A DICTIONARY-BASED COMPRESSED PATTERN MATCHING ALGORITHM

MENG-HANG HO AND HSU-CHUN YEN

Abstract. Compressed pattern matching refers to the process of, given a text in a compressed form and a pattern, finding all the occurrences of the pattern in the text without decompression. To utilize bandwidth more effectively in the Internet environment, it is highly desirable that data be kept and sent over the Internet in the compressed form. In order to support information retrieval for compressed data, compressed pattern matching has been gaining increasing attention from both theoretical and practical viewpoints. In this article, we design and implement a dictionary-based compressed pattern matching algorithm. Our algorithm takes advantage of the dictionary structure common in the LZ78 family. With the help of a slightly modified dictionary structure, we are able to do ‘block decompression’ (a key in many existing compressed pattern matching schemes) as well as pattern matching on-the-fly, resulting in performance improvement as our experimental results indicate.

Key words: compression, LZW, pattern matching

1. Introduction

As the population of the Internet users grows at a rapid pace, it is expected that the Internet be capable of delivering texts, voice, video and all kinds of multimedia information and services effectively and rapidly. To this end, the quest for faster and faster computing facilities (including CPU, memory, switching devices, and more) alone is insufficient; as one might expect, how to utilize the existing bandwidth effectively is also a key upon which the success of the Internet heavily relies. The bandwidth issue is becoming even more critical in the wireless environment, for based on the current technologies, the bandwidth capacities of wireless networks are much more limited in comparison with that in the wired environment. As a consequence, in the past decade a vast amount of research has been focusing on designing efficient mechanisms and protocols to manage and deliver data over networks of limited bandwidth, including the installation of the so-called proxy servers, for instance. A more economic way to utilize limited bandwidth effectively is, perhaps, by sending less amount of data through the network through the use of the so-called data compression mechanisms at various stages in the transmission process[16, 17]. A natural question arises: How to manipulate compressed data? By manipulation we mean the processes of compressing data, de-compressing data as well as retrieving information.

From the performance viewpoint, it is of interest and importance to investigate whether part of the above retrieving and updating steps can be carried out on the compressed data directly.

Department of Electrical Engineering, National Taiwan University, Taipei 106, Taiwan, Republic of China.
Compressed pattern matching refers to the process of, given a text in a compressed form and a pattern, finding all the occurrences of the pattern in the text without decompression. As explained in the above, compressed pattern matching could play a key role in today's Internet applications, provided that the matching can be carried out efficiently. As a consequence, compressed pattern matching has been receiving increasing attention from both theoretical and practical viewpoints in the computer science and engineering society. See, e.g., [1, 4, 6, 10]. In particular, a compressed pattern matching (for LZW compression) algorithm has been reported in [7] whose asymptotic running time outperforms the method of decompressing the text followed by an ordinary pattern matching. More recent work on compressed pattern matching can be found in [11, 12, 13].

The aim of this article is to investigate dictionary-based compressed pattern matching. The well-known LZW is a typical dictionary-based compression scheme. Throughout the rest of our discussion, LZW is assumed to be the underlying compression mechanism; our approach can be applied to other dictionary-based compression methods as well. In a dictionary-based compression, the input text is partitioned into blocks according to the contents of the dictionary (which is often constructed on-the-fly as the compression progresses). For instance, if the input "abababc" is divided into

\[
\begin{array}{ccc}
  a & bab & abc
\end{array}
\]

(where \(a\), \(bab\) and \(abc\) are assumed to be dictionary entries), then the compressed output is the concatenation of the codes associated with \(a\), \(bab\) and \(abc\). Now suppose \(aba\) is the pattern to be found. It is easy to see that \(aba\) spreads over more than one block in the compressed text, making the matching nontrivial. In fact, this is the key that makes compressed pattern matching unique and difficult. In most of the existing dictionary-based compressed pattern matching algorithms [5, 8, 9], it is needed that the contents of each block be restored completely from its code in order for the matching to proceed, and the core of those algorithms relies on manipulating the so-called active blocks (i.e., blocks that have the potential to be part of the match). In our work, we propose a novel approach using which blocks are restored on an on-demand basis. Once the restoration of a block finds a match to be impossible, the procedure terminates. To facilitate such an on-demand search, we propose a data structure for the dictionary upon which the on-the-fly restoration of a block is based. To illustrate the performance of our algorithm, some experimental results are shown.

The rest of this paper is organized as follows. Section 2 gives the basic definitions as well as an example illustrating the LZW compression mechanism. In Section 3, the details of our dictionary-based compressed pattern matching algorithm are presented. Some experimental results are given in Section 4. Section 5 concludes this paper.

2. LZW Compression Scheme

Before presenting our compressed pattern matching algorithm, we briefly explain, through an example, how the LZW compression algorithm works. The reader is referred to [2, 3, 14, 15] for more details. The flow diagram of LZW is depicted in Figure 1.
To give the reader a better feeling for how LZW functions, consider a simple input string 'abababcd'. The contents of the dictionary as well as the output are shown in Figure 2 in a step-by-step fashion under LZW.

<table>
<thead>
<tr>
<th>input</th>
<th>dictionary</th>
<th>output</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>NULL</td>
<td>NULL</td>
</tr>
<tr>
<td>ab</td>
<td>&quot;ab&quot;(102)</td>
<td>61</td>
</tr>
<tr>
<td>abab</td>
<td>&quot;ba&quot;(103)</td>
<td>61, 62</td>
</tr>
<tr>
<td>ababc</td>
<td>&quot;abc&quot;(104)</td>
<td>61, 62, 102</td>
</tr>
<tr>
<td>ababcd</td>
<td>&quot;ca&quot;(105)</td>
<td>61, 62, 102, 63</td>
</tr>
<tr>
<td>ababc</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ababc</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ababc</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 2. The contents of the dictionary and the output for input string 'abababcd' under LZW.

It is known that the ASCII code of character a is 61, b is 62, and so on. Initially the first 256 entries of the dictionary are preserved for simple characters, and those additional entries of the dictionary (called the custom dictionary) added as the algorithm progresses begin from index 258 because we use index 257 as the reinitialize point of the dictionary. The first custom dictionary is 258, whose hex code is 102. For the sake of simplicity, the first (simple) 256 entries of the dictionary are omitted in Figure 2. At each step in Figure 2, the codes annotated by a frame (i.e., box) and an underline represent the current code and the next code, respectively.
For instance, in the first entry of Figure 2, \(a\) is the current code, and \(b\) is the character to be read next.

Initially, the custom dictionary is empty; hence, the code associated with the first character \(a\) (with code 61) of the input is output. At the same time, \(ab\) (carrying the code of 102) is added to the dictionary. Now consider the second entry of Figure 2, in which the current and the next codes are \(b\) and \(a\), respectively. Up to this point, \(ba\) is not part of the dictionary; hence, the code of \(b\) (i.e., 62) is placed in the output and \(ba\) (coded as 103) is added to the dictionary. In the next step (i.e., the third entry of Figure 2), \(ab\) is already in the dictionary (whose code is 102); hence, 102 is placed in the output. The remaining steps are straightforward. Each of the

\[
61 \quad 62 \quad 102 \quad 63 \quad 104
\]

in the output (compressed) string is called a *block*.

### 3. Our Compressed Pattern Matching Algorithm

The so-called "dictionary tree" plays a critical role in the LZ-compression family. As an illustrating example, consider Figure 3. Based on this dictionary tree, the substring \(aa\) will be encoded as index (i.e., code) 2 in the output file. Likewise, \(ace\) will be turned into 6 as the result of the compression. In the compressed text, we use the leaf node's index to represent the string associated with the full path from the root to the leaf, as described above. It is apparent that in many cases, a considerable amount of space reduction can be achieved using such a dictionary-based encoding scheme.

The main difference between searching directly within a plain text, and searching within a "compressed" text is that, for the former, we only need to keep track of two states namely ‘found’ and ‘not found’, whereas in the latter, the way that decompression is done in an on-the-fly fashion requires more information, such as ‘partially found’, to be kept. More will be said about this as our discussion progresses.

![A dictionary tree.](null)

As stated in the introduction, a key design in our algorithm lies in its ability to ‘restore’ block contents on an on-demand basis while matching is in progress. To facilitate such a restoration process, the following data structure is used as the building block of the dictionary tree. Each node contains three fields, namely,
char, index, and parent, where char stores the current character, index is the auto-incremented index (code) associated with the string along the path from the root to the current node, and parent is a pointer pointing to its parent node. In the very beginning, the dictionary has only the root node. Each time a character is read in, we will then look up the dictionary to see if it exists; if not, a new ‘node’ is added with its parent pointing to the parent node.

Now we are in a position to describe our compressed pattern matching algorithm in detail. Figure 4 depicts the basic structure of many of the compressed pattern matching algorithm found in the literature [6].

Unlike the plain-text pattern matching in which the text is scanned on a character-by-character basis, compressed pattern matching requires processing the compressed text in a block-by-block fashion. As a consequence, the notion of the so-called ”partial match” is central to many of the compressed pattern matching algorithms reported in the literature [5]. In our subsequent discussion, we show how the information of a partial match can be maintained in an effective way, which in turn provides an improvement in the performance of compressed pattern matching.

Before we do pattern matching for compressed texts directly, it is helpful to think about the decompression procedure first. Following our previous discussion, we know that a compressed text is presented by a sequence of chunks, and each chunk, in fact, is the index of the leaf node representing the string along associated full path in the dictionary. We look up the ”parent field” of the node, find its parent recursively till reaching the root. In fact, the underlying idea behind many of the recent compressed pattern matching algorithms deals with finding conditions under which part of a block constitutes a potential partial match. In this paper, we introduce a new auxiliary data structure to speed up the pattern matching procedure for compressed texts.

A way to do pattern matching in a compressed text, from the previous discussion, involves decoding each block, getting the string from the dictionary tree, then...
attempting to find a matching. For a block compressed from the string "ababaca", the corresponding path in the dictionary tree is depicted in Figure 5. To decompress this block, we traverse the path upwards until the root is reached. Now a question arises: Can we do the ‘restoration’ of the block in a smart way if pattern matching, instead of decompression, is our goal.

Figure 5. The path associated with the block ababaca in the dictionary tree.

While retrieving a block and getting the first character of the block, can we make some decision to speed up the procedure? To motivate this, consider a block associated with the string "zzz", and suppose our pattern is "aaa". If the first retrieved character of the block (i.e., the rightmost character) is "z", and that the block is 3 characters of length, the restored string of the block is clearly of the form "[.]|[.]z", meaning that the block is impossible to be matched by the pattern. To realize the above idea, we modify the dictionary structure mentioned earlier so that the ‘length’ of a block is easily obtainable.

It should be noted that in decompressing a compressed text, the dictionary is created on-the-fly from scratch. In our design, the compression algorithm remains unchanged. During the process of our compressed pattern matching, each time a new block is read, the associated node, if not already exists, is added to the dictionary tree and the data structure is modified in the normal way. In addition, an integer is attached the node to record the depth of the dictionary tree. A simple example of such a modified dictionary tree is shown in Figure 6, in which the number in each node represents the depth of the associated block.
To see why the additional ‘depth’ information helps, consider an example in Figure 7 in which a block of depth 5 with its last character equals Z is shown. The block is therefore of the form:

```
* * * Z
```

Figure 7. An example.

Even though the complete contents of the block may not be known, we do know that we have a string with 5 characters, and the last character is ‘Z’. Thinking the problem recursively, we will get the block information in the reverse order; for example, if we get a block and the associated string is ”aadaz”, the traversal sequence should look like the following:
Since we have the information of the length of the block being retrieved, and as explained earlier, block characters are retrieved in the reverse order, the basic compressed pattern matching algorithm stated in Figure 4 can be modified as below.

We now describe our pattern matching algorithm in greater detail. First we redefine our pattern matching within one block as follows. For example, if we have the pattern "abac" and the context "abacadaeaf", we want to develop a way which can deal with processing the text in the order of "faeadacaba", while the pattern is also scanned in the order of "caba". To do so, the concept of the so-called reverse string matching automaton is used.

Like a regular string matching automaton, for every pattern P, we find the reverse automaton and the transition table for it. Because we scan the string in the reverse order, first we need to reverse the pattern P to pattern Q. For example, if the pattern P is "cbaba", the reverse pattern Q will be "ababc". Once we know the reverse string, we can form the reverse string matching automaton as shown in the following graph.

Notice that the transition table of the pattern is pre-processed. Let |P| = m, i.e., the length of pattern P. Now suppose we have retrieved a block and got the last character, say "x", and the length of the string U associated with the block. Let |U| = n.
First, we consider the situation when the so-called "active blocks" are absent, meaning that the preceding block of the current block does not contribute to a 'partial match.' Consider Figure 9, in which the pattern dej···hij is to be matched against the current block ef𝑔···ℎ𝑘. As mentioned in our earlier discussion, a key feature in our matching algorithm is that the contents of the current block is retrieved on an on-demand basis from the rightmost character of the block. Since no active block is involved in this case, our matching algorithm finds the longest prefix of the pattern that matches the suffix of the block (i.e., substring dejehk in this example). The matching procedure begins at the \( m \)-th position of the pattern, against the rightmost position of the block. The procedure terminates once the desired longest prefix, say \( P' \), of the pattern (which matches the suffix of the block) is found. If \( P' \neq P \), then \( P' \) is called a partial match and the block involved is called an active block. In the next phase of the matching, the remaining pattern (of length \( n - m \)) will be compared to the successive block (i.e., block ef𝑔ab in Figure 9).

Now suppose our matching procedure reaches a point at which a partial match is in existence. See Figure 10, in which the shaded area labelled 'orig active' refers to the contents of the existing active blocks. In Figure 10, the current block is that containing egxpg. Since there is a mismatch between the current block and the pattern (marked 'X' in Figure 10), active blocks have to be modified, and such a modification can be carried out simply by using the KMP algorithm. (The contents of the new active blocks are also illustrated (i.e., the shaded area labelled 'new active') in Figure 10.) Now consider the case when no mismatch occurs in the process of the current block. We have either:

1. the pattern is exhausted, then a complete match is found, or
2. the current block is exhausted, then the current block becomes part of the new active blocks.

Depending on the lengths of the current block and the pattern, we need to differentiate between the following two cases:
Figure 10. The case when active blocks exist.

- $n \leq m$ (i.e., block string is shorter than the pattern)
  In this case, since the block is shorter than the pattern, it is impossible to find the complete match. What we try to achieve is to find the longest prefix of the pattern. We can achieve this goal by the help of our decompression algorithm. Since we know the block length, we can do pattern matching on the pattern from the location equaling the block length. That is, suppose we have a block with length $n$, the matching is simply done by decoding a character from the block (by traversing the dictionary tree upwards), then try to match the pattern according the sequence of locations (counting from the left) "$n", "n-1", "n-2" till the leftmost character of the pattern. If a match is found in the transition table, we then move to the next state of the reverse string matching automaton, look up the next character, follow the transition table again recursively until either a mismatch occurs, or the automaton reaches its end-point (meaning a match has been found).

- $n \geq m$ (i.e., block string is longer than the pattern)
  In this case, we may have a "pattern found" within the block string. For example, if the block string is "dexyza", and we want to find the pattern "xyz", even the last character is 'a', we cannot skip the block immediately because we still have to consider the possibility of having "xyz" inside the block string. In fact, we need to scan at most $n$ times even we do string matching which returns a false. Processing the last $m$ characters of the string (the first $m$ position of the block string) can be regarded as the previous case.

Since we have understood the situation where no active block happens, we can now consider how to do pattern matching with our algorithm under "active" situation. In fact, the situation is quite similar, since we have found an active block, which means the pattern we need to match now is shorter than the original one. For instance, suppose the string is "hello world", and we got an active block called "hello w", then the remaining pattern needed to match should become "orld", but now the situation is a little different, now if we want to extend the active part, we need to scan a block which can be appended to the active part. The detailed algorithm (different from the above) is describe below.
Consider block $\leq$ pattern, we have only two situations; one is that the block is completely the prefix of the remaining pattern needed to match, and the other is not. If not, we can discard the active status, then shift with KMP automaton. And in this case, we regard the remaining pattern as the new pattern, and then do pattern matching like "no-active" status but if we got a character which is "match miss", we can cancel the active status. If block $\geq$ pattern, the only difference is we need to do pattern matching from the end of the pattern, and the end of the block, scan with the reverse order, if the remain block $<$ pattern, we can treat the situation as block $\leq$ pattern, if we find a match miss, we can cancel the active status.

4. Implementation and Experiment

In our implementation, in order to balance between the penalty of long dictionary and the compression ratio while compressing normal-size data, we use 12 bits as a chunk of the characters (i.e., block size). Although one could have used length such as "10 bits" or "13 bits" instead, our selection allows the decompression process to be handled as quickly and as simple as possible. We implement the compression environment in ANSI C, allowing us to handle the nibble (half byte) by left shift 4 bits and right-shift 4 bits. It is clear that it becomes costly if we use other length instead, since we then need to maintain more and more low-level bit-oriented work, resulting in a significant slow-down in speed. LZW compression, a member of the LZ family, is our underlying compression scheme.

The experiment is designed as follows. First of all, we need a reference of the "standard pattern matching" to compare with our "compressed pattern matching". In fact, there are a number of tools to do pattern matching, and each has some ‘tricks’ of improving the performance. Notable examples include grep, egrep and fgrep in UNIX and LINUX. We need a standard pattern matching to verify the correctness and compare with the compressed pattern matching and our improved algorithm. We implement the general pattern matching using the following straightforward algorithm.

From [3], the algorithm is as follows:

\[\begin{align*}
n &\leftarrow \text{length}[T] \\
m &\leftarrow \text{length}[P] \\
\text{for } s &\leftarrow \text{to } n-m \\
\text{do if } P[1..m]=T[s+1..s+m] \\
\text{then print "pattern occurs with shift" } s
\end{align*}\]

In our implementation, the algorithm is pretty much the same with the following slight modification. Since we open the file with the standard C system call, and then we read each character directly, we have no idea about the length of T. But the pattern is pre-defined by user input, so our implementation model has been slightly modified as follows:

\[\begin{align*}
n &\leftarrow \text{length}[T] \\
m &\leftarrow \text{length}[P] \\
\text{read each char until } EOF \text{ happen} \\
\text{do if } P[1..m]=T[s+1..s+m] \\
\text{then print "pattern occurs with shift" } s
\end{align*}\]
As we shall see in our experimental results, our algorithm is capable of reducing the number of comparisons, in comparison with the original compressed pattern matching. Hence, in the brute-force comparison, our outputs include matching location, total context characters read, and total comparison counts. Our comparison is defined as: each character read in and compare it with the character in the proper location of the pattern. And in the compressed pattern matching, we will then output the total block read, total characters read, and total comparison happened. Because the running performance may depend on the implementation trick, pattern relation and pattern internal correlation, so we adjust the performance not depend on the matching machine time, but we see that exactly what is the real overhead of the compressed pattern matching and how we improve by using our algorithm. The number of characters skipped is shown as follows:

<table>
<thead>
<tr>
<th>file type(byte/block)</th>
<th>strlen=5</th>
<th>strlen=6</th>
<th>strlen=7</th>
<th>strlen=8</th>
</tr>
</thead>
<tbody>
<tr>
<td>implement header code(1337/887)</td>
<td>432</td>
<td>463</td>
<td>470</td>
<td>482</td>
</tr>
<tr>
<td>implement source code(5990/4274)</td>
<td>4286</td>
<td>4290</td>
<td>4292</td>
<td>4312</td>
</tr>
<tr>
<td>latex source code(18543/12358)</td>
<td>12123</td>
<td>12130</td>
<td>12130</td>
<td>12140</td>
</tr>
</tbody>
</table>

5. Conclusion

In this paper, we have studied the issue of compressed pattern matching, and have designed an algorithm (and data structure) so that ‘block decompression’ (which is key in compressed pattern matching) can be carried out on-the-fly. Because the current block string length is known, we can make pattern matching more efficient than the traditional way. Just as we mentioned before, although the ultimate goal is to compare with direct string matching, decreasing the size of the data will have a significant performance gain when the data are stored in separate locations on the Internet. Since we are capable of doing pattern matching under adaptive dictionary based compressed document efficiently, one of the applications of our design is transferring HTML and XML like "mark up" language in the wireless environment. Because the bandwidth of wireless cannot be expanded unlimitedly like in the wired environment, how to use the bandwidth with efficiency, and faster responding time, will be an important issue. We feel that our algorithm can help people store their information in the compressed form, while the information retrieval can still be performed efficiently.

References


