Computer Communications 145 (2019) 126–136

Contents lists available at ScienceDirect

Computer Communications

journal homepage: www.elsevier.com/locate/comcom

MultiPLZW: A novel multiple pattern matching search in LZW-compressed data

Monther Aldwairi [a,b,*], Abdulmughni Y. Hamzah [a], Moath Jarrah [a]

[a] Computer and Information Technology Faculty, P.O. Box 3030, Jordan University of Science and Technology, Irbid 22110, Jordan
[b] College of Technological Innovation, P.O. Box 144534, Zayed University, Abu Dhabi, United Arab Emirates

ARTICLE INFO

Keywords:
Aho–Corasick algorithm
Compressed data
LZW compression
Pattern matching
Algorithm complexity

ABSTRACT

Searching encrypted or compressed data provides security and privacy without sacrificing efficiency. It has many applications in cloud storage, bioinformatics, IoT, unmanned aerial vehicles and drones. This paper introduces a novel, simple, and efficient algorithm to locate all occurrences of a set of patterns in LZW-compressed data, in a single pass. The algorithm comprises a preprocessing phase and a subsequent search phase. It uses a modified version of the generalized suffix tree, a lookup table, a mapping table, and a history tree. The proposed algorithm is superior in terms of the time complexity, while maintaining a space complexity of the same order as the best of existing algorithms. The time complexity is $\mathcal{O}(n+m+r)$, which is proportional to the length of the LZW-compressed data, where n is the length of the compressed data, m is the total size of the patterns, and r is the number of pattern occurrences in the compressed data. The space complexity is $\mathcal{O}(m^2+t+r)$, where t is the size of the dictionary table that is used during compression. Experimental results show a significant improvement in search time, approximately twice as fast, compared to decompressing and then searching using Aho–Corasick algorithm. Also, results on various dataset sizes, demonstrate the algorithm's superior scalability, which improves as the size of the dataset increases.

1. Introduction

Managing, processing, and manipulating data efficiently on a large-scale is invaluable in different large scale storage environments. Operations in such environments need these qualities in order to accommodate the huge amount of data and the large number of data queries [1,2]. This usually calls for the data to be stored in a compressed form due to capacity constraints as in a cloud environment, where users pay money based on the data size. Hence, operations need to query the data in such a way as not to require decompression in order to save time and space. This issue proves troublesome in large-scale datasets as decompression costs become more extravagant in terms of both time and space [3].

Nowadays, Lempel–Ziv–Welch (LZW) [4] is one of the most popular and prevalent compression algorithms; mainly due to its simplicity, high compression ratio, and time/space efficiency. LZW is adopted in many applications (e.g. GIF and UNIX) [5,6]. LZW is often used in both image and text compression, as it is a lossless compression algorithm. Thus, being able to perform pattern matching search on LZW-compressed data is something worth pursuing.

In LZW-compression algorithm, a codeword is an index to a dynamic dictionary. This adaptive peculiarity of the LZW compression leads to assigning different codewords for the same sequence of characters according to their positions in the input stream. This makes pattern matching in LZW-compressed data, a rather complicated task [7]. Unlike traditional pattern matching, it requires close attention and precision [8–10].

1.1. Motivation

Different efforts have been made towards similar goal of searching compressed data [11,12]. However, the existing work either targets the first occurrence of a pattern or a set of patterns in LZW-compressed data [13,14], or is not as easy to implement and as efficient as the one proposed here [15,16]. Furthermore, some solutions are not capable of operating directly on compressed data and require modifications to the LZW-compression algorithm itself [17]. These solutions and others are inconvenient and maybe sometimes impossible to realistically carry out because of the wide adoption of LZW-compression in many applications and systems. On the other hand, some existing efforts have provided only theoretical approaches without giving implementation considerations and/or experimental evaluation [18]. Details of the existing works are provided in Section 3.

In this paper, we design and implement a novel multiple patterns search algorithm called MultiPLZW that can find all occurrences of

* Corresponding author at: Computer and Information Technology Faculty, P.O. Box 3030, Jordan University of Science and Technology, Irbid 22110, Jordan.
E-mail addresses: munzer@just.edu.jo, monther.aldwairi@zu.ac.ae (M. Aldwairi).

https://doi.org/10.1016/j.comcom.2019.06.011

Available online 26 June 2019

patterns in LZW-compressed data. The algorithm, in essence, mimics the behavior of the Aho–Corasick algorithm (AC) [19], with the distinction and advantage of being able to be used directly in LZW-compressed data. MultiPLZW utilizes pattern preprocessing phase and builds two data structures. The first one is a tailored finite automaton from the generalized suffix tree of the set of patterns. The second is a table similar to the transition table in AC algorithm. A scanning phase follows the preprocessing phase and uses a set of functions that utilize the results of the preprocessing phase. The scanning phase uses two important data-structures which are: *a lookup table* and *a history tree*. These two data structures are incrementally updated.

MultiPLZW can be applied directly on LZW-compressed data and able to report all occurrences of patterns in a single pass and in a linear time. The time is proportional to the size of the compressed data plus the size of the given patterns. The proposed algorithm has better time and space complexity than all existing counterparts. It has an overall time complexity of $\mathcal{O}(n+m+r)$, where n is the compressed text length, m is the total size of the patterns, and r is the number of patterns occurrences in the compressed text. In addition, it has space complexity of $\mathcal{O}(m^2+t+r)$, where t is the size of the dictionary table.

Another advantage of our algorithm resides in its capability to perform searching in compressed-data without decompressing. The other counterparts algorithms perform decompression and then search using AC algorithm, which is undesirable particularly in cloud environment where large sizes are costly. Herein, we also provide implementation details and practical considerations of our approach. We also evaluate the proposed algorithm theoretically, as well as experimentally. Results show that the proposed algorithm outperforms other approaches in terms of efficiency and scalability [20].

The rest of this paper is organized as follows: Section 2 introduces the reader to the preliminaries and theoretical background. Section 3 presents the related works. Section 4 illustrates the proposed algorithm and analyzes its complexity. Section 5 evaluates the algorithm implementation experimentally. Finally, Section 6 concludes the work.

2. Preliminaries and background

Notations. The following notations will be used throughout the paper. Let Σ denotes an *alphabet* of size equals to $|\Sigma|$. $\alpha \mid \alpha \in \Sigma$ denotes a single *letter* (i.e *character*). $\rho \mid \rho = \alpha_0\alpha_1 \cdots \alpha_\gamma$ denotes a *pattern* of length γ (i.e $\gamma = |\rho|$). Π denotes a set of patterns such that $\Pi = \rho_1, \rho_2, \ldots, \rho_u$, where u is the number of patterns in Π. $m \mid m = \sum_{i=1}^{u} \gamma_i$ denotes the length of all patterns in Π. $\omega \mid \omega \in N$ denotes a *codeword*, and $\tau(\omega)$ denotes a substring that is substituted by ω. $P \mid P = \alpha_0\alpha_1\alpha_v$ denotes the plain data (i.e text or images), and $\Gamma \mid \Gamma = \omega_0\omega_1 \cdots \omega_\kappa$ denotes the equivalent LZW-compressed data. Hence, we can represent that data as $\omega_\sigma = w_{\sigma-1}||\alpha$, $\omega_{\sigma-1} = w_{\sigma-2}||\alpha$, ..., $\omega_1 = \alpha$ where α denotes a character in Σ and $\omega_\kappa \in \Phi_\chi$.

LZW-compression is a well-known and prevalent lossless compression algorithm [21] developed by Abraham Lempel, Jacob Ziv, and Terry Welch [22]. A set of LZW-compressed data, Γ, consists of a sequence of integers called *codewords*, and is represented as $\Gamma = \omega_0\omega_1 \ldots \omega_u$. Each codeword $\omega_i \mid i \geq 0,\ i < u$ represents a coded substring over the alphabet set Σ, which is denoted by $\tau(\omega_i) \mid \tau(\omega_i) = \alpha_0\alpha_1 \ldots \alpha_q$. As a value, a codeword $\omega_i \mid \omega_i \in \Gamma$ is either a pointer to a previous codeword $\omega_j \mid \omega_j \in \Gamma,\ j < i$ or an immediate letter, $\alpha \mid \alpha \in \Sigma$. More specifically, if it holds that $\omega_i < |\Sigma|$, then it is an immediate letter in Σ: $\omega_i = \alpha \mid \alpha \in \Sigma$. However, if it holds that $\omega_i \geq |\Sigma|$, then it is a pointer to the previous codeword $\omega_j \mid j = \omega_i - |\Sigma|$: $\omega_{\omega_i-|\Sigma|}$. In Γ, a $\tau(\omega_i)$ is a concatenation of a substring of the indicated codeword (i.e $\tau(\omega_{(\omega_i-|\Sigma|)})$) and the first letter α_0 in the substring of the next codeword $\omega_{(\omega_i-|\Sigma|)}$ (i.e $\omega_{(\omega_i-|\Sigma|+1)}$). For example: let Σ be a character set in UTF-8 coding of which $|\Sigma| = 256$, let $P =$ "*abcabcabc*" be the plain text, and let $\Gamma =$ "97 98 99 256 258 257" be the corresponding LZW-compressed text, where $\omega_0 = 97$, $\omega_1 = 98$, $\omega_2 = 99$, $\omega_3 = 256$, $\omega_4 = 258$, and $\omega_5 = 257$. Consequently, ω_0, ω_1, and ω_2 represent the immediate ASCII code of the letters a, b, c, respectively; while w_3, w_4, and w_5 are pointers to $\omega_{\omega_3-|\Sigma|} = \omega_{256-256} = \omega_0$, $\omega_{\omega_4-|\Sigma|} = \omega_{258-256} = \omega_2$, and $\omega_{\omega_4-|\Sigma|} = \omega_{257-256} = \omega_1$, respectively. In addition, the following holds correct, $\tau(\omega_3) = \tau(\omega_0)||\alpha_0 \mid \alpha_0 \in \tau(\omega_1) = a||b = ab$, $\tau(\omega_4) = \tau(\omega_2)||\alpha_0 \mid \alpha_0 \in \tau(\omega_3) = c||a = ca$, and $\tau(\omega_3) = \tau(\omega_1)||\alpha_0 \mid \alpha_0 \in \tau(\omega_2) = b||c = bc$.

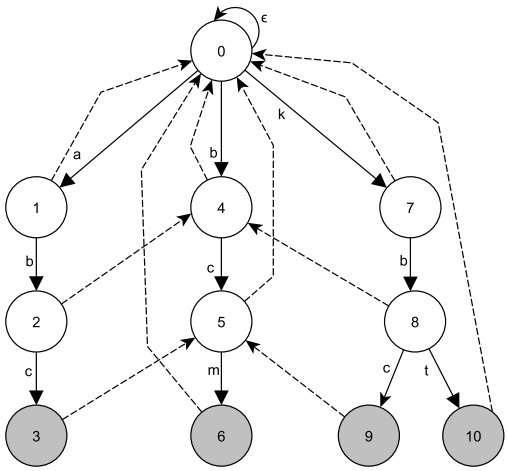

Fig. 1. AC-trie for keys: {"*abc*", "*bcm*", "*kbc*", "*kbt*"}.

Aho–corasick trie. The Aho–Corasick trie (AC-trie) is an automated finite state machine trie that can be used to find all occurrences of multiple patterns in a single pass over a plain text P [23]. To construct an AC-trie, the patterns are organized in a trie, where each node represents the sequence of letters along the path starting from the root of the trie. As with a traditional trie, an edge in AC-trie represents a single letter, and a node may have more than one outgoing edge with a different single-letter for each. However, an AC-trie is augmented with additional directed links between its internal nodes. In an AC-trie, a node is linked to the node that has the longest possible prefix matching with its longest possible suffix. These links implement the *failure* function and are called *failure links*. Failure links are not labeled (unlabeled). They allow for fast node transitions when none of the outgoing edges matches the input sequence. Furthermore, failure links provide a mechanism for revealing overlapping patterns. In AC-trie, each node/state represents a prefix of a certain pattern. For each pattern, there is a unique state that represents it and is called the *final state*. If the next letter in the pattern matches an outgoing edge, then the transition is carried out accordingly using the function *goto* causing a transition through the labeled directed edge. Otherwise, the *failure* function is executed resulting in a transition through the unlabeled *failure link* [24]. A sample of AC-trie for the terms (or keys) "*abc*", "*bcm*", "*kbc*", "*kbt*" over Σ is depicted in Fig. 1.

Solid arrows in Fig. 1 represent the ordinary trie links that are used by the function *goto*, and the dashed arrows represent the *failure links*. The symbol ϵ denotes any character that belongs to the lexicon Σ other than {"*a*", "*b*", "*k*"}.

AC-trie-based pattern matching algorithms start from the *initial state* (i.e. *root*), and proceed according to the functions *goto* and *failure*. When the current node (visited state) is a *final state*, an algorithm reports a pattern match using the function *output*. The construction of AC-trie has a time and space complexity of $\mathcal{O}(m)$[25].

Generalized suffix trie. A generalized suffix trie (GST) is a trie of all suffixes of a set of strings, where each suffix is considered a separate key to the trie [26]. The construction of a GST has a time and space complexity of $\mathcal{O}(m)$ in a serial implementation when using Ukkonen's algorithm [27]. In a parallel implementation, it has a time and space complexity of $\mathcal{O}(log^2 n)$ when using the algorithm in [28]. Besides [27] and [28], the authors of [29] provided a better implementation of the serial construction of GST algorithm.

3. Related work

Since the late '90 s until early 2000 s, the problem of pattern matching in compressed data has been considered by many researchers who introduced several solutions for various compression algorithms. Authors of [30–32] have considered pattern matching in BWT-compressed text. Eilam-Tsore and Vishkin have targeted pattern matching in compression algorithms that are subjected to multi-linear transformations [33]. Amir et al. in [34–36] have addressed pattern matching in the two-dimensional version of run-length algorithm. Farach and Thorup have addressed a single pattern matching for LZ77 compression algorithm [13,37]. Miyazaki et al. in [38] and Karpinski et al. in [39] have considered the pattern matching of strings in terms of straight-line programs. A survey in earlier pattern matching techniques for compressed text and images was provided by Bell et al. in [40]. Recently, many researchers have made contributions towards a similar goal. Tao et al. in [41] have addressed lossy compression. Gagie et al. have introduced a technique for approximate pattern matching in LZ77-compressed texts [42]. Buluş et al. in [43] have introduced a compression algorithm that inherently accommodates pattern matching. Gagie et al. in [44] have addressed pattern matching in self-indexing LZ77-based compression. Beal and Adjeroh have defined pattern matching in parameterized compression algorithms [45–47]. A more recent survey on pattern matching in compressed text and images was introduced by Adjeroh et al. in [48]. Other recent works that target pattern matching in uncompressed data include [49–52], and [53]. The problem of pattern matching becomes more complicated when a compression algorithm is adaptive; wherein the encoding of a substring depends on the location in which it appears. LZW compression is an adaptive algorithm.

Amir, Benson, and Farach [14] made progress in pattern matching for LZW-compressed data in particular. However, they introduced an algorithm to find only the first occurrence of a specific pattern. The time complexity is $\mathcal{O}(n \log \mu + \mu)$ or $\mathcal{O}(n + \mu^2)$, depending on the amount of memory space that is being used; where μ stands for the pattern size and n stands for the LZW-compressed text size. Later, the authors modified on their algorithm in order to report all occurrences of a specific pattern [54]. However, the time complexity was still the same. A more general case concerning Lempel–Ziv compression was presented by Farach and Thorup, who reduced the time complexity that is needed to find the first occurrence of a pattern in the case of LZW to $\mathcal{O}(n \log \frac{n}{2} + \mu)$ [13]. Kida et al. in [15] have improved on Amir's algorithm to have the ability to report all occurrences of multiple patterns—this algorithm is henceforth referred to as Kida's algorithm. Kida's algorithm uses Aho–Corasick automaton, in addition to a GST to preprocess patterns. Hence, Kida's algorithm comprises two phases: the preprocessing phase, then the scanning phase. In the algorithm, Kida et al. showed that the preprocessing phase has a time and space complexity of $\mathcal{O}(m^3)$. Furthermore, they showed that it can be optimized to $\mathcal{O}(m^2)$, where m denotes the total patterns' size. As it was analyzed and investigated by Tao et al. in [16], the scanning phase in Kida's algorithm takes a time complexity of $\mathcal{O}(n + t + r)$. Thus, the overall time complexity of Kida's algorithm is $\mathcal{O}(n + m^3 + t + r)$ or $\mathcal{O}(n + m^2 + t + r)$, depending on whether the preprocessing phase is optimized or not. In addition, it has an overall space complexity of $\mathcal{O}(t + m^3)$ or $\mathcal{O}(t + m^2)$, depending on the preprocessing phase, where t denotes the *LZW-trie* size [55].

Another approach, which is based on Aho–Corasick for multiple patterns matching in LZW-compressed data, was introduced by Tao et al. which we henceforth refer to as Tao's algorithm [16]. Tao et al. have succeeded in improving the preprocessing time to be $\mathcal{O}(m)$. However, the scanning phase complexity has become worse when is compared to Kida's algorithm. The scanning time complexity of Tao's algorithm is $\mathcal{O}(nm + mt + r)$. This is because, for each codeword, it needs to update the corresponding entry in the *LZW-trie*; and for each update on *LZW-trie* entry, it needs to apply its last label to the *AC* automaton y times, where y denotes the number of states in the *AC* automaton whose space complexity equals to $\mathcal{O}(m)$. One more limitation of Tao's algorithm is that, it requires further decompression of codewords whose representation string lengths exceed the length of the minimal pattern of a given set. It works by traversing the Aho–Corasick automaton one symbol at a time to detect the internal occurrences of input patterns. This is to avoid stepping over existing pattern occurrences. Later on, Pawel Gawrychowski in [18], showed that it is possible to find the first occurrence of patterns in LZW-compressed text in a time complexity of $\mathcal{O}(n \log m + m)$ or $\mathcal{O}(n + m^{1+\epsilon})$. As a result, it can be either linear in m or linear in n; and hence, he demonstrated the two extremes of possible trade-offs. Gawrychowski's approach is based on suffix array and binary search tree structures. Although Gawrychowski analyzes his approach from the theoretical perspective, it lacks implementation considerations and experimental proof. Another limitation is that, it only finds the first occurrence of patterns. Our algorithm overcomes the aforementioned limitations and outperforms the exiting algorithms in both, time and space complexity.

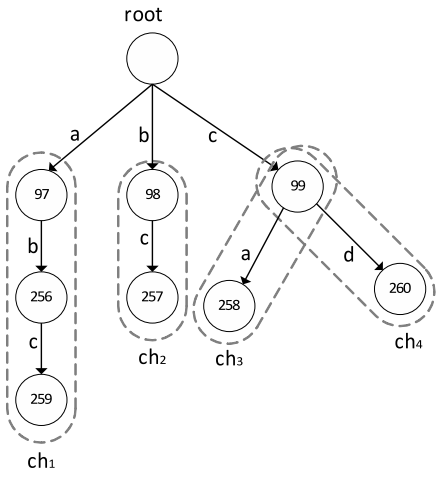

Fig. 2. A sample LZW-trie.

4. Description of the proposed MultiPLZW algorithm

The MultiPLZW algorithm has been designed and implemented based on the following definitions and discussion, which are formally defined below.

Definition 1. A **chain** of codewords is defined to contain a group of codewords that are located on a specific path in the *LZW-trie*. Typically, a specific chain x, is indicated as ch_x, where $x \in N$. For example, in the LZW-trie that is shown in Fig. 2, we have $ch_1 = \{97, 256, 259\}$, $ch_2 = \{98, 257\}$, $ch_3 = \{99, 258\}$ and $ch_4 = \{99, 260\}$.

Definition 2. The chain ch_x to which ω_i belongs is called the containing *chain* of ω_i, or the *chain* of ω_i.

Definition 3. A specific *codeword*, $\omega_i \mid \omega_i \in \Gamma$ is identified by a sequence number. Typically, there are two sequence numbers for ω: the first one is given based on its occurrence sequence in Γ, and the other one is given based on its occurrence sequence in its containing chain (i.e. ch_x). The latter is distinguished from the former by referring to the chain that it belongs to. For example, we say $\omega_{(i \in ch_x)}$ to indicate a codeword whose position sequence is i in the chain ch_x; and ω_i to indicate a codeword whose position sequence is i in Γ.

Definition 4. A **full-pattern prefix** or a **full-pattern suffix** $\rho_q \mid \rho_q \in \Pi$ indicates the set of patterns that amount to the whole ρ_q.

Definition 5. An **internal occurrence** of $\rho_q \mid \rho_q \in \Pi$ at ω_k indicates that ρ_q has occurred in $\tau(\omega_k)$.

Definition 6. Let $\rho_q = chunk_0 chunk_1 \cdots chunk_w$; where $chunk_v = \alpha_0\alpha_1 \cdots \alpha_u$ and α_e is a letter in Σ. A concatenated occurrence of $\rho_q \mid \rho_q \in \Pi$ at ω_k indicates that $chunk_w$ has occurred as a prefix for $\tau(\omega_k)$, while $chunk_0 \cdots chunk_{w-1}$ have occurred as a suffix for the concatenation $\tau(\omega_{k-w})|| \cdots ||\tau(\omega_{k-2})||\tau(\omega_{k-1})$.

Discussions. If α denotes a single letter in Σ and $equals(X, Y)$ denotes $X = Y$. According to LZW algorithm, it holds that

$$\forall(i \in ch_x)\big[equals\big(\tau(\omega_i),\ \tau(\omega_{(i-1)})||\alpha\big)\big].$$

In addition, let $occurs(X, Y)$ denotes that pattern X occurs in codeword Y and $\tau(\omega_{(i\in ch_x)})$ is a subset of $\tau(\omega_{(k\in ch_x)})$, where $k > i$, then it holds that:

$$\forall i\forall(k > i)\big[occurs(\rho_q, \omega_{(i\in ch_x)}) \implies occurs(\rho_q, \omega_{(k\in ch_x)})\big].$$

$\tau(\omega_{k\in ch_x})$ adds just a single letter, α, that does not exist in the $\tau(\omega_{i\in ch_x})$ where $i = k - 1$. Hence, it follows that:

$$\forall k\forall(i = k-1)\Big[\exists^{\leq 1}\rho_q\Big(occurs(\rho_q, \omega_{(k\in ch_x)}) \wedge \neg occurs(\rho_q, \omega_{(i\in ch_x)})\Big)\Big].$$

Furthermore, if $suffix_s(X, Y)$ denotes that a substring X is the suffix s of a substring Y, and LZW-AGST = { $suffix_1(\rho_1), \ldots,\ suffix_\gamma(\rho_1),\ suffix_1(\rho_2), \ldots,\ suffix_\gamma(\rho_2),\ suffix_1(\rho_u), \ldots,\ suffix_\gamma(\rho_u)$ } denotes the set of keys of the LZW-AGST-trie, where $suffix_x(\rho_y)$ denotes the suffix x of the pattern y. Based on the definition of LZW-AGST-trie, the function $goto(\omega_i)$ fails to make a transition if $\tau(\omega_i) \notin$ LZW-AGST. Hence, since $\tau(\omega_{(i\in ch_x)})$ is a subset of $\tau(\omega_{(k\in ch_x)})$, where $k > i$, then it holds that

$$\forall i\forall(k \geq i)\big[\big(goto(\omega_{i\in ch_x}) = fails\big) \iff$$

$$\big(\forall q\forall s\neg suffix_s\big(\tau(\omega_{k\in ch_x}),\ \rho_q\big)\big].$$

Finally, if $\exists i\big(goto(\omega_{i\in ch_x}) = fails\big)$, then a new *concatenated occurrence* for $\rho_q \mid \rho_q \in \Pi$ cannot be attained any more in $\tau(\omega_{k\in ch_x}) \mid k \geq i$. As a result, a transition to the initial state is performed (i.e $lastState \leftarrow root$).

In this paper, we consider the problem of promptly locating all occurrences of multiple patterns, Π, in a given LZW-compressed data, Γ, without having to decompress Γ. The entries of Γ are codewords that dynamically substitute coded strings in the corresponding P. Our proposed algorithm (MultiPLZW) preprocesses the augmented Π by applying the automaton to the generalized suffix tree (GST) of Π, which will be explained in the following subsection—this trie is called *LZW-AGST-trie*. In addition, the preprocessing also involves building another table, which we call *mapTable*. The algorithm then performs the search phase using the functions *goto*, *failure*, *next*, *updateMatchingHistory*, *updateLookupTable*, and *output*. It accomplishes multiple patterns matching in LZW-compressed data in a linear time complexity, proportional to the sum of total patterns length and the size of the LZW-compressed data.

MultiPLZW exploits the structure of Γ and the relationships between codewords to perform multiple pattern matching. It uses two types of transitions over *LZW-AGST-trie*, which are: *tran*1 and *tran*2. *tran*1 performs transitions by looking at individual codewords, while *tran*2 works based on Γ. For a given $\omega_{i\in ch_x}$, *tran*1 represents the sequence

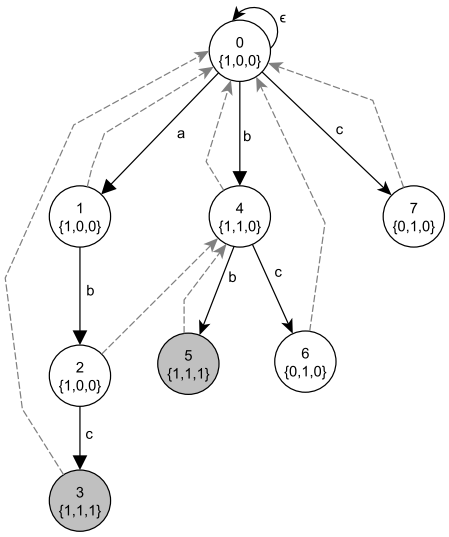

Fig. 3. LZW-AGST-trie for keys: {"abc", "bb"}.

of transitions of substring $\tau(\omega_{i\in ch_x})$ in *LZW-AGST-trie* starting from the *root*. *tran*1 exploits transitions that were performed by previous codewords in the same chain ch_x. Hence, it represents the last transition of the last character in $\tau(\omega_{i\in ch_x})$ starting from the stopping node/state (end-up state) of $\omega_{(i-1)\in ch_x}$. *tran*2 uses *tran*1 to accomplish the continuous traversing over *LZW-AGST-trie* with respect to previous codewords in Γ. This is done in such a way that, for a given ω_i, *tran*2 starts from the end-up node of ω_{i-1}. To clarify, it starts from the previous codeword in the same Γ, but not necessarily the previous codeword in the same ch (refer to Definition 2). *tran*2 is performed in a single step, leveraging the *mapTable* (which is defined in the following subsection) that links each node in *LZW-AGST-trie* to its equivalent node in the full-pattern path in *LZW-AGST-trie*.

4.1. Data structure

4.1.1. The LZW-AGST-trie

Since entries of compressed text data represent a sequence of letters, the suffixes of each pattern are added to the LZW-AGST-trie instead of full patterns. This allows us to utilize the transitions that occur in previous codewords. Traversing can starting from the last state of the last codeword that leads to the current codeword in the same *chain*. Consequently, a single transition through the *LZW-AGST-trie* is equivalent to multiple transitions that represent a sequence of letters of the augmented codeword.

In *LZW-AGST-trie*, an edge represents a single letter, and a path represents a sequence of letters/edges (a substring). A node represents the sequence of letters that produce an incoming path starting from the *root* node. In *LZW-AGST-trie*, a node contains the flags: {*Prefix, Suffix, Final*}; these flags are computed according to the following rules.

- *Rule* 1. $Prefix \leftarrow 1$, if a node represents a prefix for any of the augmented patterns, otherwise $prefix \leftarrow 0$.
- *Rule* 2. $Suffix \leftarrow 1$, if a node represents an end letter for any suffix of the augmented patterns, otherwise $suffix \leftarrow 0$.
- *Rule* 3. $Final \leftarrow 1$, if a node represents a full-pattern, otherwise $Final \leftarrow 0$.

The following example explains how we build the *LZW-AGST-trie*. Fig. 3 shows the trie for the pattern set: {"*abc*","*bb*"}. In the figure, the solid arrows represent the *goto-links* that invoke a function called *goto*. The dashed arrows represent the *failure-links* that invoke a function called *failure*. *goto-links* are similar to the links in *AC-trie*. However, the *failure-links* are those that result from forcing the destination to be a

Table 1
MapTable for LZW-AGST-trie for {"abc", "bb"}.

Node ID	0	1	2	3	4	5	6	7
0	0	1	2	3	4	5	0	0
1	0	0	0	0	2	0	3	0
2	0	0	0	0	0	0	0	3
3	0	0	0	0	0	0	0	0
4	0	0	0	0	5	0	0	0
5	0	0	0	0	0	0	0	0

prefix of a $\rho \mid \rho \in \Pi$. The *LZW-AGST-Trie* differs from *AC-trie* where we require having all suffixes of all patterns. The *failure* function detects any potential overlapping of patterns. The non-prefix sequences do not contain the startup sequence of the corresponding patterns that were derived from. Thus, as shown in Fig. 3, the destination of **node 3** is not **node 6**; because **node 6** does not represent a prefix for any of the patterns: "*abc*" and "*bb*". *LZW-AGST-trie* can be constructed in a time complexity of $\mathcal{O}(md)$ using the naive algorithm shown by Algorithm 5. In addition, it can be constructed in a time complexity of order m using Ukkonen's algorithm.

4.1.2. The map table

mapTable is a data structure that maps nodes of the *LZW-AGST-trie* to their corresponding locations in the paths that produce patterns. The nodes represent non-prefix sequences of patterns. In other words, the table reports the next node when a particular substring of letters is augmented, while the current traverse points to a particular node in the corresponding full-pattern path. At the end, the traversing task reaches to a node, which is a concatenation of substrings. For example, in Fig. 3, if the current traversing points to *node 1* along the path "$a \rightarrow b \rightarrow c$", and the given node is **node 6** (which represents substring "*bc*"), then the next node is the concatenation of "*a*||*bc*" (**node 3**). The mapping table of the example in Fig. 3 is illustrated in Table 1. *mapTable* is used by the function (*next*) during the search phase. *mapTable* rows are indexed by the ID of the last node in the traversing of $tran2$ —assuming it is **node** ϱ. The columns are indexed using the ID of the visited node during the execution of $tran1$–assuming it is **node** φ. An entry of *mapTable* represents the ID of a node that corresponds to a concatenation of $\tau(\varrho)||\tau(\varphi)$. For a set of patterns Π, the construction of *mapTable* has a time complexity of $\mathcal{O}(md)$ when using Algorithm 6; and $\mathcal{O}(m)$ when using Ukkonen's algorithm. The space-complexity of *mapTable* is $\mathcal{O}(n^3)$ when using Algorithm 6, or $\mathcal{O}(n^2)$ using Ukkonen's algorithm.

4.1.3. The lookup table

Let $P =$ "$\ldots bc \ldots bcg \ldots$" denotes a given set of plain data over Σ, and $\Gamma =$ "$\ldots \omega_j \ldots \omega_i \ldots$" denotes the corresponding LZW-compressed data; where $\omega_j = ASCII(b)$, $\omega_i = |\Sigma| + j$, $\tau(\omega_j) =$ "b", and $\tau(\omega_i) =$ "bc". Let ch_x denotes a particular chain of codewords in Γ. Accordingly, ω_j and ω_i have the relationship: $\tau(\omega_i) =$ "b"||"c" $= \tau(\omega_j)||\alpha \mid \alpha$ *is the single letter* 'c'. Based on the discussion in Section 4, $i = j + 1$ exists in the same *chain*, ch_x.

Consequently, the transition path of ω_i follows the same transition path of ω_j with one additional step due to the symbol α. The lookup table (*lookupTable*) in essence is intended to exploit this relationship in order to avoid a multi-step transition path of a particular codeword and replace it with a single-step path. In addition, the lookup table serves as the dictionary table that is used in compression/decompression, chains failure status, and matching history bookmarking.

Similar to the dictionary table, which is used during data compression, the *lookupTable* is created initially during the preprocessing phase. It then gets destroyed and rebuilt many times during the search phase. This happens because it has to have the same size as the dictionary table which is bounded. This permits any codeword in Γ to find a corresponding location in *lookupTable*. The length equals to 2^w, where w is the codeword size. LZW algorithm uses dictionary tables of length between 2^{12} and 2^{16} [56]. The LZW dictionary table length is stored in a special metadata along with the LZW-compressed data. It can be extracted and used to determine the required length of the *lookupTable*. Similar to the LZW-compression dictionary table, *lookupTable* is composed of two parts. The first part contains the first $|\Sigma|$ entries, which are initialized at the beginning of the search phase, as shown by Table 2. The $2^w - |\Sigma|$ entries are updated regularly, according to every entry in Γ during the search phase. In addition, an entry in the *lookupTable* has five essential fields, which are also shown in Table 2. The field *fchar* of an entry $\Sigma + i$ contains the first letter in $\tau(\omega_i)$. The field *lchar* contains the letter that resides after $\tau(\omega_i)$ in Γ. The field *endup-state* contains the end-up state/node in *LZW-AGST-trie* of the transition path of ω_i. The field f indicates whether the transition of an $\omega_v \mid \omega_v \in ch_x$, ω_i, $v \leq i$ has reached over to a failure link. This field has a value of zero in the same chain unless a transition of a codeword results in encountering a failure link, where the value is changed to 1. Such information is useful in order to find whether a codeword can be a potential suffix for a given patterns. Afterwards, only *final-state* nodes are stored in the matching history data structure. The field *anchor* works as a bookmark that points to a node in the history tree. For the *LZW-AGST-trie* shown in Fig. 3, the corresponding entries of codewords ω_j and ω_i in *lookupTable* are the ones that are located in positions $|\Sigma| + j$, and $|\Sigma| + i$, respectively, as depicted by Table 2.

The parameters *lchr* and *endup-state* are used by the functions *goto* and *failure*, where they indicate the transitions that have been made by previous codewords of the same chain, ch_x, that ω_i belongs to. Hence, there is no need to decompress ω_i to follow the transition path from the beginning (i.e *root* in *LZW-AGST-trie*). The final step in a transition path of ω_i is performed, which starts from the end node of the last codeword $\omega_j \mid \omega_j \in ch_x$. *endup-state* and *lchar* of ω_j are needed to perform the final step in the transition path of ω_i. Both of these data are acquired using the current $codeword$ as an index to the entry of the *lookupTable*. In the aforementioned example, to perform the transition for ω_j whose corresponding entry in *lookupTable* is $|\Sigma| + j$, *goto* or *failure* functions use the contents of the *codeword* field ($ASCII(b)$) as an index in order to find *endup-state* and *lchr*. Likewise, to perform the transition for ω_i whose corresponding entry in *lookupTable* is $|\Sigma| + i$, function *goto* or *failure* uses $|\Sigma| + i$ as an index to the *lookupTable*, which corresponds to ω_i.

While searching into Γ, if a *final-state* node or a suffix node in *LZW-AGST-trie* is encountered and the failure status is zero, then function *updateMatchingHistory* uses ω_i in the field *codeword* to lookup the history node. If the parameter f of ω_i equals to zero in the *lookupTable*, then the failure status is zero. This case indicates that:

$\exists \rho_s \mid \tau(\omega_i)$ *is a potential suffix for* $\rho_s \mid \rho_s \in \Pi$. A new child node is added to indicate a matching (or a potential matching).

Function *output* uses the parameter *anchor* of ω_i, as a bookmark that points to the node in *history-tree*. Hence, it only traverses the relevant pattern matching paths.

4.1.4. The history tree

During the search in Γ, a single transition is performed for $\omega_i \mid \in \Pi$ over *LZW-AGST-trie*. This is equivalent to multiple shorter steps, as ω_i encodes multiple letters in P. In order to catch all pattern matching in $\tau(\omega_i)$, the history of pattern matches in previous codewords that belong to the same chain are used to report the matches in ω_i. The history tree is a directed and fully-connected graph in which all paths end-up at a single node (i.e. the initial node or the root), as shown in Fig. 4. The history tree is built incrementally during the search phase, where nodes are added to the *historyTree* when pattern matchings occur. Along the search phase, the *historyTree* is destroyed and re-initialized each time when the *lookupTable* is destroyed and rebuilt. *historyTree* data structure is used to make insertion and searching fast and easy. It allows entries to be arranged into paths according to their chains. Successive codewords in a particular chain are in the same path and in the same sequence. Thus, searching the *historyTree* for a particular codeword traverses only the relevant nodes in the path.

Table 2
Lookup table.

	fchr	*lchr*	*endup*	*flag*	*anchor*
0	*char*(0)	*char*(0)	0 (*root*)	0	0 (*root*)
1	*char*(1)	*char*(1)	0 (*root*)	0	0 (*root*)
⋮	⋮	⋮	⋮	⋮	⋮
ASCII(*b*)	*b*	*b*	0 (*root*)	0	0 (*root*)
⋮	⋮	⋮	⋮	⋮	⋮
$\|\Sigma\|-1$	*char*($\|\Sigma\|-1$)	*char*($\|\Sigma\|-1$)	0 (*root*)	0	0 (*root*)
$\|\Sigma\|$					
⋮	⋮	⋮	⋮	⋮	⋮
$\|\Sigma\|+j$	*b*	*c*	4	0	0 (*root*)
⋮	⋮	⋮	⋮	⋮	⋮
$\|\Sigma\|+i$	*b*	*g*	6	0	0 (*root*)
⋮	⋮	⋮	⋮	⋮	⋮
$t-1=2^w-1$					

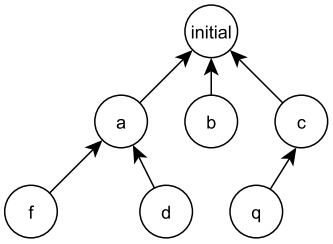

Fig. 4. History-tree.

As mentioned earlier in Section 4, a ω_i in Γ has at most one new *pattern-matching* that does not exist in the set $\{\omega_j \mid j < i,\ \omega_j \in ch_x\}$. Hence, only codewords that have new pattern matches are added to the *historyTree*. In the case, a new pattern matching occurs in $\omega_i \mid \omega_i \in ch_x$, a new node is added to the *historyTree* as a child of the node that corresponds to the latest recorded $\omega_j \mid j < i,\ \omega_j \in ch_x$. The new node of ω_i can be directly accessed using the field *anchor* of *lookupTable*. Afterwards, a reference to the newly added node is stored in the field *anchor* of the lookup table with index $i + |\Sigma|$ (it is the corresponding entry for ω_i in *lookuptable*).

The search in the *historyTree* for ω_i, starts from a node that corresponds to ω_i. Afterwards, the search goes up through the same path until the *root* node of *historyTree*. This results in reporting all pattern matches that occur in ω_i.

4.2. Functions

4.2.1. The goto function

The *goto* function is essential in the automaton machine to transfer control to the next state through an edge with a label that matches a given letter. The time complexity of the *goto* function is $\mathcal{O}(1)$. The entries of the LZW-compressed data, Γ, represent strings of the letters, $\tau(\omega_i) = \alpha_0\alpha_1...\alpha_q$. Multiple transitions are accomplished for a string $\tau(\omega_i)$ in a single step. For ω_i and ω_{i-1}, the following is correct; $goto(\omega_{i\in ch_x}) = goto(\omega_{(i-1)\in ch_x}) \rightarrow goto(\alpha)$, where α is the last letter in $\tau(\omega_{i\in ch_x})$ and both ω's belong to the same chain. The $goto(\omega_{(i-1)\in ch_x})$ and α are always retrieved from the *lookupTable* in a constant time. The function *goto* is invoked for each $\omega_i \mid \omega_i \in \Gamma$ on behalf of the *tran*1, and it returns the proper next state, which is called *newState*. The pseudo code of the function is outlined in Algorithm 1.

Algorithm 1: The *goto* function

```
function GOTO(ω_i)
    newState ← lookupTable[ω_i].state → child[lookupTable[ω_i].lchr]
    return newState
end function
```

4.2.2. The updatelookuptable function

The pseudo code of the function *updateLookupTable* is shown by Algorithm 2. It updates the fields of the corresponding *lookupTable* entry of $\omega_{i\in ch_x}$ for future referencing by $\omega_{(i+1)\in ch_x}$.

Algorithm 2: The *updateLookupTable* function

```
procedure updateLookupTable(k, ω_i, ω_(i+1), newState, anchor, f)
    lookupTable[k].state ← newState
    lookupTable[k].anchor ← anchor
    lookupTable[k].f ← f                                   % f: status flag %
    lookupTable[k].fchr ← lookupTable[ω_i].fchr            % fchr stands for first character %
    if exists(ω_(i+1)) then
        lookupTable[k].lchr ← lookupTable[ω_(i+1)].fchr    % lchr: last character %
    else
        lookupTable[k].lchr ← lookupTable[ω_i].fchr
    endif
end procedure
```

4.2.3. The failure function

The *failure* function is similar to the one in AC algorithm. It is used to perform transitions through the *failure* links when the *goto* function fails to make a transition. Similar to the *goto* function, *failure* uses *lookupTable* to determine the start node of the *tran*1 transition in the *LZW-AGST-trie*.

4.2.4. The next function

The *next* function is used in our MultiPLZW solution to enable the task of *tran*2 transitions. The function advances the transition over paths that compute to full patterns in *LZW-AGST-trie*. The transition works as follows. Let *lstState* denotes the last state that has been encountered by the function *next* at a previous codeword (i.e ω_{i-1}); and *newState* is the state of the function *goto* at the current codeword (i.e ω_i), then *next* performs *tran*2 transition to the node with an ID equals to the intersection of row *lstState.ID* and column *newState.ID* in the *mapTable*. The function *next* is used also by the function *output* in order to report the *concatenated* occurrences of patterns.

The function *next* has two forms that are expressed mathematically by formula (1) and formula (2), where Q is initially set to the value of *lstState*. The function *next* uses formula (1) for $\omega_{k\in ch_x}$, when $\exists\omega_{(i\in ch_x)} \mid i \le k$ takes the value of $goto(\omega_{(i\in ch_x)}) = fails$. For all other cases, the function uses formula (2). The flag f takes the previous entries' values in *mapTable*. For the entry of $mapTable[\omega_{k\in ch_x}]$, if f is *true* in $mapTable[\omega_{(k-1)\in ch_x}]$, or if *goto* at ω_k results in a *failure* (i.e $goto(\omega_{k\in ch_x}) = fails$), then the flag f is set to *true*. This flag indicates whether the function *goto* has resulted in a $failure$ state at a previous *codeword* in the same *chain* or not. In other words, it indicates whether the expression: $\exists(i \le k)\big(goto(\omega_{i\in ch_x}) = fails\big)$ is evaluated to *true* or $false$. Subsequently, it is used to determine whether to use formula (1) or (2) in the *next* function.

$$next(root, newState) = mapTable(root.ID, newState.ID) \tag{1}$$

$$next(Q, newState) = \begin{cases} Z = mapTable(Q.ID, newState.ID), & Z \neq NULL \text{ or } Q = root \\ next(failure(Q), newState), & \text{otherwise} \end{cases} \tag{2}$$

4.2.5. The updateMatchingHistory function

The *updateMatchingHistory* function is depicted in Algorithm 3. The function stores any newly detected pattern occurrence, *internal* or *concatenated* for a pattern ρ_q at $\omega_{i\in ch_x}$, in order to be referenced later by the function *output* at $\{\omega_{k\in ch_x} \mid k \geq i\}$. Thus, it allows for the detection of all patterns that occur in $\tau(\omega_{k\in ch_x})$ without the need to go through the process of decompressing $\omega_{k\in ch_x}$. An entry to the *historyTree* is added in two situations. The first occurs when the function *goto* hits a *final-state* node in the *LZW-AGST-trie*. This indicates an *internal* occurrence of ρ_q at $\omega_{i\in ch_x}$. The second, is when the function *goto* hits a node that represents a *suffix* for $\rho_q \mid \rho_q \in \Pi$ in the *LZW-AGST-trie*; and the entry flag *f*, in the *lookupTable*, which corresponds to $\omega_{i\in ch_x}$, is equal to 0. This indicates that the function *goto* has never returned a failed state over $\{\omega_{j\in ch_x} \mid j \leq i, j \geq 0\}$ (i.e $\exists (j \leq i)\big(goto(\omega_j) \neq fails\big)$).

According to the discussion in Section 4, it holds that:

$\exists\ \rho_q \mid \tau(\omega_i)\ is\ a\ suffix\ for\ \rho_q \mid \rho_q \in \Pi$. This entry is stored in order to check for potential *concatenated* occurrence at $\{\omega_{k\in ch_x} \mid k \geq i\}$. When adding a new entry to the *historyTree*, it is inserted under the node that corresponds to $\omega_{(i-1)\in ch_x}$ (its *ancestor*). The function *updateMatchingHistory* also returns a reference to the *historyTree* entry that corresponds to $\omega_{i\in ch_x}$. It occurs when updating the field *anchor* in the *lookupTable* entry of $\omega_{i\in ch_x}$ (i.e entry $i + |\Sigma|$).

Algorithm 3: The *updateMatchingHistory* function

```
function updateMatchingHistory(k, ω_i, newState, f)
        % newState is the output of goto at ω_i, lastEntry is the last entry to lookupTable at ω_i %
    lookupEntry ← lookupTable[k]
    ancestor ← lookupTable[ω_i].anchor
     if isFinal(newState) ∨ [isSuffix(newState) ∧ (f = 0)]  then
                                  % a new entry in historyTree references by anchor %
    anchor ← AddChild(ancestor, lookupEntry )
     else
    anchor ← ancestor
      endif
    return anchor
end function
```

4.2.6. The output function

The *output* function is depicted by Algorithm 4. It reports all patterns that occur in $\tau(\omega_{i\in ch_x})$ through the inspection of pattern occurrences history of $\{\ \omega_{j\in ch_x} \mid j \leq i, j \geq 0\ \}$. The *output* function starts checking the history of pattern occurrences at node $\omega_{i\in ch_x}$ in the *historyTree*. It traverses the tree upwards until reaching the first level. Hence, the searching, in the history at $\omega_{i\in ch_x}$, takes a time that is proportional to the number of pattern occurrences in $\tau(\omega_{i\in ch_x})$.

Algorithm 4: The *output* function

```
procedure output(lstState, k)
        % lstState: is the state in LZW-AGST-trie that was reached by next in the last iteration%
    V ← lookupTable[k].anchor          % anchor: is a reference to a vertex in history-tree %
  while V ≠ root  do                   % root: is the root of historyTree %
   if isFinal(V.entry)  then
    reportMatching()                   % report pattern matching details %
   elseif isFinal(next(lstState, V.entry))  then
    reportMatching()
   endif
   endif
    V ← V.parent
  endwhile
end procedure
```

4.3. MultiPLZW phases

MultiPLZW has two phases: preprocessing and search. the following subsection details each phase and explains the algorithm used.

4.3.1. The preprocessing phase

During the preprocessing phase, the *LZW-AGST-trie* and its corresponding *mapTable* are constructed based on the given Π. This task can be accomplished either using the two naive algorithms shown in Algorithms 5 and 6, or using Ukkonen's algorithm [27].

In Ukkonen's algorithm, a node in the trie should have two children or more. Hence, the paths that represent full patterns must stay uncompressed to avoid collisions in the *mapTable*. In addition, the suffixes nodes (i.e the nodes that correspond to the most recent letters for each $\rho_i \mid \rho_i \in \Pi$) should be independent.

Algorithm 5: A naive algorithm to create *LZW-AGST-trie*

```
procedure CreateGST(Π)                 % Π: a set of patterns %
    foreach ρ ∈ Π  do                  % ρ: denotes a single pattern in Π %
    tmp1 ← GST.root
     foreach c_i ∈ ρ | i ≥ 0, i < length(ρ)  do
      if tmp1.child[c_j] = NULL  then
    tmp1.child[c_j] ← newNode
      endif
    tmp1 ← tmp1.child[c_i]
    tmp2 ← GST.root
     foreach c_j ∈ ρ | j > i, j < length(ρ)  do
      if tmp2.child[c_j] = NULL  then
    tmp2.child[c_j] ← newNode
      endif
    tmp2 ← tmp2.child[c_j]
     endfor
     endfor
     endfor
end procedure
```

Algorithm 6: A naive algorithm to create *mapTable*

```
procedure CreateMapTable(Π)            % Π: a set of patterns %
    foreach ρ ∈ Π  do                  % ρ: denotes a single pattern in Π %
    tmp1 ← GST.root
     foreach c_i ∈ ρ | i ≥ 0, i < length(ρ)  do
    tmp1 ← tmp1.child[c_i]
    tmp2 ← GST.root
    tmp3 ← tmp1
     foreach c_j ∈ ρ | j > i, j < length(ρ)  do
    tmp2 ← tmp2.child[c_j]
    tmp3 ← tmp3.child[c_j]
    mapTable[tmp1.ID][tmp2.ID] ← tmp3
   endfor
   endfor
   endfor
end procedure
```

4.3.2. The search phase

A pattern matching is said to have occurred in Γ, if substrings for one of the given patterns span one or more successive codewords $(\omega_i, \omega_{i+1}, \omega_{i+2}, ..)$ in Γ. The MultiPLZW searching algorithm is depicted by Algorithm 7. A match of a pattern occurs when the transition hits a final state in the *LZW-AGST-trie*. There are two types of transitions. The first one is implemented using the function *next* in order to guide the transition through full-pattern paths of the *LZW-AGST-trie* (i.e these paths can result in complete patterns). A final state or a set of final states can cause a hit —we refer to this transition as *tran*2. The second type is implemented using the *goto* function. It enables the execution to be performed in a single step—we refer to this transition as *tran*1.

A single entry ω_i in Γ represents a sequence of letters. Also, a single transition is equivalent to having multiple transitions for the functions: *next* and *goto*. This results in the three functions: *goto*, *failure*, and *next*, finding multiple patterns matching in Γ. They operate in time proportional to the size of Γ. Note here that there is no need to decompress Γ, which is one of the objectives of our solution.

Using the sequence $\tau(\omega_i)$, a transition step of the *goto* function leads directly to the last state. However, stepping over intermediate states in *LZW-AGST-trie* may result in missing certain pattern matches, especially if $\tau(\omega_i)$ is long. In order to prevent this, we record a matching history of previous codewords that belong to the same chain of the augmented ω_i. The history record is used to report all pattern matches that occur

in $\tau(\omega_i)$. The *updateMatchingHistory* function makes keeps the history available, updated, and constantly accessible in a time proportional to the number of pattern matches that exist in $\tau(\omega_i)$. Finally, the *output* function is used to report all occurrences.

Algorithm 7: MultiPLZW search algorithm: finds multiple patterns Π in LZW-compressed data Γ

```
procedure searchMultiPLZW(Π, Γ)                    %Π: a set of patterns %
    Initialize − LookupTable()
    Initialize − HistoryTree()
    lstState ← AGST.root
    i ← 0
    n ← |Γ|
  while i < n  do                    % Γ: denotes LZW-compressed data %
    ω_i ← Γ[i]                       % ω_i: denotes codeword i in Γ %
    k ← i + |Σ|
    newState ← goto(ω_i)
    f_1 ← 0
    if newState = NULL  then
    newState ← failure(ω_i)
    f_1 ← 1
    endif
    f ← (lookupTable[ω_i].f ∨ f_1)
    anchor ← updateMatchingHistory(k, ω_i, newState, f)
    updateLookupTable(k, ω_i, ω_(i+1), newState, anchor, f)
    output(lstState, k)
    lstState ← next(lstState, newState)
    i ← i + 1
  endwhile
end procedure
```

4.4. Complexity analysis

This section presents the time and space complexities of all algorithm stages.

4.4.1. Time complexity

The time complexity of the preprocessing phase is explained as follows. Let Π denotes a set of patterns $\rho_1, \rho_2, \ldots, \rho_u$ of size u; μ_i = is the size of $\rho_i \mid \rho_i \in \Pi$; m = is the total size of $\Pi = \sum_{i=1}^{u} \mu_i$; and $f_1 = \sum_{i=1}^{u} {\mu_i}^2$. According to Algorithms 5 and 6, the processing of ρ_i has a time complexity of $\mathcal{O}({\mu_i}^2)$. The time complexity to process the entire Π is $\mathcal{O}\left(\sum_{i=1}^{u} {\mu_i}^2\right) = \mathcal{O}(f_1)$. Let d = denotes the size of the longest $\rho_i \mid \rho_i \in \Pi$, then it follows that $\forall i,\ d \geq \mu_i \mid i \geq 1,\ i \leq u$. In addition, if $f_2 = \sum_{i=1}^{u}(d \times \mu_i)$, then it follows that $f_1 \leq f_2$, and $\mathcal{O}(f_1) = \mathcal{O}(f_2) = \mathcal{O}\left(\sum_{i=1}^{u}(d \times \mu_i)\right) = \mathcal{O}\left(d \times \sum_{i=1}^{u} \mu_i\right) = \mathcal{O}(d \times m)$

The time to construct the *LZW-AGST-trie* and *mapTable* is $\mathcal{O}(md)$ when using the two naive algorithms shown in Algorithms 5 and 6, where d is the depth of *LZW-AGST-trie* and equals to the size of the longest $\rho_i \mid \rho_i \in \Pi$; m is the total size of the patterns. It is worth mentioning here that the *LZW-AGST-trie* and *mapTable* can be constructed in a time complexity of $\mathcal{O}(m)$ when using the Ukkonen's algorithm.

The time complexity of the search phase is explained as follows. Let γ_i denotes the number of patterns that occur in a given ω_i, n denotes the size of the given Γ, and r denotes the number of patterns that occur in $\Gamma = \sum_{i=1}^{n}(\gamma_i)$. For each loop of the MultiPLZW algorithm, the functions: *goto*, *failure*, *next*, *updateLookupTable*, and *updateMatchingHistory* have a time complexity of $\mathcal{O}(c)$, where c is a constant. The function *output* has a time complexity of $\mathcal{O}(\gamma_i)$. This entails that the time complexity for a single loop over a given ω_i is $\mathcal{O}(\gamma_i + c)$. Consequently, the time complexity to search multiple patterns over the entire Γ is $\mathcal{O}\left(\sum_{i=1}^{n}(\gamma_i + c)\right) = \mathcal{O}\left(c \times n + \sum_{i=1}^{n}(\gamma_i)\right) = \mathcal{O}(c \times n + r) = \mathcal{O}(n + r)$.

The overall time complexity of the algorithm is the addition of the preprocessing and search time. Hence, the time complexity of MultiPLZW equals to $\mathcal{O}(md) + \mathcal{O}(n + r) = \mathcal{O}(n + md + r)$. The time is reduced to $\mathcal{O}(n + m + r)$ when using the Ukkonen's algorithm in the preprocessing phase.

Table 3
Time complexity comparison.

Algorithm	The preprocessing complexity	The search complexity	Total complexity
Kida's algorithm	$\mathcal{O}(m^3)$ or $\mathcal{O}(m^2)$	$\mathcal{O}(n + t + r)$	$\mathcal{O}(n + m^3 + t + r)$ or $\mathcal{O}(n + m^2 + t + r)$
Tao's algorithm	$\mathcal{O}(mt)$	$\mathcal{O}(nm + r)$	$\mathcal{O}(nm + mt + r)$
The proposed algorithm (MultiPLZW)	$\mathcal{O}(md)$ or $\mathcal{O}(m)$	$\mathcal{O}(n + r)$	$\mathcal{O}(n + md + r)$ or $\mathcal{O}(n + m + r)$

Table 4
Space complexity comparison.

Algorithm	The preprocessing complexity	The search complexity	Total complexity
Kida's algorithm	$\mathcal{O}(m^3)$ or $\mathcal{O}(m^2)$	$\mathcal{O}(m^2 + t)$	$\mathcal{O}(m^3 + t)$ or $\mathcal{O}(m^2 + t)$
Tao's algorithm	$\mathcal{O}(m)$	$\mathcal{O}(mt)$	$\mathcal{O}(mt)$
The proposed algorithm (MultiPLZW)	$\mathcal{O}(m^2 d)$ or $\mathcal{O}(m^2)$	$\mathcal{O}(t)$	$\mathcal{O}(m^2 d + t)$ or $\mathcal{O}(m^2 + t)$

4.4.2. Space complexity

The following details the space complexity of the MultiPLZW algorithm. Let Π denotes the set of patterns $(\rho_1, \rho_2, \ldots, \rho_u)$, d = denotes the size of the longest $\rho_i \mid \rho_i \in \Pi$, and m denotes the total size of all patterns in Π. The space complexity to build the *LZW-AGST-trie* for the set Π is $\mathcal{O}(md)$, or $\mathcal{O}(m)$ when using the Ukkonen's algorithm.

The *mapTable* depends of the structure of the *LZW-AGST-trie*. The number of nodes in full-pattern paths of *LZW-AGST-trie* is m, and the total size of *LZW-AGST-trie* is md or m when using the Ukkonen's algorithm in the preprocessing phase. Thus, the space complexity of *mapTable* is $\mathcal{O}(m^2 d)$ or $\mathcal{O}(m^2)$ when using the Ukkonen's algorithm. The *lookupTable* has a size of t, where t is the size of the dictionary table that was used during compression. On the other hand, the *historyTree*'s size is proportional to t. Consequently, the total space complexity that is needed by MultiPLZW is $\mathcal{O}(m^2 d + t)$ when using the naive algorithms, or $\mathcal{O}(m^2 + t)$ when using the Ukkonen's algorithm to build the *LZW-AGST-trie*.

5. Comparison and experimental evaluation

In following subsections we present a thorough Time and space complexity analysis, comparison to related work, and the performance of the proposed algorithm in terms of runtime against competing algorithms.

5.1. Time and space complexity comparison

The time and space complexities of our algorithm (MultiPLZW) are compared with the two most relevant algorithms: Kida's and Tao's algorithms. Table 3 shows the time complexity comparison and Table 4 shows the space complexity comparison, where Multi-PLZW is superior in both.

As shown by Table 3, the time complexity of our proposed algorithm (MultiPLZW) is $\mathcal{O}(n + md + r)$ when using the naive algorithms, or $\mathcal{O}(n + m + r)$ when using the Ukkonen's algorithm. The proposed algorithm outperforms other relevant algorithms. In addition, it achieves better preprocessing time complexity. This makes it a better choice for applications that require query-response tasks in storage environment such as the cloud environments where the capacity constraint is critical [57].

The proposed algorithm manages to achieve a significant improvement in time complexity, while maintaining acceptable benchmarks

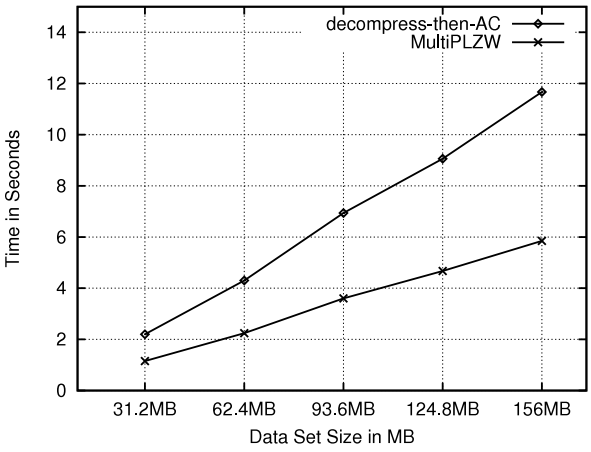

Fig. 5. Comparison of average search time including preprocessing versus dataset size.

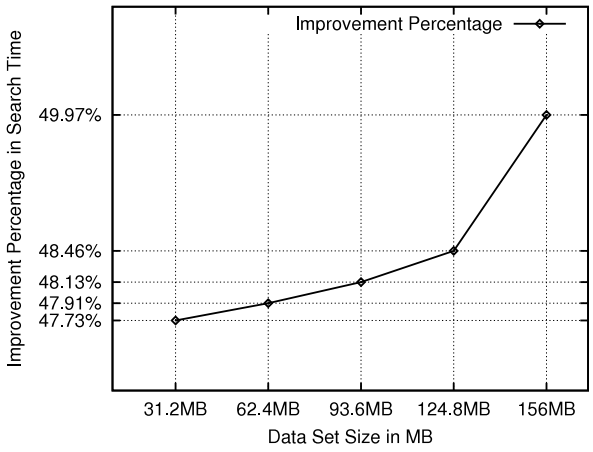

Fig. 6. Percentage of improvement in search time versus dataset size.

with regards to space complexity. We observe that the proposed algorithm has the same order as the space complexity of Kida's algorithm, and is better than Tao's algorithm. Generally, queries tend to have a small number of short patterns [16], leading to the case of $m^2 < (m \times t)$. The relatively small differences in space complexity are mostly overshadowed by the time requirements of in storage environments.

5.2. Experimental evaluation

This subsection presents the performance of the proposed algorithm experimentally against decompressing then searching the data using Aho–Corasick algorithm. Aho–Corasick is the counterpart multiple pattern matching algorithm for plain data (i.e. uncompressed data). The proposed algorithm was implemented in C++ using POSIX API and GCC compiler. Unlike related work, the implementation is open source and make available to researcher via a GitHub repository [58].

The experiments were performed on a 2.4 GHz Linux PC with 4 GB of RAM. As a testing benchmark, real world biostatistics data, that was gathered by Z. Pervez et al. in [59], are used. The data was collected from six healthy adults i.e., both male and female of age between 20 to 26 with average weight of 60 kg and average height of 164 cm for nine different activities (i.e., Taking Medicine, Teeth Brushing, Eating, Smoking, Running, Jogging, Walking, Walking Downstairs, and Walking Up-stairs). The plain numerical dataset has a total size of 156 MB. The 156 MB file was divided into five parts, each of which consists of 32.2 MB of plain data. These parts were added to the evaluation one by one over time in order to test the performance scalability.

Ultimately, we ended up processing five datasets: 31.2 MB, 62.4 MB, 93.6, 124.8 MB, and 156 MB. The compression ratios for these five datasets are between 2.7 and 2.8. The size of the pattern set was 50 patterns, each of which has a length between 5 and 10 ASCII characters. In the conducted experiments, the time is plotted using the Linux built-in *time* command. The memory usage is analyzed using the *massif* profiling tool of *Valgrind* framework [60].

Fig. 5 shows the average execution time of the search process (including the preprocessing time) for the proposed algorithm (MultiPLZW). The results are plotted against the average execution time that is required to decompress the data, and then perform multiple-pattern matching using AC algorithm. The reported average execution time over 20 runs were 1.15, 2.24, 3.60, 4.67, and 5.85 s for the proposed algorithm, for the data sets of size 32.2 MB, 62.4 MB, 93.6 MB, 124.8 MB, and 156 MB, respectively. In comparison, the counterpart achieves average execution time of 2.2, 4.3, 6.94, 9.06, and 11.67 s. These results demonstrate the considerable improvement in search time of our proposed algorithm, which amounts nearly half of the time to decompress-then-search. Moreover, we observe greater comparative improvement, that is scalability, when dataset grows larger.

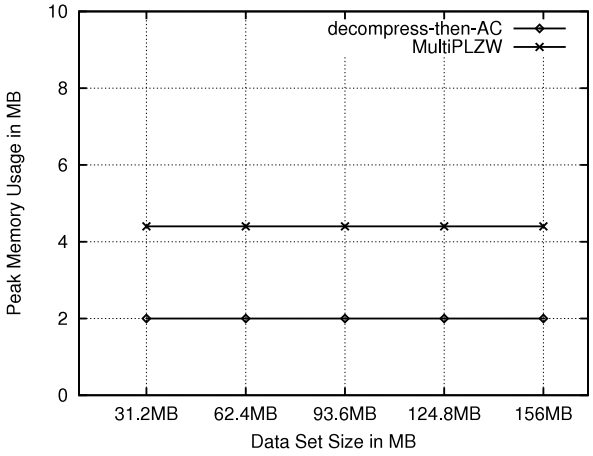

Fig. 7. Memory usage comparison of *MultiPLZW* against *decompression-then-AC*.

Fig. 6 demonstrates the superior scalability of the proposed algorithm. We observe that, with an increase in dataset size, the performance gap between the two algorithms is magnified. Furthermore, when considering a linear increase in dataset size, the variant in improvement percentage is multiplied. This increase in performance is explained by the fact that the time to decompress-then-search increases as the dataset size becomes larger because of the preprocessing and compression ratio. Hence, the proposed algorithm is desirable in big datasets. The percentage of improvement was calculated according to Eq. (3), where A is the response time of decompressing-then-AC, while B is the response time of our proposed algorithm.

$$improvement = \frac{A - B}{A} \times 100 \tag{3}$$

Fig. 7 demonstrates the peak memory usage for the proposed algorithm in comparison with decompressing-then-searching using AC algorithm for the five aforementioned datasets. It can be concluded that the proposed algorithm, on average, requires approximately 2.2 times the memory space that is required for decompressing the data then using AC algorithm to perform multiple pattern matching. That is small price to pay for faster search and better scalability.

6. Conclusion

In this work, we introduce a linear solution for fast multiple pattern matching in LZW-compressed data. The proposed algorithm mainly

aims at improving response time of queries over big datasets. The proposed algorithm uses a customized automated generalized suffix trie, in conjunction with a lookup table, mapping table, and history tree. The proposed algorithm demonstrates the best time complexity benchmark when is compared to its counterparts. The algorithm simultaneously maintains a space complexity as efficient as the best of its competitors. Experimentally, the algorithm shows a significant improvement in time over the common approach of decompressing data and then searching using Aho–Corasick algorithm. Moreover, the results show the trend of being magnified in performance improvement as the dataset becomes larger. This proves its notable scalability and efficiency when dealing with big datasets, and thus its strong viability in storage environments.

Acknowledgment

This work was supported by Zayed University, United Arab Emirates Research Office, Research Cluster Award # R17079.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

[1] A.J. Younge, G. Von Laszewski, L. Wang, S. Lopez-Alarcon, W. Carithers, Efficient resource management for cloud computing environments, in: Green Computing Conference, 2010 International, IEEE, 2010, pp. 357–364.

[2] S. Sivathanu, L. Liu, M. Yiduo, X. Pu, Storage management in virtualized cloud environment, in: Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on, IEEE, 2010, pp. 204–211.

[3] B. Aldawsari, T. Baker, D. England, Trusted energy-efficient cloud-based services brokerage platform, Int. J. Intell. Comput. Res. 6 (2015) 630–639, http://dx.doi.org/10.20533/ijicr.2042.4655.2015.0078.

[4] T.A. Welch, A technique for high-performance data compression, Computer 6 (17) (1984) 8–19.

[5] H. Wolf, K. Froitzheim, Webvideo. a tool for www-based teleoperation, in: Industrial Electronics, 1997. ISIE'97., Proceedings of the IEEE International Symposium on, Vol. 1, IEEE, 1997, pp. SS268–SS273.

[6] Y. Wu, S. Lonardi, W. Szpankowski, Error-resilient lzw data compression, in: Data Compression Conference, 2006. DCC 2006. Proceedings, IEEE, 2006, pp. 193–202.

[7] T. Gagie, P. Gawrychowski, S.J. Puglisi, Approximate pattern matching in lz77-compressed texts, J. Discrete Algorithms 32 (2015) 64–68, http://dx.doi.org/10.1016/j.jda.2014.10.003, StringMasters 2012 & 2013 Special Issue (Volume 2). URL http://www.sciencedirect.com/science/article/pii/S1570866714000719.

[8] M. Aldwairi, M.A. Alshboul, A. Seyam, Characterizing realistic signature-based intrusion detection benchmarks, in: Proceedings of the 6th International Conference on Information Technology: IoT and Smart City, in: ICIT 2018, ACM, New York, NY, USA, 2018, pp. 97–103, http://dx.doi.org/10.1145/3301551.3301591, URL http://doi.acm.org/10.1145/3301551.3301591.

[9] K. Huang, D. Zhang, Z. Qin, Accelerating the bit-split string matching algorithm using bloom filters, Comput. Commun. 33 (15) (2010) 1785–1794, http://dx.doi.org/10.1016/j.comcom.2010.04.043, URL http://www.sciencedirect.com/science/article/pii/S0140366410002215.

[10] M. Aldwairi, W. Mardini, A. Alhowaide, Anomaly payload signature generation system based on efficient tokenization methodology, Int. J. Commun. Antenna Propag. (IRECAP) 8 (5) (2018) http://dx.doi.org/10.15866/irecap.v8i5.12794, https://www.praiseworthyprize.org/jsm/index.php?journal=irecap&page=article&op=view&path.

[11] D. Adjeroh, T. Bell, A. Mukherjee, Pattern matching in compressed texts and images, Found. Trends Signal Process. 6 (23) (2013) 97–241, http://dx.doi.org/10.1561/2000000038.

[12] G. Navarro, N. Prezza, Universal compressed text indexing, Theoret. Comput. Sci. 762 (2019) 41–50, http://dx.doi.org/10.1016/j.tcs.2018.09.007, URL http://www.sciencedirect.com/science/article/pii/S0304397518305723.

[13] M. Farach, M. Thorup, String matching in lempelziv compressed strings, Algorithmica 20 (4) (1998) 388–404.

[14] A. Amir, G. Benson, M. Farach, Let sleeping files lie: Pattern matching in z-compressed files, J. Comput. System Sci. 52 (2) (1996) 299–307.

[15] T. Kida, M. Takeda, A. Shinohara, M. Miyazaki, S. Arikawa, Multiple pattern matching in lzw compressed text, in: Data Compression Conference, 1998. DCC'98. Proceedings, IEEE, 1998, pp. 103–112.

[16] T. Tao, A. Mukherjee, Pattern matching in lzw compressed files, IEEE Trans. Comput. 54 (8) (2005) 929–938.

[17] W. Rytter, Application of lempel–ziv factorization to the approximation of grammar-based compression, Theoret. Comput. Sci. 302 (1–3) (2003) 211–222.

[18] P. Gawrychowski, Simple and efficient LZW-compressed multiple pattern matching, J. Discrete Algorithms 25 (2014) 34–41.

[19] A.V. Aho, M.J. Corasick, Efficient string matching: an aid to bibliographic search, Commun. ACM 18 (6) (1975) 333–340.

[20] T. Baker, B. Aldawsari, M. Asim, H. Tawfik, Z. Maamar, R. Buyya, Cloud-senergy: A bin-packing based multi-cloud service broker for energy efficient composition and execution of data-intensive applications, Sustainable Comput.: Inf. Syst. (2018) URL http://eprints.leedsbeckett.ac.uk/5046/.

[21] S. Funasaka, K. Nakano, Y. Ito, Fast lzw compression using a gpu, in: Computing and Networking (CANDAR), 2015 Third International Symposium on, IEEE, 2015, pp. 303–308.

[22] M.R. Nelson, LZW Data compression, Dr. Dobb's J. 14 (10) (1989) 29–36.

[23] P. Barcaccia, A. Cresti, S. De Agostino, Pattern matching in text compressed with the ID heuristic, in: Data Compression Conference, 1998. DCC'98. Proceedings, IEEE, 1998, pp. 113–118.

[24] M. Aldwairi, A.M. Abu-Dalo, M. Jarrah, Pattern matching of signature-based IDS using myers algorithm under mapreduce framework, EURASIP J. Inf. Secur. 2017 (2017) 9, URL http://dblp.uni-trier.de/db/journals/ejisec/ejisec2017.html#AldwairiAJ17.

[25] C. Lin, J. Li, C. Liu, S. Chang, Perfect hashing based parallel algorithms for multiple string matching on graphic processing units, IEEE Trans. Parallel Distrib. Syst. 28 (9) (2017) 2639–2650, http://dx.doi.org/10.1109/TPDS.2017.2674664.

[26] P. Bieganski, J. Riedl, J.V. Carlis, E.F. Retzel, Generalized suffix trees for biological sequence data: Applications and implementation, in: HICSS (5), 1994, pp. 35–44.

[27] E. Ukkonen, On-line construction of suffix trees, Algorithmica 14 (3) (1995) 249–260.

[28] J. Shun, G.E. Blelloch, A simple parallel cartesian tree algorithm and its application to parallel suffix tree construction, ACM Trans. Parallel Comput. 1 (1) (2014) 8.

[29] Z. Chen, R. Fowler, A.W.-C. Fu, C. Wang, Fast construction of generalized suffix trees over a very large alphabet, Lecture Notes in Comput. Sci. (2003) 284–293.

[30] D. Adjeroh, A. Mukherjee, M. Powell, T. Bell, N. Zhang, Pattern matching in BWT-compressed text, in: Data Compression Conference, Snow Bird, Utah, 2002, p. 445.

[31] T. Bell, M. Powell, A. Mukherjee, D. Adjeroh, Searching BWT compressed text with the boyer-moore algorithm and binary search, in: Data Compression Conference, 2002. Proceedings. DCC 2002, IEEE, 2002, pp. 112–121.

[32] P. Ferragina, G. Manzini, An experimental study of an opportunistic index, in: Proceedings of the Twelfth Annual ACM-SIAM Symposium on Discrete Algorithms, Society for Industrial and Applied Mathematics, 2001, pp. 269–278.

[33] T. Eilam-Tzoreff, U. Vishkin, Matching patterns in strings subject to multi-linear transformations, Theoret. Comput. Sci. 60 (3) (1988) 231–254.

[34] A. Amir, C. Benson, Efficient two-dimensional compressed matching, in: Data Compression Conference, 1992. DCC'92., IEEE, 1992, pp. 279–288.

[35] A. Amir, G. Benson, Two-dimensional periodicity and its applications, in: Proceedings of the Third Annual ACM-SIAM Symposium on Discrete Algorithms, Society for Industrial and Applied Mathematics, 1992, pp. 440–452.

[36] A. Amir, G.M. Landau, U. Vishkin, Efficient pattern matching with scaling, J. Algorithms 13 (1) (1992) 2–32.

[37] J. Ziv, A. Lempel, A universal algorithm for sequential data compression, IEEE Trans. Inf. Theory 23 (3) (1977) 337–343.

[38] M. Miyazaki, A. Shinohara, M. Takeda, An improved pattern matching algorithm for strings in terms of straight-line programs, in: Combinatorial Pattern Matching, Springer, 1997, pp. 1–11.

[39] M. Karpinski, W. Rytter, A. Shinohara, An efficient pattern-matching algorithm for strings with short descriptions, Nord. J. Comput. 4 (2) (1997) 172–186.

[40] D. Adjeroh, T. Bell, A. Mukherjee, Pattern Matching in Compressed Texts and Images, Now Publishers Inc., Hanover, MA, USA, 2013.

[41] D. Tao, S. Di, Z. Chen, F. Cappello, Exploration of pattern-matching techniques for lossy compression on cosmology simulation data sets, in: International Conference on High Performance Computing, Springer, 2017, pp. 43–54.

[42] T. Gagie, P. Gawrychowski, S.J. Puglisi, Approximate pattern matching in lz77-compressed texts, J. Discrete Algorithms 32 (2015) 64–68.

[43] H. Buluş, A. Carus, A. Mesut, A new word-based compression model allowing compressed pattern matching, Turk. J. Electr. Eng. Comput. Sci. 25 (5) (2017) 3607–3622.

[44] T. Gagie, P. Gawrychowski, J. Kärkkäinen, Y. Nekrich, S.J. Puglisi, Lz77-based self-indexing with faster pattern matching, in: Latin American Symposium on Theoretical Informatics, Springer, 2014, pp. 731–742.

[45] R. Beal, D. Adjeroh, P-suffix sorting as arithmetic coding, J. Discrete Algorithms 16 (2012) 151–169.

[46] R. Beal, D.A. Adjeroh, The structural border array, J. Discrete Algorithms 23 (2013) 98–112.

[47] R. Beal, D. Adjeroh, Compressed parameterized pattern matching, Theoret. Comput. Sci. 609 (2016) 129–142.

[48] D. Adjeroh, T. Bell, A. Mukherjee, et al., Pattern matching in compressed texts and images, Foundations and Trends® in Signal Processing 6 (2–3) (2013) 97–241.
[49] H. Hu, K. Zheng, X. Wang, A. Zhou, Gfilter: A general gram filter for string similarity search, IEEE Trans. Knowl. Data Eng. 27 (4) (2015) 1005–1018, http://dx.doi.org/10.1109/TKDE.2014.2349914.
[50] Y. Wu, C. Shen, H. Jiang, X. Wu, Strict pattern matching under non-overlapping condition, Sci. China Inf. Sci. 60 (1) (2017) 012101.
[51] M. Drory Retwitzer, M. Polishchuk, E. Churkin, I. Kifer, Z. Yakhini, D. Barash, Rnapattmatch: a web server for RNA sequence/structure motif detection based on pattern matching with flexible gaps, Nucleic Acids Res. 43 (W1) (2015) W507–W512.
[52] Y. Wu, Y. Tong, X. Zhu, X. Wu, NOSEP: nonoverlapping sequence pattern mining with gap constraints, IEEE Trans. Cybernet. (2017).
[53] C.-D. Tan, F. Min, M. Wang, H.-R. Zhang, Z.-H. Zhang, Discovering patterns with weak-wildcard gaps, IEEE Access 4 (2016) 4922–4932.
[54] P. Barcaccia, A. Cresti, S. De Agostino, Pattern matching in text compressed with the ID heuristic, in: Data Compression Conference, 1998. DCC'98. Proceedings, IEEE, 1998, pp. 113–118.
[55] M. Mohammad, E.-B. Lin, Gibbs effects using daubechies and coiflet tight framelet systems, Contemp. Math. 706 (2018) 271–282, http://dx.doi.org/10.1090/conm/706/14209.
[56] Lempel-Ziv-Welch (LZW) Encoding Discussion and Implementation, http://michael.dipperstein.com/lzw/ (cited June 2019).
[57] R. Ma, J. Li, H. Guan, M. Xia, X. Liu, Endas: Efficient encrypted data search as a mobile cloud service, IEEE Trans. Emerg. Top. Comput. 3 (3) (2015) 372–383, http://dx.doi.org/10.1109/TETC.2015.2445101.
[58] MultiPLZW Source Code, https://github.com/abdulmughniHamzah/MultiPLZW (cited June 2019).
[59] Z. Pervez, M. Ahmad, A.M. Khattak, S. Lee, T.C. Chung, Privacy-aware relevant data access with semantically enriched search queries for untrusted cloud storage services, PLoS One 11 (8) (2016) e0161440.
[60] Valgrind Home, http://valgrind.org/ (cited June 2019).