Compressed pattern matching is an emerging research area that aims in searching patterns efficiently in the compressed files with minimal (or no) decompression. In this paper, we report our work on multiple-pattern matching in LZW compressed files using Aho-Corasick algorithm. The algorithm takes $O(mt+n+r)$ time with $O(mt)$ extra space, where $n$ is the size of the compressed file, $m$ is the size of the pattern length, $t$ is the size of the LZW trie and $r$ is the number of occurrences of the patterns. Extensive experiments have been conducted to test the performance of our algorithms. The results showed that our multiple-pattern matching algorithm is practically the fastest among all approaches when the number of patterns is not very large. Therefore, our algorithm is preferable for general string matching applications. The proposed algorithm is efficient for large files and it is particularly efficient when being applied on archival search if the archives are compressed with a common LZW trie.

1 INTRODUCTION

Data compression has been widely used to store the huge amount of data nowadays. On account of efficiency (in terms of both space and time), there is a need to keep the data in compressed form for as much as possible, even when it is being searched. This has led to calls for compressed pattern matching (CPM), whereby search operations are performed directly on the compressed data without initial decompression. Although there are many variations of the CPM, the CPM is generally defined as: Given the compressed format $S.Z$ of a text string (or an image) $S$ and a pattern string (or a sub-image) $P$, report the occurrences of $P$ in $S$ with minimal (or no) decompression of $S.Z$.

A survey on earlier works on compressed pattern matching can be found in [4]. Recent important CPM works include a series of BWT-based approaches [1,5,7], namely, Compressed-Domain Boyer-Moore, Binary Search, Suffix Arrays, q-grams and FM-Index. These approaches have been implemented and compared in [8]. Among these approaches, FM-Index is made search-aware at the price of sacrificing the compression performance. All other approaches cannot be applied directly on Bzip2 (an efficient commercialized BWT compression utility) compressed files in that they require the entire file to be compressed as one block. As can be seen from [8], this dramatically degrades the time efficiency of the compression and decompression. Besides, all the above BWT-based CPM approaches are “partial” compressed-domain pattern matching because they all require the compressed files to be partially de-compressed and the partial-decompression causes an overhead of the pattern matching. When the number of patterns to be searched is small, this overhead can be extremely expensive.
The CPM algorithms based on the LZ-family compression have also been conducted in the last decade. The research of searching LZ-compressed files is very important because the LZ compressions are among the most efficient and popular compressions. Their excellent time/compression efficiency and easy implementation have gained them a large popularity in the commercial world (e.g. the ZIP utilities). Farach and Thorup proposed a randomized algorithm [6] to determine whether a pattern is present or not in LZ77 compressed text in time \(O(m+n\log2(u/n))\), where \(u\) is the raw file size, \(n\) is the compressed file size and \(m\) is the length of the pattern. Navarro and Raffinot [9] proposed a hybrid compression between LZ77 and LZ78 that can be searched in \(O(\min(u, n\log m)+r)\) average time, where \(r\) is the total number of matches; Amir [2] proposed an algorithm which runs in \(O(n\log m)\) -- “almost optimal” or in \(O(n+m^2)\), depending on how much “extra space” being used, to search the first occurrence of a pattern in the LZW encoded files. Amir’s algorithm has been well-recognized not only because of its “almost-optimal” or near “optimal” performance, but also because it works directly with the LZW compression without having to modify it – this is a great advantage because keeping the popular implementations of the LZW and avoiding the re-compression of the LZW-compressed files are highly desirable. Amir’s original algorithm has been implemented in [3] where the algorithm was enhanced to report all pattern occurrences. A naive multiple-pattern matching method was also implemented in [3]. In [11-12], Amir’s algorithm has been extended by Kida for multiple-pattern matching by using Aho-Corasick algorithm and it takes \(O(n+m+t+r)\) time and \(O(m^2+t)\) extra space for the algorithm to report the pattern occurrences. Experimental results from [11-12] show that the algorithm is nearly twice as fast as the decompress-and-search algorithm.

In this paper, we report a novel multiple-pattern matching algorithm using Aho-Corasick algorithm and our algorithm is closely compared with the multiple-pattern matching algorithm developed by Kida [11-12]. Extensive experiments have been conducted to test the search performance of our approaches and to compare with other well-known compressed pattern matching algorithms, particularly the BWT-based algorithms and Kida’s algorithm. The results show that our multiple-pattern matching algorithm is competitive among the best of the compressed-pattern matching algorithms and works practically fastest when the number of patterns to be searched is not very large. Our multiple-pattern matching algorithm, therefore, is preferable for general string matching applications. The proposed algorithm is efficient for large files and it is particularly efficient when being applied on archival search if the archives are compressed with a common LZW trie.

The plan for the remainder of the paper is as follows. In section 2 and section 3, we briefly introduce the LZW algorithm and the Aho-Corasick algorithm, respectively. In section 4, we present the proposed algorithm and compare it with the algorithm in [11-12]. The experimental results are reported in section 5. Section 6 concludes the paper.

## 2 LZW COMPRESSION

Let \(S=c_1c_2c_3...c_u\) be the uncompressed text of length \(u\) over alphabet \(\Sigma=\{a_1, a_2, a_3,...,a_q\}\), where \(q\) is the size of the alphabet. We denote the LZW compressed format of \(S\) as \(S.Z\) and each code in \(S.Z\) as \(S.Z[i]\), where \(1 \leq i \leq n\).
The LZW compression algorithm uses a tree-like data structure called a “trie” to store the dictionary generated during the compression processes. Each node on the trie contains a node number, which is a unique ID in the range of \([0, n+q]\) and a label, which is a symbol from the alphabet \(\Sigma\). The chunk of a node is then defined as the string on the path from the root to the node.

During compression, LZW algorithm scans the text and finds the longest sub-string that appears in the trie as the chunk of some node \(N\) and outputs \(N\) to \(S.Z\). The trie then grows by adding a new node under \(N\) and the new node’s label is the next un-encoded symbol in the next. Obviously, the new node’s chunk is node \(N\)’s chunk appended by the new node’s label. An example that was used in [2] is presented in Fig. 1 to illustrate the trie structure. The decoder constructs the same trie and uses it to decode \(S.Z\). Both the compression and decompression (and thus and trie construction) can be done in time \(O(u)\).

$$\begin{align*}
S &= aabbaabbabcccccc \\
S.Z &= 1,1,2,2,4,6,5,3,11,12;
\end{align*}$$

**Fig. 1.** An example of the LZW trie.

The LZW trie can be reconstructed from \(S.Z\) in time \(O(n)\) without explicitly decoding \(S.Z\) in the following manner [2]: When the decoder receives a code \(S.Z[i]\), assuming \(S.Z[i-1]\) has already been received in the previous step \((2 \leq i \leq n)\), a new node is created and added as a child of node \(S.Z[i-1]\). The node number of the new node is \(i-1+q\) and the label of the new node is the first symbol of node \(S.Z[i]\)’s chunk.

### 3 AHO-CORASICK ALGORITHM

Aho-Corasick algorithm [10] is a classic solution for multiple-pattern matching. Pattern matching is performed using an automaton called \(AC\) automaton, which is constructed based on the patterns. The \(AC\) automaton for a set of patterns \(\{aa, ab, abc\}\) is shown in Fig. 2.

In an \(AC\) automaton, each edge is labeled with a symbol and edges coming out from a node (state) have different labels. If we define \(R(v)\) of a state \(v\) as the concatenation of labels along the path from the root to state \(v\), the following is true: for each pattern \(P\) in the pattern set, there is a state \(v\) that \(R(v) = P\), and this state is called a final state; each state represents a prefix of some pattern; for each leaf state \(v\), there is some pattern \(P\) in the pattern set so that \(R(v) = P\). For instance, pattern “aa” is represented by state 2 and leaf state 4 represents pattern “abc”. Both states 2 and states 4 are final states.
For each state, a $\text{goto}(v, a)$ function is defined which gives the state entered from state $v$ by matching symbol $a$. For instance, $\text{goto}(1, a) = 2$ means that the state to be entered is 2 if we match symbol $a$ from state 1. A failure link $f(v)$ (indicated as dotted line in Fig. 2) is also defined for each state and it gives the state entered when mismatch happens. The failure link $f(v)$ points to a state that represents the longest proper suffix of $R(v)$. Thus, when mismatch happens, by following the failure link, we will be able to continue the matching process since the state to be entered also corresponds to a prefix of some pattern. Finally, an $\text{out}(v)$ function is defined for state $v$ that gives the patterns recognized when entering that state.

When searching the patterns, the AC automaton starts from the root of the automaton and processes one symbol from the input text $T$ at a time. Through the $\text{goto}$ functions and the failure links, the automaton changes its current state from one to another. The automaton reports the pattern occurrence if a final state is entered. The search algorithm is given below:

```c
v = root;
WHILE (not end of input file T)
{
    get next symbol a from T;
    WHILE (!\text{goto}(v, a))
    {
        v = f(v);
        v = \text{goto}(v, a);
        IF !\text{out}(v) THEN report patterns in \text{out}(v)
    }
}
```

The construction of the automaton takes time and space $O(m)$ where $m$ is the total length of the patterns. The search takes time $O(u)$ where $u$ is the size of the input text. Thus, the overall computational time of Aho-Corasick algorithm is $O(u+m)$. 
4 THE PROPOSED ALGORITHM

The Algorithm

To be able to search the patterns in the compressed files, we need an AC automaton that is able to process the compressed symbols, or specifically, the LZW codes.

For this purpose, we now define a state-transition list for each node of the LZW trie. Each entry in the state-transition list is in the form of \( v_1 \to v_2 \), which indicates that state \( v_1 \) will be changed to state \( v_2 \) if the chunk of the current trie node is fed to the AC automaton. State \( v_1 \) in the list should exhaust all possible AC automaton states.

For the initial LZW trie, since each node’s chunk is simply its label, we may immediately create the state-transition list for a node by feeding its label to the AC automaton. Consider the same example in Fig. 1 and we assume the patterns are: \{aa, ab, abc\}, of which the AC automaton is shown in Fig. 2, the initial LZW trie with the state-transition list is shown in Fig. 3. For example, the state-transition list for node 1 is \{0-1, 1-2, 2-2, 3-1, 4-1\}. It means that when node 1 (of label “a”) is received, if the current state is 0, the next state will be 1; if the current state is 1, the next state will be 2; and so forth. Since the “from” state \( v_1 \) exhausts all possible states and they are ordered increasingly, the state-transition list can be written as \{1, 2, 2, 1, 1\} for node 1.

During the search, the algorithm linearly scans the compressed data. Each time a code is received, a new node is added to the LZW trie, as have been discussed in section 2. The new node’s state-transition list, instead of being obtained by feeding its chunk to the AC automaton, can be computed directly from its label and its parent only. For example, when node 4 (whose label is “a”) is added under node 1, we do not have to construct the state-transition list by feeding its chunk “aa” to the AC automaton. Instead, a better approach is to feed only its label “a” to the AC automaton and starting matching that label from the corresponding \( v_2 \) state in its parent’s state-transition list. Therefore, we will be able to obtain the new state-transition list: \{2; 2; 2; 2\} from the parent node’s state-transition list \{1, 2, 2, 1, 1\} because the states 1, 2, 2, 1, and 1, when receiving symbol “a”, will be changed to state 2, 2, 2, 2 and 2, respectively.

The complete LZW trie for the above example (we ignored the state-transition information for some nodes because those node are not referenced during the compression) is shown in Fig. 4.
In Fig. 3 and Fig. 4, the state-transition list is stored directly under a LZW trie node to better illustrate our idea. In actual implementation, instead of having a list for each node, a single table is constructed for the whole trie and we call this table the state transition table. Each row of the state transition table corresponds to one state in the AC automaton; each column of the state transition table corresponds to one node in the LZW trie. The state transition table of the initial trie in Fig. 3 is shown below:

<table>
<thead>
<tr>
<th>State\Node</th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

We now show how to perform pattern matching for the same example using the state-transition table. At the beginning of the search, the current state is set as 0. When $S.Z[1]=1$ is received, using the state transition table, the current state is changed to 1; when the second code 1 is received, the station transition table indicates that the current state is changed to 2. The complete transition is shown in the following table, where the bolded numbers in the table indicate the final states of the AC automaton.

<table>
<thead>
<tr>
<th>Node</th>
<th>1</th>
<th>1</th>
<th>2</th>
<th>2</th>
<th>4</th>
<th>6</th>
<th>5</th>
<th>3</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>State</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>3</td>
<td>4</td>
<td>0</td>
</tr>
</tbody>
</table>

Thus, patterns occur whenever a final state is entered.

However, the above table only shows five final states while there are six pattern occurrences in the above example. An occurrence of “ab” is missing from the above table. The reason is that, when we compute the third entry ($v1$ is 2) of the state-transition list of node 6, by matching the label “b” from the parent node’s corresponding $v2$ state,
state 3, the computed state is 0, which is not a final state. However, the intermediate state, state 3, is a final state. Thus, a final state is “skipped” during the transitions and this is why the second occurrence “ab” is not reported. It can be tell that a final-state skipping happens only if the corresponding entry in a node’s ancestor node’s transition list is a final state. Thus, the problem can be fixed by adding a flag for each entry in the state transition list of a node and it is set as true if the corresponding entry of the node’s ancestor is a final state. This flag is inheritable by a node’s offsprings. During the search, if the flag is on, the current node’s chunk needs be processed symbol by symbol by the Aho-Corasick automaton so we will not miss any pattern occurrence.

Analysis and Comparison

The state transition table construction takes time and space $O(mt)$ where $t$ is the LZW trie size and $m$ is the total length of the patterns. The search time depends on how many final states are skipped during the search process and is proportional to $r$, the number of occurrences of the pattern. Thus, the search time is $O(n+r)$ and the total processing time of our algorithm is $O(n+mt+r)$. The extra space used in our algorithm is solely the cost on the state transition table, i.e. $O(mt)$, which is independent of the file size.

It would be very interesting to compare our algorithm with an algorithm developed by Kida et.al., as the later has the same general idea as ours. Kida’s algorithm is also able to perform the state transition by taking the compressed data. Their algorithm first constructs the GST (General Suffix Trie) and the AC automaton of the patterns. The algorithm then relies on two main functions: $Next(q,s)$ and $Output(q,s)$ that computes the next AC state from state $q$ by taking string $s$ and outputs all patterns that ends in $q.s$, respectively. Note that string $s$ is limited to only those are represented as LZW trie nodes. It takes $O(m^2+t)$ time and space for the construction of each function. During the search, the output function enumerates the patterns in time proportional to the number of pattern occurrences, i.e. $O(r)$. Overall, Kida’s algorithm takes time $O(n+m^2+t+r)$ and extra space of $O(m^2+t)$.

In a practical implementation, the size of the dictionary is constant. Therefore, the time and space complexity of our algorithm will be $O(n+m+r)$ and $O(m)$, respectively, contrasting to Kida’s $O(n+m^2+r)$ time and $O(m^2+t)$ space algorithm. The experimental results will section 5 will give further comparison of the two algorithms from the practical performance point of view.

Applying on Archival Search

The proposed algorithm works particularly efficient for archival search if the archives are compressed using a common LZW trie. In [13], a two-pass public trie scheme is proposed for archive compression where the archives, because of their similar characteristics, are compressed using a common LZW trie, which is pre-computed based on a small set of the data from the archives. The common trie is stored together with the compressed data and does not need to be reconstructed during the decompression. Experimental results have shown that the compression performance in [13] is better than gzip. If we apply the proposed algorithm on the archives compressed from [13], we will only need to pre-process the common LZW trie once before the search is started for the
entire collection of the records. The pre-processing takes the pre-computed LZW trie and computes the state transition table, by scanning the LZW trie and applying the AC automaton for each node. In comparison to the size of the archives, which might be of hundreds of megabytes, the cost on the state transition table construction is very small.

5 EXPERIMENTAL RESULTS

All experiments were conducted on a PC with the following configuration: CPU: Intel(R) Pentium(R) 4 1.80GHz; cache size: 512KB; total memory: 756MB. The OS is Linux 2.4.20. All data were obtained as the average of 50 runs. For pattern matching, each run uses a different set of patterns. Patterns are English words randomly selected from the file being searched.

Search performance

Fig. 5. The search performances for the original LZW compression.

In Fig. 5, the search performances of six different search algorithms are plotted. Three of them are the BWT-based CPM algorithms including bwt-binary, bwt-qgram and bwt-suffix. Three of them are LZW-based algorithms: lzw-ac, which is the proposed algorithm; lzw-amir, which is the implementation of the algorithm from [3]; lzw-decompress-then-ac, which decompresses the LZW compressed file first and then applies the Aho-Corasick algorithm. A magnified view the result is shown on the upper right corner in Fig. 5.

As can be seen from Fig. 5, when only a few patterns are to be searched, our approach reports the results almost instantly. The search time of our approach slowly increases with the number of search patterns increases. When the number of patterns to be searched is not very large (less than 140 in Fig. 5), our approach has the best performance among all; when the number of the patterns is larger than that threshold, only the decompress-
then-ac approach is better than our approach. The reason that decompress-then-ac outperforms our approach for large number of patterns is that the state transition table construction time in our approach increases very fast when the number of patterns increases. The lzw-amir algorithm, when searching just few patterns, has the performance next to our algorithm. However, when the number of patterns gets large, its performance dramatically decreases. The BWT-based approaches are slow in general because their huge overhead of partial decompression.

The lzw-decompress-then-ac approach can be improved by combining the decompression and the search together. Instead of waiting for the whole file to be decompressed, we can feed the characters to the AC automaton immediately after they are decoded. We call this approach lzw-decompress-ac and a magnified view of its performance is shown on the lower right corner in Fig. 5. It can be seen that the new implementation improves the search speed quite a lot. However, even with this improved version, the proposed algorithm still works the best when the number of patterns is approximately under 20.

Archival Search

In Fig. 6(a), the search performances for archival search where the archives are compressed using a common LZW trie are shown. Only the performances of two algorithms are plotted: the proposed algorithm and its most competing algorithm -- the lzw-decompress-ac approach. The benchmarks are Wall Street Journal (1987-1989) and the uncompressed data size is about 270 MB. The trie is built based on a small set of the collection and is used for compressing the whole collection. It can be seen that the proposed algorithm is faster when the number of patterns is less than approximately 90. This is a dramatic improvement comparing to the result in Fig. 5.

![Figure 6](image-url)

**Fig. 6.** The search performances
Comparison with Kida’s Algorithm

Fig. 6(b) compares the search performances of our lzw-ac algorithm and Kida’s algorithm. It can be seen that, when the length of the patterns is under approximately 40, our algorithm works faster than Kida’s algorithm. However, when the length of the patterns is larger than 40, Kida’s algorithm is faster. We think the competition of the two algorithms is highly affected by the pre-processing time of the patterns. In our algorithm, the pre-processing time is \(O(mt)\) while in Kida’s algorithm the pre-processing takes time \(O(m^2+t)\). The hidden constant in the big \(O\) notation and the size of the LZW trie \(t\) will decide the performances of the two algorithms.

6 CONCLUSIONS

In this paper, we propose a novel compressed multiple-pattern matching algorithm for LZW compressed files using Aho-Corasick algorithm. The experiments have shown that our algorithm works practically faster than other existing compressed pattern-matching algorithms when the number of pattern is not very large. Therefore, the proposed algorithm is very efficient for general string matching applications. The proposed algorithm works particularly efficient for archival search when the archives are compressed using a common trie.

REFERENCES

5. Tim Bell, Matt Powell, Amar Mukherjee and Don Adjeroh “Searching BWT compressed text with the Boyer-Moore algorithm and binary search”, Data Compression Conference, Snow Bird, Utah, April 2002, pp. 112-121