SECURITY AND COMMUNICATION NETWORKS
Security Comm. Networks. 2008; **1**:325–335
Published online 6 June 2008 in Wiley InterScience
(www.interscience.wiley.com) DOI: 10.1002/sec.32

An efficient pattern matching scheme in LZW compressed sequences

Tsern-Huei Lee*,† and Nai-Lun Huang†

Department of Communication Engineering, National Chiao Tung University, Hsinchu, Taiwan 300, ROC

Summary

Compressed pattern matching (CPM) is an emerging research field addressing the problem: given a compressed sequence and a pattern, process the sequence with minimal (or no) decompression to find the pattern occurrence(s) in the uncompressed sequence. It can be applied to detect malwares and confidential information leakage in compressed files directly. In this paper, we report our work of CPM in Lempel–Ziv–Welch (LZW) compressed sequences. We propose an efficient bitmap-based realization of the Amir–Benson–Farach algorithm. We also generalize the algorithm to find all pattern occurrences and report their absolute positions in the uncompressed sequence. Experiments are conducted to test the space requirements of our proposed generalization and two related CPM schemes which can also be realized with bitmaps. Results show that our proposed generalization requires the least amount of storage for moderate and long patterns. We also conduct experiments to compare the throughput performance of our proposed generalization with these two related CPM schemes and the decompress-then-search scheme. Results show that our proposed generalization outperforms the decompress-then-search scheme significantly. When scanning a file with pattern occurrences, our proposed generalization performs slightly better than the two related CPM schemes. The difference is significant when scanning a file with no pattern occurrence. Copyright © 2008 John Wiley & Sons, Ltd.

KEY WORDS: bit-parallelism; compressed pattern matching; information search and retrieval; LZW compression; malwares detection; string matching

1. Introduction

As the population of communication network users grows at a rapid rate, it is expected that the network be capable of delivering data more effectively. In other words, how to utilize the transmission bandwidth efficiently is a key upon which the success of the communication network heavily relies. Obviously, an economic way to utilize limited bandwidth efficiently is to send smaller amount of data by using data compression mechanisms. Accordingly, compressed pattern matching (CPM) that performs pattern search directly on the compressed data without decompression gains more and more attention. The CPM problem is often defined as: given a compressed sequence and a pattern, process the sequence with minimal (or no) decompression to find the pattern occurrence(s) in the uncompressed sequence. One possible application of CPM is to detect malware if the signature of the malware is treated as the pattern to

*Correspondence to: Tsern-Huei Lee, Department of Communication Engineering, National Chiao Tung University, Hsinchu, Taiwan 300, ROC.
†E-mail: tlee@banyan.cm.nctu.edu.tw
†E-mail: nellen.cm938@nctu.edu.tw

Copyright © 2008 John Wiley & Sons, Ltd.

be searched. Another possible application is for the detection of confidential information leak where the pattern represents a unique string of the confidential information.

Since Lempel–Ziv–Welch (LZW) algorithm [1] is one of the most effective and popular lossless compression algorithms, CPM in LZW compressed sequences is quite important. In the last decade, several related researches have been conducted. The first CPM algorithm which finds the first pattern occurrence in an LZW compressed file was presented in Reference [2]. It takes $O(n+m^2)$ time and space, where n and m are, respectively, the lengths of the compressed sequence and the pattern. This algorithm is now well known and will be referred to as the Amir–Benson–Farach (ABF) algorithm in this paper. The analysis in Reference [2] shows that one can trade between the amount of extra space required and the time used with different implementations. However, Reference [2] proposed the algorithm without realizing it (according to Tao and Mukherjee's personal communication with Amir [3]). Thus, no experimental results are available to show the practical performance. Details of the ABF algorithm are presented in Section 3. In Reference [3], the ABF algorithm is extended to find all pattern occurrences. The basic idea is to use a flag to indicate that complete pattern occurs inside a compressed data block, in addition to checking pattern occurrences across two consecutive blocks. However, there is no clear description of how to implement the extended algorithm. In Reference [4], another CPM algorithm was proposed to do decompression and pattern matching on-the-fly. It still needs partial decompression, which may result in relatively high computation complexity. Reference [5] presented a general scheme to find all pattern occurrences in sequential blocks and realized the scheme by using the technique of bit-parallelism. This scheme can be applied to Ziv–Lempel family [1,6,7], including LZ77, LZ78, and LZW compressed sequences and will be referred to as the Navarro–Raffinot (NR) scheme in this paper. Another bitmap-based implementation for finding all pattern occurrences in LZW compressed sequences was independently proposed in Reference [8], which will be referred to as the Kida scheme. The NR scheme and the Kida scheme will be briefly described in Section 6. These two schemes, which can be realized with bitmaps, are variations of the ABF algorithm and will be compared with our work in Section 7.

In this paper, we present our work of CPM in LZW compressed sequences. We propose a bitmap-based realization of the ABF algorithm. Moreover, a generalization of the ABF algorithm to find all pattern occurrences and report their absolute positions in the uncompressed sequence is presented. Experiments are conducted to compare both throughput and space performance of our proposed generalization with the NR scheme, the Kida scheme, and the decompress-then-search scheme. Results show that our proposed generalization is significantly faster than the decompress-then-search scheme and slightly faster than the NR scheme and the Kida scheme when there are pattern occurrences. When searching a file with no pattern occurrence, our proposed generalization has the best throughput performance. Moreover, our proposed generalization has better space performance than both the NR scheme and the Kida scheme for moderate and long patterns.

The rest of this paper is organized as follows. Sections 2 and 3 give reviews of LZW and ABF algorithms, respectively. The proposed bitmap-based realization of the ABF algorithm is presented in Section 4, followed by the generalization for all pattern occurrences in Section 5. Section 6 describes the most related works, that is, the NR scheme and the Kida scheme. Experimental results and comparisons are shown in Section 7. Finally, Section 8 concludes this paper.

2. The LZW Compression Algorithm

In this section, we briefly review the LZW compression algorithm and the corresponding decompression procedure [1]. The notations used here are similar to those in Reference [2]. Let $S = c_1c_2c_3\cdots c_u$ be the uncompressed sequence (or text) of length u over alphabet $\Sigma = \{a_1, a_2, a_3, \ldots, a_q\}$, where q is the size of the alphabet. The LZW compressed format of S is $S \cdot Z$ and each code in $S \cdot Z$ is $S \cdot Z[i]$, where $1 \leq S \cdot Z[i] \leq n+q-1$ for $i = 1, \cdots, n$. The pattern being searched is $P = p_1p_2p_3\cdots p_m$, where m denotes the length of P and $p_i \in \Sigma$ for $1 \leq i \leq m$. For convenience, we use the notation S_1S_2 to denote the concatenation of two strings S_1 and S_2.

The LZW is a dictionary-based compression algorithm that uses a trie T_S to generate the compressed sequence. Each node on T_S contains:

- A node number: a unique number in the range $[0, n+q-1]$. ('node N' or 'N' represents 'the node numbered N' in this paper.)
- A label: a symbol in $\Sigma \cup \{NULL\}$.

Copyright © 2008 John Wiley & Sons, Ltd.

DOI:10.1002/sec

- A chunk: the string that the node represents. It is simply the concatenation of the labels on the path from root to the node.

T_S and the compressed sequence are constructed as follows:

1. T_S is initialized as a $(q+1)$-node trie consisting of a root node numbered 0 and labeled *NULL* and q child nodes numbered $1, 2, \cdots, q$. Child node i is labeled a_i.
2. During compression, the LZW algorithm finds the longest prefix of the uncompressed sequence that is a chunk represented by some node N on T_S and outputs N to $S \cdot Z$. T_S is then grown by adding a new node as a child of N. The new node's label is the next unencoded symbol in the sequence.

At the end of compression, there are $n+q$ nodes on T_S.

The decompression procedure constructs the same trie T_S and uses it to decode $S \cdot Z$. It is obvious that both compression and decompression can be done in time $O(u)$. The following observation makes it possible to construct T_S from $S \cdot Z$ in time $O(n)$ without decoding $S \cdot Z$ [2]. Note that, in order to construct T_S from $S \cdot Z$, an additional symbol is stored in each node. This additional symbol is the first symbol of the node's chunk.

Observation. *The code $S \cdot Z[l]$, $1 \leq l \leq n-1$, causes creation of a new node numbered $l+q$ as a child of node $S \cdot Z[l]$.*

- *The first symbol of $l+q$'s chunk is that of $S \cdot Z[l]$'s chunk.*
- *The last symbol or the label of node $l+q$ is the first symbol of $S \cdot Z[l+1]$'s chunk. (If $S \cdot Z[l+1] = l+q$, then the first symbol of $S \cdot Z[l+1]$'s chunk is the same as that of $S \cdot Z[l]$'s chunk.)*

3. The Amir–Benson–Farach Algorithm

The Amir–Benson–Farach (ABF) algorithm [2] is an effective scheme which finds the first pattern occurrence in LZW compressed sequence without decompression. To facilitate pattern matching, the following terms of a node on T_S are defined with respect to pattern P.

Definition 1. *A chunk is a prefix chunk if it ends with a nonempty prefix of P. Similarly, a chuck is a suffix chunk if it begins with a nonempty suffix of P.*

Definition 2. *A chunk is an internal chunk if it is an internal substring of P. That is, the substring $p_i \cdots p_j$ is an internal chunk if $1 \leq i \leq j \leq m$.*

Definition 3. *The prefix number of a chunk is the length of the longest pattern prefix the chunk ends with. Similarly, the suffix number of a chunk is the length of the longest pattern suffix the chunk begins with.*

Definition 4. *The internal range $[i, j]$ of a chunk indicates that the chunk is the internal chunk $p_i \cdots p_j$ if $1 \leq i \leq j \leq m$, or not an internal chunk if $i = j = 0$.*

If a node's chunk is a prefix chunk, a suffix chunk, or an internal chunk, the node is called a prefix node, a suffix node, or an internal node, respectively. Prefix number $= 0$, suffix number $= 0$, or internal range $= [0, 0]$ means that the node is not a prefix node, a suffix node, or an internal node, respectively.

The ABF algorithm consists of the preprocessing part and the compressed text scanning part which are described separately below:

A. Preprocessing:
The pattern is preprocessed to allow answering the following queries:

1. Let S_1 be a pattern prefix with prefix number P_x and S_2 be a string with internal range I.

$$Q_1(P_x, I) = \text{prefix number of } S_1S_2 \tag{1}$$

2. Let S_1 be a pattern prefix with prefix number P_x and S_2 be a nonempty pattern suffix with suffix number S_x.

$$Q_2(P_x, S_x) = \begin{Bmatrix} i, & i \text{ is the smallest index of } S_1S_2 \\ & \text{where the pattern occurs} \\ 0, & \text{no pattern occurs in } S_1S_2 \end{Bmatrix} \tag{2}$$

3. Let S_1 be an internal substring of P and $\alpha \in \Sigma$.

$$Q_3(S_1, \alpha) = \text{internal range of } S_1\alpha \tag{3}$$

Copyright © 2008 John Wiley & Sons, Ltd.

DOI:10.1002/sec

B. Compressed text scanning:
The compressed text scanning part is further divided into two components: the LZW trie construction and the pattern search. When constructing T_S, each node is assigned a node number, the first symbol of its chunk, a label, a prefix number, a suffix number, and an internal range. The pattern search part keeps track of the largest partial match and finds out if the partial match can be extended to a complete match. The compressed text scanning procedure is described below.

Initialize: variable $Prefix \leftarrow 0$
for $l = 1$ to n do
(Let P_x, S_x, and I denote node $S \cdot Z[l]$'s prefix number, suffix number, and internal range, respectively.)

1. LZW trie construction:
   1.1. Add a new node numbered $l + q$ to T_S as a child node of $S \cdot Z[l]$. Let α denote the label of node $l + q$.
   1.2. The first symbol of $l + q$'s chunk is that of $S \cdot Z[l]$'s chunk.
   1.3. Set α as the first symbol of $S \cdot Z[l+1]$'s chunk. (If $S \cdot Z[l+1] = l + q$, then the first symbol of $S \cdot Z[l+1]$'s chunk is the same as that of $S \cdot Z[l]$'s chunk.)
   1.4. If $S \cdot Z[l]$ is an internal node, set $l + q$'s internal range $[i, j]$ as $Q_3(S_1, \alpha)$, where S_1 denotes the string represented by $S \cdot Z[l]$. Otherwise, set $l + q$'s internal range $[i, j]$ as $[0, 0]$.
   1.5. If $j = m$, set $l + q$'s suffix number as $m - i + 1$. Otherwise, set $l + q$'s suffix number as S_x.
   1.6. Set $l + q$'s prefix number as $Q_1(P_x, I_\alpha)$, where I_α is the internal range of α.
2. Pattern search:

If $Prefix = 0$, $Prefix \leftarrow P_x$
Else,
  If $S_x \neq 0$,
    // Check the pattern occurrence with $Q_2(Prefix, S_x)$
    If $Q_2(Prefix, S_x) \neq 0$, a pattern occurrence is found
  If $I \neq [0, 0]$, $Prefix \leftarrow Q_1(Prefix, I)$. Else, $Prefix \leftarrow P_x$

To answer query Q_3, we need to construct the suffix trie of P, denoted by ST_P. Note that there are m nonempty suffixes of P and the number of nodes in ST_P is $O(m^2)$. Moreover, there is a unique node on ST_P which represents a specific substring of P (even if the substring appears multiple times in P). It is clear that query $Q_3(S_1, \alpha)$ can be easily answered by tracing ST_P.

One can reduce the space complexity of ST_P as follows. A node on ST_P is said to be explicit if and only if (iff) either it represents a suffix of P or it has more than one child node. The nodes that are not explicit are said to be implicit. One can construct the compacted ST_P which contains only explicit nodes of the uncompacted ST_P by eliminating all implicit nodes in between two explicit nodes. The space complexity can be reduced because the number of explicit nodes on the uncompacted ST_P is $O(m)$ [2].

Queries Q_1 and Q_2 can be answered in constant time during text scanning if two tables, each has $O(m^2)$ entries, are constructed in advance [2]. Each entry in the two tables requires $O(\log_2 m)$ bits. Obviously, when m is large, these two tables require significant amount of memory. In Section 4, we present an efficient realization which requires only $O(m)$ bitmaps, each has m bits, to answer queries Q_1 and Q_2. Then, we generalize the ABF algorithm to find all pattern occurrences in Section 5.

4. Bitmap-Based Realization

Let us consider the implementation of query Q_2 first. Given a pattern $P = p_1 p_2 p_3 \cdots p_m$ of length m, we need two sets of bitmaps where each bitmap has m bits. The first set, called prefix bitmaps, consists of m bitmaps that correspond to the m possible prefix numbers $0, 1, 2, \ldots, m-1$. Let $A_i = a_i^1 a_i^2 \cdots a_i^m$ denote the ith prefix bitmap which corresponds to prefix number $i - 1$. We assign $a_i^k = 1$ iff $k \leq i$ and $p_{i-k+1} \cdots p_{i-1}$ is a nonempty prefix of P, that is, $p_{i-k+1} \cdots p_{i-1} = p_1 \cdots p_{k-1}$. Note that $p_{i-k+1} \cdots p_{i-1}$ represents a null string if $k = 1$. Clearly, with the assignment, we have $a_i^1 = 0$ for all i, $1 \leq i \leq m$, $a_i^i = 1$ if $1 < i \leq m$, and $a_i^j = 0$ if $j > i$.

The second set of bitmaps, called suffix bitmaps, consists of $m - 1$ bitmaps which correspond to the $m - 1$ possible suffix numbers $1, 2, \ldots, m-1$. Again, the size of each suffix bitmap is m bits. Let $B_i = b_i^1 b_i^2 \cdots b_i^m$ be the ith suffix bitmap which corresponds to suffix number i. Assign $b_i^k = 1$, iff $k \geq m - i + 1$ and $p_{m-i+1} \cdots p_{2m-i-k+1}$ is a nonempty suffix of P, that is, $p_{m-i+1} \cdots p_{2m-i-k+1} = p_k \cdots p_m$. In other words, $b_i^k = 1$ iff the length-$(m - k + 1)$ prefix of

Copyright © 2008 John Wiley & Sons, Ltd.

DOI:10.1002/sec

$p_{m-i+1} \cdots p_m$ is a nonempty suffix of P. Similarly, with the assignment, we have $b_i^{m-i+1} = 1$ and $b_i^j = 0$ if $j < m - i + 1$.

We now show that query $Q_2(P_x, S_x)$ can be answered with the two sets of bitmaps. Let $P_x = i - 1$ and $S_x = k$. In other words, we have $S_1 = p_1 \cdots p_{i-1}$, $S_2 = p_{m-k+1} \cdots p_m$, and $S_1S_2 = p_1 \cdots p_{i-1}p_{m-k+1} \cdots p_m$. Note that $S_1 = p_1 \cdots p_{i-1}$ represents a null string if $i = 1$. To answer query Q_2, we first perform the bitwise AND operation of A_i and B_k. Let $R = r_1r_2 \cdots r_m$ denote the result, that is, $R = A_i \otimes B_k$, where $\otimes$ represents the bitwise AND operation. If $i > 1$ and there is a cross-boundary pattern occurrence starting at the jth position of S_1, then it must hold that $p_j \cdots p_{i-1}$ is a prefix of P and $p_{m-k+1} \cdots p_{2m-k-i+j}$ is a suffix of P. Since $p_j \cdots p_{i-1}$ is a prefix of P, we have $a_i^{i-j+1} = 1$. Similarly, $p_{m-k+1} \cdots p_{2m-k-i+j}$ is a suffix of P implies $b_k^{i-j+1} = 1$. Consequently, the pattern occurrence can be detected because it holds that $r_{i-j+1} = 1$. To determine the first pattern occurrence, we need only identify the rightmost 1 of R. Assume that the rightmost 1 of R occurs in the lth position, that is, $r_l = 1$ and $r_i = 0$ for $l + 1 \le i \le m$, then the first pattern occurrence is found starting at the $(|S_1| - l + 2)$th position of S_1. There is no pattern occurrence crossing the boundary of S_1 and S_2 if $r_x = 0$ for all x, $1 \le x \le m$. In case that $i = 1$, that is, S_1 is a null string, we have $a_i^x = 0$ for all x, $1 \le x \le m$, which implies $r_x = 0$ for all x, $1 \le x \le m$. Note that the implementation can actually find all cross-boundary pattern occurrences. This function will be used in the generalization to find all pattern occurrences presented in the next section.

Let us consider the implementation of query Q_1. A third set of m-bit bitmaps are required. For convenience, we number the nonempty suffixes of P so that suffix k is of length k, $1 \le k \le m$. We need a bitmap to be associated with each node on the compacted ST_P. Consider the bitmap $C_N = c_N^1 c_N^2 \cdots c_N^m$ associated with a particular node N. Assign $c_N^i = 0$ for all i, $1 \le i \le m$, if node N is the root node. The bitmap associated with the root node is for the internal range [0, 0]. Assume that node N is not the root node. It is clear that node N represents a unique nonempty substring of P. Assign $c_N^{m-k+1} = 1$, iff node N represents suffix k or the node which represents suffix k is a descendent node of N. Note that the above assignment results in $c_N^{m-k+1} = 1$ iff the string represented by node N is a nonempty prefix of suffix k.

With the prefix bitmaps and the bitmaps associated with the nodes on the compacted ST_P, one can now answer query $Q_1(P_x, I)$. Let M be the node on the LZW trie T_S which represents string S_2 with internal range I. Also, let N be the node on the compacted ST_P which either represents S_2 or the string it represents is the shortest string represented by any node on the compacted ST_P which contains S_2 as a prefix. Node M contains a pointer which points to the bitmap associated with node N. To answer query $Q_1(P_x, I)$, we perform the bitwise AND operation of the prefix bitmap corresponding to prefix number P_x and the bitmap pointed to by the pointer stored in node M. Let $R = r_1r_2 \cdots r_m$ denote the result of the bitwise AND operation. If $r_i = 0$ for all i, $1 \le i \le m$, then $Q_1(P_x, I)$ returns the prefix number of node M. Assume that $r_i = 1$ for at least one i. The answer of $Q_1(P_x, I)$ equals $k - 1 + \mathrm{Dep}(M)$ if $r_k = 1$ and $r_i = 0$, $k + 1 \le i \le m$, where Dep(M), the depth of node M, denotes the length of the chunk represented by node M.

The correctness of the above implementation for query $Q_1(P_x, I)$ can be proved as follows. Assume that $P_x = i - 1$ so that $S_1 = p_1 \cdots p_{i-1}$. If $i > 1$ and the longest pattern prefix that is a suffix of S_1S_2 starts at the jth position of S_1, then it holds that $p_j \cdots p_{i-1}$ is a prefix of P and suffix $m - i + j$ contains S_2 as a prefix. As a result, we have $a_i^{i-j+1} = 1$ and $c_N^{i-j+1} = 1$ which implies $r_{i-j+1} = 1$. In other words, such a prefix can be detected by the bitwise AND operation. Since we are looking for the longest pattern prefix, the rightmost 1 of R is selected. If the rightmost 1 of R is r_k, the symbol p_{i-k+1} starts the longest pattern prefix whose length is equal to $k - 1 + \mathrm{Dep}(M)$. If $r_i = 0$ for all i, $1 \le i \le m$, then the longest pattern prefix is completely contained in S_2 and its length is equal to the prefix number of node M. Therefore, the above implementation does result in correct answer for query Q_1.

To implement query Q_3, we need the suffix trie ST_P. The answer of Q_3 can be obtained by tracing ST_P and the space complexity of ST_P can be reduced to $O(m)$, as mentioned before.

Below are two examples.

Example 1. Let $P = abcab$. Tables I and II show the prefix bitmaps and the suffix bitmaps of P, respectively. As an example of query $Q_2(P_x, S_x)$, assume that $S_1 = abca$ and $S_2 = bcab$. Consequently, we have $P_x = 4$, $S_x = 4$, and $R = 01001$. For this example, the first pattern occurrence starts at the first position of S_1. In fact, as indicated by the two 1's appeared in R, there are two pattern occurrences in S_1S_2.

Example 2. Let $P = ababc$. Table III shows the prefix bitmaps of P. For ease of description, we use the

Copyright © 2008 John Wiley & Sons, Ltd.

DOI:10.1002/sec

Table I. Prefix bitmaps.

P_x	Prefix	Bitmap #	Bitmap
0	*NULL*	1	00000
1	*a*	2	01000
2	*ab*	3	00100
3	*abc*	4	00010
4	*abca*	5	01001

Table II. Suffix bitmaps.

S_x	Suffix	Bitmap #	Bitmap
1	*b*	1	00001
2	*ab*	2	00010
3	*cab*	3	00100
4	*bcab*	4	01001

uncompacted suffix trie ST_P of P as illustrated in Figure 1. The bitmaps associated with the explicit nodes of ST_P are given in Table IV. As an example of query $Q_1(P_x, I)$, assume that $S_1 = abab$ and $S_2 = ab$. Consequently, we have $P_x = 4$, and $I = [1, 2]$ (or [3, 4]). In our implementation, $I = [1, 2]$ (or [3, 4]) is represented by node 2 of ST_P. Since $P_x = 4$ corresponds to the prefix bitmap 00101 and the bitmap associated with node 2 of ST_P is 10100, we have $R = 00100$. As a result, the answer of query Q_1(4, [1, 2]) (or Q_1(4, [3, 4])) is $3 - 1 + |S_2| = 2 + 2 = 4$. As another example, if $S_1 = ab$ and $S_2 = bab$, then we have $P_x = 2$ and $I = [2, 4]$ (the bitmap to be used is the one associated with node 9 of ST_P). Therefore, $R = 00100 \otimes 01000 = 00000$. In this case, the answer of query Q_1(2, [2, 4]) is 2, that is, the prefix number of *bab*.

Let us consider now examples of query $Q_3(S_1, \alpha)$. Assume that $S_1 = ab$ which is represented by node 2 of ST_P. If $\alpha = b$, then we have $Q_3(S_1, \alpha) = [0, 0]$ because there is no transition from node 2 to any node with label *b*. However, if $\alpha = c$, then we have $Q_3(S_1, \alpha) = [3, 5]$ which is represented by node 10 of ST_P.

Table III. Prefix bitmaps.

P_x	Prefix	Bitmap #	Bitmap
0	*NULL*	1	00000
1	*a*	2	01000
2	*ab*	3	00100
3	*aba*	4	01010
4	*abab*	5	00101

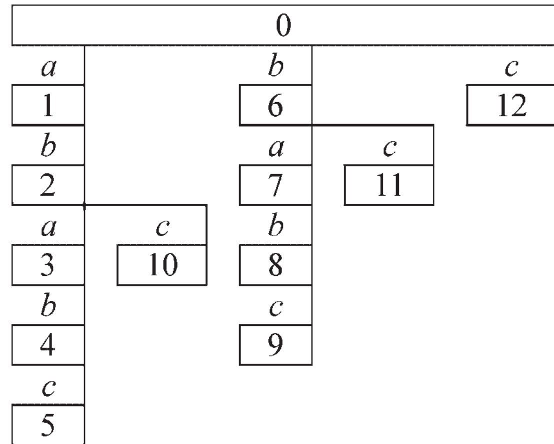

Fig. 1. The uncompacted suffix trie ST_p of $P = ababc$.

Table IV. Bitmaps associated with explicit nodes.

Explicit node	Bitmap
0	00000
2	10100
5	10000
6	01010
9	01000
10	00100
11	00010
12	00001

5. Generalization to All Pattern Occurrences

The pattern occurrence checking in the ABF algorithm is only performed cross two consecutive data blocks to report the first occurrence. To generalize the ABF algorithm to find all pattern occurrences, we need to consider all pattern occurrences cross two consecutive data blocks and those inside a data block as well. Our implementation presented in Section 4 allows detection of all pattern occurrences cross two consecutive data blocks. Therefore, the remaining work is to detect all pattern occurrences inside a data block. The generalization is designed to also report the absolute positions of pattern occurrences.

To detect all pattern occurrences inside a data block, we add two fields, called pattern inside flag (PIF) and pattern inside pointer (PIP), to every node on the LZW trie T_S. The PIF flag is an indication of existence of pattern occurrences inside a node's chunk and the PIP pointer is used for backtracking to find the positions of all pattern occurrences inside the chunk. For the root node, its PIF is 0 and its PIP pointer points

Copyright © 2008 John Wiley & Sons, Ltd.

DOI:10.1002/sec

to the node itself. Assume that a new node M is to be added as a child of node N. The PIP pointer of M inherits the PIP value of N if N is not a final node, that is, a node whose chuck ends with the complete pattern P. To identify final nodes, we let the prefix number of a final node equal m. In case N is a final node, the PIP pointer of M points to N. Similarly, the PIF of M inherits that of N unless the PIF of N is 0 and M is a final node. In this case, we set the PIF of M to 1. It is not hard to see that, with these additional fields, one can trace back the LZW trie to find all pattern occurrences inside a chuck. The trace-back ends once a node with PIP pointer points to the root node, that is, PIP = 0, is reached.

Note that, since we allow the prefix number of a node to be equal to m, we need to add an additional prefix bitmap corresponding to prefix number m. The contents of the bitmap are assigned with the same algorithm described in Section 4. It is clear that the value of the variable *Prefix* may equal m too. However, it does not cause any problem because the bitmap corresponding to prefix number m is the same as the bitmap corresponding to prefix number k, such that $p_{m-k+1} \cdots p_m$ is the longest suffix of P which is also a proper prefix of P, that is, a prefix which is not P itself. Note that $p_{m-k+1} \cdots p_m$ represents a null string if $k = 0$.

For convenience, we also allow the suffix number of a node to be equal to m. As a consequence, another bitmap corresponding to suffix number m is added to the set of suffix bitmaps. Again, the contents of the added suffix bitmap are assigned according to the algorithm described in Section 4 and the additional suffix bitmap does not cause any problem because $a_i^1 = 0$ for all i, $1 \leq i \leq m + 1$.

To report the absolute positions of pattern occurrences, we can rely on the depth fields of nodes on the LZW trie T_S and a global variable *COUNT* which stores the number of bytes in text S that have been scanned. Computation of the depth field is simple. The depth of the root node is 0. When node M is added as a child of node N, the depth of M equals that of N plus one. With the depth fields, we know the position of a node inside a chuck, which, together with the global variable *COUNT*, can be used to determine the absolute positions of pattern occurrences. The overall generalized algorithm is described below.

A. Preprocessing:
The prefix bitmaps and the suffix bitmaps are computed. Also, the compacted suffix trie ST_P of pattern P with the associated bitmaps are determined.
B. Compressed text scanning:

Initialize: $Prefix \leftarrow 0$, $COUNT \leftarrow 0$
for $l = 1$ to n do
(Let P_x, S_x, I, F, and D denote node $S \cdot Z[l]$'s prefix number, suffix number, internal range, PIF, and depth, respectively.)

1. LZW trie construction:
   Add a new node numbered $l + q$ to T_S as a child of $S \cdot Z[l]$ and compute its prefix number, suffix number, internal range, the first symbol, label, depth, PIF, and PIP.
2. Pattern search:
   If $S_x \neq 0$,
   Check cross-boundary occurrences with the bitwise AND operation for query $Q_2(Prefix, S_x)$.
   Let $R = r_1 r_2 \cdots r_m$ be the result of the bitwise AND operation.
   for $k = 1$ to m do
       If $r_k = 1$, report the position: $COUNT - k + 2$
   If $F = 1$ and $P_x = m$, report the position: $COUNT + D - m + 1$
   $N \leftarrow S \cdot Z[l]$'s PIP
   While $N \neq 0$
       Report the position: $COUNT + \mathrm{Dep}(N) - m + 1$
       $N \leftarrow N$'s PIP
   $Prefix \leftarrow Q_1(Prefix, I)$ // Note that the answer of $Q_1(Prefix, I)$ is P_x, if the result of bitwise AND operation for $Q_1(Prefix, I)$ is all-zero
   $COUNT \leftarrow COUNT + D$

Note that the algorithm can be applied in a communication network environment where data arrive in packets. It can take the reordered packets and scan their payloads on-the-fly. The only modification is that intermediate states should be saved after a packet is processed and fetched to continue the scanning process when a new packet is received. The intermediate states include *Prefix*, *COUNT*, and the partial code that has not been processed if a code is separated and contained in consecutive packets.

6. Related Works

A different bitmap-based implementation for LZW compressed pattern matching was proposed in Reference [8]. For convenience, we call it the Kida scheme. Two functions, that is, F and Output, were defined for the scheme. The F function defines state transitions from a set of active states with an input chunk and the

Copyright © 2008 John Wiley & Sons, Ltd.

DOI:10.1002/sec

Output function emits matching results associated with state transitions. Its detailed operation can be found in Reference [8].

To implement the F function, we need an m-bit bitmap $\hat{M}(u)$ for each LZW node, where $\hat{M}(u)$ denotes the bitmap for the node representing chunk $u = u_1 \cdots u_{|u|}$ such that the ith bit of $\hat{M}(u)$ is a 1 iff $p_1 \cdots p_i = u_{|u|-i+1} \cdots u_{|u|}$ or $p_{i-|u|+1} \cdots p_i = u_1 \cdots u_{|u|}$. To implement the Output function, we need an m-bit bitmap $U(u)$ and two values $lps(u)$ and $prev(u)$ for each LZW node. $U(u)$ denotes the bitmap for the node representing chunk u such that the ith bit of $U(u)$ is a 1 iff $1 \leq i \leq \min\{|u|, m-1\}$ and the mth bit of $\hat{M}(u[1:i])$ is a 1. $lps(u)$ denotes the longest prefix of u that is a pattern suffix and $prev(u)$ denotes the longest proper prefix of u whose suffix is the pattern P.

Another scheme which can find all pattern occurrences and be implemented with bitmaps was presented in Reference [5]. For convenience, we call it the Navarro–Raffinot (NR) scheme. The NR scheme is a general technique to perform pattern matching in the text presented as a sequence of chunks which either have just one symbol or are formed by concatenating previously seen chunks. The chunks are processed one by one. A description, denoted by $D(B) = (L, O, Su, Pr, M)$, is computed for each new chunk B, where

- $L = |B| =$ the length of B in symbols
- $O = \mathrm{Offs}(B) =$ the length in symbols of the text that had been processed when B appeared
- $Su = \mathrm{Suff}(B) = \{|x|, P = xBy\} \cup \{|x|, |x| > 0 \wedge |z| > 0 \wedge P = xz \wedge B = zy\}$
- $Pr = \mathrm{Pref}(B) = \{|xB|, P = xBy \wedge |y| > 0\} \cup \{|z|, |z| > 0 \wedge |y| > 0 \wedge P = zy \wedge B = xz\}$
- $M = \mathrm{Matches}(B) = \{|x|, B = xPy\}$

The NR scheme can be realized with two sets of bitmaps, Pref(B) and Suff(B), for every chunk B. The length of every bitmap for Pref(B) and Suff(B) is equal to m, the pattern length. When applied to pattern search in LZW compressed sequences, the numbers of bitmaps for Pref(B) and Suff(B) are equal to the number of nodes on the LZW trie, just like the Kida scheme. The matches inside each chunk B, that is, Matches(B), can be represented by either bitmaps or arrays of numbers. We assume that Matches(B) are represented by arrays of numbers. The reason to represent Matches(B) as an array of numbers is that $O(\log_2 b)$ bits take less space than $O(b)$ bits, where b is the maximum chunk length.

Once the description of the new chunk is computed, it is used to update the state of the search. The state of the search contains the matches that have already occurred and the potential matches in progress, that is,

- $\mathrm{Res}(S') = \{|x|, S' = xPy\}$
- $\mathrm{Active}(S') = \{|x|, |x| > 0 \wedge |y| > 0 \wedge P = xy \wedge S' = zx\}$

where S' denotes the text already processed at any moment of the search. When the search process is over, it holds that $S' = S$, the original text. Hence, when the text processing is complete, Res(S) is the answer. The initial state of the search is $\mathrm{Res}(\varepsilon) = \mathrm{Active}(\varepsilon) = \emptyset$, and $S' = \varepsilon$, where ε denotes the empty string.

Both the Kida scheme and the NR scheme are variations of the ABF algorithm and use the technique of bit-parallelism for realization. The two related works are compared with ours in the following section.

7. Experimental Results

In this section, we compare the space requirement and the throughput performance of our proposed generalized scheme, the NR scheme, the Kida scheme, and the one that performs decompression followed by pattern searching with the Knuth–Morris–Pratt (KMP) algorithm [9].

Consider the space requirement first. In our proposed generalized scheme, the number of bitmaps, including prefix bitmaps, suffix bitmaps, and the bitmaps associated with the nodes on the compacted suffix trie ST_P is $O(m)$. Each bitmap has m bits. The number of nodes on the compacted suffix trie ST_P is $O(m)$. Each node can be identified by a number of size $O(\log_2 m)$ bits. The LZW trie T_S takes space $O(t)$, where t is the number of nodes on T_S. The prefix number, the suffix number, and the internal ranges stored in every node of T_S are replaced by three pointers, each of size $O(\log_2 m)$ bits, which point to the appropriate bitmaps. The node number and the PIP pointer of each LZW node take space $O(\log_2 t)$ bits. The depth of each node takes space $O(\log_2 b)$ bits, where the maximum chunk length b is equivalent to the maximum node depth on T_S. Therefore, the space complexity of our generalized algorithm is $O(m^2 + t\log_2 m + t\log_2 t + t\log_2 b + m\log_2 m)$ bits.

In the Kida scheme, each LZW node carries two m-bit bitmaps, $\hat{M}(u)$ and $U(u)$. Therefore, the total

Copyright © 2008 John Wiley & Sons, Ltd.

DOI:10.1002/sec

number of bitmaps required is $O(t)$ and the bitmaps take space $O(tm)$, which increases proportional to the size of the LZW trie. In addition, each LZW node carries $lps(u)$, $prev(u)$ and a node number, and the suffix trie ST_P is required for computing $lps(u)$. Similarly, we can use the compacted version of ST_P, which takes space $O(m\log_2 m)$. Each $lps(u)$ takes space $O(\log_2 m)$ bits; each $prev(u)$ and each node number take space $O(\log_2 t)$ bits. In order to compute the matching positions in the uncompressed text, the depth of each LZW node needs to be stored, which takes space $O(\log_2 b)$ as our proposed generalized scheme. In summary, the space complexity of the Kida scheme is $O(tm + t\log_2 m + t\log_2 t + t\log_2 b + m\log_2 m)$ bits. The term $O(tm)$ indicates that the space requirement increases significantly as m increases if t is large. In a system where each character is represented by 8 bits, t is initially 256 before compression starts. After compression, t is often much larger than 256 for a large file.

In the NR scheme, the number of elements stored in Res(S) is $O(r)$ and each element requires $O(\log_2 u)$ bits, where r is the number of pattern occurrences in the text. Each LZW node carries a description. Thus, it requires $O(t)$ descriptions and each of them carries an identifier of size $O(\log_2 t)$ bits. Besides, each of the $O(t)$ description contains five elements, L, O, Su, Pr, and M. The element L is equivalent to the depth field in our generalized scheme and requires $O(\log_2 b)$ bits. Assume that the number of pattern occurrences inside an LZW chunk is upper bounded by a constant so that the size of the element M is $O(\log_2 b)$ bits. Note that there are $O(t)$ bitmaps of Su and $O(t)$ bitmaps of Pr, each of the bitmaps has m bits. The space requirement of these two sets of bitmaps is $O(tm)$, which increases proportional to t as the Kida scheme. Another significant difference between the NR scheme and our generalized scheme is that r affects the space requirement of the NR scheme, but not ours. This effect will be studied later. Since the element O in the description is not necessary for every node, we assume that it is omitted and instead a global counter *COUNT* is adopted in comparison. In summary, the space complexity of the NR scheme is $O(tm + t\log_2 t + t\log_2 b + r\log_2 u)$ bits.

In the following experiments, all variables and constants are taken into account to compare the space requirement in practice, unless special test case is stated. In the first test case, we ignore the space requirement of the NR scheme caused by r, that is, we intentionally let $r = 0$. The text used in test case 1 is dfrgntfs.exe (an executable file in Windows) whose uncompressed size is 104,960 bytes. Comparison of the space requirements of the three schemes for test case 1 is shown in Figure 2. It can be seen that in comparison with the NR scheme, our generalized scheme requires less storage if the pattern length is longer than 25; in comparison with the Kida scheme, it requires less storage if the pattern length is longer than 10.

In the following test cases, we take the influence of r into account to compare the space requirement of the NR scheme and our proposed scheme. We use case2.txt (a randomly generated text file with 3,000 patterns inserted) of uncompressed size 296,126 bytes and case3.txt (another randomly generated text file with 18,000 patterns inserted) of uncompressed size 1,705,714 bytes as the texts in test cases 2 and 3, respectively. Figures 3 and 4 show, respectively, the space requirements under these two test cases for different pattern lengths. As shown in Figures 2–4,

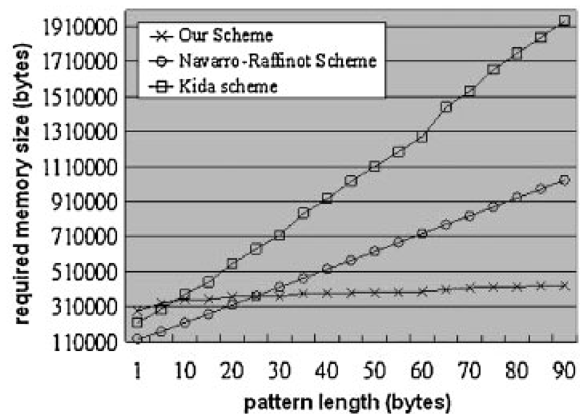

Fig. 2. Comparison of space requirements for test case 1.

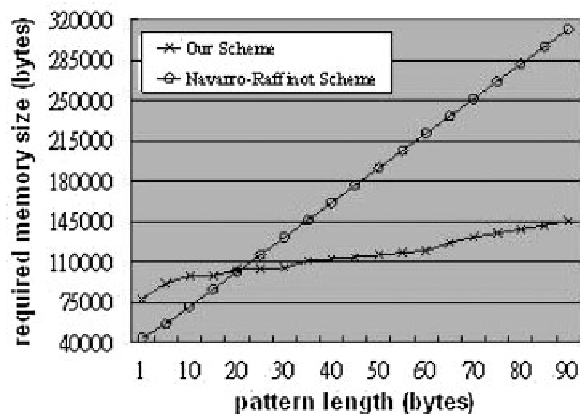

Fig. 3. Comparison of space requirements for test case 2.

Copyright © 2008 John Wiley & Sons, Ltd.

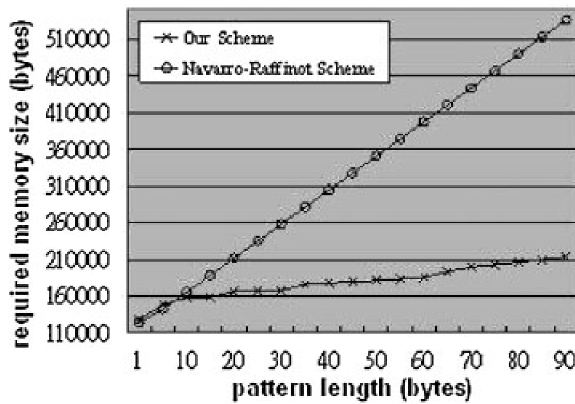

Fig. 4. Comparison of space requirements for test case 3.

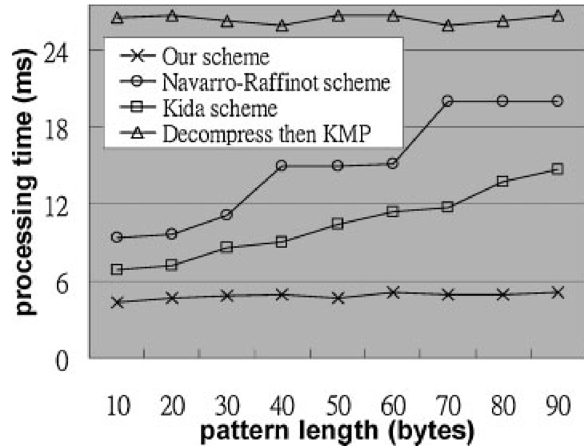

Fig. 6. Performance comparison for test case 4.

our proposed scheme is preferable for applications with moderate and long patterns, such as ClamAV signatures [10].

In Figure 5, we show the space requirements for pattern length $m = 25$ with various numbers of patterns inserted in dfrgntfs.exe. The uncompressed size of the modified dfrgntfs.exe is 249,860 bytes. As one can see, the space requirement of the NR scheme increases as r increases, whereas the space requirement of our proposed scheme is insensitive to the value of r.

Next, we consider the throughput performance. We implemented each scheme in C++ and carried out the experiments on a PC with an Intel Pentium 4 CPU operated at 2.80 GHz with 512 MB of RAM running Microsoft Windows XP operating system. In the fourth test case, we use dosx.exe (an executable file in Windows) as our text which contains no pattern at all. The uncompressed size of dosx.exe is 53,856 bytes. In the fifth test case, we insert various numbers of patterns with $m = 4$ in dosx.exe at randomly selected positions. The experimental results of test cases 4 and 5 are shown in Figures 6 and 7, respectively. As one can see, the processing time of our scheme is the least in test case 4. The reason is that in our scheme, the prefix bitmap for each LZW node is a bitmap in the set of prefix bitmaps that can be computed in the preprocessing stage, and so are the suffix bitmap and the bitmap associated with the node on the compacted suffix trie ST_P. Therefore, the bitmaps for each LZW node are precomputed in our scheme, whereas they are computed in the text scanning stage in both the Kida and the NR schemes. Figure 7 shows that, for test case 5, the performance of our scheme is only slightly better than those of the Kida and the NR schemes. The reason is that the time

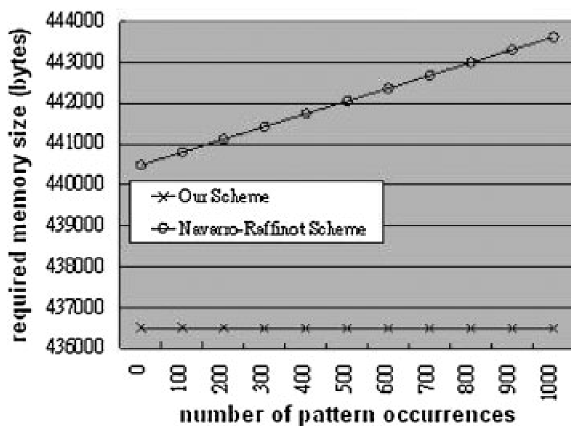

Fig. 5. Comparison of space requirements for different number of pattern occurrences (m is fixed to be 25).

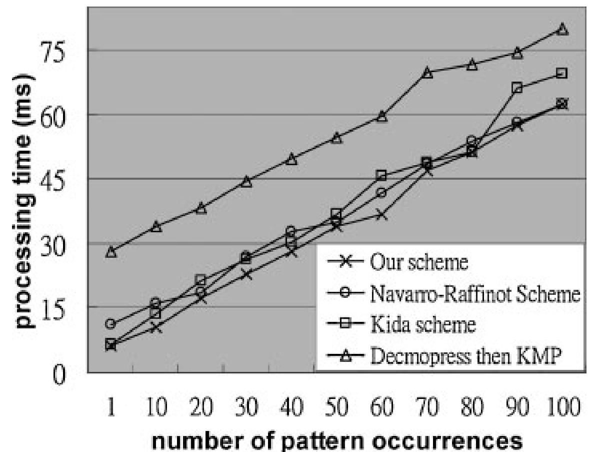

Fig. 7. Performance comparison for test case 5.

Copyright © 2008 John Wiley & Sons, Ltd.

DOI:10.1002/sec

for reporting search results dominates the performance when there are pattern occurrences in the text. Also, the experiments show that our scheme, the Kida scheme, and the NR scheme have better performance than the decompress-then-search algorithm in both test cases 4 and 5 because the former three schemes do not need to decompress the compressed file.

As our proposed scheme, all the related schemes can be modified for a communication network environment. The processing times of our proposed scheme and the related schemes are influenced by the arrival pattern of packets. If the next packet arrives before the current packet finishes being processed, the performance of each scheme is approximately the same as that presented above. The difference is slight increase of processing time spent in fetching the packets.

8. Conclusion

In this paper, we report our work on compressed pattern matching which can process LZW compressed sequences with no decompression to report all pattern occurrences in the uncompressed sequences. It can be applied to detection of malwares and confidential information leak in LZW compressed files directly. A summary of the highlights of our work is:

- We present an efficient bitmap-based realization of the well-known ''almost optimal'' ABF algorithm.
- Our realization is then generalized to detect all pattern occurrences and report the absolute match positions.
- Precise analysis is presented to compare the space complexity of our proposed scheme with two related CPM schemes, namely, the Kida scheme and the NR scheme. The space complexity of our scheme is $O(m^2 + t\log_2 m + t\log_2 t + t\log_2 b + m\log_2 m)$, whereas that of the Kida and the NR schemes are $O(tm + t\log_2 m + t\log_2 t + t\log_2 b + m\log_2 m)$ and $O(tm + t\log_2 t + t\log_2 b + r\log_2 u)$, respectively.
- Experiments are conducted to compare the space requirements and throughput performance of our proposed generalized scheme with the decompress-then-search scheme, the Kida scheme, and the NR scheme. In throughput performance comparison, our scheme is slightly better than the Kida scheme and the NR scheme when scanning a file with pattern occurrences. For a file with no pattern occurrence, our scheme has the best throughput performance, which is about five times better than that of the decompress-then-search scheme.
- In the space performance comparison, our scheme requires less storage than both the Kida scheme and the NR scheme for patterns of length longer than 25. Therefore, our proposed scheme is preferable for applications with moderate and long patterns, such as ClamAV signatures. Moreover, the space requirement of our proposed scheme is insensitive to the number of pattern occurrences in the text.

An interesting further research topic is to design efficient algorithms to match regular expressions in LZW or other Ziv–Lempel family compressed sequences.

References

1. Welch TA. A technique for high-performance data compression. *IEEE Computer* 1984; **17**(6): 8–19.
2. Amir A, Benson G, Farach M. Let sleeping files lie: pattern matching in Z-compressed files. *Journal of Computer and System Sciences* 1996; **52**: 299–307.
3. Tao T, Mukherjee A. Pattern matching in LZW compressed files. *IEEE Transactions on Computers* 2005; **54**(8): 929–938.
4. Ho MH, Yen HC. A dictionary-based compressed pattern matching algorithm. In *IEEE Proceedings of the 26th Annual International Computer Software and Applications Conference*, Oxford, England, 2002; 873–878.
5. Navarro G, Raffinot M. A general practical approach to pattern matching over Ziv-Lempel compressed text. In *Proceedings of 10th Annual Symposium on Combinatorial Pattern Matching*, Springer-Verlag: London, UK. *Lecture Notes in Computer Science* 1999; **1645**: 14–36.
6. Ziv J, Lempel A. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory* 1977; **23**: 337–343.
7. Ziv J, Lempel A. Compression of individual sequences via variable-rate coding. *IEEE Transactions on Information Theory* 1978; **24**: 530–536.
8. Kida T, Takeda M, Shinohara A, Arikawa S. Shift-And approach to pattern matching in LZW compressed text. In *Proceedings of 10th Annual Symposium on Combinatorial Pattern Matching*, Springer-Verlag: London, UK. *Lecture Notes in Computer Science* 1999; **1645**: 1–13.
9. Knuth DE, Morris JH, Pratt VR. Fast pattern matching in strings. *SIAM Journal on Computing* 1977; **6**(2): 323–350.
10. Clam AntiVirus (ClamAV) website http://www.clamav.net

Copyright © 2008 John Wiley & Sons, Ltd.

DOI:10.1002/sec