An On-line Variable Length Binary Encoding [1]

Tinku Acharya **Joseph F. Já Já**

Institute for Systems Research and

Institute for Advanced Computer Studies

University of Maryland

College Park, MD 20742

{*acharya, joseph*}*@umiacs.umd.edu*

Abstract

We present a methodology of an on-line variable-length binary encoding of a set of integers. The basic principle of this methodology is to maintain the prefix property amongst the codes assigned on-line to a set of integers growing dynamically. The prefix property enables unique decoding of a string of elements from this set. To show the utility of this on-line variable length binary encoding, we apply this methodology to encode the LZW codes. Application of this encoding scheme significantly improves the compression achieved by the standard LZW scheme. This encoding can be applied in other compression schemes to encode the pointers using variable-length binary codes.

1 Introduction

The basic idea behind any lossless data compression technique [1] is to reduce the redundancy in data representation. The two general categories of text compression techniques are *statistical coding* and *dictionary coding*. The statistical coding is based on the statistical probability of occurrence of the characters in the text, *e.g.* Huffman coding [2], Arithmetic coding [3] etc. In dictionary coding, a dictionary of common words is generated such that the common words appearing in the text are replaced by their addresses in the dictionary. Most of the adaptive dictionary based text compression algorithms belong to a family of algorithms originated by Ziv and Lempel [4, 5], popularly known as LZ coding. The basic concept of all the LZ coding algorithms is to replace the substrings (called phrases or words) with a pointer to where they have occurred earlier in the text. Different variations of these algorithms have been described in [6] which differ in the way the dictionary is referenced and how it is mapped onto a set of codes. This mapping is a code function to represent a pointer. The size of this pointer is usually fixed and determines the size of the dictionary. As a result, the same number of bits are transmitted for each pointer, irrespective of the number

[1] Technical Report : CS-TR-3442, UMIACS-TR-95-39

of phrases in the dictionary at any stage. This affects the compression performance at the beginning when more than half of the dictionary is empty. The most popular variation is the LZW algorithm [7] for text compression. We will describe a methodology of an on-line variable-length binary encoding of a set of integers and apply this methodology to the LZW codes to enhance the compression ratio. The variable-length on-line binary encoding scheme can be applied to other LZ encoding schemes as well.

In section 2, we formulate an on-line variable length binary encoding of a set of integers. We propose a solution to this problem, assuming that the correlation amongst the elements as well as their statistics are not known. The encoding will maintain the prefix property in order to uniquely decode any string formed by the elements of the set. We describe the LZW algorithm and redundancy in binary encoding of its output in sections 3 and 4 respectively. In section 5, we use the proposed binary encoding scheme to encode the LZW codes for variable-length binary encoding of text. The decoding operation to recover the original text will be discussed in section 6. We present the experimental results in section 7.

2 An On-line Variable Length Binary Encoding Problem

Before we formulate the on-line variable length binary encoding problem, let us define a data structure called the **phase in binary tree** and its properties below.

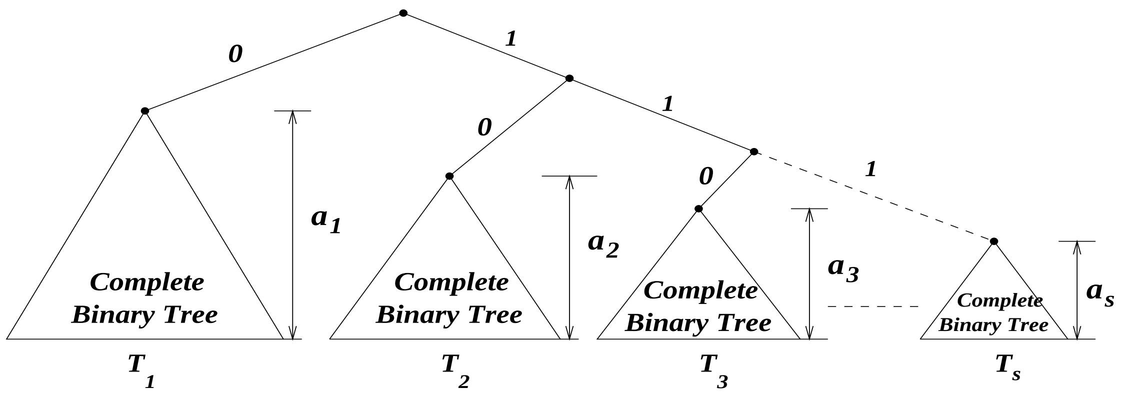


Figure 1: A phase in binary tree.

Definition 1: Given a positive integer N expressed as $N = \sum_{m=1}^{s} 2^{a_m}$, where $a_1 > a_2 > \cdots > a_s \geq 0$ and $s \geq 1$, a **phase in binary tree**[2] T with N leaf nodes is a rooted binary tree that consists of s **complete binary trees**[3] $T_1, T_2, \cdots, T_s$ of heights $a_1, a_2, \cdots, a_s$

[2] A broken triangle in this paper represents a phase in binary tree.

[3] A **complete binary tree** of height h is a binary tree with 2^h leaf nodes and $2^h - 1$ internal nodes. A complete binary tree is represented by a solid triangle in this paper.

respectively, arranged as shown in Figure 1.

Lemma 1: A phase in binary tree T with N leaf nodes is unique and it contains $N-1$ internal nodes.

Proof : The structure of the phase in binary tree with N leaf nodes is based on the representation of the positive integer N as

$$N = \sum_{m=1}^{s} 2^{a_m}$$

where $a_1 > a_2 > \cdots > a_s \geq 0$ and $s > 0$. Expression of N suggests that $a_1, a_2, \cdots, a_s$ are the positions of bit 1 when N is represented as a normalized unsigned binary number using $a_1 + 1$ bits. Since the representation of an integer in the form of a normalized unsigned binary number is always unique, the structure of the phase in binary tree with N leaf nodes is also unique.

According to the definition of a phase in binary tree, T consists of s complete binary subtrees $T_1, T_2, \cdots, T_s$ as shown in Figure 1 and the number of nodes in the complete binary tree T_m is $2^{a_m+1} - 1$, where a_m is the height of T_m for every $1 \leq m \leq s$. Hence sum of the number of nodes of all the above complete binary subtrees is

$$\sum_{m=1}^{s} (2^{a_m+1} - 1) = 2\sum_{m=1}^{s} 2^{a_m} - s = 2N - s$$

The remaining nodes in T form the path starting from the root node of T to the root node of T_{s-1}, consisting of $s-1$ nodes including the root node of T as shown in Figure 1. As a result, total number of nodes in the phase in binary tree T is $2N - s + (s-1) = 2N - 1$. Since T has N leaf nodes, the number of internal nodes in T including the root node is $2N - 1 - N = N - 1$. $\square$

Corollary 1: Height of the above phase in binary tree is $a_1 + 1$ if $s > 1$ and a_1 if $s = 1$.

Corollary 2: A complete binary tree is a special case of a phase in binary tree.

Definition 2: Given a set of n binary codes $V = \{V_1, V_2, \cdots, V_n\}$, such that $|V_i| \leq \lceil \log_2 n \rceil$ for every $1 \leq i \leq n$, the $value(V_i)$ of the binary code $V_i \in V$ is defined by the decimal value of V_i formed by appending $\lceil \log_2 n \rceil - |V_i|$ number of 0's at the end of V_i.

Definition 3: Given a phase in binary tree T with n leaf nodes, we can associate a unique set of binary codes $V = \{V_1, V_2, \cdots, V_n\}$ with the leaves by labeling every left edge of T by 0 and every right edge by 1 such that V_i is the sequence of 0's and 1's in the unique path from the root node to the ith leaf node of T, where the leaves are indexed consecutively from left to right. This set of binary codes is called a **phase in binary encoding** of size n.

Corollary 3: For a phase in binary encoding $V = \{V_1, V_2, \cdots, V_n\}$ of size n, the following relations always holds : $value(V_i) < value(V_{i+1})$ and $|V_i| \geq |V_{i+1}|$ for every $1 \leq i < n$.

Corollary 4: If V^a and V^b are two phase in binary encodings of sizes n_1 and n_2 respectively such that $n_1 < n_2$, $|V_i^a| \leq |V_i^b|$ and $|V_i^a| = |V_i^b| \Rightarrow V_i^a = V_i^b$ for every $1 \leq i \leq n_1$.

We formulate a problem of on-line variable length binary encoding of pointers arising in a dynamically growing dictionary here and present a solution with the aid of the above definition of a phase in binary tree. To make the problem more general, we represent this dynamically growing dictionary of pointers as a growing sequence of distinct elements in increasing order. The sequence is extended in every step by concatenating a number of elements at the end of the sequence and preserving the ordering. Let $P^{i-1} = \{\alpha_1, \alpha_2, \cdots, \alpha_{\eta_{i-1}}\}$ be the sequence of $\eta_{i-1} > 0$ distinct elements such that $\alpha_j < \alpha_{j+1}$ for every $1 \leq j < \eta_{i-1}$ at step $i-1$, $1 \leq i \leq t$ where t is a positive integer constant. At step i the sequence $P^i = \{\alpha_1, \alpha_2, \cdots, \alpha_{\eta_{i-1}}, \alpha_{\eta_{i-1}+1}, \cdots, \alpha_{\eta_i}\}$ is formed by concatenating the sequence $\{\alpha_{\eta_{i-1}+1}, \cdots, \alpha_{\eta_i}\}$ at the end of P^{i-1}. We want to map the sequence P^i dynamically into a set of variable length binary codes $C^i = \{C_1^i, C_2^i, \cdots, C_{\eta_i}^i\}$ maintaining certain prefix properties explained below.

Problem: Develop an on-line algorithm to generate the binary encoding $C^i = \{C_1^i, C_2^i, \cdots, C_{\eta_i}^i\}$ for the elements of P^i at any step i, such that

(1) No C_j^i is a prefix of C_l^i, for $1 \leq l \neq j \leq \eta_i$.

(2) No C_j^i is a prefix of C_l^{i-1}, for $1 \leq l \neq j \leq \eta_{i-1}$.

Solution: We can develop the above binary encoding C^i at step i by mapping the sequence P^i into the leaf nodes of a phase in binary tree B_i with η_i leaf nodes. After construction of the phase in binary tree B_i, we label every left edge by 0 and every right edge by 1. The sequence of 0's and 1's in the unique path from the root node of B_i to the jth leaf node from left is the binary code C_j^i of α_j in P^i. If η_i is expressed as $\eta_i = \sum_{m=1}^{s} 2^{a_m}$, where $a_1 > a_2 > \cdots > a_s \geq 0$, the first 2^{a_1} elements of P^i will be encoded by $a_1 + 1$ bits, the next 2^{a_2} elements will be encoded by $a_2 + 2$ bits, the following 2^{a_3} elements will be encoded by $a_3 + 3$ bits, and so on. But the last 2^{a_s} elements will be encoded by $a_s + s - 1$ bits.

Since the encoding is generated from the binary tree and each code is represented by a leaf node of the corresponding phase in binary tree, no code in any step can be a prefix of another code *i.e.* condition 1 of the problem holds. The condition 2 is obvious due to the properties of the phase in binary codes as in corollary 4.

2.1 Incremental Construction of the Phase in Binary Tree

The phase in binary tree B_i to represent the binary encoding C^i of the ordered sequence P^i in step i can be constructed incrementally by little systematic modifications of the phase in binary tree B_{i-1} generated in the previous step. This is explained below in detail. The number of elements in the concatenating subsequence $\{\alpha_{\eta_{i-1}+1}, \cdots, \alpha_{\eta_i}\}$ is $m_i = \eta_i - \eta_{i-1}$. This subsequence of m_i elements is concatenated at the end of the sequence $P^{i-1} = \{\alpha_1, \alpha_2, \cdots, \alpha_{\eta_{i-1}}\}$ to form the sequence $P^i = \{\alpha_1, \alpha_2, \cdots, \alpha_{\eta_{i-1}}, \alpha_{\eta_{i-1}+1}, \cdots, \alpha_{\eta_i}\}$ in step i. Several cases might arise depending upon the values of η_{i-1} and m_i :

case 1: η_{i-1} is a power of 2, say $\eta_{i-1} = 2^k$, $k \geq 0$ and $m_i \leq \eta_{i-1}$.

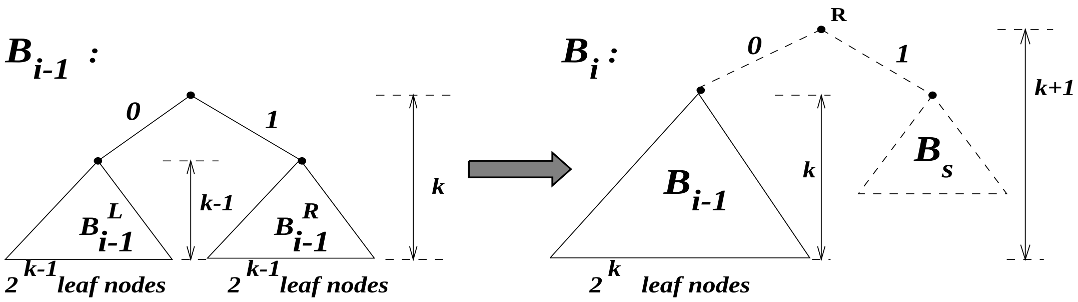


Figure 2: Construction of B_i from B_{i-1} for **case 1**.

According to the definition, B_{i-1} is a complete binary tree of height k with 2^k leaf nodes as shown in Figure 2. To generate B_i from B_{i-1}, we first construct a phase in binary subtree B_s with m_i leaf nodes corresponding to the concatenating subsequence $\{\alpha_{\eta_{i-1}+1}, \cdots, \alpha_{\eta_i}\}$. Then we create the root node **R** of B_i whose left child is the root node of B_{i-1} and the right child is the root node of B_s as shown in Figure 2. As a result, first η_{i-1} binary codes in C^i, *i.e.* $\{C^i_1, C^i_2, \cdots, C^i_{\eta_{i-1}}\}$ are generated by appending a 0 at the beginning of the corresponding codes in C^{i-1}. The last m_i codes in C^i, *i.e.* $\{C^i_{\eta_{i-1}+1}, , \cdots, C^i_{\eta_i}\}$ will be the binary sequences represented by the leaf nodes of the right subtree B_s in B_i.

case 2: η_{i-1} is a power of 2, say $\eta_{i-1} = 2^k$, $k \geq 0$ and $m_i > \eta_{i-1}$.

In this case m_i can be expressed as $m_i = n + \sum_{j=0}^{s-1} 2^{k+j}$, where $n < 2^k$. We construct s complete binary trees $B_i^0, B_i^1, \cdots, B_i^{s-1}$ of heights $k, k+1, \cdots, k+s-1$ respectively and a phase in binary tree B_s with n leaf nodes. The leaf nodes of B_i^0 will represent the first 2^k elements of the subsequence $\{\alpha_{\eta_{i-1}+1}, \cdots, \alpha_{\eta_i}\}$, the leaf nodes of B_i^1 will represent the next 2^{k+1} elements and so on. The last n elements of the concatenating subsequence will be represented by the leaf nodes of B_s. Then we connect these trees with B_{i-1} to construct B_i as shown in Figure 3 below.

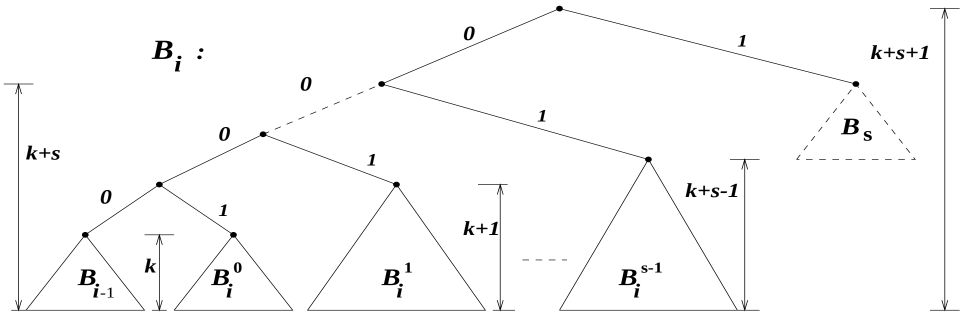


Figure 3: The phase in binary tree B_i in **case 2**.

As a result, the left subtree of the root node of B_i is a complete binary tree of height $k+s$ which can easily be verified from Figure 3. Note that the first η_{i-1} binary codes in C^i corresponding to the η_{i-1} elements in P^{i-1} will be formed by simply appending $s+1$ 0's at the beginning of the corresponding codes in C^{i-1}. The last m_i binary codes $\{C^i_{\eta_{i-1}+1}, \cdots, C^i_{\eta_i}\}$ in C^i are the binary sequences of 0's and 1's represented by the leaf nodes of $B_i^0, B_i^1, \cdots, B_i^{s-1}$ and B_s in B_i.

case 3: $\eta_{i-1} = 2^{k_1} + 2^{k_2} + \cdots + 2^{k_s}$, where $k_1 > k_2 > \cdots > k_s \geq 0$ and $m_i \leq 2^{k_s}$.

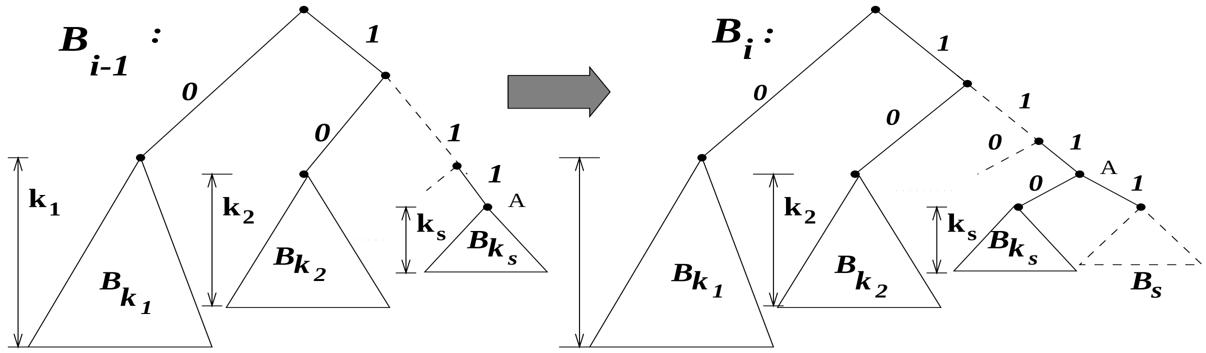


Figure 4: The phase in binary trees in **case 3**.

According to the construction of the phase in binary tree B_{i-1} with η_{i-1} leaf nodes, it will consist of the complete binary subtrees $B_{k_1}, B_{k_2}, \cdots, B_{k_s}$ of heights $k_1, k_2, \cdots, k_s$ respectively as shown in Figure 4. As a result, height of B_{i-1} is k_1+1. To construct B_i from B_{i-1}, a phase in binary tree B_s is constructed using m_i leaf nodes. The leaf nodes of this B_s represent the m_i concatenated elements in the sequence. B_s is then joined with B_{i-1} by converting B_{k_s} to be the left child and B_i as the right child of the node A as shown in Figure 4. Hence first $\eta_{i-1} - 2^{k_s}$ codes in C^i will be identical to the corresponding codes in C^{i-1} and the rest of the codes are determined by the binary sequences corresponding to the leaf nodes of B_{k_s} and B_s respectively in B_i.

case 4: $\eta_{i-1} = 2^{k_1} + 2^{k_2} + \cdots + 2^{k_s}$, where $k_1 > k_2 > \cdots > k_s \geq 0$ and $m_i > 2^{k_s}$.

This is a combination of the above three cases. In this case the complete binary tree B_{k_s} in B_{i-1} is first considered and the procedure as in case 2 is iterated until the height of the complete binary tree becomes $k_s + 1$ or all the elements of the concatenating subsequence have been considered depending upon the value of m_i. The new complete binary tree of height $k_s + 1$ is then considered, if all the elements in the concatenating subsequence has not been considered yet. If any element in the concatenating subsequence remains not assigned to a leaf node, the same procedure as in case 2 is again applied. The above procedure is iterated until all the elements in the concatenating subsequence have been represented by a leaf node in B_i.

3 The LZW Algorithm

Let $S = s_1 s_2 \cdots s_m$ be a string (or text) over the alphabet $\Sigma = \{a_1, a_2, \cdots, a_q\}$. The LZW algorithm maps S into the compressed string $c(S) = p_1 p_2 \ldots p_n$, where p_i is a positive integer and $p_i \leq n + q - 1$, for $i = 1, \ldots, n$. This mapping can be achieved with the aid of a "*dictionary trie*". This *dictionary trie* (T) is constructed on-line during the compression of the text as shown in Figure 5. Each node (N) in T at any step represents a substring which has been visited earlier into the already encoded text. This substring is found by concatenating the characters in the label of each node on the path from the root node to N. Each node is numbered by an integer which is used as a pointer (code value) to replace a matching substring into the text to form the codes in the output compressed text. The trie T is initialized on a $q+1$ rooted tree where the root is labeled $(0, \lambda)$ to represent the null string (λ). The root has q children nodes labeled $(1, a_1), (2, a_2), \ldots, (q, a_q)$ respectively to represent q single character strings. This is shown as T_1 in Figure 5. The input text is examined character by character and the longest substring in the text which already exists in the trie, is replaced by the integer number associated with the node representing the substring in the trie. This matching substring is called a **prefix string**. This prefix string is then extended by the next character to form a new prefix string. A child node is created at the node representing the matching substring in the trie. This new child node will now represent the new prefix string. In the algorithm described below, we express a prefix string by the symbol ω and the input character by K. In each step of the *Loop*, the next character K is read and the extended string ωK is tested to find whether the string ωK already exists in the trie. If it already exists, the extended string becomes the new prefix string ω. If ωK does not exist in the trie, it is inserted into the trie, the code value of ω is transmitted as compressed data and the character K becomes the first character of the next string ω.

The LZW Compression Algorithm :

```
begin
    Initialize the trie with single-character strings;
    Initialize ω with the first input character;
    Loop : Read next input character K;
        if no such K exists (input exhausted) then
            Output the code value of ω;
            EXIT from the Loop;
        end if;
        if ωK exits in the trie then
            ω ← ωK;
        else        /* The phrase ωK doesn't exist in the trie */
            Output the code value of ω;
            Insert the phrase ωK into the trie;
            ω ← K;
        end if;
    end Loop;
end.
```

If we assume that the size of the dictionary is big enough to accommodate all the parsed strings, the number of bits to encode each pointer is $\lceil \log_2 (n+q) \rceil$, *i.e.* if the pointer size is predetermined to be k-bits, we can accommodate a maximum of 2^k parsed strings in the dictionary and reinitialize the dictionary trie each time $(2^k - q)$ pointers are output.

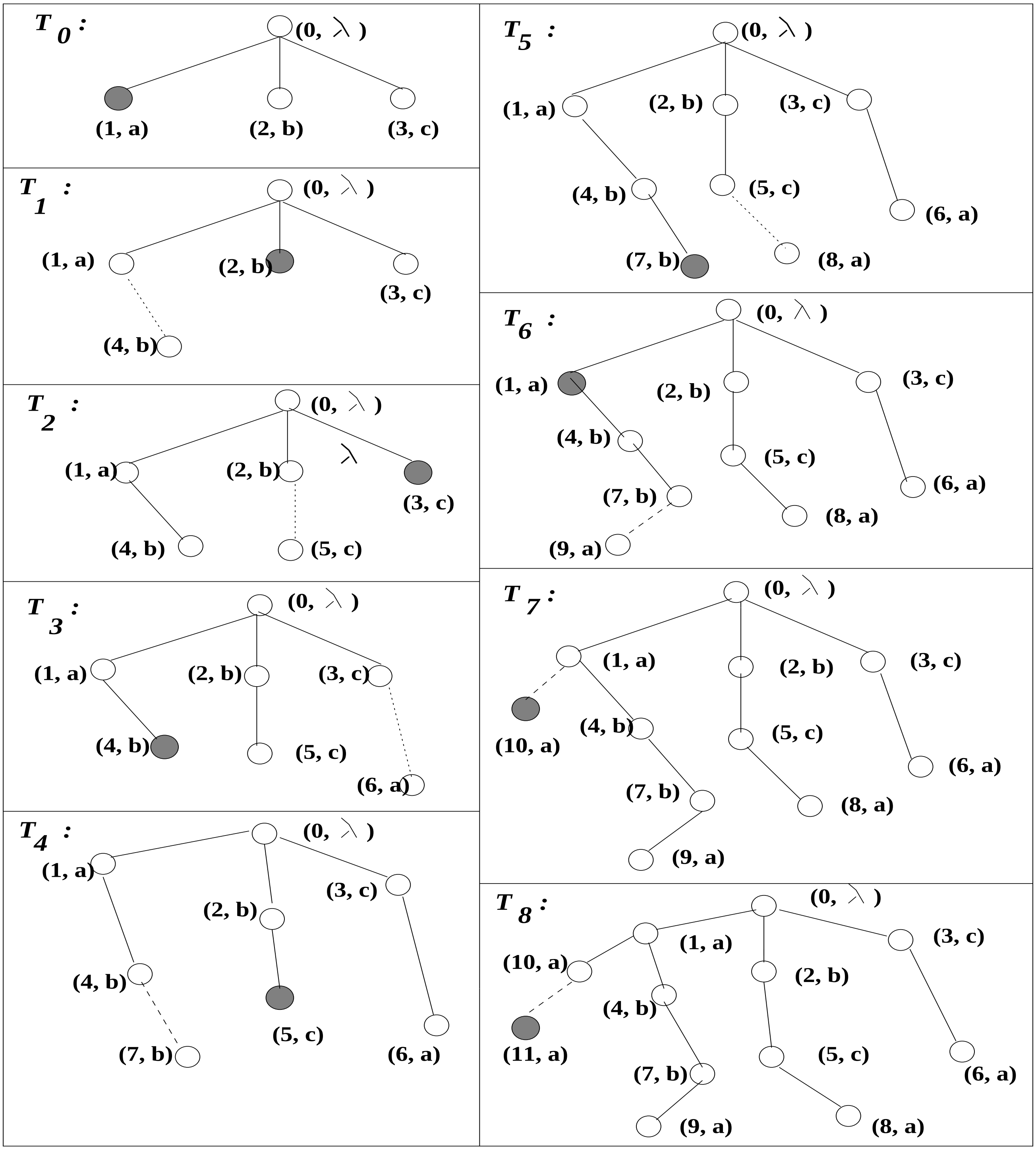


Figure 5: Example of the *dictionary Trie* using LZW algorithm.

Example: $\Sigma = \{a, b, c\}$, $S = abcabbcabbaaaaaa$. $c(S) = 1, 2, 3, 4, 5, 7, 1, 10, 11$.

The steps and the corresponding trie data structure to encode the above string S are shown in Figure 5. In each step $i \geq 0$, we output the pointer p_i which we find from the label associated with the dark node in trie T_i. T_i is then modified to T_{i+1} by inserting a node to represent the new prefix string which is shown by the node lead by the dotted line in T_{i+1}. Also note that the number of nodes in T_i is always $i + q$ and the pointer p_i to output at step i is $1 \leq p_i \leq i + q$. Since the final dictionary trie T_8 contains 11 prefix strings (represented by 11 nodes), each pointer will be encoded by $\lceil 11 \rceil = 4$ bits each. As a result, the size of the compressed string $c(S)$ is 36 bits.

4 Redundancy in Binary Encoding of the Output Codes

The size of the pointer in LZW algorithm is predefined and hence the same fixed number of bits are output in each LZW code irrespective of the number of entries in the dictionary. As a result a large number of bits are used unnecessarily when the number of phrases into the dictionary is less than half of its maximum size. In the LZC algorithm (which is a variant of the LZW algorithm used in UNIX-Z compression), string numbers are output in binary [8]. The number of bits used to represent the string number in any step varies according to the number of phrases (say M) currently contained in the dictionary. For example, when the value of M is in the range 256 to 511 each string number can be represented using a 9 bits binary number and when M becomes in the range 512 to 1023, each string number will be represented as a 10 bits binary number and so on. The scheme is also sub-optimal because a fractional number of bits is still wasted unless M is a power of 2. For example, when M is 513, it is possible to encode the first 512 string numbers in the dictionary (0 through 511) using 10 bits while 512 and 513 can still be represented as 2-bit binary numbers '10' and '11' respectively, without violating the prefix property because the first 512 binary codes will start with the bit 0 and the last two codes starts with the binary bit 1. The methodology can be developed as a special case of the phase in binary encoding scheme, we proposed in section 2. This is explained in the following section.

5 Allocation of the Phase in Binary Codes in LZW output for variable-length binary encoding

In each step of the LZW algorithm, only one phrase may be inserted into the dictionary, *i.e.* only one pointer is appended into the pointer list. We can express the set of pointers in step

i as a sequence of $q+i$ integers $P^i = \{1, 2, \cdots, q+i\}$ in increasing order. As a result, this can be considered as a special instance of the proposed on-line variable length binary encoding problem presented in section 3. Here the length of the subsequence being concatenated at every step is 1, specifically the integer $q+i$ is appended at the end of the sequence P^{i-1} in step i to form the extended sequence P^i. Hence the sequence P^i of length $q+i$ can be mapped into the leaf nodes of a phase in binary tree with $q+i$ leaf nodes as described in section 2. As a result, the LZW codes (*i.e.* the pointers) in every step can be encoded by the phase in binary codes to form a variable-length binary encoding of text. We first show with the same example how to assign the variable length binary codes to the pointers in each step of the compression. We later describe a decoding algorithm that uniquely generates the original LZW codes. The steps are shown in Figure 6. The single character strings are represented by the *dictionary trie* T_0 with the set of pointers $P^0 = \{1, 2, 3\}$ as shown in Figure 6. These pointers can be mapped into the variable length binary encoding $C^0 = \{00, 01, 1\}$ represented by the phase in binary tree B_0. The set of binary codes of the pointers to the single character strings maintain the prefix property, *i.e.* no code is a prefix of another code into the set and the output pointer 1 can be encoded with 2 bits '00'. In the next step the set of pointers is $P^1 = \{1, 2, 3, 4\}$, which can be mapped into the set of binary encodings $C^1 = \{00, 01, 10, 11\}$ represented by the binary tree B_1 as shown in Figure 6. Hence the output pointer 2 can be encoded using two bits '01'. Following the same procedure the sets of pointers $P^2, P^3, \cdots, P^8$ can be mapped into the sets of binary encodings $C^2, C^3, \cdots, C^8$ obtained from the phase in binary trees $B_2, B_3, \cdots, B_8$ respectively as shown in Figure 6. Accordingly, the output pointers 3, 4, 5, 7, 1, 10 and 11 in the next seven consecutive steps can be encoded as '010', '011', '100', '110', '0000', '11' and '11' respectively. Hence the compressed string $c(S)$ can be encoded as 00 01 010 011 100 110 0000 11 11 using 24 bits, instead of 36 bits using the fixed-length encoding.

6 Decoding of the Binary Codes

During the decompression process using the LZW algorithm, logically the same trie is created as in the compression process. But generation and interpretation of the binary tree is a little tricky although the same binary tree will be essentially generated as in the compression process. To decompress the binary codes, we start with the same phase in binary tree B_0 and the corresponding trie T_0 as in the compression process. In each step i, a string of characters (say S_i) is regenerated. In step i, the phase in binary tree B_i is traversed starting from the root node. The left child (or the right child) is traversed if the next input bit is 0 (or 1) until a leaf node in B_i is reached. The label of this leaf node in B_i is the pointer to a node in T_i which represents the regenerated substring S_i to output. In step i, the phrase formed by concatenating the previous output substring S_{i-1} (only exception during the first step, $i = 0$ and $S_{-1} = \lambda$) and the first character (say K) of the present output substring S_i, *i.e.* $S_{i-1}K$ is inserted into the trie T_i to create the new trie T_{i+1}. If the substring $S_{i-1}K$ already exists in T_i, the trie T_{i+1} will be identical to T_i. The new node (if any) in T_{i+1} will be represented by the pointer $q + i$ in T_{i+1}. The phase in binary tree B_i is then incremented to B_{i+1} and the new leaf node B_{i+1} is labeled by the integer $q + i + 1$. The same procedure is repeated until all the bits in binary code $c(S)$ have been exhausted.

It should be mentioned here that the decoding operation in LZW is handled differently in a special case. This special case occurs whenever the input string contains a substring of the form $K\omega K\omega K$, where $K\omega$ already appears in the trie. Here K is a single character from the alphabet and ω is a prefix string. This is explained in detail in the original paper of LZW [7]. If the original input string (S) contains a substring of this form, during the decoding operation of our proposed binary encoding, we will find that the decoded label from the binary tree is not yet created in the trie in the corresponding step (say step j). In this case, we know that the output string S_{j-1} (obtained in the immediate previous step $j - 1$) was of the form $K\omega$. Hence we need to output a substring S_j of the form $K\omega K$ in the present step and insert the prefix string $S_j = K\omega K$ into the trie T_j to form T_{j+1} and the corresponding node representing S_j in T_{j+1} will be marked by $q + j$.

We describe the above decoding process using the same example to decode the binary code $c(S)$ = 00 01 010 011 100 110 0000 11 11. This binary code $c(S)$ was obtained by compressing $S = abcabbcabbaaaaaa$ using the proposed phase in binary encoding scheme. We start with the phase in binary tree B_0 with 3 leaf nodes and the trie T_0 consisting of the single character strings as shown in Figure 7. Here the alphabet is $\Sigma = \{a, b, c\}$ and hence size of Σ is $q = 3$. In the first step ($i = 0$), we traverse the phase in binary tree B_0 starting from the root node and reach to the leaf node of label 1 after reading the first two input bits 00. This label 1 indicates a pointer in trie T_0 which represents the output substring S_0='a'. The binary tree B_0 is modified to B_1 to represent a phase in binary tree with 4 leaf nodes and the new leaf node is now labeled by $q + i + 1 = 4$ as shown in the phase in binary tree B_1 in Figure 7. In the next step ($i = 1$), the B_1 is traversed and the next two input bits 01 lead to the leaf node 2 in B_1. The node of label 2 in trie T_1 represents the output string S_1='b'. This character concatenated with the previous output substring 'a' forms the new prefix string 'ab' which is inserted into the trie to form T_2. This is shown by the broken edge in T_2 in Figure 7. The new pointer to the prefix string 'ab' is $q + i = 4$. The phase in binary tree B_1 is now incremented to form the phase in binary tree B_2 with 5 leaf nodes. The new leaf node is labeled by $q + i + 1 = 5$ as shown in B_2 in Figure 7. In the following step ($i = 2$), B_2 is traversed and the next three input bits 010 lead to the leaf node of label 3 which represents the output substring S_2='c' in T_2. This is concatenated with the previous output substring S_1='b' to form the new prefix string 'bc' and inserted into the dictionary trie to form T_3. The new pointer to the prefix string 'bc' in T_3 is labeled by $q + i = 5$. The phase in binary tree B_2 is now incremented to form the phase in binary tree B_3 with 6 leaf nodes. The new leaf node is labeled by $q+i+1 = 6$ in B_3. B_3 is then traversed and the next three input bits 011 lead to the leaf node of label 4 which represents the output substring S_3='ab' as shown in T_3. The first character(a) of this output substring is concatenated with the previous output string S_2='c' to form the new prefix string 'ca' which is inserted into the trie to form T_4. The pointer of this prefix string 'ca' in T_4 is now $q + i + 1 = 6$. The phase in binary tree B_3 is now incremented to the phase in binary tree B_4 with 7 leaf nodes and the new leaf node is labeled by $q + i + 1 = 7$. Following the same procedure in next three steps (*i.e.* $i = 4, 5, 6$), we can regenerate the decoded output substrings S_4='bc', S_5='abb' and S_6='a' for the binary sequences 100, 110 and 0000 from the phase in binary trees B_4, B_5, B_6 and the corresponding tries T_4, T_5 and T_6 respectively. In step 7 ($i = 7$), we will traverse the binary tree B_7 and the next two input bits 11 will lead to the leaf node marked by the integer 10 in B_7. But there is no node in the trie T_7 labeled by the pointer 10. This arises due to the special case of appearance of a substring $K\omega K\omega K$, as we described above. Hence the output substring should be of the form $K\omega K$. In this case, $K\omega$ is the output substring S_6='a' in the previous step. As a result, $K = a$ and $\omega = \lambda$ here. So we output the substring S_7=$K\omega K = aa$ and insert the prefix string 'aa' into the trie represented by the node labeled by the pointer $q + i = 10$ to form T_8. The same situation arises, for the next two input bits 11. By traversing the tree B_8 using the bits 11, we reach to the leaf node of label 11. But there is no pointer 11 in trie T_8. Hence the special case arises, where $K = a$ and $\omega = a$. Hence output substring will be $S_8 = K\omega K = aaa$. Since all the input bits are exhausted the decoding process stops. Now concatenating all the output substrings above, we find that the decoded string is *abcabbcabbaaaaaa* which is the original string S.

7 Experimental Results

We have implemented our scheme and tested it with texts of different sizes and characteristics. In all the cases, the phase in binary encoding method significantly improves the performance of the raw LZW technique with fixed-length pointer size of 12 bits and starting the dictionary all over again after it is full. We have performed our experiment with different maximum allowable height of the binary tree. The best performance is achieved when the binary tree is allowed to grow to a maximum height of 15. In our implementation, we allowed the binary tree to grow until it becomes a complete binary tree of height 15 and start all over again after that. We presented the experimental results in the table below to compare the compression performance based on the LZW method and our proposed variable-length encoding scheme which we call the LZWAJ scheme. The experiment was performed on the files obtained from the University of Calgary text corpus in addition to several other text files. The experimental results of our scheme illustrate the significant improvements obtained by using our scheme over the LZW algorithm.

(Sizes are expressed in nearest Kilo-bytes)

TEXT	Original	LZW	LZWAJ
bib	111	64	45
book1	768	446	346
book2	610	346	259
geo	102	84	77
news	377	246	188
obj1	21	14	13
obj2	246	143	123
paper1	53	31	24
paper2	82	45	35
progc	39	22	18
progl	71	32	25
progp	49	19	18
trans	93	50	36
Average	202	119	93

8 Conclusion

In this paper, we have presented a novel methodology for an on-line variable-length binary encoding of a growing set of integers by mapping the integers into the leaf nodes of a special binary tree called the phase in binary tree. The characteristics of the phase in binary tree is that the left subtree is always a complete binary tree and the height of the right subtree is no bigger than the height of the left subtree. This phase in binary tree can be represented as a composition of a number of height balanced complete binary trees. The binary codes are generated in such a way that prefix property between two consecutive steps is always maintained for the sake of unique decoding operation to regenerate the original text. We have used this methodology to further enhance the compression performance using the LZW scheme to show the effectiveness of the newly defined phase in binary encoding. As a result, a variable-length binary encoding of the text is possible using the popular LZW codes and the phase in binary encoding. The experimental results show that we can achieve much better compressions than these obtained by using the standard LZW algorithm.

References

[1] Storer, J. A."Data Compression: Methods and theory." *Computer Science Press*, Rockville, MD, 1988.

[2] Huffman, D.,"A Method for the Construction of Minimum Redundancy Codes," *Proc. IRE*, Vol. 40, 1952, pp. 1098-1101.

[3] Witten, I.H., Neal, R., and Cleary, J.G.,"Arithmetic coding for data compression," *Communication of the ACM*, 30(6), 520-540, June 1987.

[4] Ziv, J. and Lempel, A.,"A Universal Algorithm for Sequential Data Compression," *IEEE Trans. on Info. Theory*, IT-23, 3, May 1977, pp. 337-343.

[5] Ziv, J., and Lempel, A.,"Compression of Individual Sequences Via Variable-rate Coding," *IEEE Trans. Info. Theory*, IT-24, 5, September 1978, pp. 530-536.

[6] Bell, T. C., Cleary, J. G. and Witten, I. H.,"Text Compression," *Prentice Hall*, NJ, 1990.

[7] Welch, T.,"A Technique for High-Performance Data Compression," *IEEE Computer*, 17 (6), 8-19, June 1984, pp. 8-19.

[8] Horspool, R. N.,"Improving LZW," *Data Compression Conference*, 1991, pp. 332-341.

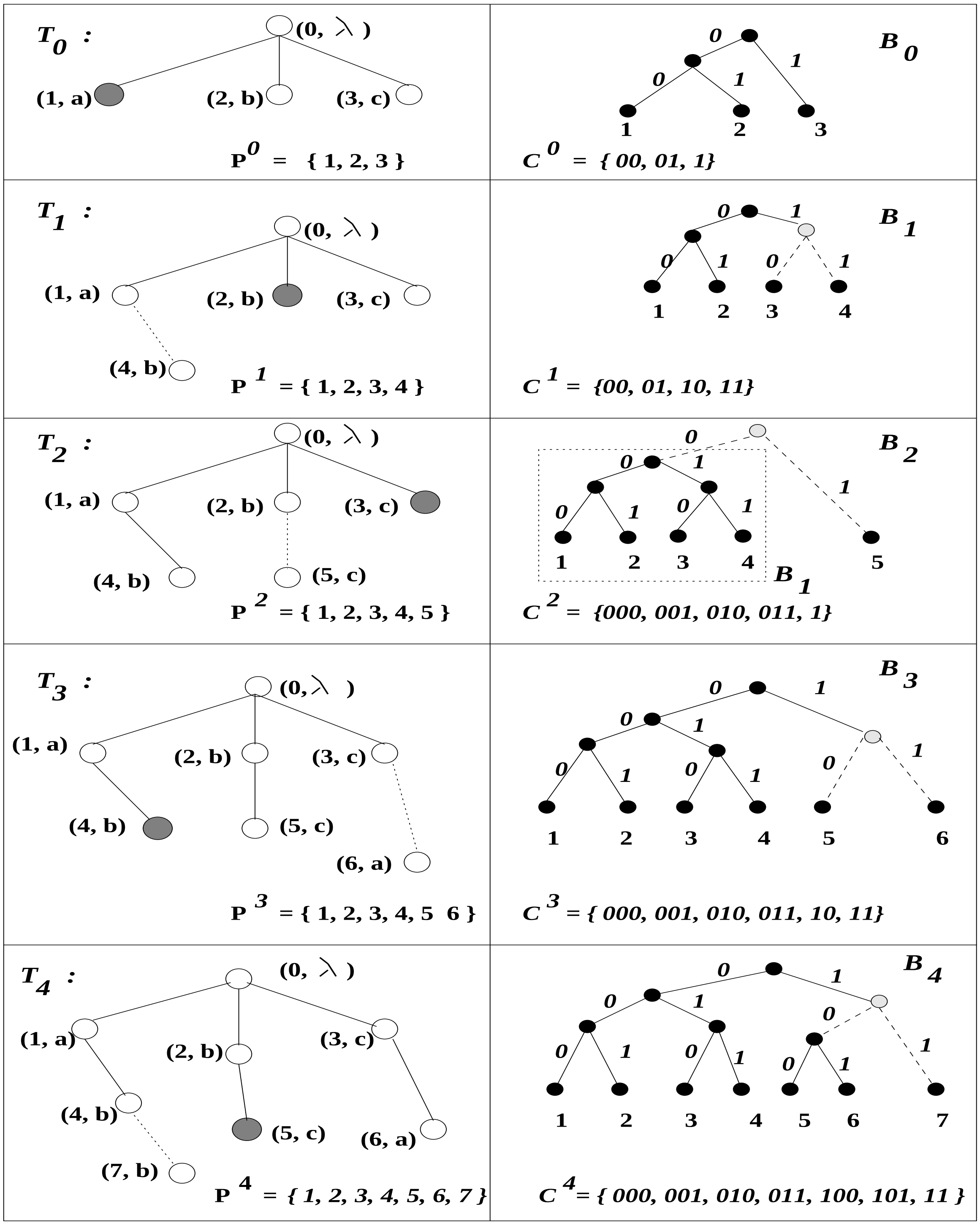
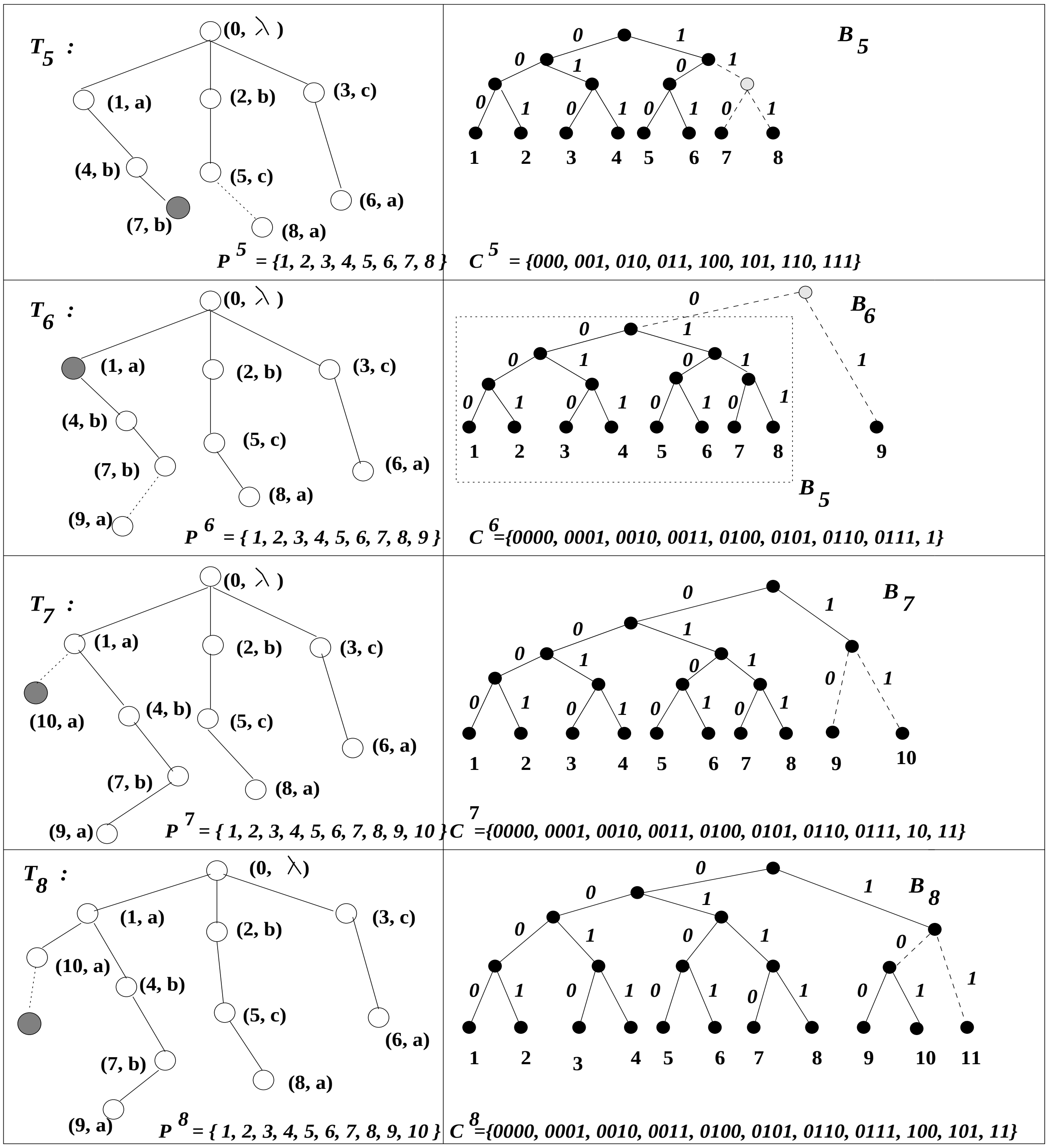


Figure 6: Example of the LZWAJ coding (Continued in the next page also).



Example of the LZWAJ encoding (Continued).

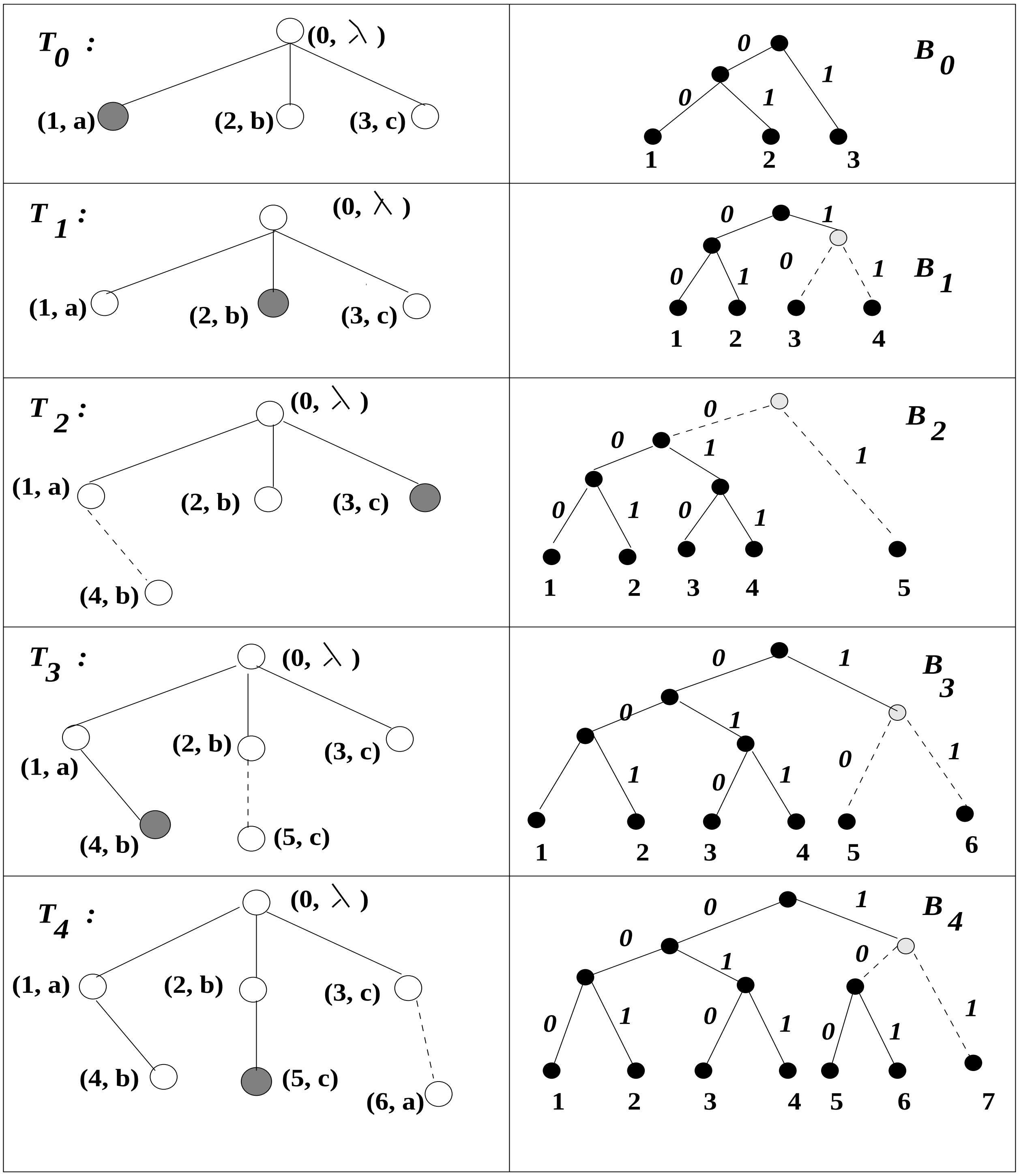
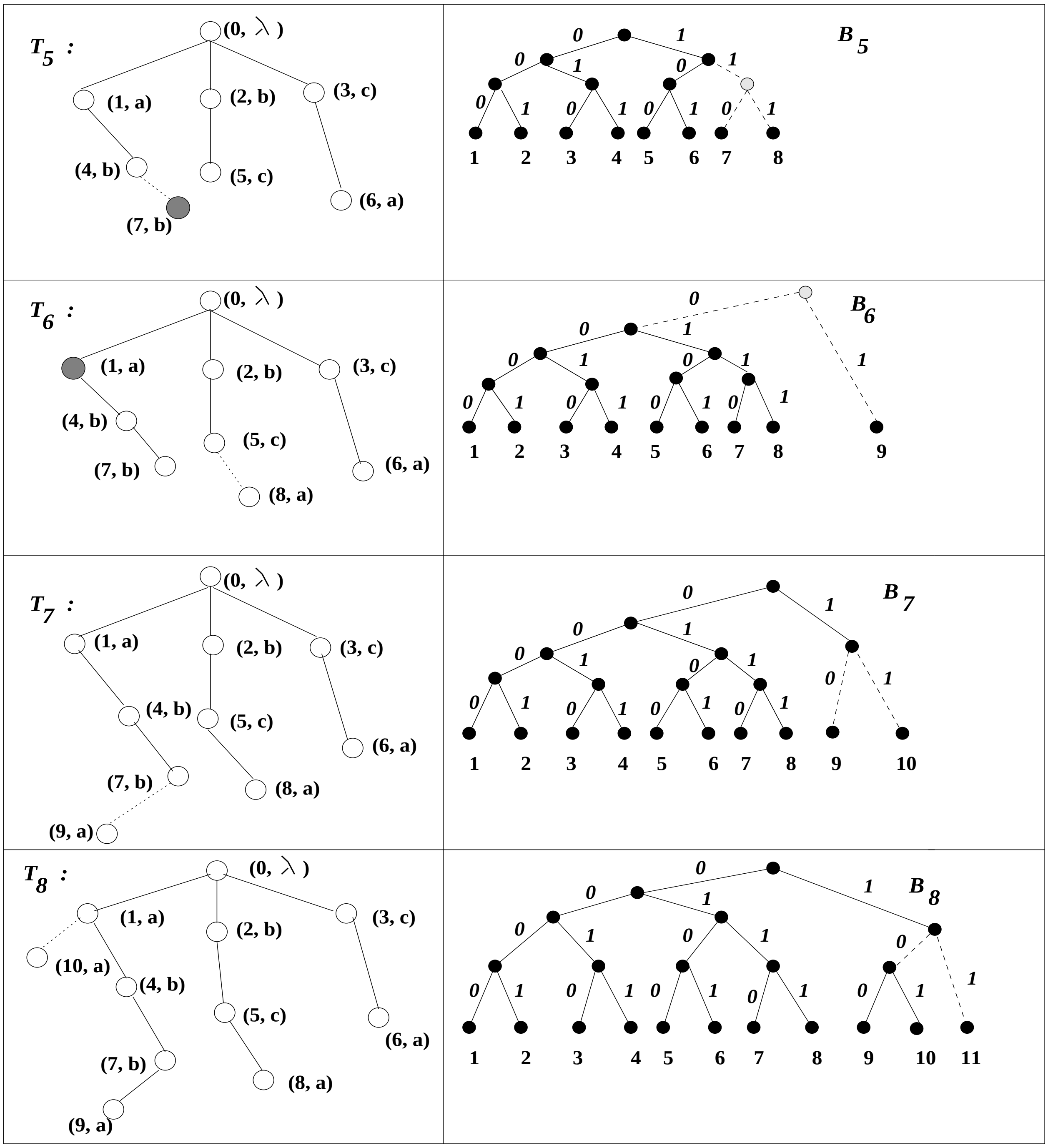


Figure 7: Example of the LZWAJ decoding of binary codes.



Example of the LZWAJ decoding (Continued).