LZW Based Compressed Pattern Matching

Tao Tao, Amar Mukherjee

School of Electrical Engineering and Computer Science
University of Central Florida, Orlando, Fl.32816 USA
Email: (ttao+amar)@cs.ucf.edu

Abstract

Compressed pattern matching is an emerging research area that addresses the following problem: given a file in compressed format and a pattern, report the occurrence(s) of the pattern in the file with minimal (or no) decompression. In this paper, we report our work on compressed pattern matching in LZW compressed files. The reported work is based on Amir's well-known "almost-optimal" algorithm but has been improved to search not only the first occurrence of the pattern but also all other occurrences. The improvements also include the multi-pattern matching and a faster implementation for so-called "simple patterns". Extensive experiments have been conducted to test the search performance and to compare with the BWT-based compressed pattern matching algorithms. The results showed that our method is competitive among the best compressed pattern matching algorithms. LZW is one of the most efficient and popular compression algorithms used extensively and our method requires no modification on the compression algorithm. The work reported in this paper, therefore, has great economical and market potential.

1. Introduction

The importance of fast retrieval of information from the compressed data has been gaining attention because of the increasing amount of data that is being stored in the compressed format. With the phenomenal growth of the Internet, improvements in raw computer processing power, and developments in data storage and communications technology, huge amounts of multimedia data have become available, even to the casual user. 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. Another fundamental problem is to develop *search-aware* compression algorithms, in which compression is performed in such a way as to support later searching on the compressed data. Although there are many variations of the CPM, such as approximate CPM, CPM with "don't cares" and multi-pattern 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 [Bell2001]. Recently, [Bell2002, Adjeroh2002, Ferragina2001] have developed a series of BWT-based approaches to CPM including: Compressed-Domain Boyer-Moore, Binary Search, Suffix Arrays, q-grams and FM-Index. These approaches have been implemented and compared by [Firth2002] and the experimental results shown that they are the most competitive CPM algorithms among all reported CPM works. However, 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 so-called "bsmp" compression. As can be seen from [Firth2002], the bsmp compression dramatically degrades the time efficiency of the compression and decompression. Beside these problems, 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 nowadays. Their excellent time/compression efficiency and easy implementation have gained them a large popularity in the commercial world (e.g. the ZIP utilities and COMPRESS utility). Among the LZ-family based CPM algorithms: Farach and Thorup proposed a randomized algorithm [Farach98] to determine whether a pattern is present or not in LZ77 compressed text in time $O(m+n*log_2(u/n))$; Navarro and Raffinot [Navarro99] proposed a hybrid compression between LZ77 and LZ78 that can be searched in $O(min(u, n*logm)+r)$ average time, where r is the total number of matches; Amir [Amir96] proposed an algorithm which runs in $O(nlogm+m)$ -- "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. Barcaccia [Barcaccia98] extended Amir's work to an LZ compression method that uses the so-called "ID heuristic". Among the above LZ-based CPM algorithms, 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. But, unfortunately, Amir's algorithm has never been implemented because of its apparent complexity [Personal communication with Amir, 2003] and thus no experimental results are available to show the practical performance of the algorithm. A major limitation of Amir's algorithm, however, is that it is only able to report the first occurrence of the pattern. Multiple pattern matching is not addressed.

In the present paper, we first report a complete implementation of Amir's algorithm and make it practically useful by incorporating all the basic functionalities that a realistic pattern matching algorithm should possess viz. multiple occurrence of a pattern or multiple patterns matching in the compressed domain. We also report a faster implementation for so-called "simple patterns". Extensive experiments have been conducted to test the search performance and to compare with the BWT-based compressed pattern matching algorithms. The results showed that our method is competitive among the best compressed pattern matching algorithms. In view of the facts that LZW is a universal compression algorithm and our method requires no modification on the compression algorithm, we believe our proposed LZW based CPM algorithm will be readily adopted by compression community with large potential economic benefits.

The plan for the remainder of the paper is as follows. In section 2, we introduce Amir's compressed pattern matching algorithm. In section 3, we present our work that is considered as an enhancement of Amir's work. In Section 4, we report the experimental results. Section 5 concludes the paper.

2. LZW Compression and Amir's Algorithm

In the rest of this paper, we will use notations similar to those used in [Amir96]. Let $S=c_1c_2c_3\ldots\ c_u$ be the uncompressed text of length u over alphabet $\Sigma=\{a_1, a_2, a_3, \ldots, a_q\}$, 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 $0 \le i \le n$. We also denote the pattern as $P=p_1p_2\ p_3\ldots\ p_m$, where $pi \in \Sigma$ for $1 \le i \le m$, m is the length of pattern P.

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: a unique ID in the range $[0, n+q]$; thus, "a node with node number N" and "node N" are sometimes used interchangeably in this paper.
- A label: a symbol from the alphabet Σ,
- A chunk: the string that the node represents. It is simply the string consisting of the labels on the path from the root to this node.

For example, in Figure 1, the leftmost leaf node's node number is **8**; its label is '*b*'; and its chunk is "*aab*". At the beginning of the trie construction, the trie has q+1 nodes, including a root node with node number **0** and a *NULL* label and q child nodes each labeled with a unique symbol from the alphabet. During compression, LZW algorithm scans the text and finds the longest sub-string that appears in the trie as the chunk of a node N and outputs the node number of N as the code in $S.Z$. 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

N's chunk appended by the new node's label. At the end of the compression, there are $n+q$ nodes in the trie.

An example that was used in [Amir96] is presented in Figure 2 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)*.

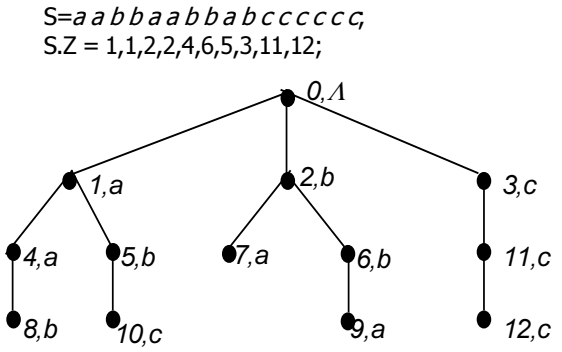


Figure 1 LZW Trie Structure

The following important observation makes it possible to construct the trie from *S.Z* in time *O(n)* without explicitly decoding S.Z:

Observation: When the decoder receives code S.Z[i], assuming S.Z[i-1] has already been received in the previous step ($2 \le i \le n$*, 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. For S.Z[1], no new node is created.*

Amir's algorithm performs the pattern matching directly on the trie. Because the above observation, no explicit decompression of the data is required. To facilitate the pattern matching, the following terms of a node in the trie are defined with respect to the pattern:

- A chunk is a *prefix chunk* if it ends with a non-empty pattern prefix; the *representing prefix* of a prefix chunk is the longest pattern prefix it ends with.
- A chunk is a *suffix chunk* if it begins with a non-empty pattern suffix; the *representing suffix* of a suffix chunk is the longest pattern suffix it begins with.
- A chunk is an *internal chunk* if it is an *internal sub-string* of the pattern, i.e. the chunk is $p_i...p_j$ for $i>1$ and $j \le m$. If $j=m$, the internal chunk also becomes a suffix chunk.

If a node's chunk is prefix chunk, suffix chunk or internal chunk, the node is called a *prefix node*, *suffix node* or *internal node*, respectively. To represent a node's representing prefix, a *prefix number* is defined for the node to indicate the length of the representing prefix, a value of 0 means that the node is not a prefix node; Similarly, to represent a node's representing suffix, a *suffix number* is defined for the node to indicate the length of the representing suffix, a value of 0 means that the node is not a suffix node; To represent a node's internal chunk status, an *internal range [i, j]* is defined to indicate that the node's chunk is an internal chunk $p_i...p_j$; a internal range [0, 0] means that the node is not an internal node.The prefix number, suffix number and internal range are computed for a node when the node is being added to the trie:

(a) The new node's internal range is computed as function Q3(I_P, a), where I_P is the internal range of its parent and a is its label;.

(b) If the result from step (a) tells that the new node is not only an internal node, but also a suffix node (j=m), set its suffix number as m-i+1. Otherwise the new node's suffix number is set as its parent's suffix number.

(c) The new node's prefix number is computed as function Q1(P_P, a), where P_P is the prefix number of its parent and a is its label.

When the new node's label is ***not*** in the pattern, the above computations can be easily answered; when the new node's label is in the pattern, the operands of function *Q1()* and *Q3()* are all sub-strings of the pattern. Since the number of the sub-strings of a given pattern is finite, we can pre-compute the results for all possible combinations of the operands. In [Amir96], this pre-processing of the pattern is done by Knuth-Morris-Pratt automaton [Knuth77] and the suffix-trie. The pre-processing takes time $O(m^2)$. Once the preprocessing is done, (a)-(c) can be answered in constant time.

The pattern matching is performed simultaneously as the trie is growing, as described in the following algorithm:

Pre-process the pattern.

Initialize trie and set global variable Prefix =NULL.

For i=2 to n, perform the followings after receiving code S.Z[i] (we will refer it as the current node)

Step 1. Add a new node in the trie as described in the above observation and compute the new node's prefix number, suffix number and internal range.

Step 2. Pattern Matching:

(a) If Prefix =NULL, set variable Prefix as current node's prefix number.

(b) If Prefix !=NULL and the current node is a suffix node, check the pattern occurrence from Prefix and the current node's representing suffix S; this checking is defined as function Q2(Prefix, S).

(c) If Prefix !=NULL and the current node's chunk is an internal chunk, compute Prefix as Q1(Prefix, I) where I is the current node's internal range.

(d) If Prefix !=NULL and the current node's chunk is not an internal chunk, set Prefix as the current node's prefix number.

Note that function *Q2()* can also be pre-processed by the KMP automata and the suffix trie of the pattern because both its two operands are sub-strings of the pattern.The algorithm has a total of $O(n+m^2)$ time and space complexity. A tradeoff between the time and space alternatively gives a $O(nlogm+m)$ time and $O(n+m)$ space algorithm.

3. The Proposed Method

Based on the work presented in section 2, we propose a CPM method that improves the original algorithm by adding important features such as searching all occurrences of the patterns and performing multiple pattern-matching simultaneously.

3.1 Reporting All Pattern Occurrences

In the pattern-matching algorithm presented in the previous section, the pattern occurrence checking is performed only at step 2(b). It is obvious that the algorithm assumes that the detected pattern crosses the boundary of the nodes. Thus, if the pattern occurs inside a node, as shown in the Figure 3, the above algorithm will not be able to find it.

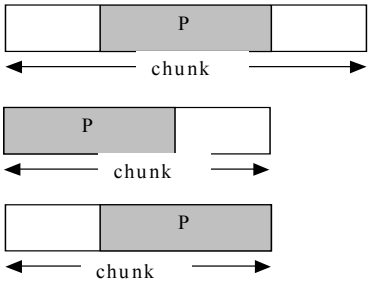


Figure 2 Illustration of the Pattern Inside a Node

The third case can be easily fixed by checking the value of variable *Prefix*: if it is equal to the length of the pattern, a pattern occurrence is found. For the first occurrence of a pattern, the first two cases shown in Figure 2 never happen. We can prove it by contradiction:

Suppose P's fist occurrence is in node *A* and it shows as the case in Figure 2, if node *B* is the parent of node *A*, it must have the same chunk as node *A* except the last symbol from node *A*. This is to say that *P* also occurs in node *B*. Since node B is added before node *A*, an earlier occurrence of *P* happens, this contracts with the assumption that *P*'s first occurrence is in node *A*.

The above discussion explains why Amir's algorithm reports only the first occurrence of *P*. To report all the occurrences of the pattern, we propose that, for each sub-pattern, a *PIC (stands for "pattern is contained") flag* is maintained to indicate that the pattern is a sub-string of the node's chunk. For example, if the chunk of a node is "*bcdef*", the PIC flag of the node is set as *true* with respect to pattern "*bcd*" while it is set as *false* with respect to pattern "*abc*".

Similar to prefix number, suffix number and internal range, each time a new node is added to the trie, its PIC flag is updated, after its prefix number and suffix number have been computed:

- *If its parent's PIC flag is on, a node's PIC flag is also on; otherwise, if the prefix number of the node equals to the length of the pattern, set the PIC flag on;*

The reasoning of the above operation is illustrated in Figure 3.

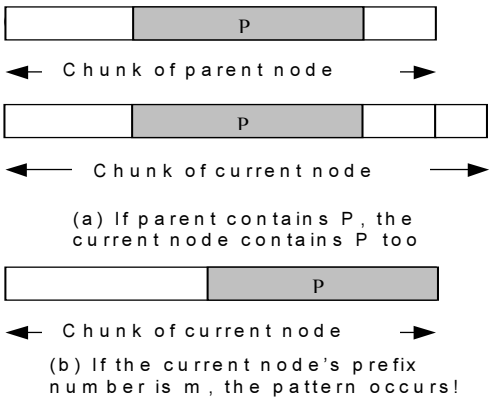


Figure 3 PIC Flag Update

The above operation turns on the PIC flag whenever it detects that the prefix number of a node equals to the length of the pattern. Once a node's PIC flag is on, all of its offspring's PIC flag is also on. By using the PIC flag, it is easy to identify the pattern occurrences inside the nodes. We will discuss pattern matching using PIC flag further in section 3.4.

3.2 Multiple Pattern Matching

Multiple pattern matching is often a useful feature in many applications such as when a Boolean query with many terms is performed or when interactive pattern searching is desired. Once the common "overhead" of searching different patterns is done, such as pre-processing of the compressed data or computation of the auxiliary array, it can be used for search many patterns. Thus, multiple pattern matching is not equivalent to performing pattern matching multiple times.

Amir's algorithm can be easily enhanced to perform multiple patterns matching by searching the multiple patterns simultaneously. The method requires each node to maintain a set of {*prefix number, suffix number, internal range, PIC flag*} for each pattern being searched. This method is practically feasible and has been tested through our implementation. However, since all the patterns have to be searched simultaneously, it is not able to provide searching refinement or interactive searching.

3.3 A Simple Implementation for "Simple Patterns"

Definition: *A simple pattern is a pattern that no symbol appears more than once.*

The examples of simple pattern are: *result, thus, world* and the examples of non-simple patterns are: *telephone, hello, pattern*. The following theorem is true for any simple pattern:

Theorem: Query Q1, Q2 and Q3 can be done in constant time for a simple pattern, where Q1 takes a pattern prefix P_P and an internal sub-string of the pattern I_P and returns the representing pattern prefix of P_PI; Q2 takes a pattern prefix P_P and a pattern suffix S_P and returns the first occurrence of the pattern. Q3 takes an internal sub-string of the pattern I_P and a symbol **a** from the pattern and it returns the position where I_Pa occurrences in the pattern.

If we denote P_P as $p_1...p_h$ $(h \le m)$, I_P as $p_i...p_j$ $(i>1$ and $j \le m)$, S_P as $p_{(m-k+1)}...p_m$ $(m-k+1 \ge 1)$, for any simple pattern P:

$$Q1(P_P, I_P) = \begin{cases} j & if(h+1=i) \\ 0 & else \end{cases}$$

$$Q2(P_P, S_P) = \begin{cases} 1 & if(h=m) \\ h+1 & elseif(k=m) \\ 1 & elseif(h+k=m) \\ 0 & else \end{cases}$$

$$Q3(I_P, a) = \begin{cases} [i, j+1] & if(a = p_{j+1}) \\ [0,0] & else \end{cases}$$

All the above equations can be easily proved. Thus, to search simple patterns, we simply plug the above implementations of Q1, Q2 and Q3 in the CPM algorithm presented in section 2. No pre-processing of the pattern is needed and thus the algorithm runs in time O(n+m).and in space $O(n+m)$.

3.4 The Proposed Algorithm

In [Amir96], internal number is represented as [i,j] where i and j represents the starting position and ending position of the internal chunk in a pattern, respectively. However, this representation has a problem in that the internal sub-string may appear in a pattern more than once. For example, if the internal chunk is "*abc*" and the pattern is "*a**abc**a**abc**d*", how can we represent the internal chunk? For simple patterns, this is never the case. However, for non-simple pattern cases, a more reasonable representation is needed. Fortunately, recall that in Section 2, a suffix trie is constructed as part of the pattern pre-processing and a sub-string of the pattern can be represented by a

unique node in the suffix trie, we then represent the internal chunk as a node number in the suffix trie. We now only give our searching algorithm as below:

If (pattern is a simple pattern) Use Tao's queries and pattern matching functions; Else, Preprocess the pattern using suffix trie and KMP automota; Use Amir's queries and pattern matching functions.

Initialize trie, set Prefix =NULL for each pattern
For i=2 to n Do:

1. Trie Updating:
(a) When receive a code S.Z[i], add a new node to the trie as described before.
(b) The new node's suffix number for a pattern is set as its parent's suffix number for that pattern SP_j
(c) For the new node's parent is an internal node with chunk I and the new node's label is ***a****, the new node's internal number is set as Q3(I, a); If the new node's chunk is also a suffix of* SP_j*, the suffix number for* SP_j *needs to be reset. The detection of the suffix chunk is easy: if the suffix trie is built, simply check if the internal number is a leaf node of the suffix tire; If no suffix trie (for simple patterns), check the right position of the internal chunk.*
(d) If the new node's parent is a prefix node with chunk P and the new node's label is ***a****, the new node's prefix number is set as* $Q1(P_{SPj}, a)$*; Otherwise check if* ***a*** *is a prefix of the pattern and set the new node's prefix number correspondingly.*
(e) The new node's PIC flag is set as we discussed in section 3.1

2. Pattern Matching:
(a) If Prefix =NULL

- *Set Prefix as the representing prefix of current node (the parent node of the new node).*
- *If the PIC flag of the current node is set, a pattern occurrence is found.*

(b) If Prefix !=NULL

- *If one of the following two conditions is true, a pattern occurrence is found:*
  (a) If the current node is a suffix node Q2(Prefix, the representing suffix of the current node) is true,
  (b) If the PIC flag of the current node is set
  In either case, we need to set Prefix as the representing prefix of the current node to prepare for searching the next pattern occurrence.
- *If none of the above two conditions are true:*

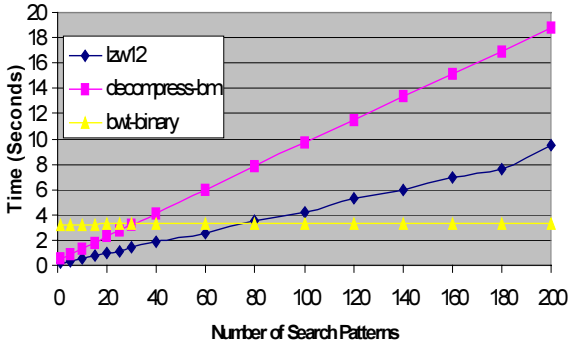
(a) If the current node is an internal node, set Prefix = Q1(Prefix, internal chunk); Otherwise set Prefix as the current node's representing prefix.

4. Experimental Results

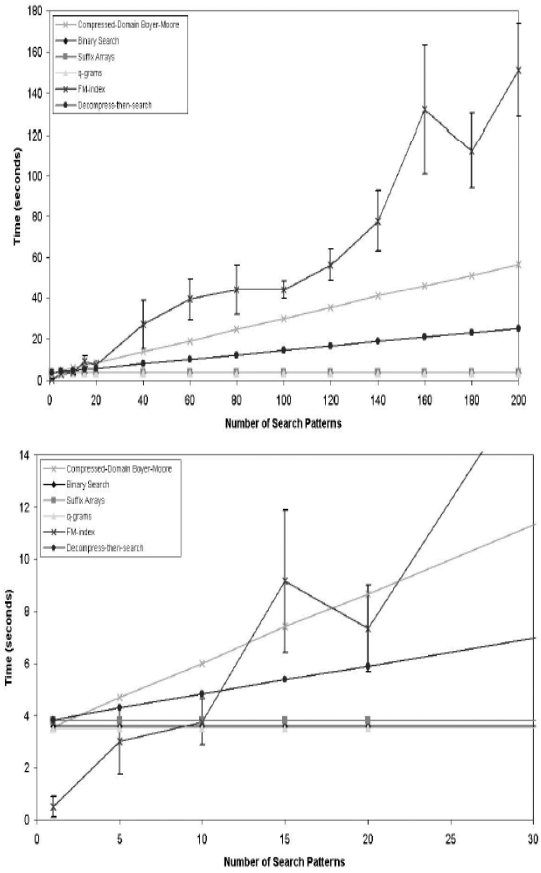
All experiments were conducted on a PC with the following configuration: CPU: Intel(R) Pentium(R) 4 1.80GHz; cache size: 512KB; total memory: 756MB. The OS is Linux 2.4.20

All data were obtained as 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. All file sizes are in bytes

- **Search Performance**



In the above chart, the searching performances of there different CPM algorithms are plotted: *lzw12*, which is our method; *decompress-bm*, which decompresses the LZW compressed file first and then applies the Boyer-Moore searching method on the decompressed file; *bwt-binary*, which is the BWT-based binary search algorithm proposed in [Bell2002] To have a better understanding of the searching performance of our method, we also "copy-and-paste" the results from [Firth2002] below. The performances depicted in the chart, from top to bottom, are of FM-Index, compressed domain Boyer-Moore, decompress-then-search, suffix array, binary search and q-gram, respectively. A magnified view of the results is also shown below. The performances of our approach can be compared with all BWT-based approaches based on binary search performance, which is depicted in both the chart above and the charts below.

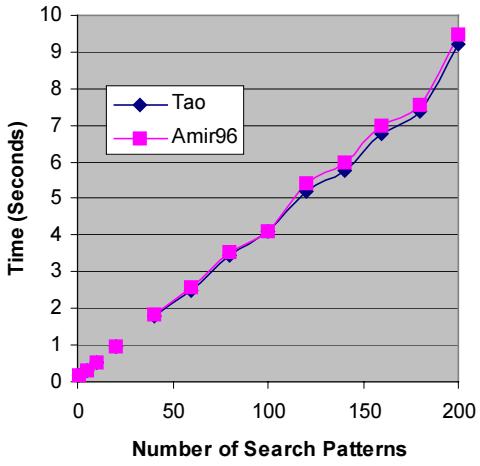


The charts show that our method outperforms all other CPM algorithms when the number of patterns is no more than 70-80 in this experiment.

- **"Simple Pattern" Matching**

For simple patterns, the implementation proposed in [Amir96] and the implementation proposed in section 3.3 are tested and compared. The following chart shows the results.

The performances of the two implementations are very close when the number of search patterns is small. Only when the number of patterns is large enough we can observe the advantage of the proposed implementation.



5. Conclusion and Future Work

A summary of the highlights of our work is:

- It is the first implementation of Amir's well-known algorithm and a CPM algorithm based on LZW algorithm.
- It overcomes the limitation of the original algorithm that only the first occurrence of the pattern is reported.
- It enhances the original method by providing multi-pattern matching.
- The proposed algorithm is "smarter" in that it automatically detects patterns that we define as "simple patterns" and applies a faster implementation of the searching algorithm.
- Extensive experiments have been conducted on the performance of our method and comparisons have made with the BWT-based algorithms. The results show that our work is competitive among the best CPM works.

Further research can be conducted for a more efficient multiple pattern matching method and for the approximate pattern matching algorithm. It is also possible to split any pattern into sub-patterns that are "simple patterns" to improve the searching performance.

Acknowledgement

The work has been partially supported by National Science Foundation grants IIS-0312724 and IIS-0207819.

References

[Adjeroh2002] D.A. Adjeroh, A. Mukherjee, M. Powell, T.C. Bell and N. Zhang, "Pattern matching in BWT-compressed text", Data Compression Conference, Snow Bird, Utah, April 2002, P. 445.

[Amir96] A. Amir, G. Benson and M. Farach, "Let sleeping files lie: Pattern matching in Z-compressed file", *Journal of System Sciences*, 52**:** 299-307, 1996.

[Barcaccia98] P. Barcaccia, A Cresti, S.D. Agostino, "Pattern matching in text compressed with the ID heuristic", *Data Compression Conference*, 113-118, 1998.

[Bell2001] T.C. Bell, D.A. Adjeroh and A. Mukherjee, "Pattern matching in compressed text and images", May 2001, available at:
(http://www.csee.wvu.edu/~adjeroh/progress/cpmPapers/acmSurvey2001.pdf).

[Bell2002] 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

[Farach98] M. Farach and M. Thorup, "String matching in Lempel-Ziv compressed strings", *Algorithmica*, **20**: 388-404, 1998

[Ferragina2001] P. Ferragina and G. Manzini, "An experimental study of an opportunistic index", *Proceedings, 12th ACM-SIAM Symposium on Discrete Algorithms, SODA 2001* pp. 269–278, 2001

[Firth2002] Andrew Firth, "A comparison of BWT approaches to compressed-domain pattern matching", Honours report at the University of Canterbury, available at:
(http://www.cosc.canterbury.ac.nz/research/reports/HonsReps/2002/hons_0205.pdf)

[Knuth77] D. E. Knuth, J. H. Morris V. R. Pratt, *SIAM J. Comput. 6*, 323, 1977

[Navarro99] G. Navarro and M. Raffinot, "A general practical approach to pattern matching over Ziv-Lempel compressed text", *Proceedings, Combinatorial Pattern Matching*, *LNCS* 1645: pp. 14-36, 1999.