must be identical to the original. By text we mean data is in the form of a sequence of characters drawn from an input alphabet. Application of lossless text compression includes the compression of spoken or written language text, numerical data, data base information, etc., where even the loss of a single bit may not be acceptable.

Correspondence to: S. De Agostino, Computer Science Department, Brandeis University, Waltham, MA 02254, USA. Email: sergio@cs.brandeis.edu.

SSDI 0304-3975(93)E0142-Q

Textual substitution methods (often called "LZ" methods due to the work of Lempel and Ziv [7]) are among the most practical and effective for lossless text compression. *Textual substitution* replaces substrings in the text with *pointers* to copies that are stored in a *dictionary*. The encoded string will be a sequence of pointers and uncompressed characters. The *static* method is when the dictionary is known in advance [5, 11]. By contrast, with the *dynamic* method (often called "LZ2" method due to the work of Ziv and Lempel [13]) the dictionary may be constantly changing as the data is processed. A special way to change dynamically is the *sliding* dictionary method (often called "LZ1" method due to the work of Lempel and Ziv [8]), where the dictionary is a window that passes continuously from left to right over the input.

In this paper we consider the LZ2 method. The original Ziv–Lempel algorithm, that we call *LZ2 algorithm*, achieves the maximum compression obtainable with a finite state encoder when the length of the input string goes to infinity. Two standard variations of this algorithm that are more effective in the practical cases are the *next-character heuristic* [9, 12] and the *first-character heuristic* [10]. While efficient parallel algorithms (polylogarithmic time and polynomial number of processors) have been designed for compression with static and sliding dictionaries [2, 3], we show that the LZ2 algorithm, the next-character heuristic and the first-character heuristic are hardly parallelizable.

We need to introduce the notions of P-*complete problem* and *P-complete* algorithm that we adopt as defined in [4, 1]. Let P denote the set of problems solvable in polynomial sequential time and NC denote the set of problems solvable in polylogarithmic parallel time using a polynomial number of processors. A problem $L \in \mathrm{P}$ is said to be P-complete if every other problem in P can be transformed to L with an NC-*reduction* (a reduction that belongs to NC). It follows that a P-complete problem does not belong to NC unless NC = P. An algorithm T is said to be P-complete if the problem $L_T = \{(x, i, b)$: *the ith bit of* $T(x)$ *is* $b\}$ is P-complete. We prove that the LZ2 algorithm, the next-character heuristic and the first-character heuristic are P-complete. Since we strongly believe that NC and P are different, an efficient parallelization of such algorithms is unlikely.

2. The compression algorithms

The LZ2 algorithm learns substrings by reading the input string from left to right with a so-called *incremental parsing* procedure. The dictionary is initially empty. This procedure adds a new substring to the dictionary as soon as a prefix of the still unparsed part of the string does not match a dictionary element. So, the last character of the new substring is left uncompressed while the prefix is replaced with a pointer to the dictionary (see example below). The uncompressed characters left by the LZ2 algorithm guarantee progress of the reading of the string and do not cost anything in terms of asymptotical performance since the pointer size goes to infinity. In practice, we do not want to leave characters uncompressed. This can be avoided by initializing the dictionary with the alphabet characters. The next-character heuristic also parses the string from left to right with a greedy procedure. It finds the longest match in the current position and updates the dictionary by adding the concatenation of the match with the next character. The first-character heuristic differs in the way it updates the dictionary. The new element is defined as the concatenation of the last match with the first character of the current match.

Example. *abababaaaa*

LZ2 algorithm
parsing: a, b, ab, aba, aa, a;

dictionary: a, b, ab, aba, aa;

coding: $0a$, $0b$, $1b$, $3a$, $1a$, $0a$.

next-character heuristic
parsing: a, b, ab, aba, a, aa;

dictionary: a, b, ab, ba, aba, $abaa$, aa;

coding: 1, 2, 3, 5, 1, 7.

first-character heuristic
parsing: a, b, ab, ab, a, a, aa;

dictionary: a, b, ab, ba, aba, aa;

coding: 1, 2, 3, 3, 1, 1, 6.

3. The P-completeness of the LZ2 algorithm

In this section we show that the LZ2 algorithm is P-complete. The P-completeness proof will be a reduction from the circuit value problem [6]. The circuit value problem is the following: Given a circuit and values for its inputs, what is the value of its output? Formally, a circuit C_n is a string $c_1 \ldots c_n$ where c_i is either an input gate with value 0 or 1, or a boolean gate. The gates are numbered topologically so if c_k receives an input from c_i, then $i<k$. We shall put the following restriction on the circuit: c_i is either an INPUT gate or a NOT gate or an OR gate for $1 \leqslant i \leqslant n-2$, c_{n-1} is a NOT gate and c_n is an OR gate. Obviously, the circuit value problem remains P-complete under these restrictions. Let us denote by $i(j)$ the number of gates c_h, with $h \leqslant i$, having the output of c_j as input and by x^k the concatenation of k symbols equal to x (x^0 is the empty string). We prove the following theorem.

Theorem 3.1. *The LZ2 algorithm is* P*-complete.*

Proof. We reduce the circuit to a binary string. A certain pointer will be in the LZ2 coding of this string iff the output of the circuit is 1. Since the string is binary, the problem is P-complete for any fixed cardinality of the alphabet. A string X_i is associated with each gate c_i with the following rules:

- c_i INPUT gate with value 1: $X_i = b^{2i}a$,
- c_i INPUT gate with value 0: $X_i = ab^{2i}a$,
- c_i NOT gate having the output of c_j as input: $X_i = b^{2j}aa^{i(j)}b^{2i}a$,
- c_i OR gate having the outputs of c_j and c_k as inputs:
  (i) c_j and c_k both either OR gate or INPUT gate:

$$X_i = b^{2j}aa^{i(j)}b^{2i}ab^{2k}aa^{i(k)}b^{2i}a,$$

  (ii) c_j either OR gate or INPUT gate and c_k NOT gate:

$$X_i = b^{2j}aa^{i(j)}b^{2i}aab^{2k}aa^{i(k)}b^{2i}a,$$

  (iii) c_j and c_k both NOT gate:

$$X_i = ab^{2j}aa^{i(j)}b^{2i}aab^{2k}aa^{i(k)}b^{2i}a.$$

Define $Y_i = ab^{2i-1}b^{2i-1}ab^{2i}ab^{2i-1}ab^{2i}$. The reduction maps each gate c_i to the string Y_iX_i, so that the circuit $C_n = c_1 \dots c_n$ is reduced to the string $X = Y_1X_1Y_2X_2 \dots Y_nX_n$.

We see now how the parsing of X simulates the circuit. By parsing Y_1 we add to the dictionary six substrings that are *a*, *b*, *ba*, *bb*, *ab*, *abb*. Then the substring X_1, associated with the input gate c_1, is added to the dictionary and is equal to *bba* (*abba*) iff its input is 1 (0). Let c_m be the first OR gate in the topological order of the circuit to receive input values both equal to 1 or 0 (if such gate is not defined, the case is much simpler and the correctness of the reduction still follows from the arguments below). We can verify that, for $2 \leqslant i \leqslant m$, when Y_i is parsed the substrings ab^{2i-1}, b^{2i-1}, ab^{2i}, $ab^{2i-1}a$ and b^{2i} are added to the dictionary and the first character of X_i starts a new dictionary element. If c_i is an INPUT gate then the substring X_i is added to the dictionary and is equal to $b^{2i}a$ ($ab^{2i}a$) iff the input value is 1 (0). If c_i is a NOT gate with input c_j then by parsing X_i the substrings $b^{2j}aa^{i(j)}$ ($b^{2j}aa^{i(j)-1}$) and $b^{2i}a$ ($ab^{2i}a$) are added iff the output value of c_i is 0 (1). Since an OR gate needs just one input equal to 1 to have 1 as its own output value, if c_i is an OR gate with inputs from INPUT gates or NOT gates then by parsing X_i the substring $b^{2i}a$ is added iff its output value is 1. It follows that these conditions verify also for the gates receiving inputs from an OR gate. The inputs of c_m are both equal to 1 (0), so the substring $b^{2m}aa$ ($ab^{2m}aa$) is added to the dictionary, where the last a is prefix of Y_{m+1}. The substrings b^{2m+1}, $b^{2m+1}a$, $b^{2(m+1)}$, ab^{2m+1} and $ab^{2(m+1)}$ are added by parsing the suffix of Y_{m+1}. Thus, the substrings $b^{2(m+1)}$ and $ab^{2(m+1)}$ are in the dictionary and the first character of X_{m+1} starts a new dictionary element. It follows that, for $1 \leqslant i \leqslant n$, we learn the substrings $b^{2(i)}$ and $ab^{2(i)}$ by parsing Y_i, the first character of X_i starts a new dictionary element and the parsing of X_i has the properties shown above, even if $i > m$. Therefore, the circuit output is 1 iff $b^{2n}a$ is added to the dictionary, since c_n is an OR gate, i.e., iff the pointer to the dictionary element b^{2n} is in the coding. Observe that the parsing of the substring Y_i provides five dictionary elements when $i \geq 2$ and, since c_{n-1} is a NOT gate, b^{2n} is the fifth of the elements provided by the parsing of Y_n. Therefore, the pointer to b^{2n} is equal to $5n+1+l+2t+4(r-1)$ where l, t and r are, the number of INPUT, NOT and OR gates, respectively. □

4. The P-completeness of the next- and the first-character heuristics

We show the P-completeness of the next-character heuristic with a reduction from the same restricted version of the circuit value problem. The circuit will be reduced again to a binary string and a certain pointer will be in the coding iff the output of the circuit is 1.

Theorem 4.1. *The next-character heuristic is* P-*complete.*

Proof. The reduction maps the circuit C_n to a string X with a prefix $P = aZ_1 \ldots Z_{2n}$, where $Z_i = ab^i a^i b^i aab^{i-1} b^i b^{i+1} aab^i$. The suffix $S = Y_1 X_1 Y_2 X_2 \ldots Y_n X_n$ is constructed defining $Y_i = aab^{2i-1}a$ and associating X_i with c_i in the following way:

- c_i INPUT gate with value 1: $X_i = ab^{2i}a$,
- c_i INPUT gate with value 0: $X_i = aab^{2i}a$,
- c_i NOT gate having the output of c_j as input: $X_i = ab^{2j}aa^{i(j)}ab^{2i}a$,
- c_i OR gate having the outputs of c_j and c_k as inputs:
  (i) c_j and c_k both either OR gate or INPUT gate:
  $$X_i = ab^{2j}aa^{i(j)}ab^{2i}aab^{2k}aa^{i(k)}ab^{2i}a,$$
  (ii) c_j either OR gate or INPUT gate and c_k NOT gate:
  $$X_i = ab^{2j}aa^{i(j)}ab^{2i}aaab^{2k}aa^{i(k)}ab^{2i}a,$$
  (iii) c_j and c_k both NOT gate:
  $$X_i = aab^{2j}aa^{i(j)}ab^{2i}aacb^{2k}aa^{i(k)}ab^{2i}a.$$

Initially, the first character a is matched in the dictionary and the substring aa is added to it. When the substring Z_i is parsed, the substrings ab^{i-1}, ba^{i-1}, ab^i, aab^{i-1}, b^i, b^{i+1}, aab^i are matched and the substrings ab^i, ba^i, ab^ia, aab^i, b^{i+1}, $b^{i+1}a$, aab^ia are added to the dictionary. The dictionary elements ab^ia and aab^ia are the ones we utilize to parse the suffix S. The substring Y_i guarantees that the first character of X_i starts a dictionary element, for $1 \leq i \leq n$. If c_i is an INPUT gate then the substring X_i is matched, which is equal to $ab^{2i}a$ ($aab^{2i}a$) iff the input value is 1 (0), and $ab^{2i}aa$ ($aab^{2i}aa$) is added to the dictionary. When c_i is a NOT gate with input c_j, the substrings $ab^{2j}aa^{i(j)}$ ($ab^{2j}aa^{i(j)-1}$) and $ab^{2i}a$ ($aab^{2i}a$) are matched while the substrings $ab^{2j}aa^{i(j)}a$ ($ab^{2j}aa^{i(j)-1}a$) and $ab^{2i}aa$ ($aab^{2i}aa$) are added to the dictionary iff the output value of c_i is 0 (1). When c_i is an OR gate, the substring $ab^{2i}a$ is matched and the substring $ab^{2i}aa$ is added to the dictionary iff its output value is 1. It follows that the circuit output is 1 iff the pointer to the dictionary element $ab^{2^n}a$ is in the coding. The parsing of the substring Z_i provides seven dictionary elements and $ab^{2^i}a$ is the third element. Since the dictionary is initialized with the alphabet and the substring aa is added at the beginning, the pointer to $ab^{2^n}a$ is equal to $14n-1$. □

The first-character heuristic is proved to be P-complete by modifying slightly the reduction for the next-character heuristic.

Theorem 4.2. *The first-character heuristic is* P-*complete.*

Proof. We change the prefix P of the string X defining $Z_i = ab^ia^ib^iaab^{i-1}b^{2^i}b^{i+1}aab^i$. The suffix S remains the same. When the substring Z_i is parsed, the substrings $ab^{i-1}, ba^{i-1}, ab^i, aab^{i-1}, b^i, b^i, b^{i+1}, aab^i$ are matched and the substrings $ab^i, ba^i, ab^ia, aab^i, b^{i+1}, b^{i+1}a, aab^ia$ are added to the dictionary. The same seven elements are added to the dictionary when Z_i is parsed and ab^ia is still the third element, as in the former proof. The parsing of the suffix S is the same as the one for the next character heuristic. Therefore a pointer equal to $14n-1$ will be in the coding iff the output of the circuit is 1. □

Acknowledgment

I would like to thank Prof. J.A. Storer for helpful comments and discussions.

References

[1] R.J. Anderson and E.W. Mayr, Parallelism and greedy algorithms, in: F.P. Preparata, ed., *Advances in Computing Research: Parallel and Distributed Computing* (JAI Press, Greenwich, 1987) 17–38.
[2] M. Crochemore and W. Rytter, Efficient parallel algorithms to test square-freeness and factorize strings, *Inform. Process. Lett.* **38** (1991) 57–60.
[3] S. De Agostino and J.A. Storer, Parallel algorithms for optimal compression using dictionaries with the prefix property, submitted.
[4] A. Gibbons and W. Rytter *Efficient Parallel Algorithms* (Cambridge Univ. Press, Cambridge, 1989) 344–373.
[5] A. Hartman, and M. Rodeh, Optimal parsing of strings, in: A. Apostolico and Z. Galil, eds., *Combinatorial Algorithms on Words* (Springer, Berlin, 1985) 155–167.
[6] R.E. Ladner, The circuit value problem is log space complete for P, *SIGACT News* **7** (1975) 18–20.
[7] A. Lempel and J. Ziv, On the complexity of finite sequences, *IEEE Trans. Inform. Theory* **22** (1976) 75–81.
[8] A. Lempel and J. Ziv, A universal algorithm for sequential data compression, *IEEE Trans. Inform. Theory* **23** (1977) 337–343.
[9] V.S. Miller and M.N. Wegman, Variations on theme by Ziv–Lempel, in: A. Apostolico and Z. Galil, eds., *Combinatorial Algorithms on Words* (Springer, Berlin, 1985) 131–140.
[10] J.A. Storer and T.G. Szymanski, Data compression via textual substitution, *J. ACM* **29** (1982) 928–951.
[11] R.A. Wagner, Common phrases and minimum text storage, *Comm. ACM* **16** (1973) 148–152.
[12] T.A. Welch, A technique for high-performance data compression, *IEEE Comput.* **17** (1984) 8–19.
[13] J. Ziv and A. Lempel, Compression of individual sequences via variable rate coding, *IEEE Trans. Inform. Theory* **24** (1978) 530–536.